Theoretical Computer Science 95 (1992) 207–229
Elsevier

Systolic convolution of arithmetic functions*

Patrice Quinton
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Yves Robert
LIP-IMAG, Ecole Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon Cedex 07, France

Communicated by D. Perrin
Received January 1989
Revised March 1990

Abstract

Quinton, P., and Y. Robert, Systolic convolution of arithmetic functions, Theoretical Computer Science 95 (1992) 207–229.

Given two arithmetic functions f and g, their convolution $h=f*g$ is defined as $h(n)=\sum_{kl=n,\,1\leqslant k,l\leqslant n} f(k)\,g(l)$ for all $n\geqslant 1$. Given two arithmetic functions g and h, the inverse convolution problem is to determine f such that $f*g=h$.

In this paper, we propose two linear arrays for the real-time computation of the convolution and the inverse convolution problem. These arrays extend the design of Verhoeff for the computation of the Möbius function μ, defined as the solution of the inverse convolution problem $\mu*g=\Delta$, where $g(n)=1$ for all $n\geqslant 1$ and $\Delta(n)=1$ if $n=1$, $\Delta(n)=0$ if $n>1$.

1. Introduction

Given two arithmetic functions f and g, their convolution $h=f*g$ is defined as $h(n)=\sum_{kl=n,\,1\leqslant k,l\leqslant n} f(k)\,g(l)$ for all $n\geqslant 1$. Given two arithmetic functions g and h, the inverse convolution problem is to determine f such that $f*g=h$. The inverse convolution problem is not always solvable; see [5] for a review.

For instance, let $E_0(n)=1$ for all $n\geqslant 1$ and $\Delta(n)=1$ if $n=1$, $\Delta(n)=0$ if $n>1$. The Möbius function μ is defined as

$$\mu(n)=1 \qquad \text{if } n=1,$$

*This work has been supported by the Research Program C3 of the French National Council for Research CNRS and by the ESPRIT Basic Research Action 3280 of the European Economic Community.

$\mu(n)=(-1)^r$ if n is the product of r distinct primes,

$\mu(n)=0$ if n is divisible by a prime square.

$d|n$ denotes that d is a divisor of n. The Möbius function can be more easily computed using the Möbius inversion formula [5]

$$\sum_{1\leqslant d\leqslant n,\, d|n} \mu(d)=\Delta(n),$$

which states that μ is the solution to the inverse convolution problem $\mu * E_0=\Delta$.

Verhoeff [16] proposes a systolic linear array for the real-time computation of the Möbius function μ. In this paper, we propose two systolic architectures for the real-time computation of the convolution and the inverse convolution of general arithmetic functions. Kung [6] introduces a systolic linear array for the computation of the sequence

$$h(n)=\sum_{k+l=n+1,\, 1\leqslant k,\, l\leqslant n} f(k)g(l), \quad n\geqslant 1,$$

where $(f(k); k\geqslant 1)$ and $(g(l); l\geqslant 1)$ are two given sequences. Such a computation corresponds to the product of two polynomials, or series, with coefficients $f(k)$ and $g(l)$.

Informally, if we want to compute $h(n)=\sum_{d|n} f(d)$, we can use a design similar to that of [6]. The problem is to inhibit the operation of the cells when they receive a pair of inputs $(f(d), h(n))$, where d is not a divisor of n, and this is the key to Verhoeff's design [16].

The general convolution problem $h=f*g$ is more difficult for two reasons:

- First we have to ensure that every component of f can meet every component of g.
- When a cell receives a pair of inputs $(f(d), h(n))$, where d is not a divisor of n, its operation has still to be inhibited. But when d is a divisor of n, we have to organize the flow of g such that $g(n/d)$ is also an input to the cell.

In this paper, we first propose a linear systolic architecture of $O(N)$ cells which solves the problem of computing $(h(n); 1\leqslant n\leqslant N)$ in time $O(N)$. This first solution is obtained using the dependence mapping method. The space–time complexity of the proposed architecture is $O(N^2 \log N)$. Then we propose another systolic architecture of only $O(N^{1/2})$ processing cells which also solves the problem in time $O(N)$. This second architecture requires $O(N \log N)$ delay cells, leading to the same space–time complexity as for the first solution. Both architectures can be extended to the solution of the inverse convolution problem, with the same performances.

Throughout the paper, we assume the reader to be familiar with the systolic model. Systolic arrays have been introduced by Kung and Leiserson [8] and consist of a large number of elementary processors (or cells), which are mesh-interconnected in a regular and modular way, and achieve high performance through extensive concurrent and pipeline use of the cells. We refer the reader to [7] for a general presentation of the systolic model.

2. A first linear systolic array

The arithmetic convolution consists in the computation of the following equation:

$$\forall n \geqslant 1, \quad h(n) = \sum_{kl=n,\ 1 \leqslant k, l \leqslant n} f(k)\, g(l), \tag{1}$$

where $f(k)$, $k \geqslant 1$ and $g(l)$, $l \geqslant 1$ are given integer functions.

Let K be a fixed parameter. Systolic arrays that compute the convolution

$$\forall n \geqslant 1, \quad h(n) = \sum_{1 \leqslant k \leqslant K} f(k)\, g(n-k+1) \tag{2}$$

are well known (see, for example, [7]). From the bidirectional design of Kung, one can easily derive a systolic array for the *unbounded convolution*, i.e. the convolution whose equation is

$$\forall n \geqslant 1, \quad h(n) = \sum_{1 \leqslant k \leqslant n} f(k)\, g(n-k+1). \tag{3}$$

The only difficulty is the loading of the coefficients g, as one cannot assume that they are preloaded in the array, as in the case of the bounded convolution.

The arithmetic convolution (1) is much harder, as the summation has only to be done for product terms $f(k)\, g(l)$ such that $kl = n$.

In what follows, we derive and prove the correctness of a systolic array which has the following characteristics:

- The array is linear and bidirectional. The number of cells needed for the computation of $h(n)$, $1 \leqslant n \leqslant N$, is N.
- The period of the array is 3, i.e. each cell works every three cycles.

We shall derive the design using a space–time transformation of a dependence graph of the algorithm, following the approach of Moldovan [10] and Quinton [11]. In Section 2.1, we recall the principles of the dependence mapping method on the example of the unbounded convolution. In particular, we explain how the coefficients $g(l)$ can be loaded in the cells. In Section 2.2, we develop the arithmetic convolution.

2.1. Unbounded convolution

Figure 1 depicts a dependence graph for the unbounded convolution. For the moment, we ignore the loading of the g coefficients. On this diagram, a coefficient $h(n)$ is computed along a diagonal line, by summing up the product terms $f(k)\, g(n-k+1)$ in increasing order of k. The result $h(n)$ is obtained at point $(n, 1)$. Deriving a systolic array from this dependence graph can be done easily using a space–time linear transformation (see [10, 11] among others). First we seek an affine (integral) schedule of the computations, i.e. a function $t(n, m)$ such that if calculation at point (n_1, m_1) depends on calculation at point (n_2, m_2), then $t(n_1, m_1) > t(n_2, m_2)$. Denote $t(n, m) = \lambda_1 n + \lambda_2 m + \alpha$, where λ_1, λ_2, and α belong to $\mathbf{Z}$, the set of integers. Let us call *dependence vector* the quantity $(n_1 - n_2, m_1 - m_2)$ for two dependent points. As the

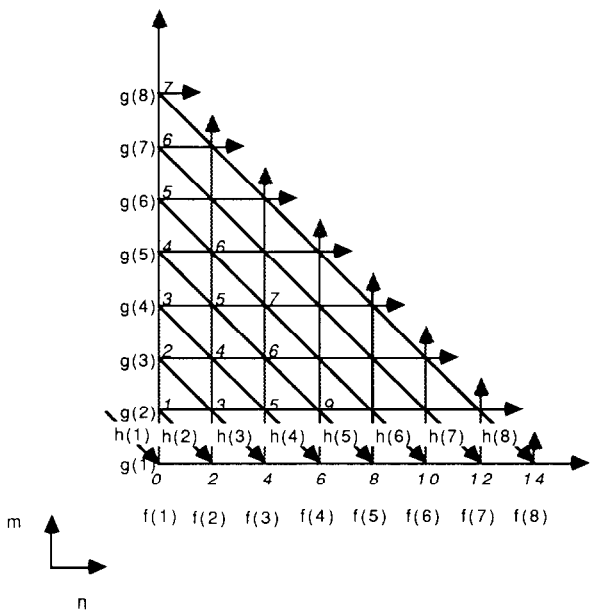


Fig. 1. Dependence graph and timing function for the convolution.

number of dependence vectors in this graph is finite (and independent of n), this condition amounts to a finite number of linear inequalities involving these vectors.

The set of possible dependence vectors is

$$\Delta = \{(0, 1), (1, -1), (1, 0)\}.$$

We must, therefore, have

$$\lambda_2 \geqslant 1; \qquad \lambda_1 - \lambda_2 \geqslant 1; \qquad \lambda_1 \geqslant 1.$$

These conditions are met optimally with $\lambda_1 = 2$ and $\lambda_2 = 1$. The coefficient α can be chosen in such a way that the computation starts at time 0 by taking $\alpha = 3$. Thus, an optimal timing-function is $t(n, m) = 2n + m - 3$.

A well-known systolic array can be obtained by projecting the array along the n axis. Figure 2 shows this systolic array which uses N bidirectionally connected cells, for the computation of $h(n)$, $1 \leqslant n \leqslant N$. The detail of the cells is shown in Fig. 3, where s is a boolean control signal.

Coefficients $h(k)$ are computed when they flow from the right to the left, and output by the leftmost cell. Note that the period of the array is 2, and this can be clearly seen on the dependence graph.

In this first design, coefficient $g(l)$ is pre-loaded in cell l. The best way to solve this problem is to fold the dependence graph along the bisectrix of the first orthant, in order to let f and g play a symmetric role with regard to the projection direction. However, to do so, it is first necessary to change slightly the way the h coefficient is

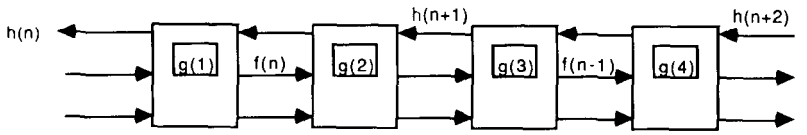
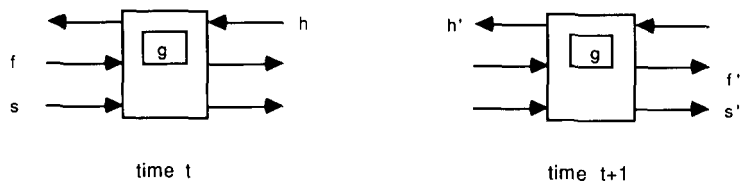


Fig. 2. Systolic array for the usual convolution.



$f' := f; s' := s;$
case
  s → $h' := f \cdot g$; {first computation}
  not s → $h' := h + f \cdot g$; {update}
esac

Fig. 3. Operation of the cells for the unbounded convolution.

summed up, so that the direction of its movement remains the same once the domain is folded. The trick is to compute the summation starting by the middle, as shown in Fig. 4.

In the new dependence graph, $g(l)$ and $f(l)$ flow together, starting from point $n = l$, $m = 1$, and are reflected along the line $n = m$. Therefore, $h(p)$, moving along the straight line $n + m = p + 1$, meets successively the pairs $(g(l), f(l))$ and $(g(p-l+1), f(p-l+1))$, when l ranges from $\lfloor p/2 \rfloor$ down to 1.

By projecting the dependence graph along the n axis, one obtains the systolic array depicted by Fig. 5, whose cells are shown in Fig. 6.

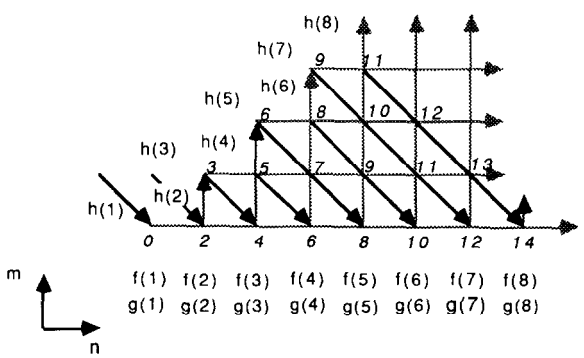


Fig. 4. Dependence graph and timing after folding.

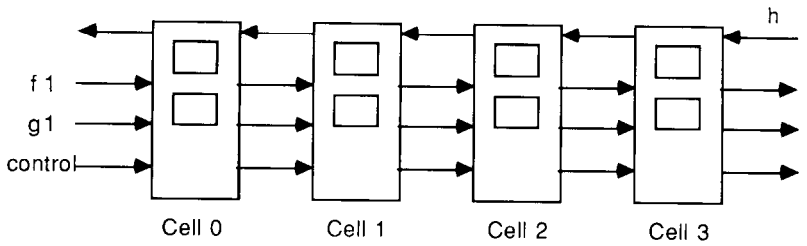
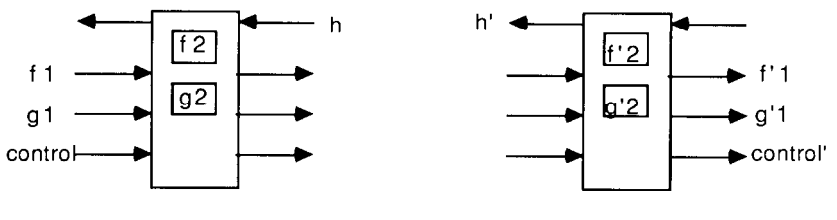


Fig. 5. Systolic array for the unbounded convolution.




```
case
    control = initodd→   {this case for the points on line n = m}
    begin
      f2 := f1; g2 := g1;   {load f1 and g1 in registers}
      h' := f1 * g2;   {compute middle term of the sum}
    end
    control = initeven→   {this case for the points on line n = m + 1}
    begin
      f'1 := f1; g'1 := g1;   {transmit f1 and g1}
      h' := f1 * g2 + f2 * g1;   {compute middle term of the sum}
    end
    otherwise→   {normal case}
    begin
      f'1 := f1; g'1 := g1;   {transmit f1 and g1}
      h' := h + f1 * g1 + f2 * g2;   {update with next term of the sum}
    end
esac
control' := control;   {transmit control}
```

Fig. 6. Detail of the cells of the systolic array for the unbounded convolution.

A few words of explanation about the control of the cells are in order. As shown in Fig. 4, depending on the parity of p, $h(p)$ starts its computation at the point $((p+1)/2, (p+1)/2)$ when p is odd, or at $(p/2+1, p/2)$ when p is even. Moreover, the pair of coefficients g, f is reflected on point $((p+1)/2, (p+1)/2)$. These peculiarities of the operation of the cell are taken care of by introducing a signal named *control* which can take the value *initodd*, *initeven* or *normal*. When *control* = *initodd*, the coefficients f and g are loaded in the registers, and h is initialized with the value $f * g$. When

control = *initeven*, h is initialized with $f1 * g2 + f2 * g1$. Finally, in the normal case, h is incremented with $f1 * g2 + f2 * g1$. The signal *control* flows along the direction given by vector (1, 1) on Fig. 4. Therefore, it visits a new cell every three cycles. Along the diagonal $n = m$, *control* = *initodd*. Along the line $n = m + 1$, *control* = *initeven*. Otherwise, *control* = *normal*.

2.2. *Arithmetic convolution*

The problem is to compute the convolution $h = f * g$. For all $n \geq 1$, we need to evaluate the sum $h(n) = \sum_{kl=n,\, 1 \leq k,\, l \leq n} f(k)\, g(l)$. The constraints are the following:

- *Communications with the host*: there is only one cell in the array communicating with the host. This cell receives the input sequences $(f(n); n \geq 1)$ and $(g(n); n \geq 1)$ and delivers the output sequence $(h(n); n \geq 1)$.
- *Real-time computation*: for all $n \geq 1$, $h(n)$ should be output by the array k units of time after the input of $f(n)$ and $g(n)$, where k is a fixed constant independent of n,
- *Modularity*: the operation of the cells should not depend upon their location in the array, nor should it depend on the indices of their inputs/outputs. Provided that this condition is met, the array can be used for the computation of an arithmetic convolution of any size.

Figure 7 shows a dependence graph for the arithmetic convolution. As in the usual convolution, coefficient $h(p)$ is computed along the diagonal line, whose equation is $n + m = p + 1$. The problem is to "route" the coefficients g and f used for computing the product terms, in such a way that the graph be uniform and that these coefficients meet on the right line. The choice that is made here consists in routing the coefficients g and f in a *symmetric way*, so that $g(n)$ and $f(n)$ meet on the first bisectrix. For example, $g(2)$ and $f(2)$ meet at point (5/2, 5/2), just on the line where $h(4)$ is summed up, and coefficients $g(3)$ and $f(3)$ at point (5, 5), which is just on the line $m = 10 - n$ where $h(9)$ is computed. The diagram shows intuitively that when the g's and the f's flow on straight lines, they meet on and only on the lines where they are used. This is confirmed by the following obvious lemma.

Lemma 2.1. *Consider the family of points*

$$X(k, l) = (G_l(k), F_k(l)) = ((k-1)(l+1)/2 + 1, (l-1)(k+1)/2 + 1),$$

where $k \geq 1$ *and* $l \geq 1$.

Then the curves $\{X(k,l)\}_{k \geq 1}$ *and* $\{X(k,l)\}_{l \geq 1}$ *are straight lines. Moreover, they meet on the line* $m = kl + 1 - n$.

Lemma 2.1 clearly states that the routing scheme shown in Fig. 7 is correct. However, as it is, this dependence graph cannot be used for deriving a systolic array, for three reasons:

(1) the underlying lattice of the graph is not integral;
(2) there exists an unbounded number of dependence vectors;

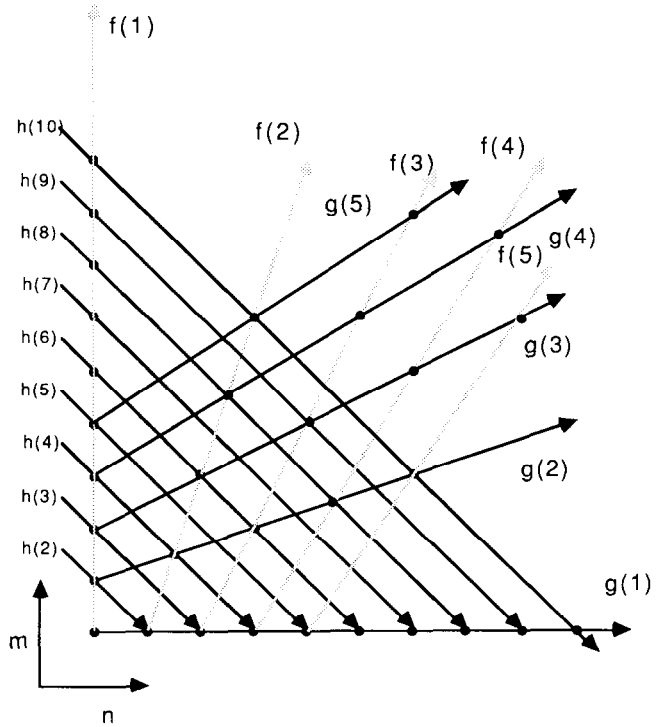


Fig. 7. Dependence graph for the arithmetic convolution.

(3) we must provide a mechanism to avoid pre-loading of the g coefficients.

Problem 1 can be solved by a dilation of the graph by 2 along both directions. We shall also solve problem 3 by applying the same trick as for the unbounded convolution, i.e. by folding the dependence graph.

Problem 2 is more involved. The solution consists in replacing the straight lines where the coefficients g and f flow by a piecewise linear curve that meets the following conditions:

(a) the curve follows a finite number of directions;

(b) all the intersection points are on the curve;

(c) the number of curves passing through a given point of the plane is bounded.

Condition (a) is necessary if one wants to obtain a finite number of dependence vectors. Condition (b) is obviously necessary to keep the properties of the previous dependence graph. Finally, condition (c) is needed for the resulting systolic array to have only a finite amount of data to be transmitted from one cell to another. This last condition is the reason why we have chosen the symmetric routing scheme (other simpler routing schemes are possible; for example, to have $f(k)$ flow on line $n=k$ and route $g(l)$ to the point $(k, kl+1)$, but in this case, condition (c) could not be met).

Let us first ignore the problem of loading the g coefficients. Figure 8 shows the dependence graph after dilation by 2 along both axes (in order for the intersection

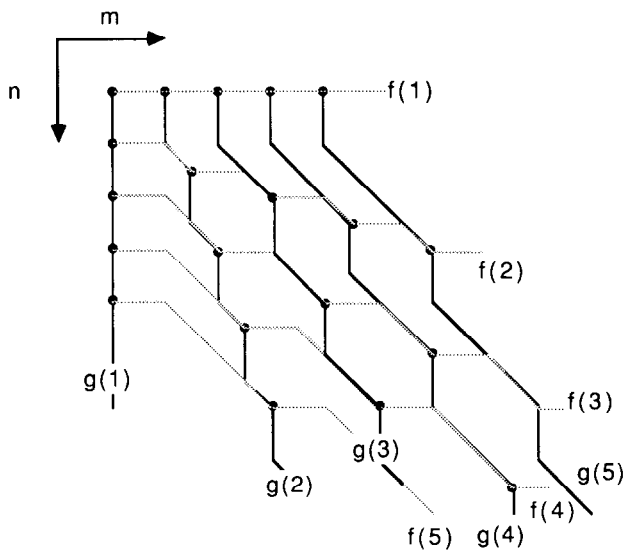


Fig. 8. Piecewise approximation of g and f movement.

points to be on an integral lattice), once the straight lines are replaced by a piecewise approximation. This approximation can be explained as follows.

Let $\{G_l'(k, i)\}_{k\geqslant 1,\, 0\leqslant i<l+1}$ be the curve, parameterized by k and i, defined by

$$G_l'(k,i)=\begin{cases}((k-1)(l+1)+i+2,\,(l-1)(k+1)+2) & \text{if } 0\leqslant i\leqslant 2,\\ ((k-1)(l+1)+4,\,(l-1)(k+1)+i) & \text{if } 2\leqslant i<l+1.\end{cases}$$

Informally, the points of this curve are obtained by starting at point $(2, 2l)$, and moving once along vector $(2, 0)$ and then $l-1$ times along vector $(1, 1)$.

Similarly, let $\{F_k'(l, i)\}_{l\geqslant 1,\, 0\leqslant i<k+1}$ be the curve, parameterized by l and i, defined by

$$F_k'(l,i)=\begin{cases}((k-1)(l+1)+i+2,\,(l-1)(k+1)+2) & \text{if } 0\leqslant i\leqslant 2,\\ ((k-1)(l+1)+4,\,(l-1)(k+1)+i) & \text{if } 2\leqslant i<k+1.\end{cases}$$

The following lemma proves that the piecewise approximation meets the previously stated conditions.

Lemma 2.2. *For all* $l, k\geqslant 0$,

(1) $\{G_l'(k, i)\}_{k\geqslant 1,\, 0\leqslant i<l+1}$, $\{F_k'(l, i)\}_{l\geqslant 1,\, 0\leqslant i<k+1}$, *and the straight line* $n+m=2(kl+1)$ *are concurrent,*

(2) *The family of curves* $\{G_l'(k, i)\}$ ($\{F_k'(l, i)\}$) *has no intersection point.*

Proof. (1) is easily seen, as $G_l'(k, 0)=F_k'(l, 0)=((k-1)(l+1)+2, (l-1)(k+1)+2)$, which is just twice the coordinates of the point X that we have already seen in Lemma 2.1. The proof of (2) is also very simple. □

The loading of the g coefficients is solved in a similar way as already seen in the unbounded convolution, i.e. by folding the dependence graph along the line $n=m$. The resulting dependence graph is shown in Fig. 9. The pairs $(f(n), g(n))$ are input along the n axis, and are reflected when they meet the line $n=m$. The dependence vectors are also shown in Fig. 9. They are

$$\Delta = \{(0, 2), (1, 1), (2, 0), (-1, 1)\}.$$

Therefore, the parameters λ_1 and λ_2 of the timing function must satisfy

$$2\lambda_2 \geqslant 1, \qquad \lambda_1 + \lambda_2 \geqslant 1,$$

$$2\lambda_1 \geqslant 1, \qquad \lambda_1 - \lambda_2 \geqslant 1.$$

The optimal (rational) solution is $\lambda_1 = 3/2$ and $\lambda_2 = 1/2$, which gives the timing function $t(n, m) = \lfloor 3n/2 + m/2 - 4 \rfloor$, as shown in Fig. 9 (the interested reader is referred to [11] for the use of rational timing functions).

The final architecture (Fig. 10) is obtained by projecting the dependence graph along n axis. We can immediately see that the number of cells needed for computing

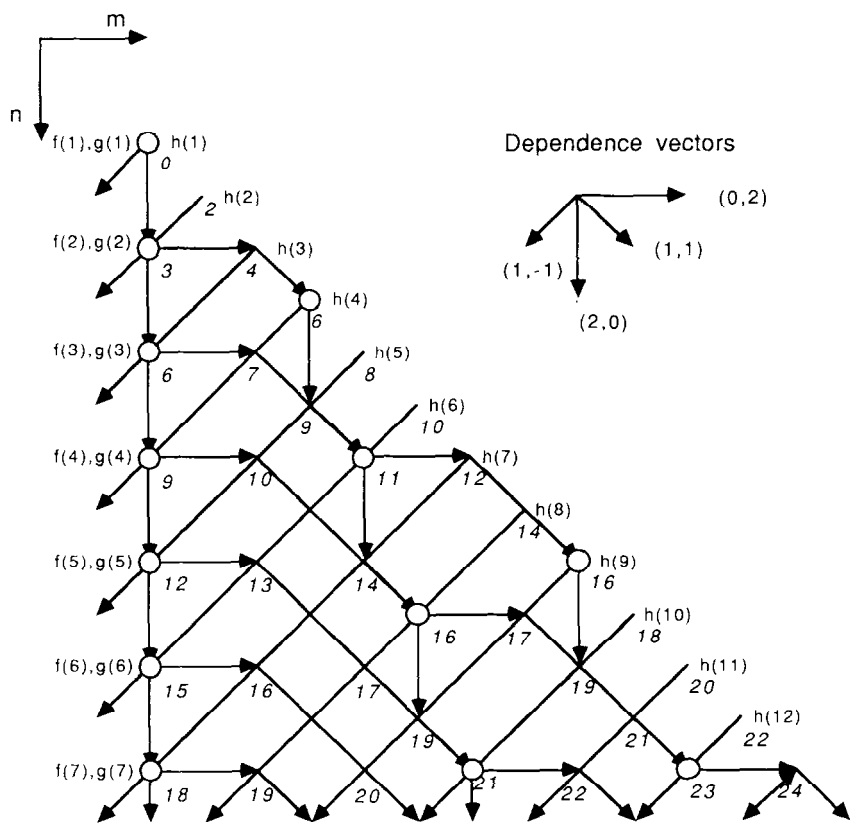


Fig. 9. Final dependence graph for the arithmetic convolution.

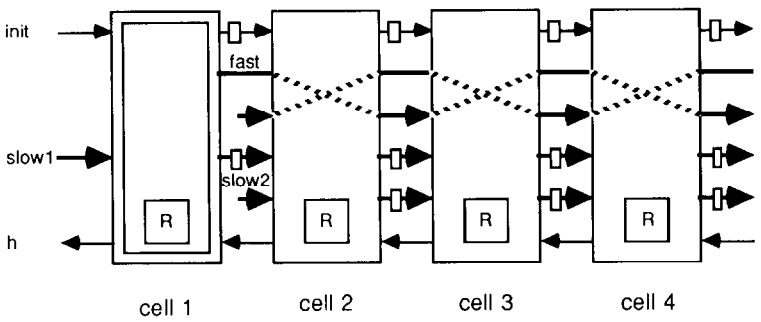


Fig. 10. Systolic array for the arithmetic convolution.

the sequence $h(k)$, $1 \leqslant k \leqslant n$ is n. Each cell of the array works only every three cycles. Each cell except for the first one has four left-to-right, and one right-to-left links:

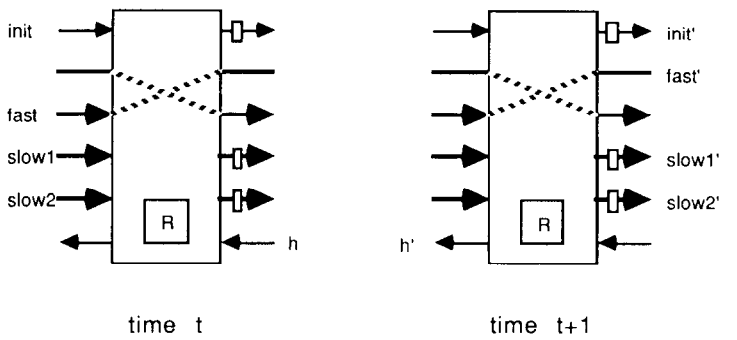
- *init* carries a initialization signal which follows the line $n=m$ on the dependence graph of Fig. 9. This signal reaches a new cell every other time;
- *fast* carries the f and g coefficients moving along direction (0, 2) on Fig. 9, as well as other informations that will be detailed later on. After being processed by a cell, these values skip the next cell and reach the second next one;
- *slow1* and *slow2* carry two pairs of f and g coefficients (as well as other informations) moving along direction (1, 1);
- h carries the value of the h coefficient under computation.

Finally, each cell is provided with a register R which keeps the pair f and g flowing along the direction (2,0) in Fig. 9: this direction being projected in the same cell, it corresponds to the storage of a value in one cell. The first cell is special. It has only a *slow1*, an h and an *init* link.

The details of the operation of the cells are shown in Fig. 11. Actually, each one of the *fast*, *slow1* and *slow2* link carries a 5-tuple (f, g, *valid*, *rank*, *count*), where f and g are coefficients, *valid* is a boolean indicating that the link effectively carries significant values, *rank* is the number of the coefficients, and *count* is an integer which will be used to determine the movement of the pair f, g. To understand the operation of the cell, it is best to refer again to the dependence graph of Fig. 9. As already seen, a pair $(f(k), g(k))$ moves once along the direction (0, 2) and then k times along the direction (1, 1). After the pair is reflected along the line $n=m$, it moves once along the direction (2, 0) and k times along the direction (1, 1). The *rank* parameter is used to remember the k value, and *count* is decreased when doing the k movements along the direction (1, 1). When *count* reaches 0, then two pairs of coefficients are available on *slow1* and *slow2* and the h coefficient is modified.

2.3. *Performances*

In fact, although each cell is activated every three cycles, the actual efficiency of the network is less than 1/3, as the number of calculations to be done (in sequential) for




```
case
    init and slow1.count ≠ 0→   {initialization and no calculation}
    begin h' := 0; init' := init end;
    init and slow1.count = 0→   {initialization and calculation}
    begin
      h' := h + slow1.f * slow1.g; R := slow1   {store slow1 in R}; init' := init
    end;
    not init and fast.valid→   {send fast on slow1, reset count}
    begin
      slow1' := fast; slow1'.count := rank − 1; init' := init; h' := h
    end;
    not init and slow1.valid and slow1.count ≠ 0 and R.valid→
    begin   {send R on slow2, decrease count of slow1}
      slow1' := slow1; slow1'.count := slow1'.count − 1;
      slow2' := R; R.valid := false; init' := init; h' := h
    end;
    not init and slow1.valid and slow1.count ≠ 0 and not R.valid→
    begin   {keep values moving on slow lines, decrease count of slow1}
      slow1' := slow1; slow1'.count := slow1'.count − 1;
      slow2' := slow2; init' := init; h' := h
    end;
    not init and slow1.valid and slow1.count = 0→
    {note that slow2 is necessarily valid}
    begin {compute, send slow1 on fast and keep slow2 in R}
      h' := h + slow1.f * slow2.g + slow2.f * slow1.g;
      R := slow2; fast' := slow1; init' := init
    end
esac
```

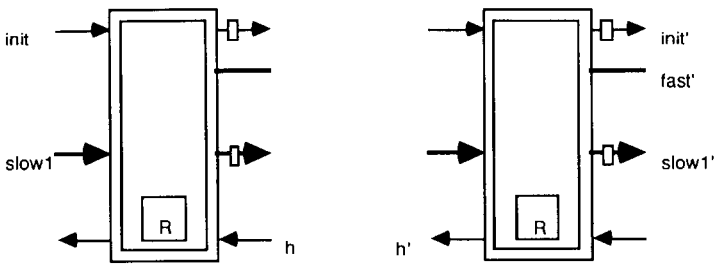
Fig. 11. Operation of cell i, $i \geqslant 2$.

computing $h(n)$ is not of the order of n: computing $(h(n);\ 1 \leqslant n \leqslant N)$ requires

$$\sum_{1 \leqslant n \leqslant N} \mathrm{div}(n) = O(N \log N) \text{ multiplications,}$$

where $\mathrm{div}(n)$ is the number of divisors of n.

In summary, this first solution uses $O(N)$ cells for processing in time $O(N)$ the computation of the sequence $h(n)$, $1 \leqslant n \leqslant N$. However, as each cell n has to make use of a counter initialized to n, the area complexity of this design is $O(N \log N)$.



time t time t+1

case
init→ {initialization: compute and store in R}
begin
$h' := slow1.f * slow1.g$; $R := slow1$; {store in R}
$init' := init$
end;
not *init*→ {first cell, normal case}
begin
$h' := h + slow1.f * R.g + R.f * slow1.g$;
$R := R$ {store in R}; $init' := init$
end;
esac

Fig. 11 (*continued*). Operation of the first cell.

3. Another systolic design

In this section, we design another systolic array of processors for the parallel computation of the convolution of two arithmetic functions. We address the inverse convolution problem in Section 3.4. The second array will look like the one shown in Fig. 12.

3.1. Half computation

First, we show how to compute the sum $h_1(n) = \sum_{kl=n,\ k \geqslant l} f(k)\, g(l)$. The other half of the arithmetic convolution will be computed similarly. We use a linear array of cells, as in Fig. 12, and number the cells from left to right.

- *Input and output (I/O) format.* For all $n \geqslant 1$, $f(n)$ and $g(n)$ enter the array at time n; $h_1(n)$ is output at time n (see Fig. 13).

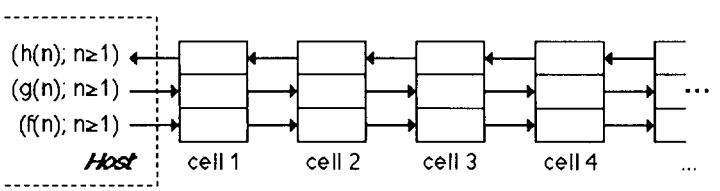


Fig. 12. The second linear array.

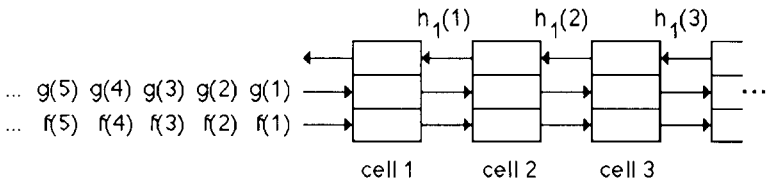


Fig. 13. Input/output format.

- *Flow of* $g(l)$. For all $l \geqslant 1$, $g(l)$ flows rightwards until it reaches cell l. It is then stored in the internal register g_reg of cell l. The systolic mechanism to realize such a flow is well known: cells are marked when their register g_reg is filled, and each $g(l)$ flows rightwards until it finds a nonmarked cell.
- *Flow of* $f(k)$. For all $k \geqslant 1$, $f(k)$ flows rightwards until cell k, where it initializes the operation of the cell; then $f(k)$ is marked nonactive. Therefore, $f(k)$ is the first active f-input of cell k for all k. When it reaches cell k, $f(k)$ meets $g(k)$ and the first product $f(k) * g(k)$ is computed for $h_1(k^2)$. According to the I/O format, $h_1(k^2)$ is an input of cell 1 at time k^2, so the product $f(k) * g(k)$ must be computed at time $k^2-(k-1)$ in cell k. The flow of the f's is organized to meet this requirement: $f(k)$ reaches cell k at time k^2-k+1. We detail hereafter the organization of the flow of f.
- *Flow of* $h_1(n)$. Since $h_1(n)$ is in cell 1 at time n, we can conceptually say it is in cell n at time 1 and moves leftwards from cell to cell at speed 1. In fact, the first product of $h_1(n)$ is computed in cell l, where l is the largest divisor of n such that $l^2 \leqslant n$, at time $n-l+1$. As it moves leftwards, $h_1(n)$ accumulates partial products $f(k) * g(l)$ in decreasing order with respect to l, in all cells l such that l divides n.

Now we need to organize the flow of the f's. We examine the first meetings that are required. In Table 1, we report the times when, and the cells where, products are computed. From Table 1 we see that cell k is activated every kth step after receiving its first f-input $f(k)$ at time k^2-k+1. In other words, $f(j)$ is input to cell k at time $kj-k+1$ for all $j \geqslant k$. For $j \geqslant k+1$, $f(j)$ is input to cell $k+1$ at time $(k+1)j-(k+1)+1$ so that $f(j)$ should be delayed by $j-2$ units of time in cell k before being output towards cell $k+1$.

There is a special processing for $f(1)$ by cell 1, which acts slightly differently from the other cells: rather than marking $f(1)$ nonactive, it deletes it (equivalently, it can mark it with some special code). As a consequence, the ith f-input to all cells except the first one should be delayed by $i-1$ units of time. The first input, namely $f(2)$, is transmitted without delay. The second input, namely $f(3)$, is delayed by one unit of time, and so on. After having deleted $f(1)$, cell 1 operates exactly as the other cells.

Right now, we only need to design a special systolic mechanism to generate these delays: then we add one of them to each cell, and the flow of the f's will be correct. Such a mechanism cannot be implemented using counters because of the modularity constraint. We use a design similar to that described in [9, 13].

The delay mechanism is depicted in Fig. 14. It is a two-column array of delay cells.

Table 1
Space–time diagram for the computation of partial products

Times	Cell 1	Cell 2	Cell 3	Cell 4	Cell 5
1	$f(1)\cdot g(1)$				
2	$f(2)\cdot g(1)$				
3	$f(3)\cdot g(1)$	$f(2)\cdot g(2)$			
4	$f(4)\cdot g(1)$				
5	$f(5)\cdot g(1)$	$f(3)\cdot g(2)$			
6	$f(6)\cdot g(1)$				
7	$f(7)\cdot g(1)$	$f(4)\cdot g(2)$	$f(3)\cdot g(3)$		
8	$f(8)\cdot g(1)$				
9	$f(9)\cdot g(1)$	$f(5)\cdot g(2)$			
10	$f(10)\cdot g(1)$		$f(4)\cdot g(3)$		
11	$f(11)\cdot g(1)$	$f(6)\cdot g(2)$			
12	$f(12)\cdot g(1)$				
13	$f(13)\cdot g(1)$	$f(7)\cdot g(2)$	$f(5)\cdot g(3)$	$f(4)\cdot g(4)$	
14	$f(14)\cdot g(1)$				
15	$f(15)\cdot g(1)$	$f(8)\cdot g(2)$			
16	$f(16)\cdot g(1)$		$f(6)\cdot g(3)$		
17	$f(17)\cdot g(1)$	$f(9)\cdot g(2)$		$f(5)\cdot g(4)$	
18	$f(18)\cdot g(1)$				
19	$f(19)\cdot g(1)$	$f(10)\cdot g(2)$	$f(7)\cdot g(3)$		
20	$f(20)\cdot g(1)$				
21	$f(21)\cdot g(1)$	$f(11)\cdot g(2)$		$f(6)\cdot g(4)$	$f(5)\cdot g(5)$

The first column is composed of cells with one input and three outputs. The operation of the cells in the first column is very simple (Fig. 15):

- the first input is output rightwards on the fast channel;
- the second input is output rightwards on the slow channel;
- all following inputs are output downwards to the next cell in the column.

The cells of the second column simply transmit their valid input, if any (to implement this, they perform an OR-operation on their two inputs, where non-specified variables have the default value *nil*). Some consecutive time-steps of the mechanism are illustrated in Fig. 16.

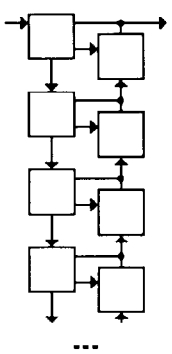
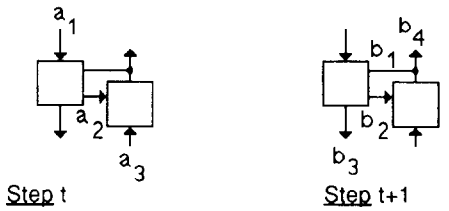


Fig. 14. Delay mechanism.



case *status* **of**
first: **begin** $b_1 := a_1$; *status*:=*second*; **end**
second: **begin** $b_2 := a_1$; *status*:=*marked*; **end**
marked: **begin** $b_3 := a_1$; **end**
$b_4 := a_2$ OR a_3;

Fig. 15. Operation of the delay mechanism.

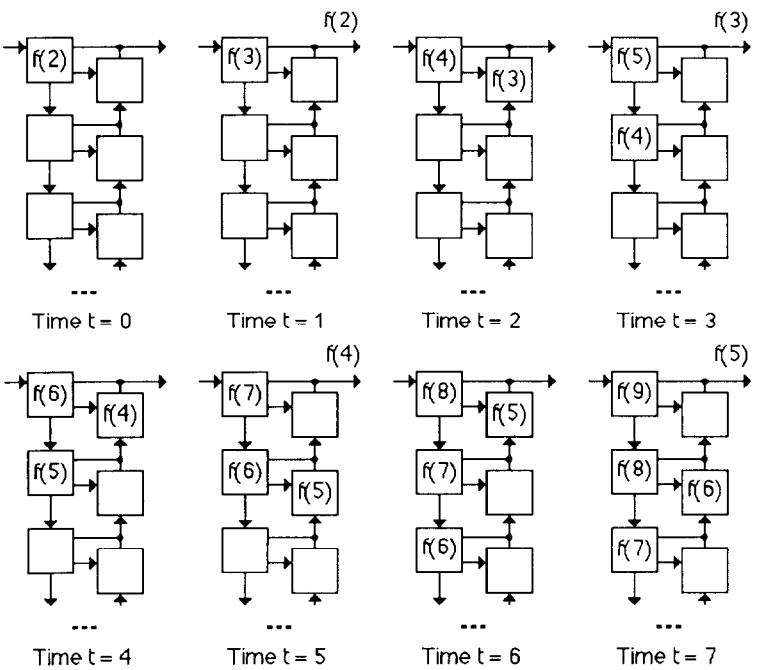
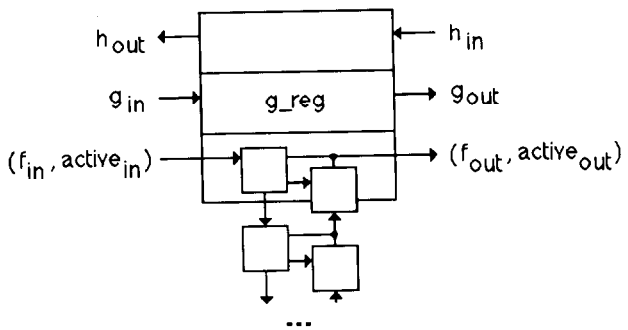


Fig. 16. Some consecutive time-steps for the delay mechanism.

The full operation of the cells for the computation $h_1(n) = \sum_{kl=n,\, k \geqslant l} f(k)g(l)$ is described in Fig. 17, where nonspecified variables have the default value *nil*. As stated above, the operation of cell 1 is slightly different since its first input is deleted.

Proof of correctness

We know that $f(k)$, $k \geqslant 1$, reaches cell l at time $kl + k - 1$. If $k \geqslant l$, $f(k)$ is active in cell l, and the product $f(k) * g(l)$ is computed. Consider the computation of $h_1(n)$: $h_1(n)$ is




```
{store g_in in g_reg if nonmarked}
if nonmarked then
  begin g_reg := g_in; nonmarked := false; end
else g_out := g_in;
{active f-input}
if active_in then
  begin
    {inactivate first f_input}
    if first_f then begin active_in := false; first_f := false; end
    {update h_in}
    h_out := h_in + g_reg * f_in;
  end
{delay all f_inputs}
(f_out, active_out) := Delay_Mechanism (f_in, active_in);
```

Fig. 17. Operation of the cells of the array.

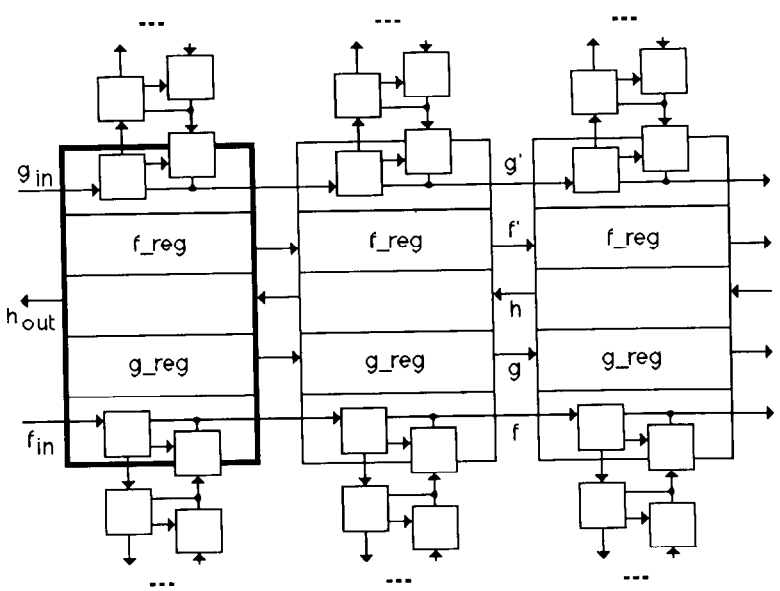
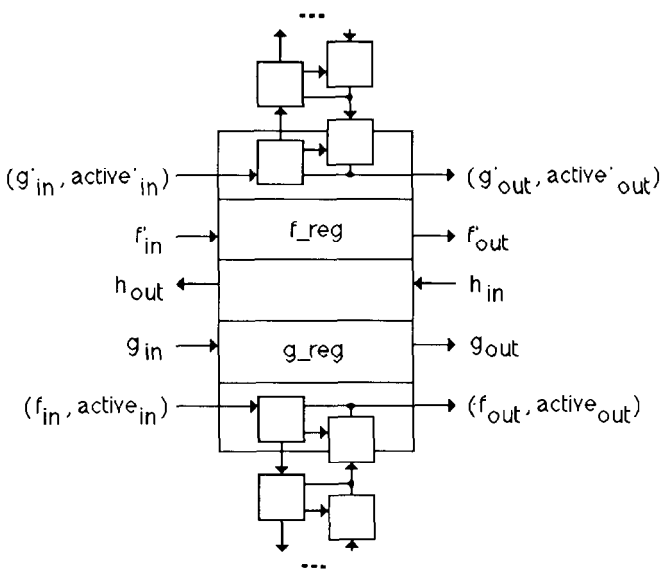


Fig. 18. Systolic array for the arithmetic convolution $h = f * g$.




```
{store g_in in g_reg and f'_in in f_reg if nonmarked}
if nonmarked then
  begin g_reg := g_in; f_reg := f'_in; nonmarked := false; end
else
  begin g_out := g_in; f'_out := f'_in; end
{active f-input and g'-input. Note that active_in = active'_in by symmetry}
if active_in then
  begin
    {inactivate f- and g'-inputs}
    if first_fg' then
      begin active_in := false; active'_in := false; first_fg' := false; end
      {update h_in}
      h_out := h_in + g_reg * f_in + f_reg * g'_in;
  end
{delay all f- and g'-inputs}
(f_out, active_out) := Delay_Mechanism (f_in, active_in);
(g'_out, active'_out) := Delay_Mechanism (g'_in, active'_in);
```

Fig. 19. Operation of the cells for the arithmetic convolution $h=f*g$.

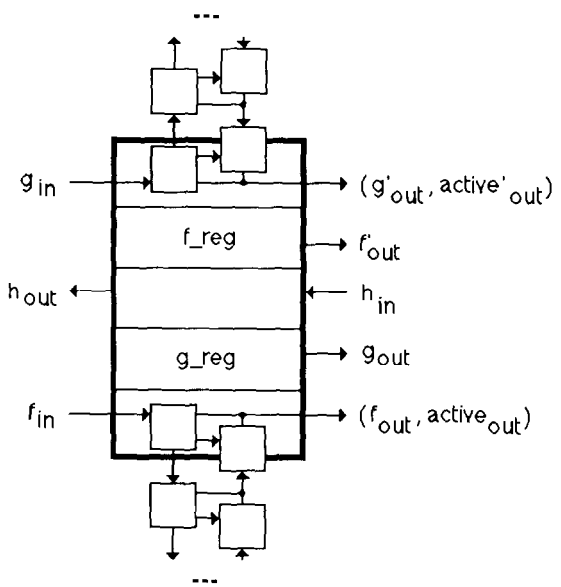
in cell l at time $n-l+1$. It meets some $f(k)$ there if and only if $kl-l+1=n-l+1$, i.e. $kl=n$. Then $h_1(n)$ is updated into $h_1(n):=h_1(n)+f(k)g(l)$ if and only if $f(k)$ is active or, equivalently, $k \geqslant l$. Therefore, the final value of $h_1(n)$ is $h_1(n)=\sum_{kl=n,\, k\geqslant l} f(k)\, g(l)$ as expected.

3.2. Systolic arithmetic convolution

For computing $h(n)=\sum_{kl=n} f(k)\, g(l)$, we use two copies of the previous array. In the first array, we compute $h_1(n)=\sum_{kl=n,\, k\geqslant l} f(k)g(l)$ as before.

In the second array, we compute $h_2(n)=\sum_{kl=n,\, k>l} g(k)f(l)$. We interchange the flows of f and g in the second array: the f's are stored in the cells, and the g's move

rightwards with delays. The only modification is that the second array should not compute products $f(k)*g(k)$, as they are already computed by the first array. We simply modify the operation of the cells (except the first one) as follows: the first time they receive an active g-input, they let $h_2(k^2):=0$ rather than $h_2(k^2):=f(k)*g(k)$. In fact, we can make things simpler by coalescing the corresponding cells of both arrays, as described in Fig. 18. Again, the first cell is slightly different because it deletes its first f-input (bottom part) and its first g-input (top part). Also, it duplicates f- and g-inputs. See Fig. 19 for the operation of all cells but the first one, and Fig. 20 for the operation of the first cell.



if *nonmarked* **then**
  begin
    {$g_{in}=g(1)$; $f_{in}=f(1)$; store and mark cell}
    $f\_reg:=f_{in}$; $g\_reg:=g_{in}$; $nonmarked:=false$;
    {delete f_{out} and g'_{out}} $f_{out}:=nil$; $g'_{out}:=nil$;
    {compute $h(1)$} $h_{out}:=f_{in}*g_{in}$;
  end
else
  begin
    {transmit g_{in} and f'_{in}} $g_{out}:=g_{in}$; $f'_{out}:=f_{in}$;
    {update h_{in}} $h_{out}:=h_{in}+g\_reg*f_{in}+f\_reg*g'_{in}$;
    {activate f-input and g'-input} $active_{in}:=true$; $active'_{in}:=true$;
    {delay all f- and g'-inputs}
    $(f_{out}, active_{out}):=Delay\_Mechanism(f_{in}, active_{in})$;
    $(g'_{out}, active'_{out}):=Delay\_Mechanism(g'_{in}, active'_{in})$;
  end

Fig. 20. Operation of first cell for the arithmetic convolution $h=f*g$.

3.3. Performances

For the computation of $(h(n);\ 1 \leqslant n \leqslant N)$, we have designed an array of $N^{1/2}$ processing cells (which perform multiply-and-adds). Note that the total number of delays is proportional to $N \log N$, and not to N, although we have only $O(N)$ inputs. To see this, consider for instance the first array. The last element that we need to consider in cell k is $f(N/k)$, to be multiplied by $g(k)$. $f(N/k)$ has been delayed by N/k units of time in cells $1, 2, \ldots, k-1$ before reaching cell k. So that we need $N/2$ delays in cell 1 (due to $f(N/2)$), $N/3$ delays in cell 2 (due to $f(N/3)$), $N/4$ delays in cell 3 (due to $f(N/4)$), and so on up to $N/N^{1/2}$ delays in the cell before the last one (due to $f(N^{1/2})$).

3.4. The inverse arithmetic convolution problem

The previous array can be very easily modified to solve the inverse arithmetic problem. This is quite similar to the technique used for moving from FIR filtering to IIR filtering [7] or from polynomial multiplication to polynomial division [6].

To compute (whenever possible) the function f such that $f * g = h$, we observe that

$$f(1) = h(1)/g(1),$$

$$f(n) = \left[h(n) - \sum_{kl=n,\ 1 \leqslant k,\ l \leqslant n,\ k \neq n} f(k)g(l) \right] \Big/ g(1) \quad \text{if } n \geqslant 1.$$

We input to the array the sequence $(g(n);\ n \geqslant 1)$ in the same format as before. We replace the input sequence $(f(n);\ n \geqslant 1)$ by the sequence $(h(n);\ n \geqslant 1)$, with the same format (Fig. 21). All cells operate exactly as before, except the first one whose program is given in Fig. 22.

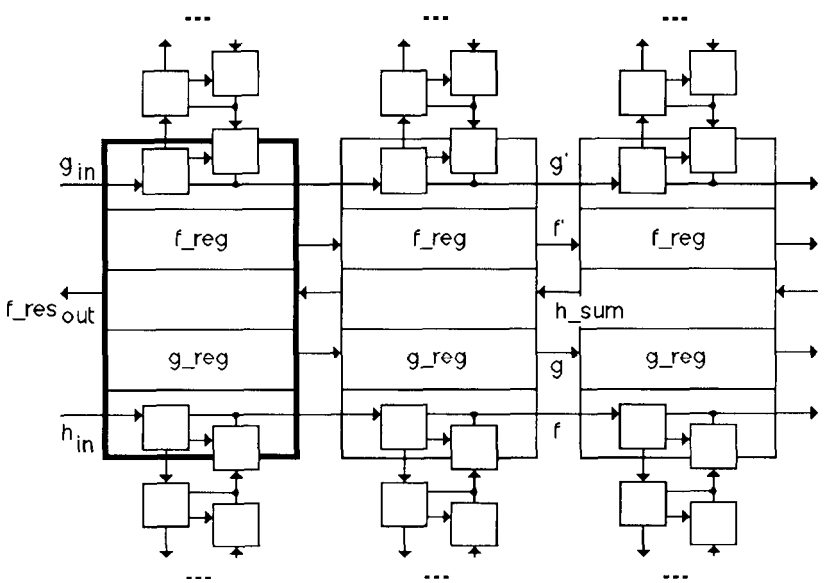
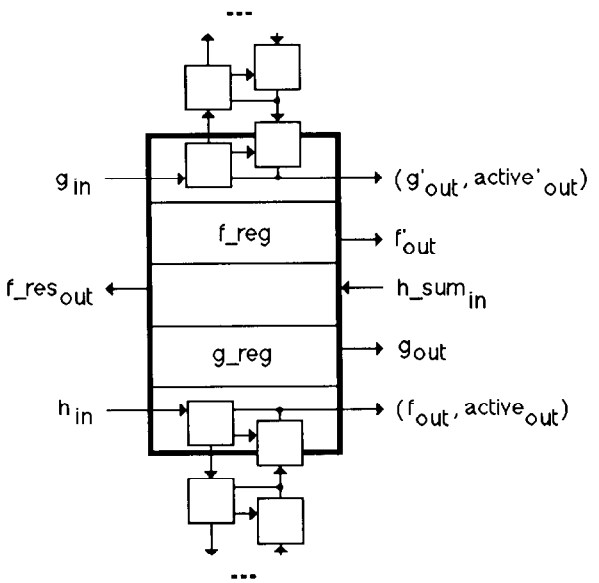


Fig. 21. Systolic array for the inverse arithmetic convolution problem.



if *nonmarked* **then**
  begin
    {$g_{in}=g(1)$; $h_{in}=h(1)$; compute $f\_res_{out}=f(1)$} $f\_res_{out}:=h_{in}/g_{in}$;
    {store and mark cell} $f\_reg:=f\_res_{out}$; $g\_reg:=g_{in}$; $nonmarked:=false$;
    {delete f_{out} and g'_{out}} $f_{out}:=nil$; $g'_{out}:=nil$;
  end
else
  begin
    {compute f_res_{out}} $f\_res_{out}:=(h_{in}-h\_sum_{in}-f\_reg*g_{in})/g\_reg$;
    {transmit g_{in} and f'_{in}} $g_{out}:=g_{in}$; $f'_{out}:=f\_res_{out}$;
    {activate f-input and g'-input} $active_{in}:=true$; $active'_{in}:=true$;
    {delay all f- and g'-inputs}
    $(f_{out}, active_{out}):=Delay\_Mechanism(f\_res_{out}, active_{in})$;
    $(g'_{out}, active'_{out}):=Delay\_Mechanism(g'_{in}, active'_{in})$;
  end

Fig. 22. Program of first cell for the inverse convolution problem.

The performances are the same as for the direct arithmetic convolution. We use $N^{1/2}$ processing cells and $O(N \log N)$ delays for the computation of the sequence $(f(n); 1 \leqslant n \leqslant N)$ with N units of time.

4. Conclusion

We have presented two linear systolic arrays for the real-time solution of the arithmetic convolution and of the inverse arithmetic convolution problem. Both

arrays extend Verhoeff's design for the Möbius function to solve the general arithmetic convolution problem. Our first design is a linear array of $O(N)$ cells that solves the problem in time $O(N)$, thereby delivering the same performances as Verhoeff's design. Our second design requires only $O(N^{1/2})$ computational cells. We believe it would be an interesting challenge to derive this second design completely automatically, using the synthesis methods of [1, 3, 10, 11] or the parallel constructs of [15, 16].

Acknowledgments

The authors thank Tom Verhoeff for bringing the arithmetic convolution problem to their attention during a workshop organized by Alain Martin in La Jolla, California, USA, on February 22–26, 1988. Tom Verhoeff presented its design for the Möbius function [16] and raised the problem of systolizing the general arithmetic convolution. We designed the first solution (reported in this paper see also [12]) two months after the workshop. At the time of this writing, we are aware of three other solutions, by Chen and Choo [2], by Duprat [4] and by Struik [14]. These three solutions are variations on the first design presented in this paper, in that they are linear arrays of $O(N)$ cells that solve the arithmetic computation problem in time $O(N)$.

References

[1] M. Chen, Synthesizing VLSI architectures: dynamic programming solver, in: K. Hwang et al., eds., *Proc. 1986 Internat. Conf. on Parallel Processing* (IEEE Computer Soc. Press, Silver Spring, MD, 1986) 776–784.

[2] M. Chen and Y. Choo, Synthesis of a systolic Dirichlet product using nonlinear contraction domain, in: M. Cosnard et al., eds., *Parallel and Distributed Algorithms* (North-Holland, Amsterdam, 1989) 281–295.

[3] J.M. Delosme and I.C.F. Ipsen, Systolic array synthesis: computability and time cones, in: M. Cosnard et al., eds., *Parallel Algorithms and Architectures* (North-Holland, Amsterdam, 1986) 295–312.

[4] J. Duprat, Private communication.

[5] H.L. Keng, Introduction to Number Theory (Springer, Berlin, 1982).

[6] H.T. Kung, Use of VLSI in algebraic computations: some suggestions, in: *Proc. 1981 ACM. Symp. on Symbolic and Algebraic Computation* (ACM, New York, 1981) 218–222.

[7] H.T. Kung, Why systolic architectures, *IEEE Trans. Comput.* **15**(1) (1982) 37–46.

[8] H.T. Kung and C.E. Leiserson, Systolic arrays for (VLSI), in: I.S. Duff et al., eds., *Proc. Symp. on Sparse Matrices Computations*, Knoxville, Tennessee (1978) 256–282.

[9] C.E. Leiserson and J.B. Saxe, Optimizing synchronous systems, in: *Proc. 22th Annual Symposium on Foundations of Computer Science* (IEEE Press, New York, 1981) 23–36.

[10] D.I. Moldovan, On the design of algorithms for VLSI systolic arrays, *Proceedings of the IEEE* **71**(1) (1983) 113–120.

[11] P. Quinton, The systematic design of systolic arrays, in: F. Fogelman et al. eds., *Automata networks in computer science: theory and applications* (Manchester University Press, 1987) 229–260.

[12] P. Quinton and Y. Robert, Systolic convolution of arithmetic functions, Research Report 449, IRISA Rennes, 1989.

[13] Y. Robert and M. Tchuente, Réseaux systoliques pour des problèmes de mots, *RAIRO Inform. Théor. Appl.* **19**(2) (1985) 107–123.

[14] P. Struik, A systematic design of a parallel program for Dirichlet convolution, Computing Science Note 89/07, Eindhoven University of Technology, 1989.

[15] J.L.A. van de Snepscheut and J.B. Swenker, On the design of some systolic algorithms, Computing Science Note 87/05, University of Groningen, The Netherlands, 1987.

[16] T. Verhoeff, A parallel program that generates the Möbius sequence, Computing Science Note 88/01, Eindhoven University of Technology, 1988.