Generic Systolic Arrays : A Methodology for Systolic Design

Pascal Gribomont
Université de Liège
Institut Montefiore B28
4000 Liège Sart-Tilman (Belgium)

Vincent Van Dongen
Centre de Recherche Informatique
1801 McGill College, Bureau 800
Montréal Canada H3A 2N4

Abstract. Several recent papers demonstrate the interest of viewing systolic algorithms as *while*-programs whose statements are synchronous multiple assignments. This approach is based on the classical invariant method and compares favourably with earlier ones, based on recurrence systems and space-time transformations. Our purpose is to use the particularities of the systolic paradigm to reduce the creativity needed to develop a systolic algorithm and its invariant. More precisely, two points are taken into account. First, the architecture is often chosen before the real beginning of the development and, second, the basic operations to be executed by individual cells are also partially known at the beginning. In fact, the development does not start from scratch, but from a "generic systolic array" (gsa), whose parameters have to be instantiated. Most systolic arrays are instances of a simple gsa that is introduced, investigated and illustrated in this paper.

1 Introduction to systolic array design

1.1 Systolic array

A *systolic array* is a regular network of similar processing units. These units, or *cells*, are connected by *channels*. Usually, each cell is connected with immediate neighbours only; this makes VLSI implementation more efficient [12].

The program executed by every cell is a loop, whose body is a finite, partially ordered set of statements. The statements specify three kinds of actions: receiving values (data) from some input channels, performing computations within the internal memory, transmitting values (results) to output channels.

The processing units act with high synchronism. This synchronism is often provided by a global, broadcasted clock, but this can lead to implementation problems. Another solution is the synchronization by communication, named *rendezvous*: a value can be transmitted from a cell to another one only when both cells are prepared to do so; there is no buffering mechanism.

Let us outline briefly the most classical application of systolic programming: the dot product of a matrix A by a vector x. The result y is described by the usual formula:

$$y_k = \sum_{j=1}^{p} A_{kj} x_j \,. \tag{1}$$

This is computed by the simple Pascal-like program

$$\text{for } k := 1 \text{ to } p \text{ do } [\ y_k := 0\,;\ \text{for } j := 1 \text{ to } p \text{ do } y_k := y_k + A_{kj}x_j\ ].$$

The complexity of this algorithm is $O(p^2)$ since the assignment $y_k := y_k + A_{kj}x_j$ is executed p^2 times. However, as many of these operations may be executed concurrently, a systolic algorithm could be of linear complexity in time and space. Each cell will repeatedly execute the assignment with data received from the neighbouring cells and the outside, and transmit data and results to neighbouring cells or the outside. An adequate systolic algorithm for this application is presented in Section 3.2.

1.2 Space-Time transformation methodology

Many approaches have been developed for systolic array design. Some of them are presented in [6]; more recent ones are [15, 8, 2, 9, 14, 16, 17, 13, 18, 19]. We will outline a commonly used method. Classically, systolic design is concerned with the parallelization of algorithms. The algorithm to be mapped is specified as a set of equations attached to integral points, and mapped on the architecture using a regular time and space allocation scheme. This approach became the basis of many studies on the synthesis of systolic arrays. The main problems that were tackled were the scheduling of the computations, the mapping of the computations on regular architectures, the partitioning schemes for fixed size arrays, and the organization of multistep algorithms.

Let us summarize the basics of the *space-time transformation* methodology, illustrated with the matrix-vector multiplication algorithm. The methodology consists of four main steps: the *index localization*, the *uniformization*, the *space-time transformation*, and the *interface design*.

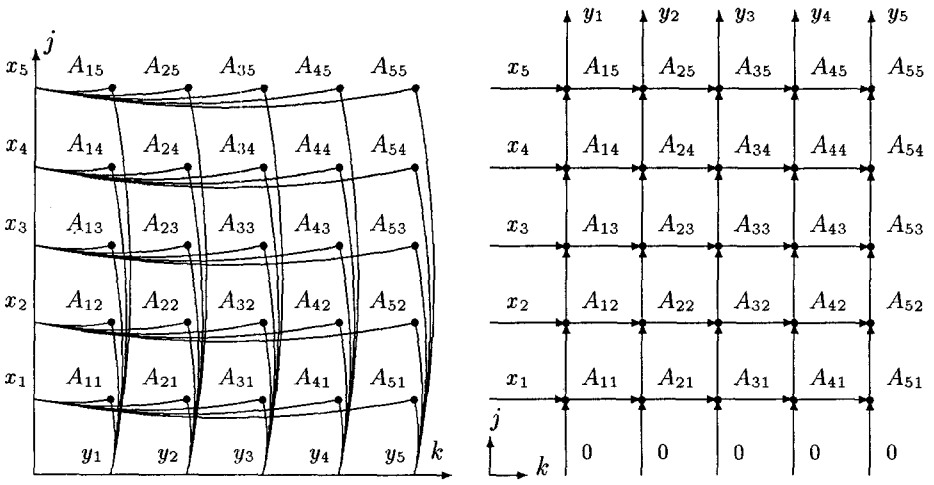

Fig. 1. The index localization, the basis of most mapping methodologies (left) A uniform dependence graph for the matrix-vector multiplication algorithm (right)

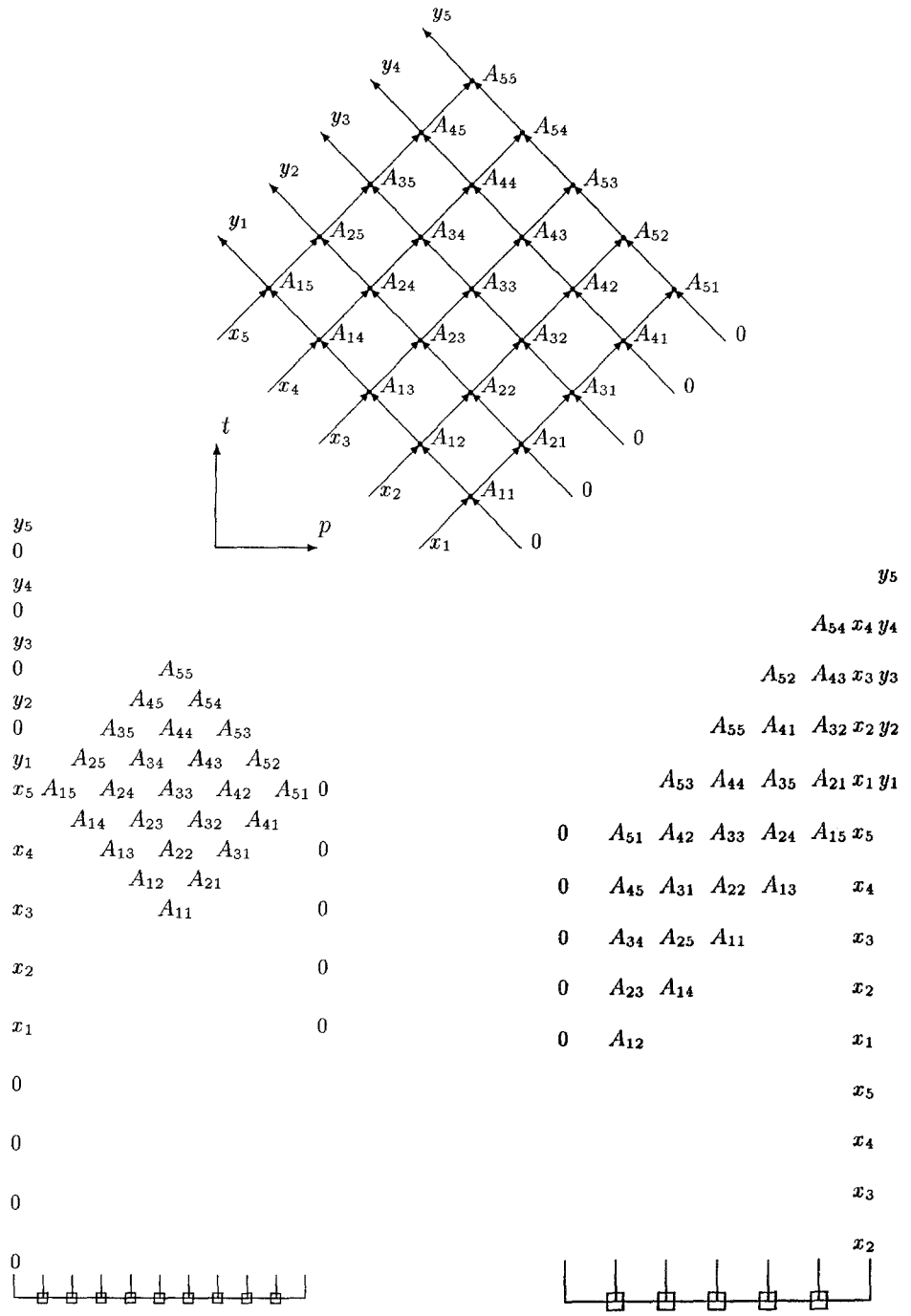

Fig. 2. Space-time diagram : when and where the computations are performed (top)
A classical array for the matrix-vector multiplication (bottom left)
A new array for the matrix-vector multiplication (bottom right)

1. *Index localization.* For the matrix-vector multiplication algorithm, the computations to be performed are defined by equation (1), which uses indices k and j. In the index localization, A_{kj} is attached to the point (k, j), x_j is attached to the point $(\_, j)$, and y_k is attached to the point $(k, \_)$. Figure 1 (left) shows graphically the result of this first step; the arcs of the graph represent the relations between the data.

2. *Uniformization.* Figure 1 (left) gives an indication on where the data need to be, and where the results are being produced. When a piece of data is needed at many places, the fan-out degree of the associated node is large. In the same way, when a result uses many data, the fan-in degree is large. The uniformization consists on reducing the fan-in and the fan-out degrees of these nodes, using the well known pipelining technique [4, 16]. In other words, step two transforms the initial algorithm into a set of *uniform recurrence equations.* Figure 1 (right) represents graphically the result of step two. At each node, the same set of computations needs to be performed:

$$y_{out} = y_{in} + A_{in} \times x_{in}\,, \quad x_{out} = x_{in}\,. \tag{2}$$

3. *Space-time transformation.* In this third step, a time and a processor allocation functions are being chosen. These two functions define respectively when and where the computations will be performed. Affine functions are well suited for the mapping of uniform recurrences on regular arrays [10]. Figure 2 (top) shows graphically the result of step three; the space-time diagram indicates when and where the computations are performed: the computation associated to the node (p, t) is performed at processor p at the time t.

4. *Interface design.* In this final step, the loading of the data and the unloading of the results are considered. There are two main techniques for doing so: either control signals are added with I/O lines, or the algorithm is slightly modified with the use of dummy variables. In the matrix-vector multiplication, we chose to propagate additional zeroes to avoid the use of control signals. The resulting classical circuit is shown in Figure 2 (bottom left), along with the data introduced in the circuit upon the time. The x_k's and the results are respectively introduced and produced at the leftmost cell. Zeroes are also introduced to the left to avoid the use of control signals for the initializations.

Many algorithms already were parallelized using this efficient technique. However, this methodology suffers from some drawbacks.

First, the algorithm must be specified as a set of recurrence equations, or nested do-loop instructions. This is not always easy to do. In particular, we will present an architecture that computes the greatest common divisor, but to which no recurrence equation is directly associated. Yet, the proof that the architecture performs the right result is given, using an invariant technique.

Second, a location in space is associated to each index value. This constraint is well suited for the synthesis of regular arrays: the data will be introduced in a regular order. However, this is not necessary and it eliminates the possibility of synthesizing other architectures. See for example the circuit described in Figure 2 (bottom right). The method used for its derivation is described hereafter;

no a priori assumption on the localization of the data was made. On the other hand, the above space-time approach could not derive such a circuit without using a very complex space-time transformation function.
The third drawback of the space-time transformation approach is the synthesis of the initializations. Here, additional zeroes were introduced to avoid the use of control signals. Thus, the initial algorithm (1) is now slightly modified; one should still use a proof technique to be sure that the results are not modified. In the technique presented hereafter, initializations are easily taken into account.

1.3 A program-oriented methodology

Various attempts have been made to overcome the drawbacks of the space-time transformation method. Those based (explicitly or not) on viewing systolic design as program design seem especially promising, for the reasons listed below.

- Most work has been done about formal methods for developing programs; these methods also apply to the development of systolic algorithms and, in particular, to the construction of an adequate while-loop.
- The notion of invariant, introduced for programs, happens to keep a prominent role in the design of systolic algorithm. (This is clearly demonstrated in [18]; an additional example is given in Section 4.2.)
- It is now widely accepted that operational notions should not appear in programming at the early design stages. For instance, control is left implicit in structured sequential programming (see e.g. [5]) and also in concurrent programming [3]. As space-time allocation is an operational notion, it is perhaps not adequate to base a method on this notion. In fact, space-time allocation should be deduced at the last stage of the design. Early introduction of operational notions often leads to exclude possible solutions of a problem.
- Time and space boundary conditions often are a problem in the design of systolic arrays; this problem is best handled with an invariant-based method.

To summarize, an adequate method of systolic design can be as follows. First, the specifications of the system are formalized with an input and an output predicates, just as in structured sequential programming. Second, a couple (program, invariant) is deduced in an incremental way from the specification. Third, the sequential while-program is further transformed into a systolic program; the statements of this program are concurrent assignments. Last, the systolic array is (easily) obtained from the systolic program. This approach (of some aspects of it) is presented and illustrated in several papers, including [2, 14, 7, 13, 18].

Such a method clearly inherits the usual problem in structured sequential programming: the development of a program together with its invariant is not easy, and some kind of creativity is often needed. Besides, further creativity is needed to transform a sequential program and its invariant into a systolic program and its invariant.

1.4 Generic systolic arrays

We propose to reduce the need of creativity by taking into account two facts about systolic arrays. First, most of systolic arrays presented in the literature are based on a common, very simple architecture. It is helpful to consider a systolic array as an instance of this architecture; systolic design is then reduced to parameter choosing. Second, the specifications of the problem frequently suggest all or part of the elementary operations to be performed by each cell; when this is the case, the remaining problem is to determine how and when data are to be pushed in the array, and when the results are to be collected out.

In this framework, the starting point will be neither a sequential algorithm, nor the description of a single process, but a *gsa* (generic systolic array), that is, some kind of partially specified systolic array (or piece of systolic array). This object is equally distant from the source problem and the target systolic implementation.

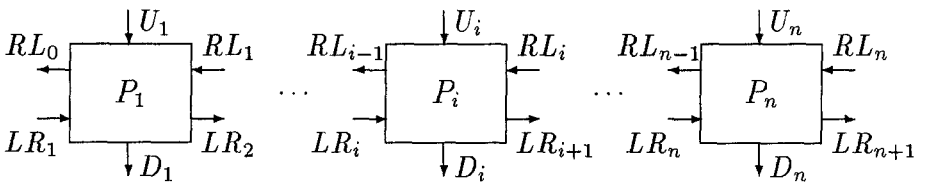

Fig. 3. The linear generic systolic array

We concentrate on one of the most common architectures. Processing units are connected in a linear array (Figure 3). Each cell is connected with its immediate neighbours; the extreme cells can exchange data and results with the outside. Furthermore, each cell can receive data from the top and transmit results to the bottom. For instance, cell P_i admits three input channels; P_i can receive data from P_{i-1} through channel LR_i (Left to Right), from P_{i+1} through RL_i, and from the outside through U_i (Up). Similarly, P_i has also three output channels, which allow transmission of results to the left and right neighbours and to the outside. In particular applications, some of these channels may be unused and suppressed. It is also possible to obtain 2-dimensional arrays by stacking several linear arrays and adequately connecting channels together. Ring, cylinder and torus arrays can be obtained in a similar way.

The *topological structure* of our gsa is fixed; it is still necessary to fix a *communication scheme*, that is, an order between the internal computation, the reception of data and the transmission of results. A simple scheme is as follows:

$$communication \ ; \ computation \tag{3}$$

A generic systolic array is a systolic array whose computation part is left unspecified. The (generic) systolic arrays described here appear as particular CSP networks (see e.g. [15]). Recall that, if C is a transmission channel, the concurrent execution of the input statement $C?x$ and of the output statement $C!e$ implements the distributed assignment $x := e$.

This paper goes on as follows. In the next section, a formal model for the gsa is introduced; its properties are stated and proved. Two applications are presented in Sections 3 and 4, with emphasis on the critical design steps.

2 Properties of the linear generic systolic array

2.1 Description of the linear generic systolic array

The *linear* gsa is determined by the following points.

- The topology of the network is depicted on Figure 3.
- The scheme of communication is scheme (3).
- The internal memory of cell P_i contains six communication registers, denoted $A[i]$, $B[i]$, $C[i]$, $D[i]$, $E[i]$ and $F[i]$. The remaining part of the memory is denoted $M[i]$; its size is independent from the size n of the network.
- The communication phase for cell P_i consists in the concurrent execution of the following statements (Figure 4):

$$LR_i?A[i]\,,\ RL_{i-1}!B[i]\,,\ U_i?C[i]\,,\ D_i!E[i]\,,\ LR_{i+1}!F[i]\,,\ RL_i?G[i]\,.$$

- The internal computation phase is not restricted; it can be modelled by a function Φ. More specifically, the computation phase consists in executing the assignment $(F, E, B, M) := \Phi(A, C, G, M)$.

Comments. An *instance* of the linear gsa is obtained by replacing Φ by a total function (therefore, systolic algorithms are deterministic and never fail).
During the communication phase, only the *input registers* A, C and G are changed; during the computation phase, only the *storage register* M and the *output registers* B, D and F are changed. The registers can consist of a single memory cell, but also of any fixed amount of memory.

The linear gsa comprises four *logical* components, which are listed below:

- The *left-to-right* component contains LR_i, $A[i]$ and $F[i]$, for all i.
- The *right-to-left* component contains RL_i, $B[i]$ and $G[i]$, for all i.
- The *up-to-down* component contains U_i, D_i, $C[i]$ and $E[i]$, for all i.
- The *storage* component contains $M[i]$, for all i.

Incomplete instances of the linear gsa are obtained by omitting one or more logical components.

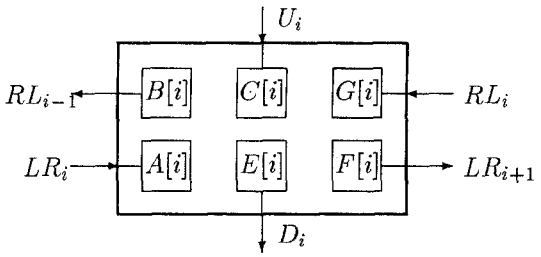

Fig. 4. A typical cell of the linear generic systolic array

Comment. The Warp machine [1] can be viewed as a gsa of size 10.

2.2 External behaviour of the linear gsa

From the outside, the linear gsa is seen as a "big cell". The input channels are LR_0, $U_0, \ldots, U_n$ and RL_n; the output channels are RL_{-1}, $D_0, \ldots, D_n$ and LR_{n+1}. The structure of the program executed by the array is

$$*(\alpha)[\textit{transmission phase};\ \textit{computation phase}]\,,$$

where "$*(0)P$" and "$*(\alpha{+}1)P$" respectively mean "*skip*" and "P; $*(\alpha)P$". A description of the transmission phase is given below:

$$\begin{array}{ll} LR_1?A[1] \parallel A[2:n] := F[1:n-1] \parallel LR_{n+1}!F[n] & \text{Left-to-right,} \\ \parallel (\|_{k=1}^{n} U_k?C[k]) \parallel (\|_{k=1}^{n} D_k!E[k]) & \text{Up-to-down,} \\ \parallel RL_0!B[1] \parallel G[1:n-1] := B[2:n] \parallel RL_n?G[n] & \text{Right-to-left.} \end{array}$$

The computation phase is modelled by

$$Cpt \;=\; \|_{r=1}^{n} \left\{ \begin{array}{lcll} F[r] & := & S(A[r], C[r], G[r], M[r]) & \text{Left-to-right,} \\ E[r] & := & T(A[r], C[r], G[r], M[r]) & \text{Up-to-down,} \\ B[r] & := & V(A[r], C[r], G[r], M[r]) & \text{Right-to-left,} \\ M[r] & := & W(A[r], C[r], G[r], M[r]) & \text{Storage.} \end{array} \right.$$

The notation "$A[i:i{+}n] := B[j:j{+}n]$" stands for "$\|_{k=0}^{n} A[i{+}k] := B[j{+}k]$" (*skip* if $n < 0$).

The gsa can work only when connected with an *environment*, providing data and collecting results. Data and results are organized in *streams*; a stream s is a sequence $(s(i) \;:\; i \in \mathbf{N}_0)$ of values. Data streams are called dL, dR and $dU_1, \ldots, dU_n$; result streams are rL, rR and $rD_1, \ldots, rD_n$ ("*d*"stands for "data" and "*r*" stands for "results"). The environment is modelled by a processing unit which executes the program

$$\begin{array}{ll} *(\alpha)\,[\; i := i+1\,; & \text{Update counter,} \\ \quad LR_1!dL(i) \parallel LR_{n+1}?rR(i) & \text{Left-to-right,} \\ \quad \parallel (\|_{k=1}^{n} U_k!dU_k(i) \parallel D_k?rD_k(i)) & \text{Up-to-down,} \\ \quad \parallel RL_0?rL(i) \parallel RL_n!dR(i) & \text{Right-to-left.} \\ \;]\,. & \end{array}$$

The initial value of i is 0.

The parallel composition of the program executed by the network and the program executed by the array is an ordinary sequential program, since all communication statements appear in matching pairs, which reduce to assignments. This program is

$$\begin{array}{l} *(\alpha)\,[\, i := i+1\,; \\ \quad A[1] := dL(i) \parallel A[2:n] := F[1:n-1] \parallel rR(i) := F[n] \\ \quad \parallel C[1:n] := dU_{1:n}(i) \parallel rD_{1:n}(i) := E[1:n] \\ \quad \parallel rL(i) := B[1] \parallel G[1:n-1] := B[2:n] \parallel G[n] := dR(i)\,; \\ \quad Cpt\,]\,. \end{array}$$

It has a very simple structure (single loop), and interesting knowledge can be gained about it, before instantiating the parameters S, T, V and W. This knowledge is conveniently summarized in the form of a recurrence system, which describes the effect of the execution of the loop body. Here is the recurrence system.

$$\begin{array}{l} F_i[r] = S(F_{i-1}[r-1], dU_r(i), B_{i-1}[r+1], M_{i-1}[r]) \\ E_i[r] = T(F_{i-1}[r-1], dU_r(i), B_{i-1}[r+1], M_{i-1}[r]) \\ B_i[r] = V(F_{i-1}[r-1], dU_r(i), B_{i-1}[r+1], M_{i-1}[r]) \\ M_i[r] = W(F_{i-1}[r-1], dU_r(i), B_{i-1}[r+1], M_{i-1}[r]) \end{array} \quad (4)$$

The value of a register X of cell r, after the ith iteration, is denoted $X_i[r]$ (the symbol X stands for F, E, B or M). The ith element of the input stream dU_r is denoted $dU_r(i)$. The equations written above hold for all r in $\{1, \ldots, n\}$ and for all $i > 0$, with the following additional conventions:

$$F_{i-1}[0] = dL(i)\ ;\ B_{i-1}[n+1] = dR(i) \quad (5)$$

This recurrence system relates successive values of the output and storage registers; the output streams of the gsa are given by the following identities:

$$rL(i) = B_{i-1}[1]\,,\ \forall r : (\,rD_r(i) = E_{i-1}[r]\,)\,,\ rR(i) = F_{i-1}[n]\,. \quad (6)$$

2.3 Invariant of a recurrence system

An *invariant* of the recurrence system $X_{i+1} = f(X_i)$ is a predicate P such that $\forall Y\,[P(Y) \Rightarrow P(f(Y))]$, that is, an invariant of the associated program

$$i := 0;\ \text{while } i < \alpha \text{ do } (X_{i+1}, i) := (f(X_i), i+1)\,.$$

The knowledge of an adequate invariant of a recurrence system can be useful, especially when the system cannot be easily solved. The interesting fact about invariants is that they summarize substantial information under a concise form. An example will be given in paragraph 5.2.

3 Design of algorithms for linear systolic arrays

A *linear systolic array* is obtained by replacing the parameters S, T, V and W by actual functions in the linear gsa. These functions describe a specific algorithm for the array.

3.1 Outline of the method

The design problem consists in adapting the gsa to a specific task. This adaptation can be performed in several steps, enumerated below.

1. The linear generic systolic array is instantiated in a linear systolic array.

2. The behaviour of the array and its environment is formally stated as a set of recurrence equations.
3. The recurrence system is solved, or an adequate invariant is found.
4. The data and results streams of the array are interpreted as data and results of the problem.

Let us comment a little about these four steps. In the first step, the designer decides what will be the computation part *Cpt*. The linear gsa allows four logical components, described respectively by functions S, T, V and W but, for some applications, one or two logical components will be enough.

The second step is mechanical: the recurrence system corresponding to the array (and its environment) is obtained by mere instantiation of the generic recurrence system (4) introduced in paragraph 3.2.

The third step can be difficult. From the practical point of view, let us observe that the discovery of an adequate invariant can be easier than the discovery of the solution of the recurrence system. This point will be illustrated later (Section 4.2; see also [7]).

The fourth step is the interpretation step. It allows to determine where and when the data are transmitted into the array, and where and when the results are collected out. This determination is simpler than in the space-time method; it requires more carefulness than creativity, since rather little choice is left.

A classical application is presented in this section, in order to demonstrate the design method.

3.2 Matrix-vector product

As recalled in paragraph 1.1, the product y of a matrix A and a vector x is obtained by executing the assignments $y_k := y_k + A_{kj}x_j$ for all k, j in $\{1, \ldots, p\}$. The initial value of every y_k is 0. Assignments on y_k and $y_{k'}$ may be performed concurrently if and only if $k \neq k'$.

Classically, the design problem for this application consists in discovering an adequate time and processor allocation for the p^2 assignments. In this framework, an adequate allocation maps each assignment onto a processing unit, and also specifies when the assignment is performed. An adequate allocation should satisfy the concurrency constraint just mentioned and also an implementation constraint: a processing unit can perform only one assignment at a time.

Many adequate allocations exist. One of them is the purely sequential one: there is only one processing unit, and the assignment $y_k := y_k + A_{kj}x_j$ is performed at time $t(k, j) := p(k-1) + j$. Interesting adequate allocations are *time-optimal* ones: due to the concurrency constraint, the time of computation is at least p, since each y_k is altered by p assignments. An optimum can be reached as follows. There are p processing units, each of them devoted to a single y_k. Each unit sequentially executes the assignment, for $j = 1, \ldots, p$.

This time-optimal allocation is not fully satisfactory. There are communication problems. Each x_j must be simultaneously broadcasted to all processing

values. As broadcasting is generally not accepted in systolic arrays, the simple time-optimal allocation will be rejected.

The critical point of the design problem for this application is here: a trade-off between time-optimality and communication-optimality must be discovered.

Let us now come back to the linear gsa. The problem of finding such a trade-off disappears, since the communication scheme is already fixed. As a matter of fact, very little choice is left. Each cell will perform the assignment $a := a + b * c$, for some a, b and c. We have only to assign the three flows available in the gsa to these three values. We are interested in linear time algorithms or, at least, in algorithms of complexity better than quadratic, so the only acceptable b-flow (matrix coefficients) is the UD-flow. Indeed, the remaining flows LR and RL allow only one input at a time. They are assigned arbitrarily; for instance, LR is assigned to a (coefficients of y) and RL is assigned to c (coefficients of x). The storage logical unit is not used. This leads to the following parameters:

$$\begin{array}{ll} S(a,b,c) = a + b * c\,, & LR\,, \\ T(a,b,c) = b\,, & UD\,, \\ V(a,b,c) = c\,, & RL\,. \end{array}$$

The corresponding recurrence system is:

$$\begin{array}{l} \forall r:\ F_i[r] = F_{i-1}[r-1] + dU_r(i) * B_{i-1}[r+1])\,, \\ \forall r:\ E_i[r] = dU_r(i)\,, \\ \forall r:\ B_i[r] = B_{i-1}[r+1]\,. \end{array}$$

We are interested in the result stream rR, which is obtained easily as follows

$$\begin{array}{l} rR(i) = F_{i-1}[n] \\ \quad = F_{i-2}[n-1] + dU_n(i) * B_{i-2}[n+1] \\ \quad = F_{i-3}[n-2] + dU_{n-1}(i-1) * B_{i-3}[n] + dU_n(i) * B_{i-2}[n+1] \\ \quad = \cdots \qquad\qquad \cdots \\ \quad = F_{i-n-1}[0] + \sum_{l=1}^{n} dU_{n-l+1}(i-l) * B_{i-l-1}[n-l+2] \\ \quad = F_{i-n-1}[0] + \sum_{l=1}^{n} dU_{n-l+1}(i-l) * B_{i-2l}[n+1] \\ \quad = dL(i-n) + \sum_{l=1}^{n} dU_{n-l+1}(i-l) * dR(i-2l+1)\,. \end{array}$$

Comment. The validity of the development is restricted by the range of the indices. The conditions are $i-n > 0$, $0 < n-l+1 \leq n$, $i-l > 0$ and $i-2l+1 > 0$, for all l in $\{1,\ldots,n\}$. These conditions reduce to $i \geq 2n$. (For $0 < i < 2n$, the value $rR(i)$ depends on the initial values of the cells; as no condition is required about the initial values, this part of the result stream cannot be used.)

Let us emphasize the fact that, for the time being, the allocation implemented by the array is not known yet. Even the relation between the size n of the systolic array and the dimension p of the matrix and of the vector, must still be fixed. This is done in the fourth and last step of the design procedure.

The value $y_k = \sum_{r=1}^{n} A_{k,r} x_r$ must be extracted from the stream rR, for all k in $\{1,\ldots,n\}$. As the first useful value of the result stream is $rR(2n)$, the

useful values could be $rR(2n)$, $rR(2n+1)$, ..., $rR(3n-1)$. More specifically, the identity

$$rR(2n-1+k) = \sum_{j=1}^{p} A_{kj} x_j$$

is matched, for $k \in \{1, \ldots, n\}$, with the identity

$$rR(2n-1+k) = dL(n-1+k) + \sum_{l=1}^{n} dU_{n-l+1}(2n-1+k-l) * dR(2n+k-2l) .$$

Several matchings are possible, and discovered easily. Obviously, n and p are equal and the dL flow must be 0. The dR contains the components of x. A simple solution consists in deliver these components in the natural order; let us choose $dL(j) = x_{j \bmod n+1}$ ("+1" is introduced because the range of $j \bmod n$ is $\{0, \ldots, n-1\}$, while the components of x are indexed in $\{1, \ldots, n\}$). This choice leads to:

$$rR(2n-1+k) = \sum_{l=1}^{n} dU_{n-l+1}(2n+k-l-1) * x_{(2n+k-2l) \bmod n+1} .$$

If n is odd, each component of x occurs exactly once in this sum. The last task consists in specifying the dU streams. The data $dU_{n-l+1}(2n+k-l-1)$ should be the matrix component $A_{k,(2n+k-2l) \bmod n+1}$. This allows the determination of $dU_i(j)$. The results are summarized below; a graphical representation is on Figure 2 (bottom right).

$$\begin{array}{lll} dL(j) = 0 & & j \in \{n, \ldots, 2n-1\} ; \\ dU_i(j) = A_{j-i-n+2,\,(j+i-n) \bmod n+1} & & i \in \{1, \ldots, n\} , \\ & & j \in \{n+i-1, \ldots, 2n+i-2\} ; \\ dR(j) = x_{j \bmod n+1} & & j \in \{1, \ldots, 3n-2\} . \end{array} \qquad (7)$$

Comments. Similar results can be obtained for even n.

Let us emphasize that the data are delivered to the systolic array in a rather strange way; moreover, the components of the vector x must be delivered twice or three times.

The components of x are output through rL_0, without modification, but with a delay of n time units.

The components of A are output through rD, without modification, but with a delay of one time unit; more specifically, $rD_r(j+1) = dU_r(j)$.

The execution is completed after $3n-1$ step.

The initial contents of the registers are arbitrary; so are the members of the input streams which do not occur in formulas (7).

4 More general systolic arrays

4.1 Formal description of generic arrays

Most systolic arrays can be obtained as combinations of linear arrays, whose communication channels are properly connected. Some important combinations are introduced now.

A *ring* consists in a single linear array whose left and right communications channels have been connected; this means that RL_0 and RL_n, on the one hand, and LR_1 and LR_{n+1} on the other hand, have been identified. A *rectangle* is obtained by stacking p identical linear arrays. Let these arrays be identified by a superscript. Channel D_i^{j+1} and channel U_i^j are identified, for $1 \leq i \leq n$ and $1 \leq j < p$. Further connections in a rectangle can lead to other interesting topologies. The *horizontal cylinder* is obtained by identifying U_i^p and D_i^1. The *vertical cylinder* is obtained transforming each linear array of a rectangle into a ring, and a *torus* is obtained by performing both up-down and left-right connections. A *square* is a rectangle where $n = p$.

The equations (4, 5, 6) are a formal description of the generic linear systolic array. A similar description can be obtained for any architecture. This task is very simple for an architecture derived from the linear one. A single case, the ring, will be considered here.

The formal description of the ring is obtained from the description of the linear array in a straightforward way. First, the identities

$$rR(i) = dL(i) \qquad \text{and} \qquad rL(i) = dR(i)$$

are introduced in the equations (4, 5, 6); second, the streams dL, dR, rL and rR are eliminated. The resulting equations are

$$\begin{aligned}
F_i[r] &= S(F_{i-1}[r-1], dU_r(i), B_{i-1}[r+1], M_{i-1}[r]) \\
E_i[r] &= T(F_{i-1}[r-1], dU_r(i), B_{i-1}[r+1], M_{i-1}[r]) \\
B_i[r] &= V(F_{i-1}[r-1], dU_r(i), B_{i-1}[r+1], M_{i-1}[r]) \\
M_i[r] &= W(F_{i-1}[r-1], dU_r(i), B_{i-1}[r+1], M_{i-1}[r])
\end{aligned} \tag{8}$$

$$F_{i-1}[0] = F_{i-1}[n] \; ; \; B_{i-1}[n+1] = B_{i-1}[1]$$

$$\forall r : (\, rD_r(i) = E_{i-1}[r] \,)$$

4.2 Greatest common divisor

Let us consider a set $E = \{x_1, \ldots, x_n\}$ of positive integers. The greatest common divisor (gcd) of these numbers can be found by execution of the well known Euclidean algorithm:

repeat until $x_i = x_j$ for all i, j:
select i, j such that $x_i > x_j$; replace x_i by $x_i - x_j$.

This algorithm always terminates and the common final value of the x_i is the requested gcd.

Let us try to implement this algorithm on a ring. The only data are the numbers; let us suppose that, initially, cell P_i contains x_i, for all i. Every cell will communicate the value it contains to, say, its right neighbour. (The right neighbour of P_n is P_1.) The computation part of cell P_i consists in comparing its value and the value received from P_{i-1} and in subtracting the smallest value

from the greatest one; sooner or later, all the values in the array will become equal to the required gcd.

This very informal idea should be formalized according to the methodology presented in Section 4. The equations for the ring can be simplified, since only two logical components are needed: the LR component, to implement the circulation of data, and the storage component, since each cell should contain a value. The resulting equations are

$$\begin{aligned} F_i[r] &= S(F_{i-1}[r-1], M_{i-1}[r]) \\ M_i[r] &= W(F_{i-1}[r-1], M_{i-1}[r]) \\ F_{i-1}[0] &= F_{i-1}[n] \end{aligned} \tag{9}$$

The functions S and W are defined as follows:

$$\begin{aligned} &S(x,y) := \text{if } x \neq y \text{ then } \max(x,y) - \min(x,y) \text{ else } x\,, \\ &W(x,y) := \min(x,y)\,. \end{aligned}$$

Let us note that the following properties hold:

$$\begin{aligned} &\forall x, y \in \mathbf{N}:\ gcd(S(x,y), W(x,y)) = gcd(x,y)\,, \\ &\forall x, y \in \mathbf{N}:\ [x \neq y \ \Rightarrow\ S(x,y) + W(x,y) < x + y]\,. \end{aligned}$$

The recurrence system reduces to:

$$\begin{aligned} F_i[r] &= \text{if } F_{i-1}[r-1] \neq M_{i-1}[r] \\ &\qquad \text{then } \max(F_{i-1}[r-1], M_{i-1}[r]) - \min(F_{i-1}[r-1], M_{i-1}[r]) \text{ else } F_{i-1}[r-1]\,, \\ M_i[r] &= \min(F_{i-1}[r-1], M_{i-1}[r]) \end{aligned}$$

with the convention $F_{i-1}[0] = F_{i-1}[n]$. The initial conditions are:

$$F_0[r] = M_0[r] = x_r\,,\quad r \in \{1, \ldots, n\}\,,\quad x_r \in \mathbf{N}_0\,.$$

Comment. Here is an example where the explicit solution of the recurrence system is not easily found (§3.1, point 3). Fortunately, such an explicit solution is not necessary, and it is sufficient to discover an adequate invariant instead. Let us introduce a notation:

$$E_i =_{def} \{F_i[1], M_i[1], F_i[2], \ldots, M_i[n-1], F_i[n], M_i[n]\}\,.$$

An interesting property of the (multi)set E_i is $gcd(E_i) = gcd(x_1, x_2, \ldots, x_n)$. The proof is by induction on i. The identity is obvious for $i = 0$. Let us suppose it is true for $i = k-1$. The identity $gcd(F_i[r], M_i[r]) = gcd(F_{i-1}[r-1], M_{i-1}[r])$ holds for all r: for all positive integers x and y, $gcd(x,y) = gcd(S(x,y), W(x,y))$. As a consequence,

$$gcd(E_i) = gcd(\bigcup_r \{F_i[r], M_i[r]\}) = gcd(\bigcup_r \{F_{i-1}[r-1], M_{i-1}[r]\}) = gcd(E_{i-1})\,.$$

Let us note Z_i the sum of the $2n$ members of the multiset E_i. The sequence $(Z_0, Z_1, \ldots, Z_n, \ldots)$ has three interesting properties. First, it is monotonically decreasing since, for all positive integer x, y, $S(x,y) + W(x,y) \leq x + y$. Second $Z_{i-1} = Z_i$ occurs if and only if $F_{i-1}[r-1] = M_{i-1}[r] = F_i[r] = M_i[r]$, for all r. Third, $Z_{i-1} = Z_i = Z_{i+1}$ occurs only if all the members of E_i are equal: the first identity implies $F_i[r] = M_i[r]$, the second one implies $F_i[r-1] = M_i[r]$, for all r.

As Z_i is always a positive integer, the sequence cannot decrease forever, and a stable state is reached after finitely many iterations.

Comments. The repetition number is still unknown. In practice, an additional circuit can be added to detect when all the registers F and M are identical; in this case, the common value is the gcd.
It is also possible to determine an adequate repetition number. With the simple definition we have adopted for S and W, it would be rather large. An improvement consists in replacing, in the definition of S, the expression "$\max(x,y) - \min(x,y)$" by "$\max(x,y)$ mod* $\min(x,y)$", where "mod*" is the usual modulo operator, except that $(nA \bmod^* A)$ is A instead of 0. In this case, $\alpha = log_\phi(\Sigma)$ is an adequate repetition number, where $\phi = (1+\sqrt{2})/2$ and $\Sigma = max_r x_r + 1$. (The Euclidean algorithm for the gcd is studied e.g. in [11].)

4.3 Other examples

Several examples have suggested that the instantiation of a generic systolic array is significantly easier than the design of a new array by the classical method. We did not encounter any example for which the instantiation method proposed here is more difficult than the space-time mapping method. However, the communication scheme considered in this paper (Equation 3) turned out to be rather restrictive and classical examples often require a slightly less elementary scheme, that is:

$$input \ ; \ computation \ ; \ output \, . \tag{10}$$

With the synchronous communication paradigm, "input" for a cell means "output" for some neighbour, so the global scheme really is

input for odd cells || output for even cells,
computation for odd cells,
input for even cells || output for odd cells,
computation for even cells.

A rather frequent additional refinement is to dissociate cycles related to distinct communication flows; an example is

a-input and b-output for odd cells || a-output and b-input for even cells,
a-computation for odd cells || b-computation for even cells,
a-input and b-output for even cells || a-output and b-input for odd cells,
a-computation for even cells || b-computation for odd cells.

The method we propose is easily adapted to any kind of communication scheme, but the choice of an adequate communication scheme is left to the designer. The solution is not unique; distinct schemes lead to distinct trade-off between the number of cells, the size of the memory and the time of a typical computation.

5 Conclusion

We presented in this paper the very simple but powerful concept of *generic systolic array*. Its properties have been stated and proved once for all, using CSP-like notations. A new methodology for the mapping of algorithms on systolic arrays is based on this concept. Because no a priori assumption is made on the localization of the data, circuits can be derived which are different from those obtained with a space-time transformation technique. Despite the fact that only simple algorithms have been derived here, this methodology is very promising.

References

1. M. Anaratone et al., "Warp architecture and implementation", Proc. 13th Int. Symp. on Computer Architecture, pp. 346-356, 1986
2. K.M. Chandy, J. Misra, "Systolic algorithms as programs", Distributed Computing 1, pp. 177-183 (1986)
3. K.M. Chandy, J. Misra, "Parallel Program Design : A Foundation", Addison-Wesley, Reading, Mass, 1988
4. M.C. Chen, "Synthesizing VLSI architectures: Dynamic programming solver", Proc. of the 1986 Int. Conf. on Parallel Processing, pp. 776-784, 1986
5. E.W. Dijkstra, "A discipline of programming", Prentice-Hall, New-Jersey, 1976
6. J.A.B. Fortes et al., "Systematic approaches to the design of algorithmically specified systolic arrays", IEEE Conf. ICASSP 85, vol. 1, pp. 300-302, 1985
7. P. Gribomont, "Proving systolic arrays", L.N.C.S. 299, pp. 185-199 (1988)
8. M. Hennessy, "Proving Systolic Systems Correct", ACM Toplas 8, pp. 344-387 (1986)
9. C.H. Huang, C. Lengauer, "An implemented method for incremental systolic design", L.N.C.S. 258, pp. 160-177 (1987)
10. R.M. Karp, R.E. Miller, S. Winograd, "The organization of computations for uniform recurrence equations", J.ACM 14, pp. 563-590 (1967)
11. D.E. Knuth, "The art of computer programming", vol. 1, Addison-Wesley, Reading, Mass, 1968
12. H.T. Kung, C.E. Leiserson, "Algorithms for VLSI processor arrays", in "Introduction to VLSI systems", Mead and Conway (Eds), Addison-Wesley, Reading, Mass, pp. 271-292, 1980
13. A.R. Martin, J.V. Tucker, "The concurrent assignment representation of synchronous systems", Parallel Computing 9, pp. 227-256 (1989)
14. C. Mongenet, G.-R. Perrin, "Synthesis of systolic arrays for inductive problems", L.N.C.S. 258, pp. 260-277 (1987)
15. M. Ossefort, "Correctness Proofs of Communicating Processes: Three Illustrative Examples from the Literature", ACM Toplas 5, pp. 620-640 (1983)
16. S.V. Rajopadhye, R.M. Fujimoto, "Systolic array synthesis by static analysis of program dependencies", L.N.C.S. 258, pp. 295-310 (1987)
17. M. Rem, "Trace theory and systolic computations", L.N.C.S. 258, pp. 14-33 (1987)
18. J.L.A. van de Snepscheut, J.B. Swenker, "On the Design of Some Systolic Algorithms", J.ACM 36, pp. 826-840 (1989)
19. V. Van Dongen, "Mapping uniform recurrences onto small size arrays" L.N.C.S. 505, pp. 191-208 (1991)