Systolic Petri Nets

Alexandre Abellard, Patrick Abellard

5

Systolic Petri Nets

Alexandre Abellard and Patrick Abellard
HandiBio EA4322, IUT, Toulon University
France

1. Introduction

In many research fields and applications requiring real time, such as signal, speech or image processing, problems are often characterized by the amount of data to deal with.
However, it can happen that real time constraints are difficult to satisfy without taking into account the intrinsic parallelism of processings to perform. Thus, diverse architectures appeared (SIMD, MIMD...), leading to different classes of architecture. Literature showed the advantages and drawbacks of each of them. Moreover, expensive costs limited their applications during a long time.
Systolic architectures defined by H.T. Kung are a particular class of parallel architectures. They constitute specialized systems characterized by a repetitive structure of identical elementary processors locally and regularly interconnected. Synchronous data circulate through the architecture, which interact at each encounter. Several important difficulties like the central memory sharing, buses access conflicts... can be therefore avoided. All problems are however not systolizable. These networks are designed for the repetitive identical processing of a huge amount of data, which is the case, for example, in many signal and image processing algorithms.
The conception of these networks has been the subject of many studies, the main ones are developped in the first part of this chapter. The limited number of systolic processors available on the market was problematic for the development of methodologies for a long time. Now, programmable components enable to be free from this problem. To ease their implementation, we developped a methodology based on a formal and universal tool (Petri Nets) that is developped in the second part of the paper.

2. Definitions

2.1 Systolic architecture

This architecture is considered as an intermediate between data flow and pipeline. It has been introduced in (Kung, 1982) and is made of a set of processors or processing cells locally interconnected. Each cell can run a simple or complex operation and links between cells are established so as to minimize the associated paths complexity.

Just like in a pipeline structure, information move in a cascade scheduling form. Communications between outside environment and systolic network are established with peripheral cells that constitute the network I/O ports.

2.2 Systolic networks properties

Systolic networks have interesting properties("Quinton & Robert, 1991") :
- Data flow coming from the environment are intensively used
- Networks association is made easier thanks to structure cascadability
- Elementary cells are not complex
- Data flow are simple and regular
- A set of elementary processings is performed synchronously on different cells

These main properties enable to simplify their implementation on VLSI once an automated and rigorous conception method has been defined.

Several kinds of nets exist, according to different basic cells (Fig. 1) : the linear array, orthogonal array and hexagonal array. All can be unidirectional or bidirectional. In this paper, we will mostly focus on linear arrays since it better suits our application, Fig. 1 therefore does not show all possible propagation directions in arrays.

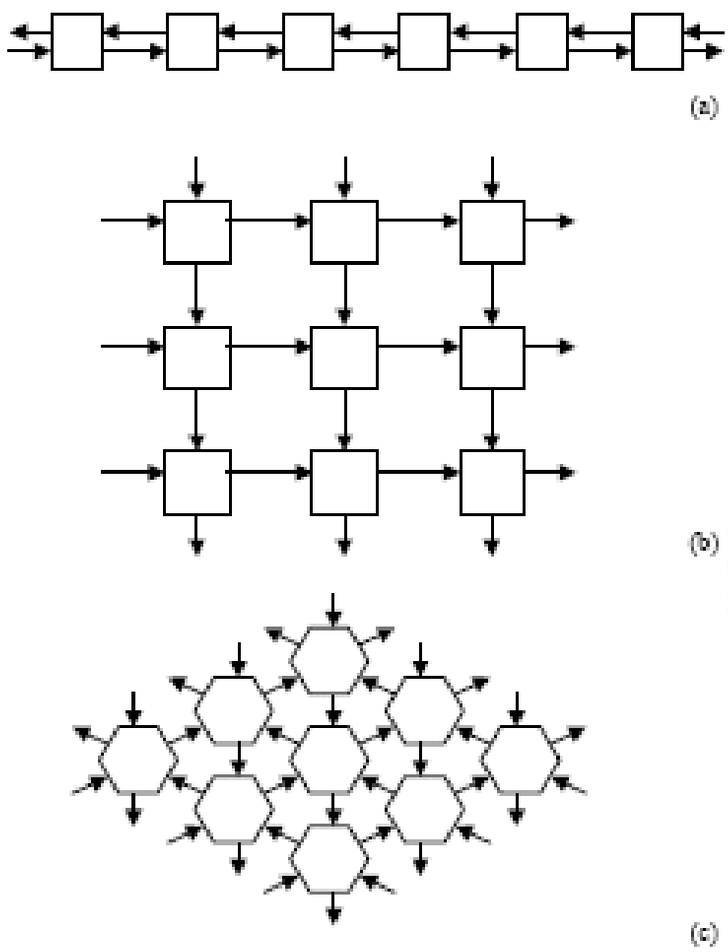


Fig. 1. Basic systolic architectures : (a) linear array, (b) squared array, (c) hexagonal array

These architectures take advantage of the massive parallelism encountered in processing applications (Johnson & Hurson, 1993). They only need a minimum of operators (Sousa, 1998) and memory accesses (Kung, 1988) thanks to a very efficient communications system (Lim & Swartzlander, 1996a). Their combination allows to obtain arrays (Lim & Swartzlander, 1996a) that can be used in a wide range of applications : Discrete Fourier Transform (Lim & Swartzlander, 1999b) (Jackson et al., 2004) (Nash, 2005), convolution (Lee & Song, 2003), filtering (Lee & Song, 2004), matrix operations (Yang et al., 2005), dynamic programming (Lee & Song, 2002).... The regularity of their structures facilitate their hardware, implementation, for instance in FPGAs (Mihu et al., 2001) (Nash, 2002) (Castro-Pareja et al., 2004).

2.3 Principles

2.3.1 Example of a linear network

Linear equation solving is done thanks to the following equation:

$$y_i^{k+1} = a_{i,k+1}.x_{k+1} + y_i^k \text{ , } 0 \le k \le n\text{-}1, 1 \le i \le n \quad (1)$$

Systolic network defined by Kung is established with a group of interconnected processors, each having 3 registers : R_y for y^k, R_a for $a_{i,k}$ and R_x for x. Each register has a connection as input and another one as output. Kung defined 2 kinds of cells : squared (Fig. 2a) and hexagonal (Fig. 2b).

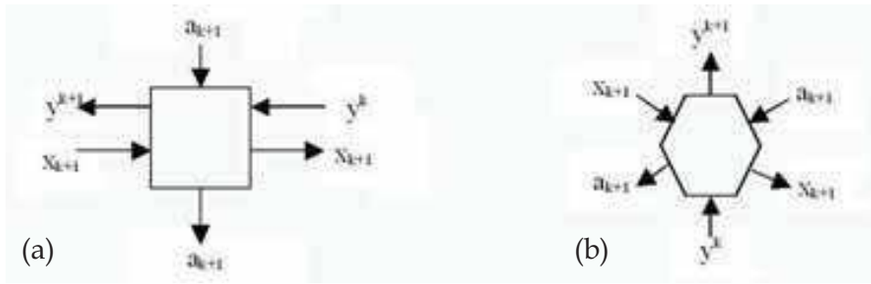


Fig. 2. Square (a) and hexagonal (b) cells

These two kinds of cells work in this operating cycle.
1 – the cell loads inputs y^k, x_{k+1} and $a_{i,k+1}$ in respective registers R_y, R_x and R_a.
2 – y^{k+1} is processed using equation (1)
3 – y_i^{k+1}, x_{k+1} and $a_{i,k+1}$ are transferred to the output
Example of the Matrix-Vector Product (MVP)

$$Y = A \,.\, X \quad (2)$$

Relations to implement are then :

$$y_i^{k+1} = a_{i,\,k+1} \,.\, x_{i,\,k+1} + y_i^k \quad (3)$$
$$y_i^0 = 0 \text{ ; } y_i = y_i^W$$

with W=dim(X), 0≤k ≤n-1, 1≤i≤n

For example, with W = 3 :

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} . \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

Equation (3) then gives :

$$y_1 = y_1^0 + y_1^1 + y_1^2 + y_1^3 \Rightarrow y_1 = a_{11}.x_1 + a_{12}.x_2 + a_{13}.x_3$$
$$y_2 = y_2^0 + y_2^1 + y_2^2 + y_2^3 \Rightarrow y_2 = a_{21}.x_1 + a_{22}.x_2 + a_{23}.x_3$$
$$y_3 = y_3^0 + y_3^1 + y_3^2 + y_3^3 \Rightarrow y_3 = a_{31}.x_1 + a_{32}.x_2 + a_{33}.x_3$$

Given an elementary recurrence relation :

$$y_i^{k+1} = a_{i,\,k+1} \,.\, x_{i,\,k+1} + y_i^k$$

successive recurrence steps are performed at consecutive instants. The use of parallelism done via several cells enable to perform a step of many different elementary recurrences at the same time. Systolic network is therefore made of a set of (2.*n*-1) linearly interconnected squared cells, each one receiving y_i^k, x_{k+1} and $a_{i,k+1}$ at each step of time t_j (Fig. 3).

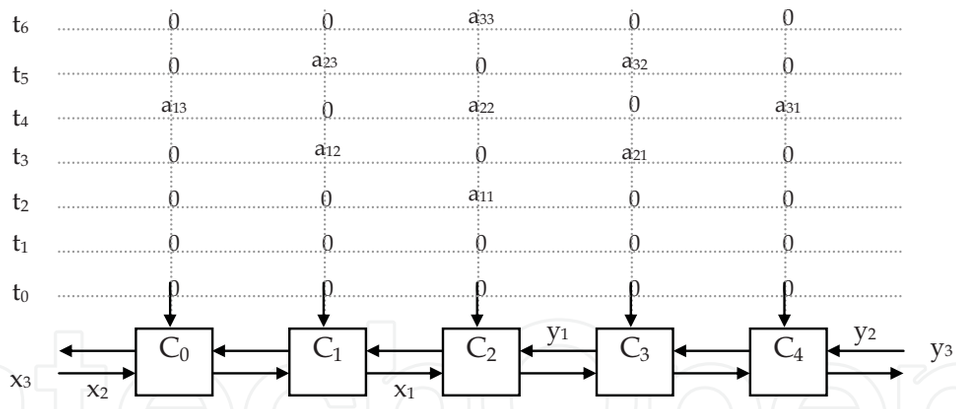


Fig. 3. Linear systolic network of matrix-vector product *Y*=*A*.*X*, *n*=3

x_1 is used by C_0 cell at t_0, then is transmitted to C_1 that processes it at t_1 and so on from left to right. A similar processing is done for y_i data from right to left. Fig. 4 shows the detail of data propagation on cells performing $y_i^{k+1} = a_{i,\,k+1} \,.\, x_{i,\,k+1} + y_i^k$.

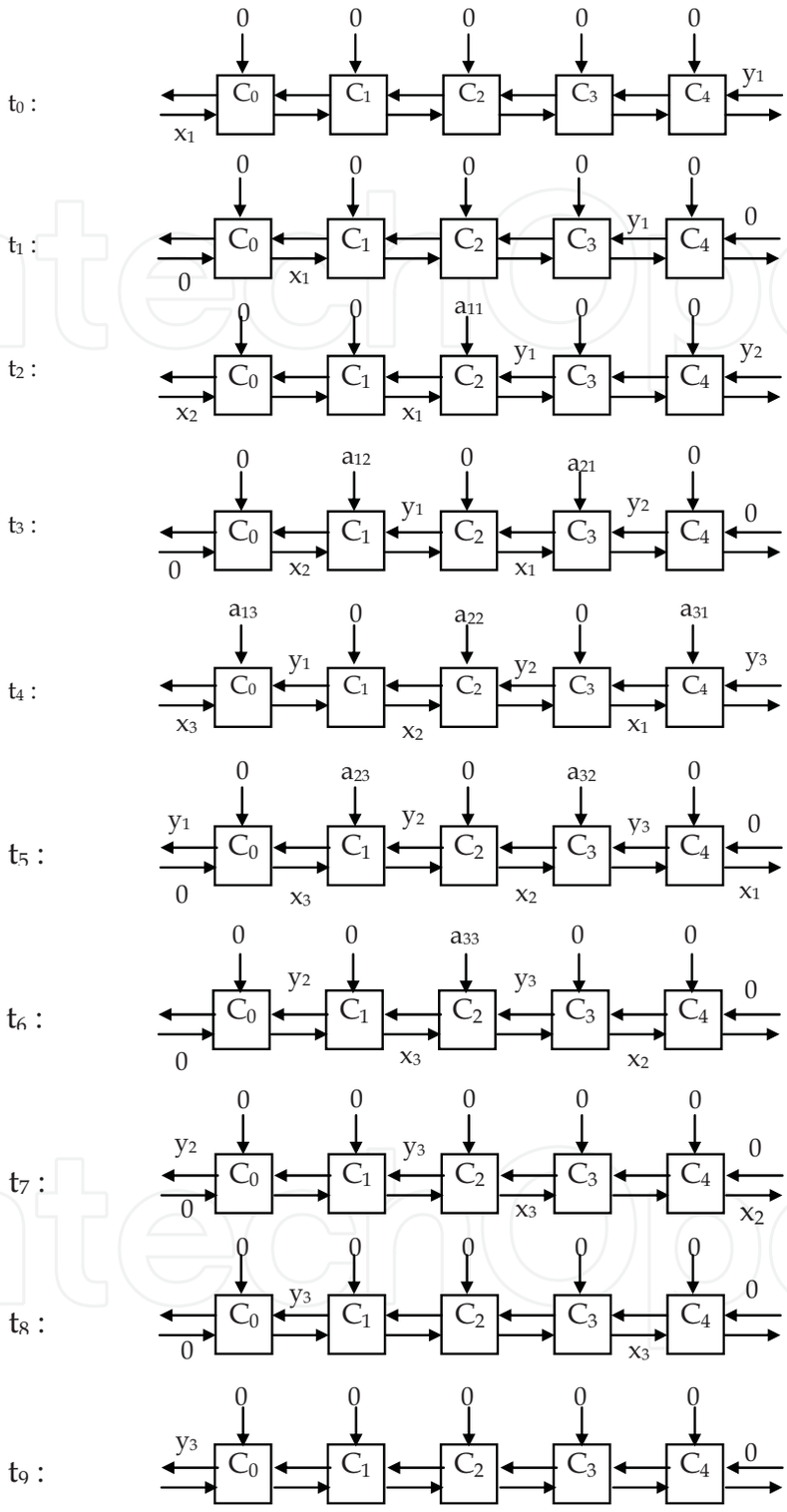


Fig. 4. Linear systolic network processing for matrix-vector product $Y=A.X$ (size 3)

2.3.2 Example of a bi-dimensional network

Consider now the matrix product : C = A.B, being sizes of A (m,n), B (n,p) and C (m,p). Each coefficient of C is processed using :

$$c_{i,j} = \text{Sum}(a_{i,k}.b_{k,j})_{\ k=1..n}\ ,\ 1 \le i \le n,\ 1 \le j \le n,\ 1 \le k \le n \qquad (4)$$

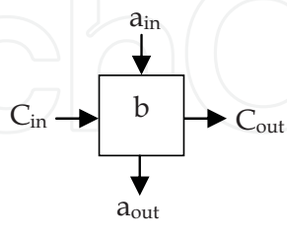


Fig. 5. Elementary cell

$$a_{out} = a_{in}\ ;\ C_{out} = C_{in} + a_{in}.b$$

This relation can be expressed via a recurrence relation like in 2.3.1. Coefficients a_{ik} propagate on j axis, and b_{kj} propagate on i axis. k is a recurrence axis that can be assimilated to a temporal axis. Elementary processings given by (4) are all identical. Network is thus made of a sole kind of cell (Fig. 5) that can be associated in square.

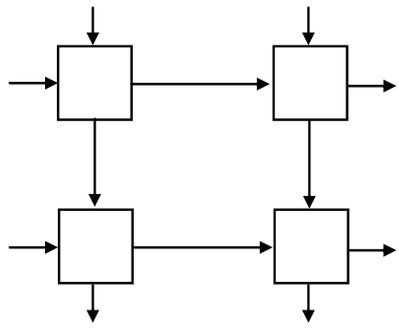


Fig. 6. Example of a 2×2 squared network

Data propagation is detailed on Fig. 7. Other organisation and data propagation possibilities exist. Hexagonal cells can also be used for this processing. In these cells, data propagate in three directions (Fig. 8) so as to be used by neighbouring ones (Fig. 9).

Conceiving systolic networks depends on the problem to be solved and the forced constraints (minimizing number of cells, data flow…). As a consequence, there is no unique method on conception. Several methods exist, some using mathematical equations of the problem to solve, others using problem algorithms. This next section will deal with these questions.

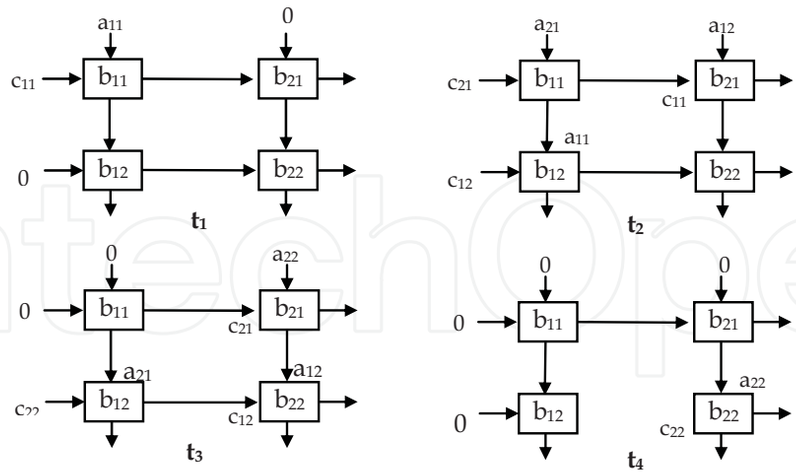
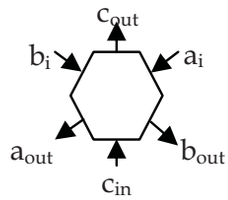


Fig. 7. Data propagation in a squared network



$$a_{out} = a_{in}$$
$$b_{out} = b_{in}$$
$$c_{out} = c_{in} + a_{in}.b_{in}$$

Fig. 8. Elementary hexagonal cell

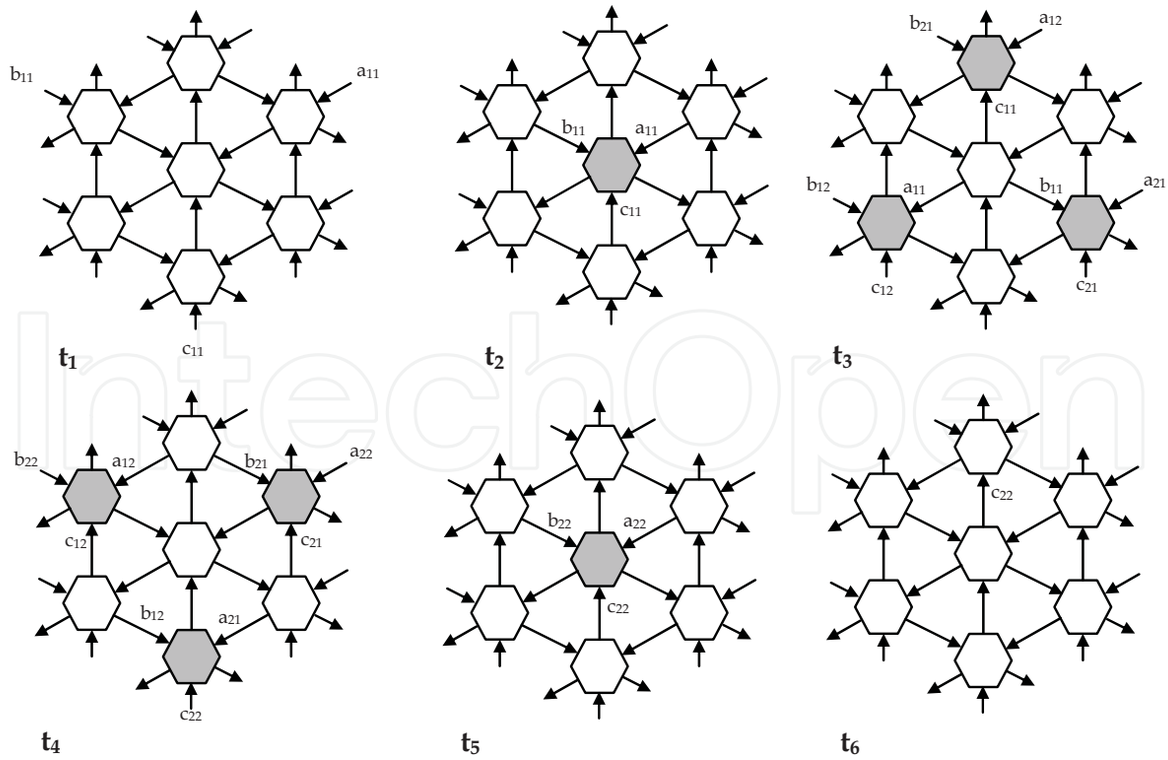


Fig. 9. Hexagonal systolic network operating cycle

3. Equation-solving based methods

Among the various approaches done, the three main ones respectively use recurrent equations, sequential algorithms transformation and fluency graphs.

3.1 Recurrent equations based method

3.1.1 Quinton method

It is based on the use of geometrical domain projection representing the processing to be done so as to define systolic structures (Quinton, 1983). It has three steps :
- Expressing the problem by a set of uniform recurrent equations on a domain $D \subset \mathbf{Z}^n$
- From this set of equations, defining a temporal function so as to schedule processings
- Defining one or several systolic architectures by applying processing allocation functions to elementary cells

These functions are determined by the different processing domain projections.

3.1.1.1 Step 1 : Creating recurrent equations

Be $\mathbf{R}^n$, the n-dimension real numbers space, $\mathbf{Z}^n$ its subset with integer coordinates and $D \subset \mathbf{Z}^n$ the processing domain. On each point z from D, a set of equations E(z) is processed :

$$\begin{aligned} u_1(z) &= f(u_1(z+\theta_1), u_2(z+\theta_2), ..., u_m(z+\theta_m)) \\ u_2(z) &= u_2(z+\theta_2) \\ &... \\ u_m(z) &= u_m(z+\theta_m) \end{aligned} \quad (5)$$

in which vectors $\theta_i \subset \Theta$ called dependency vectors are independent from z. They define which are the values where a point of the domain must take its input values. This system is uniform since θ_I does not depend on z and the couple (D, Θ) represents a dependency graph. Thus, the processing of A and B (2 n×n-matrices) is defined by :

$$c_{ij} = \mathrm{Sum}(a_{ik}.b_{kj})_{k=1..n} \text{ , } 1 \le i \le n, 1 \le j \le n$$

It can be defined by the following uniform recurrent equations system :

$$\begin{aligned} c(i,j,k) &= a(i,j,k-1)+a(i,j-1k).b(i-1,j,k) \\ a(i,j,k) &= a(i,j-1,k) \\ b(i,j,k) &= a(i-1,jk) \end{aligned} \quad (6)$$

Several possibilities to propagate data on i, j and k axis exist. a_{ik}, b_{kj} and c_{ij} are respectively independent from j, i and k, the propagation of these 3 parameters can be done following the (i,j,k) trihedron. The processing domain is the cube defined by D = {(i,j,k), 0≤i≤n, 0≤j≤n, 0≤k≤n}. Dependency vectors are θ_a = (0, 1, 0) , θ_b = (1, 0, 0) , θ_c = (0, 0, 1). With n=3, dependency graph can be represented by the cube on Fig. 10. Each node corresponds to a processing cell. Links between nodes represent dependency vectors. Other possibilities for data propagation exist.

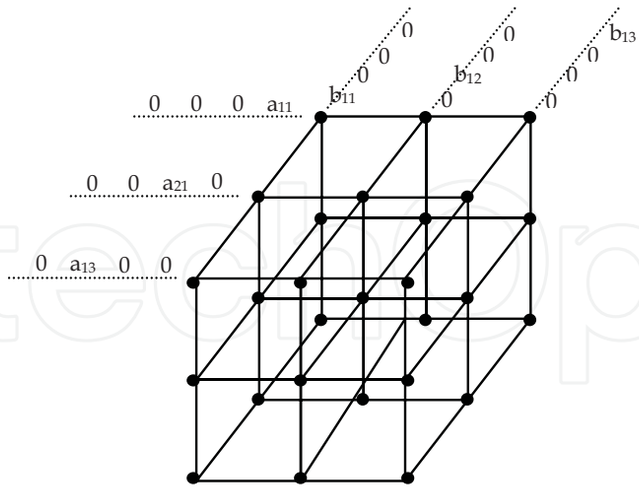


Fig. 10. Dependency domain for matrix product

3.1.1.2 Step 2 : Determining temporal equations

The second step consists in determining all possible time functions for a system of uniform recurrent equations. A time function t is from $D \subset \mathbf{Z}^n \rightarrow \mathbf{Z}^n$ that gives the processing to perform at every moment. It must verify the following condition :

If $x \in D$ depends on $y \in D$, i.e. if a vector dependency $\Theta_i = \overrightarrow{yx}$ exists, then $t(x)>t(y)$.

When D is convex, analysis enables to determine all possible quasi-affine time functions. In this aim, following definitions are used :

- D is the subset of points with integer coordinates of a convex poyedral $\underline{D}$ from $\mathbf{R}^n$.
- $\text{Sum}(\mu_i.x_i)_{i=1...m}$ is a positive combination of points $(x_1, \ldots, x_n)$ from $\mathbf{R}^n$ if $\forall i , \mu_i > 0$
- $\text{Sum}(\alpha_i.x_i)_{i=1...m}$ is a convex combination of $(x_1, \ldots, x_n)$ if $\text{Sum}(\alpha_i)_{i=1...m} = 1$
- s is a summit of $\underline{D}$ if s can not be expressed as a convex combination of 2 different points of $\underline{D}$
- r is a radius of $\underline{D}$ if $\forall x \in \underline{D}$, $\forall \mu_i \in \mathbf{R}^+$, $x+\mu_i.r \in \underline{D}$
- a radius r of $\underline{D}$ is extremal if it can not be expressed as a positive convex combination of other radii of D.
- l is a line of $\underline{D}$ if $\forall x \in \underline{D}$, $\forall \mu_i \in \mathbf{R}$, $x+\mu_i.l \in \underline{D}$
- if $\underline{D}$ contains a line, $\underline{D}$ is called a cylinder

If we limit to convex polyedral domains that are not cylinders, then the set S of summits of $\underline{D}$ is unique as well as the set R of $\underline{D}$ extremal radii. $\underline{D}$ can then be defined as the subset of points x from $\mathbf{R}^n$ with x = y + z, y being a convex combination of summits of S and z a positive combination of radii of R.

Definition 1. $T = (\lambda, \alpha)$ is a quasi-affine time function for (D, Θ) if $\forall \theta \in \Theta$, $\lambda^T.\theta \geq 1$, $\forall r \in R$, $\lambda^T.r \geq 0$, $\forall s \in S$, $\lambda^T.s \geq \alpha$

Thus, for the uniform recurrent equations system defining the matrix product, (λ,α) time functions meets the following characteristics :

$$\lambda^T=(\lambda_1, \lambda_2, \lambda_3) \text{ with } \lambda_1 \geq 1, \lambda_2 \geq 1, \lambda_3 \geq 1 \text{ and } \lambda_1 + \lambda_2 + \lambda_3 > 1.$$

A possible time function can therefore be defined by $\lambda^T = (1,1,1)$, with the following 3 radii (1,0,0), (0,1,0) and (0,0,1).

3.1.1.3 Step 3 : Creating systolic architecture

Last step of the method consists in applying an allocation function ξ of the network cells. This function $\xi=a(x)$ from D to a finite subset of $\mathbf{Z}^m$ where m is the dimension of the resulting systolic network, must verify the following condition (t : time function seen on 3.1.1.2) that guarantees that two processings performed on a same cell are not simultaneous :

$$\forall x \in D, \forall y \in D, a(x)=a(y) \Rightarrow t(x) \neq t(y).$$

Each cell has an input port $I(\mu_i)$ and an output port $O(\mu_i)$, associated to each μ_i , defined in the system of uniform recurrent equations. $I(\mu_i)$ of cell C_i is connected to $O(\mu_i)$ of cell $C_{i+a.\theta i}$ and $O(\mu_i)$ of cell C_i is connected to $I(\mu_i)$ of cell $C_{i-a.\theta i}$. Communication time between 2 associated ports is $t(\theta_i)$ time units. For the matrix product previously considered, several allocation functions can be defined. :

- ξ = (0,0,1) or (0,1,0) or (1,0,0), respectively corresponding to a(i,j,k)=k, a(i,j,k)=j, a(i,j,k)=i. Projection of processing domain in parallel of one of the axis leads to a squared shape
- ξ = (0,1,1) or (1,0,1) or (1,1,0), respectively corresponding to a(i,j,k)=j-k, a(i,j,k)=i-k, a(i,j,k)=i-j. Projection of processing domain in parallel of the bisector lead to a mixed shape
- ξ = (1,1,1). Projection of processing domain in parallel of the trihedron bisector lead to a hexagonal shape.

Li and Wah method (Li & Wah, 1984) is very similar to Quinton, the only difference is the use of an algorithm describing a set of uniform recurrent equations giving data spatial distribution, data time propagation and allocation functions for network building.

3.1.2 Mongenet method

The principle of this method lies on 5 steps (Mongenet, 1985) :

- systolic characterization of the problem
- definition of the processing domain
- definition of the generator vectors
- problem representation
- definition of associated systolic nets

3.1.2.1 Systolic characterization of the problem

The statement characterizing a problem must be defined with a system of recurrent equations in $\mathbf{R}^3$:

$$\begin{gathered} y_{ij}^k = f(y_{ij}^{k-1}, a^1, ..., a^n) \\ y_{ij}^k = v, v \in \mathbf{R}^3 \\ 0 \leq k \leq b, i \in I, j \in J \end{gathered} \quad (7)$$

in which $a^1, \ldots, a^u$ are data, I and J are intervals from $\mathbf{Z}$, k being the recurrency index and b the maximal size of the equations system.

a^q elements can belong to a simple sequence (s_l) or to a double sequence ($s_{l,l'}$), $l \in L$, $l' \in L'$, L and L' being intervals of **Z**. In this case, a^q elements are characterized by their indexes which are defined by a function h depending on i, j and k. The result of the probem is a double sequence (r_{ij}), $i \in I$, $j \in J$ where r_{ij} can be defined in two ways :
− the result of a recurrency $r_{ij} = y_{ij}^b$
− $r_{ij} = g(y_{ij}^b, a^1, ..., a^n)$
For example, in the case of resolving a linear equation, results are a simple suite y_i, , $1 \le i \le n$, each y_i being the result of the following recurrency :

$$y_i^{k+1} = y_i^k + a_{i,k+1}. x_{k+1}$$
$$y_i^0 = 0 \qquad (8)$$
$$0 \le k \le n-1,\ 1 \le i \le n$$

3.1.2.2 Processing domain

The second step of this method consists in determining the processing domain D associated to a given problem. This domain is the set of points with integer coordinates corresponding to elementary processings. It is defined from the equations system defining the problem.
Definition 2. Consider a systolizable problem which recurrent equations are similar to (7) and defined in $\mathbf{R}^3$. The D domain associated to the problem is the union of two subsets D_1 and D_2.:
- D_1 is the set of indexes values defining the recurrent equations system. b being a bound defined by the user, it is defined as $D_1 = \{ (i,j,k) \subset \mathbf{Z}^3, i \in I, j \in J, a \le k \le b\}$
- D_2 is defined as :
  - if the problem result is $(r_{ij}) : i \in I, j \in J \mid r_{ij} = y_{ij}^b$, then $D_2 = \varnothing$
  - if the problem result is $(r_{ij}) : i \in I, j \in J \mid r_{ij} = q(y_{ij}^b , a^1, ..., a^u)$,
    then $D_2 = \{ (i,j,k) \subset \mathbf{Z}^3, i \in I, j \in J, k = b+1 \}$

In the case of the MVP defined in (8), $D_1 = \{ (i,k) \in \mathbf{Z}^2 \mid , 0 \le k \le n-1, 1 \le i \le n\}$ and D_2 is empty, since an elementary result y_i is equal to a recurrency result..
Definition 3. Systolic specification of a defined problem in $\mathbf{R}^3$ from p data families implies that $D \subset \mathbf{Z}^3$ defines the coordinates of elementary processings in the canonical base (b_i, b_j, b_k). For example, concerning the MVP previously defined, $D = \{ (i,k) \in \mathbf{Z}^2 \mid , 0 \le k \le n-1, 1 \le i \le n\}$.

3.1.2.3 Generating vectors

Definition 4. Let's consider a problem defined in $\mathbf{R}^3$ from p data families, and d a data family which associated function h_d is defined in the problem systolic specification.
ψ_d is called a generating vector associated to the d family, when it is a vector of $\mathbf{Z}^3$ which coordinates are (ψ_i ,ψ_j ,ψ_k) in the canonical base BC of the problem, such as :
- for a point (i , j , k) of the D domain, $h_d(i, j, k) = h_d(i+\psi_i , j+\psi_j , k+\psi_k)$
- highest common factor (HCF) is : $HCF(\psi_i ,\psi_j ,\psi_k) = +1$ or -1

This definition of generating vectors is linked to the fact that (i, j, k) and ($i+\psi_i$, $j+\psi_j$, $k+\psi_k$) points of the domain, use the same occurrence of the d data family.
The choice of ψ_d with coordinates being prime between them enables to limit possible choices for ψ_d and to obtain all points ($i+n\times\psi_i$, $j+\psi_j$, $k+\psi_k$), $n \in \mathbf{Z}$, from any (i, j, k) point of D.
In the case of the matrix-vector product, generating vectors $\Psi_y = \Psi_a = \Psi_x = (\psi_y , \psi_a , \psi_x)$ are associated to results h_y, h_a and h_x. Generating vectors are as following :

$h_y(i,k)=h_y(i+\psi_i,\ k+\psi_k) \Leftrightarrow i = i+\psi_i \Leftrightarrow \psi_i = 0$. Moreover, $HCF(\psi_i,\ \psi_k)=\pm1$, thus $\psi_k=\pm1$. Generating vector ψ_y can therefore be (0, 1) or (0, -1).
$h_x(i,k) = i+k$. Generating vector Ψ_a must verify $h_a(i,k)=h_x(i+\psi_i,\ k+\psi_k) \Leftrightarrow i+k=i+k+\psi_i+\psi_k$
$\Leftrightarrow \psi_i = -\psi_k$. Moreover, $HCF(\psi_i,\psi_k)=+1$ or -1, thus $\Psi_a=(1,-1)$ or (-1,1)
Similar development leads to $\Psi_x=(1,0)$

3.1.2.4 Problem representation

A representation set is associated to a problem defined in $\mathbf{R}^3$. Each representation defines a scheduling of elementary processings. The temporal order relation between the processing requires the introduction of a time parameter that evolves in parallel to the recurrency, since this relation is a total order on every recurrency processings associated to an elementary processing. We thus call spacetime, the space $ET \subset \mathbf{R}^3$. with orthonormal basis (i, j, t), where t represents the time axis.

Definition 5. A problem representation in ET is given by :
- the transformation matrix P from the processing domain canonical base to the spacetime basis
- the transformation vector V such as $V=\overrightarrow{O'O}$, where O is the origin of the frame associated to the canonical basis and O' is the origin of the spacetime frame

Point coordinates in spacetime can there for be expressed from coordinates in the canonical basis :

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}_{ET} = P \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix}_{BC} + V$$

$$\Leftrightarrow \begin{pmatrix} i \\ j \\ k \end{pmatrix}_{BC} = P^{-1} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix}_{ET} - P^{-1} \cdot V$$

This representation is given by the example of the Matrix Vector Product of Fig. 11.

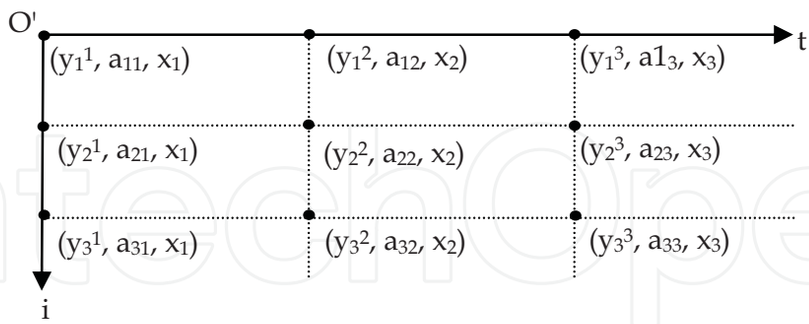


Fig. 11. Representation of the Matrix Vector Product in spacetime (t=k)

We call R_0 the initial representation of a problem, the one for which there is a coincidence between the canonical basis and the spacetime basis, i.e. P = I, I being the Identity Matrix, and V the null vector (O and O' are coinciding). For the MVP example, initial representation is given on Fig. 11.
These representations show the occurencies of a data at successive instants. Processings can be done in the same cell or on adjacent cells. In the first case, data makes a systolic network

made of functional cells in which the data can be put in the cell memory. In the second case, data circulate in the network from cell to cell.
The representation of the problem in spacetime defines a scheduling for the processing. To obtain networks with a different order, we apply transformations on the initial representation R_0. If, after a transformation, data are still processed simultaneously, a new transformation is applied until the creation of an optimal scheduling. From this representation a set of systolic networks is determined.
Applying a transformation to a representation consists in modifying the temporal abscissa of the points. Whatever the representation is, this transformation must not change the n-uple associated to the invariant points when order and simultaneity of processings is changed. The only possible transformations are thus those who move the points from the D domain in parallel to the temporal axis (O', t). For each given representation, D_t is the set of points which have the same temporal abscisse, resulting in segments parallel to (O', i) in spacetime are obtained.
The transformation to be applied consists in deleting data occurencies simultaneities by forcing their successive and regular use in all the processings, which implies that the image of all lines d_t by this transformation is also a line in the image representation. For instance, for the initial representation R_0 of the MVP, D_t straight lines are dotted on Fig. 11. One can therefore see that occurrencies of data x_k, $0 \le k \le n-1$ are simultaneously used on each point of straight line D_k with t = k. Therefore, a transformation can be applied to associate a non parallel straight line to the (O', i) axis to each D_t parallel straight line to (O', i).
Two types of transformations can be distinguished leading to different image straight lines :
- T_c for which the image straight line has a slope = +P (Fig. 12a)
- T_d for which the image straight line has a slope = -P (Fig. 12b)

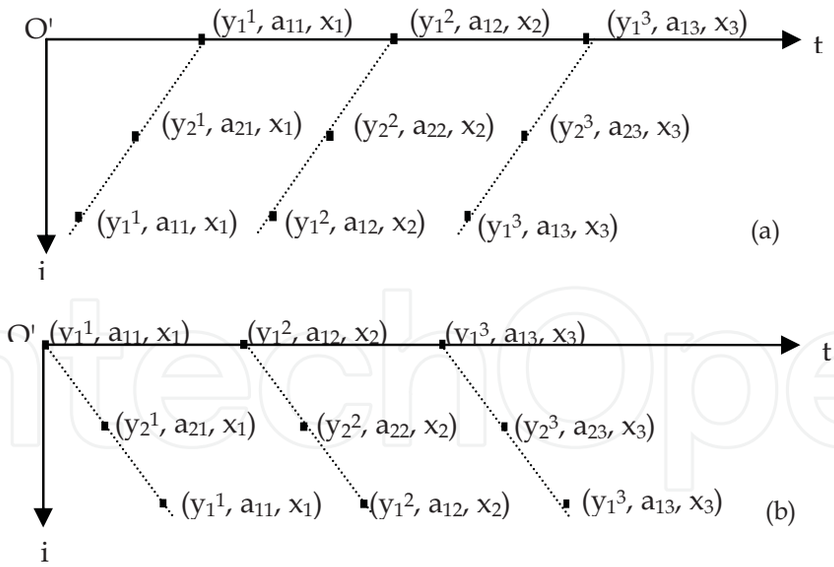


Fig. 12. Applying a transformation on the initial representation : (a) *Tc*, (b) *Td*

The application of a transformation enables to delete the occurencies use simultaneity of data, but increases the processing total execution time. For instance, for the initial representation of Fig. 11, the total execution time is t=n=3 time units, whereas for representations on Fig. 12, it is t=2.n-1 = 5 time units.

Concerning the initial representation, one can notice that 2 points of the straight line D_t having the same temporal abscisse have 2 corresponding points on the image straight line which coordinates differ by 1. It means that two initially simultaneous processings became successive. After the first transformation, no simultaneity in data occurency use is seen, since all elementary processings on D_t parallel to (O', i) use different data. Thus, no other transformation is applied. For the different representations, P (transformation matrices) as well as V (translation vectors) are :

$$P=\begin{pmatrix}1&0\\0&1\end{pmatrix} \text{ and } V=\begin{pmatrix}0\\0\end{pmatrix} \text{ for Fig. 12a.}$$

$$P=\begin{pmatrix}1&0\\1&1\end{pmatrix} \text{ and } V=\begin{pmatrix}0\\3\end{pmatrix} \text{ for Fig. 12b.}$$

3.1.2.5 Determining systolic networks associated to a representation

For a given representation of a problem, the last step consists in determining what is/are the corresponding systolic network(s). The repartition of processings on each cell of the net has therefore to be carefully chosen depending on different constraints. An allocation direction has thus to be defined, as well as a vector with integer coordinates in $\mathbf{R}^3$, which direction determines the different processings that will be performed in a same cell at consecutive instants. In fact, the direction of allocations can not be chosen orthogonally to the time axis, since in this case, temporal axis of the different processings would be the same, which contradicts the definition.

Consider the problem representation of Fig. 12a. By choosing for instance an allocation direction $\xi=(1, 0)_{BC}$ or $\xi=(1, 1)_{ET}$ and projecting all the processings following this direction (Fig. 13), the result is the systolic network shown on Fig. 14. This network is made of n=3 cells, each performing 3 recurrency steps. The total execution time is therefore $2n$-1 = 5 time units. If an allocation direction colinear to the time axis is chosen, the network shown on Fig. 15 is then obtained.

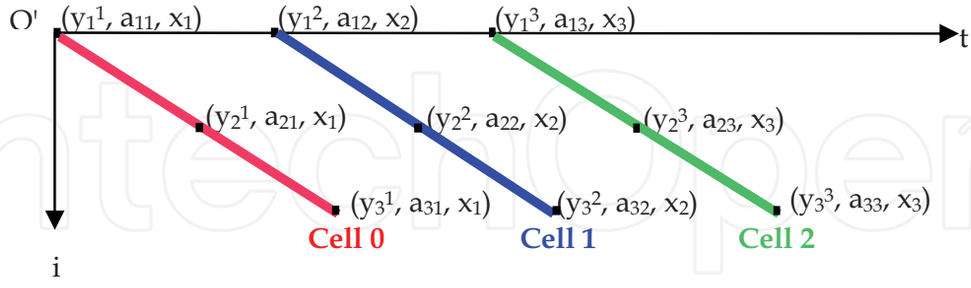


Fig. 13. Processings projection with $\xi=(1,1)_{ET}$

Other networks can be obtained by choosing another value for D_t slope. The nature of the network cells depends on the chosen allocation direction.

Cappello and Steiglitz approach (Capello & Setiglitz, 1983) is close to Mongenet. It differs from the canonical representation obtained by associating a temporal representation indexed on the recurrency definition. Each index is associated to a dimension of the

geometrical space, and each point corresponds to a n-uple of indexes in which recurrency is defined.

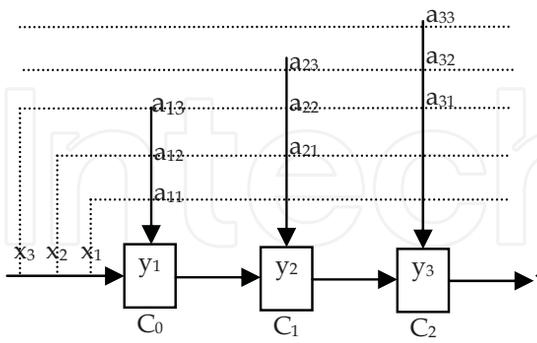


Fig. 14. Systolic network for $\xi=(1,1)_{ET}$

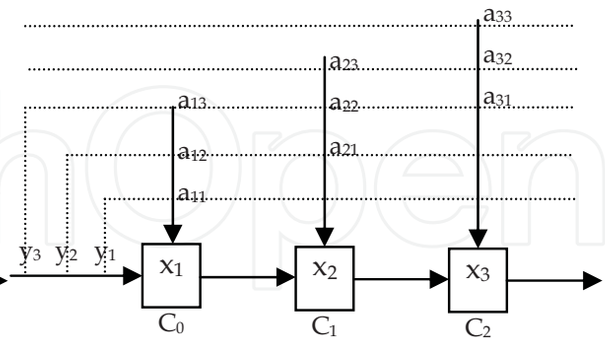


Fig. 15. Systolic network for $\xi=(0,1)_{ET}$

Basic processings are thus directly represented in the functional specifications of the architecture cells. The different geometrical representations and their corresponding architectures are then obtained by applying geometrical transformations to the initial representation.

3.2 Methods using sequential algorithms

Among all methods listed in (Quinton & Robert, 1991), we'll detail a bit more the Moldovan approach (Moldovan, 1982) that is based on a transformation of sequential algorithms in a high-level language.

The first step consists in deleting data diffusion in the algorithms by moving in series data to be diffused. Thus, for (n×n)-matrices product, the sequential algorithm is :

$$\forall i \mid 1\le i\le n, \forall j \mid 1\le j\le n, \forall k \mid 1\le k\le n, c_{new}(i,j)=c_{old}(i,j) + a(i,k).b(k,j) \quad (9)$$

If one loop index on variables a, b and c is missing, data diffusion become obvious. When pipelining them, corresponding indexes are completed and artificial values are introduced so that each data has only one use. New algorithm then becomes :

$$\forall i \mid 1\le i\le n, \forall j \mid 1\le j\le n, \forall k \mid 1\le k\le n$$
$$a^{j+1}(i, k) = a^{j}(i, k)$$
$$b^{i+1}(k, j) = b^{i}(k, j)$$
$$c^{k+1}(i, j)= c^{k}(i, j)+ a^{j}(i, k).b^{i}(k, j)$$

The algorithm is thus characterized by the set L^n of indexes of n overlapped loops. Here,

$$L^3 = \{ (k,i,j) \mid 1\le k\le n, 1\le i\le n, 1\le j\le n \}$$

which corresponds to the domain associated to the problem.

The second step consists in determining the set of dependency vectors for the algorithm. If an iteration step characterized by a n-uple of indexes $I(t) = \{i^1(t), i^2(t), ..., i^n(t)\} \in L^n$ uses a

data processed by an iteration step characterized by another n-uple of indexes $J(t)= \{ j^1(t), j^2(t), ..., j^n(t) \} \in L^n$, then a dependency vector $DE(t)$ associated to this data is defined :

$$DE(t) = J(t) - I(t)$$

Dependency vectors can be constant or depending of L^n elements. Thus, for the previous algorithm, processed data $c^k(i,j)$ at the step defined by (i, j, k-1) is used at the step (i, j, k). This defines a first dependency vector $d_1=(i, j, k) - (i, j, k-1) = (0, 0, 1)$. In the same way, step (i, j, k) uses the $a^j(i, k)$ data processed at the step (i, j-1, k) as well as the $b^i(j, k)$ data processed at the step (i-1, j, k). The two other dependency vectors of the problem are therefore $de^2=(0,1,0)$ and $de^3=(1,0,0)$.
The next step consists in applying on the $<L^n, E>$ structure a monotonous and bijective transformation T (E is the order imposed by the dependency vectors), defined by :

$$T : <L^n, E> \rightarrow <L_T^n, E_T>$$

T is partitionned into :

$$\Pi : L^n \rightarrow L_T^k, k<n$$
$$S : L^n \rightarrow L_T^{n-k}$$

k gives the dimension of Π and S. It is such as the function results in the order E_T. Thus, the k first coordinates of J and L_T^n depend on time, whereas the following n-k coordinates are linked to the algorithm geometrical properties. For obtaining planar results, n-k must be less or equal than 2.
In the case that the algorithm made of n loops is characterized by n constant dependency vectors

$$DE = \{de_1, de_2, ... , de_n\}$$

the transformation T is chosen linear, i.e. $J = T . I$

If v_i is the dependency vector de_j after transformation, $V_i = T. DE_j$, the system to solve is $T.DE = \Delta$, $DE = \{ v_1, v_2, ..., v_m \}$. Necessary and sufficient conditions for existence of a valid transformation T for such an algorithm are :
- $v_i = DE_i[c_j]$, c_j being the HCF of the d_j elements
- $T.DE = \Delta$ has a solution
- The first non-zero element of v_j is positive

Therefore, in our exemple of matrix product, dependency vectors are defined by :

$$D=\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

A linear transformation T is such as $T = \Delta$. The first non-zero element of v_j being positive, we consider $\Pi . d_i > 0$ and k =1 in order to size Π and S, with :

$$T=\begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix}$$

In this case, $\Pi.de_i = t_{1i} > 0$. Thus, we choose for $t_{1i,}$ i=1, ..., 3, the lowest positive values, i.e. $t_{11} = t_{12} = t_{13} = 1$. S is determined by taking into account that T is bijective and with a matrix made of integers, i.e. Det(T) = ±1 . Among all possible solutions, we can choose :

$$T=\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

This transformation of the indexes set enables to deduce a systolic network :
- Functions processed by the cells are deduced from the algorithm mathematical expressions. An algorithm similar to (9) contains instructions executed for each point of L^n. Cells are thus identical, except for the peripherical ones. When loop processings are too important, the loop is decomposed in several simple loops. The corresponding network therefore requires several different cells.
- The network geometry is deduced from function S. Identification number for each cell is given by $S(I) = (j^{k+1}, ... , j^n)$ for $I \in L^n$. Interconnections between cells are deduced from the n-k last components of each dependency vector v_j after being transformed :

$$v_j^s = S(I + DE_j) - S(I)$$

When T is linear :

$$v_j^s = S.DE_j$$

For each cell, v_j^s vectors indicate the identification number of the cell for the variable associated to the vector. The network temporal processing is given by :

$$\Pi : L^n \rightarrow I_T^k$$

The elementary processing corresponding to $I \in L^n$ is performed at $t=\Pi(I)$. The communication time for a data flow associated to the dependency vector DE_j is given by $\Pi(I+DE_j) - \Pi(I)$, which is reduced to $\Pi(DE_j)$ when T is linear.
Using the integer k for sizes of Π and S with the lowest possible value, the number of parallel operations is increased at the expense of cells number. Thus, when considering the matrix product defined with the following linear transformation :

$$T=\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

S is defined by :

$$S\begin{pmatrix}k\\i\\j\end{pmatrix}=\begin{pmatrix}0&1&0\\0&0&1\end{pmatrix}.\begin{pmatrix}k\\i\\j\end{pmatrix}=\begin{pmatrix}i\\j\end{pmatrix}$$

The network is therefore a bidimensional squared network (Fig. 1c).
Data circulation are defined by S.DE_j. For the c_{ij} data, dependency vector is

$$de_1=\begin{pmatrix}1\\0\\0\end{pmatrix}\Rightarrow S.de_1=\begin{pmatrix}0\\0\end{pmatrix}$$

Therefore, data remain in cells.
For the a_{ik} data, dependency vector is :

$$de_2=\begin{pmatrix}0\\0\\1\end{pmatrix}\Rightarrow S.de_2=\begin{pmatrix}0\\1\end{pmatrix}$$

a_{ik} circulate horizontally in the network from left to right.
Similarly, we can find :

$$S.de_3=\begin{pmatrix}1\\0\end{pmatrix}$$

and deduce that b_{kj} circulate vertically in the network from top to bottom.

3.3 Fluency graphs description

In this method proposed by Leiserson and Saxe (Leiserson & Saxe, 1983), a circuit is formally defined as an oriented graph $G = (V, U)$ which summits represent the circuit functional elements. A particular summit represent the host structure so that the circuit can communicate with its environment. Each summit v of G has a weight $d(v)$ representing the related cell time cycle. Each arc $e = (v, v')$ from U has an integer weight $w(e)$ which represents the number of registers that a data must cross to go from v to v'.
Systolic circuits are those for which every arc has at least one related register and their synchroniszation can be done with a global clock, with a time cycle equal to Max($d(v)$).
The transformation which consists in removing a register on each arc entering a cell, and to add another on each arc going out of this cell does not change the behaviour of the cell concerning its neighborhood.
By the way, one can check that such transformations remain invariant the number of registers on very elementary circuit.
Consequently, a necessary condition for these transformations leading to a systolic circuit, is that on every elementary circuit of the initial graph, the number of registers is higher or equal to the number of arcs. Leiserson and Saxe also proved this condition is sufficient.
Systolic architecture condition is therefore made in 3 steps :

– defining a simple network w in which results accumulate at every time signal along paths with no registers

– determining the lowest integer k. Thus, the resulting newtork w_k obtained from w by multiplying by k the weights of all arcs is systolizable. w_k has the same external behaviour than w, with a speed divided by k.
– systolizing w_k using the previous transformations

This methodology is interesting to define a systolic architecture from an architecture with combinatory logic propagating in cascade. Main drawback is that the resulting network often consists of cells activated one time per k time signals. This means the parallelism is limited and execution time is lenghtened.
Other methods use these graphs :
- Gannon (Gannon, 1982) uses operator vectors to obtain a functional description of an algorithm. Global functional specificities are viewed as a fluency graph depending on used functions and operators properties, represented as a systolic architecture
- Kung (Kung, 1984) uses fluency graphs to represent an algorithm. The setting up of this method requires to choose the operational basic modules corresponding to the functional description of the architecture cells.

4. Method based on Petri Nets

In previously presented methods, the thought process can almost be always defined in three steps :
– rewriting of problem equations as uniform recurrent equations
– defining temporal functions specifying processings scheduling in function of data propagation speed
– defining systolic architectures by application of processings allocation functions to processors

To become free from these difficulties that may appear in complex cases and in the perspective of a method enabling automatic synthesis of systolic networks, a different approach has been developped from Architectural Petri Nets (Abellard et al., 2007) (Abellard & Abellard, 2008) with three phases :
– constitution of a Petri Net basic network depending on the processing to perform
– making of the Petri Net in a systolic shape (linear, orthogonal or hexagonal) defining data propagation

4.1 Architectural Petri Nets

To take into account sequential and parallel parts of an algorithm, an extention of Data Flow Petri Nets (DFPN) (Almhana, 1983) has been developped : Architectural Petri Nets (APN), using Data Flow and Control Flow Petri Nets in one model. In fact Petri Nets showed their efficiency to model and specify parallel processings and on various applications, including hardware/software codesign (Barreto et al., 2008) (Eles et al., 1996) (Gomes et al., 2005) (Maciel et al., 1999) and real-time embedded systems modeling and development (Cortés et al., 2003) (Huang & Liang, 2003) (Hsiung et al., 2004) (Sgroi et al., 1999). However, they may be insufficient to reach the implementation aim when available hardware is either limited in resources or not fully adequate to a particular problem. Hence, APN have been designed to limit the number of required hardware resources while taking advantage of the chip performances so that the importance of execution time lengthening may be non problematic

(Abellard, 2005). Their goal is on the one hand to model a complete algorithm, and on the other hand, to design the interface with the environment. Thus, in addition with operators used for various arithmetic and logic processing, other have been defined for the Composition and the Decomposition in parallel of data vectors.

4.1.1 Defactorized operators

4.1.1.1 Compose

It proceeds to the ordered regrouping of d input data T'_1 to T'_d of a same type into an output vector $[T'_1 ... T'_d]$ (Fig. 16).

$$Co(T'_1, \ldots, T'_d) \rightarrow [\, T'_1, \ldots, T'_d \,]$$

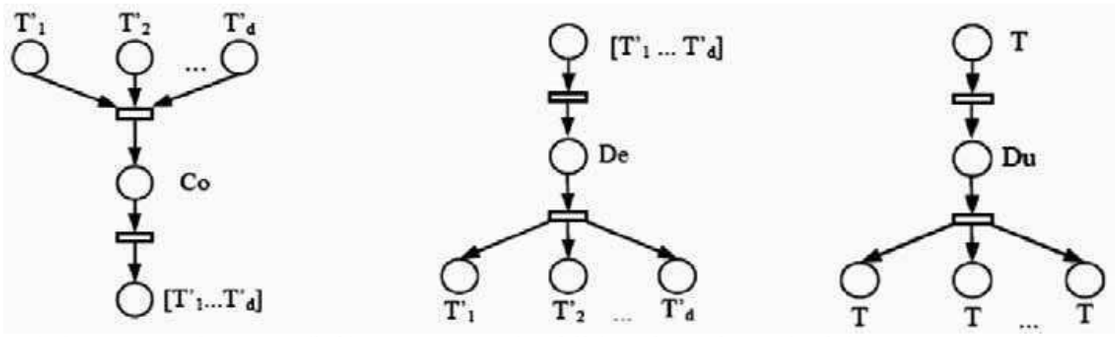


Fig. 16. Operators : Compose (left), Decompose (middle) and Duplicate (right)

4.1.1.2 Decompose

It proceeds to the decomposition of a vector $[T'_1...T'_d]$ into its d elements T'_1 to T'_d (Fig. 16).

$$De([T'1 ... T'd]) \rightarrow T'_1, \ldots, T'_d$$

4.1.1.3 Duplicate

It proceeds to the duplication of input data to d subnets as in Data Flow Petri Nets, different operators can not use the same set of data (Fig. 16).

4.1.1.4 Example of a Matrix Vector Product

An example of application of these operators is given on Fig. 17 with a MVP. One can easily see that the more important are the sizes of matrix and vector, the more important is the number of operators in the Net (and consequently the required hardware ressources).

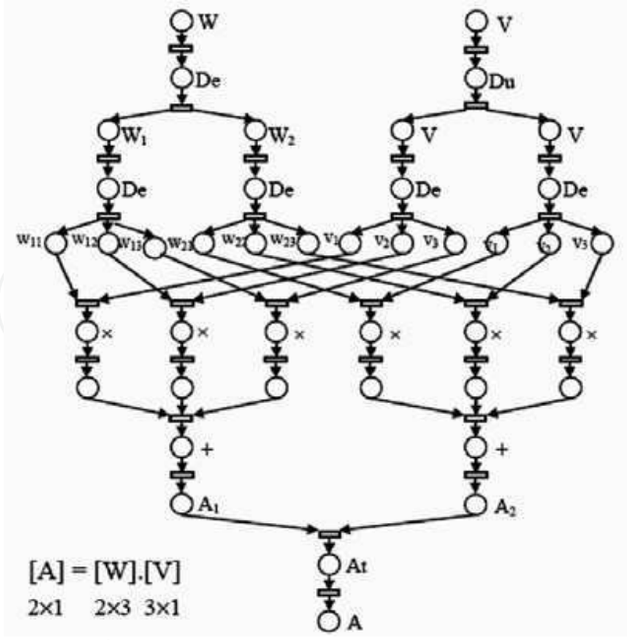


Fig. 17. Data Flow Petri Net of a MVP

The use of classic DFPN leads to an optimal solution as regards the execution time, thanks to an unlimited quantity of resources. However, a problem may appear. In fact, although these operations are simple taken separately, their combination may require relatively important amount of hardware resources, depending on the data type of the elements, and on the input matrix and vector sizes. We therefore have to optimize the number of cells prior to execution time. This is not a major drawback with a programmable component which has short execution times for real time controls. In order to limit as more as possible the resources quantity, we defined the Architectural Petri Nets (APN), that unify in a unique model Data Flow and Control Flow.

4.1.2 Factorization concept

The decomposition of an algorithm modelled with DFPN into a set of operations leads to the repetition of elementary identical operations on different data. So, it may be interesting to replace the repetitive operations by a unique equivalent subnet in which input data are enumerated and output data are sequentially produced. This leads us to define the concept of factorized operator which represents a set of identical operations processing differentsequential data.

Each factorized operator is associated to a factorization frontier splitting 2 zones : a slow one and a fast one. When the operations of slow zone are executed one time, those of the fast zone are executed n times during the same lapse of time.

Definition 6. A T-type element is represented by a vector of d1 elements, all of T'-type. Each T' type element may be also a vector of d_2 T''-type elements, and so on.

Definition 7. A Factorized Data Flow Petri Net (FDFPN) is a 2-uple (R, F) in which R is a DFPN and F a set of factorization frontiers F = {FF_1 , FF_2, ... FF_n}.

4.1.3 Factorized operators

The data enumeration needs to use a counter for each operator. An example is given on Fig. 18. Various factorized operators that are used in our descriptions are described in next sections.

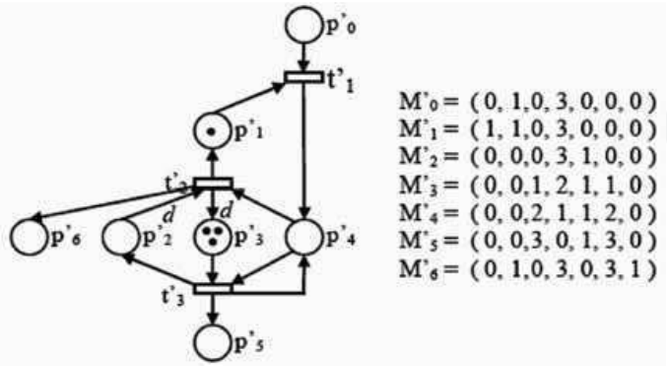


Fig. 18. Counter from 0 to n-1 (here n=3)

4.1.3.1 Separate

It is identified by Se and it proceeds to the factorization of a Data Flow in an input vector form [T'_1...T'_d] by enumerating the elements T'_1 to T'_d. A change of the input data value in the operator corresponds to d changes of the output data value. The Separate operator allows to go through a factorization frontier by increasing the data speed : the down speed of the input data of Separate is d times greater than the upper speed of output data. d output data (fast side) correspond to one input data (slow side) as the result of the input data elements enumeration synchronized with an internal counter (which sole p'_0 and p'_6 places are represented for graphic simplification).

Thus, a factorization frontier FF defined by a Separate operator dissociates the slow side from the fast side (Fig. 19a). A graphic simplified representation, where places coming from counter are not represented, is adopted on Fig. 19b. In a FDFPN, the operator Separate corresponds to the factorized equivalent of Decompose defined in 4.1.1.2.

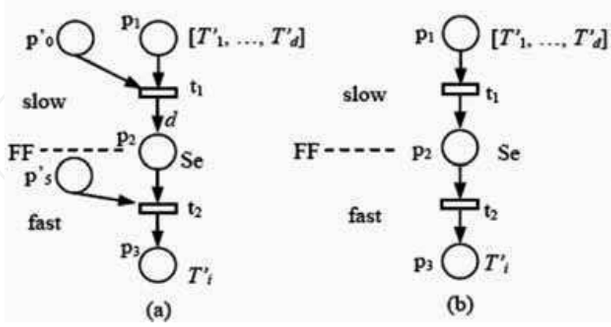


Fig. 19. Separate operator

4.1.3.2 Attach

It is identified by At and it proceeds to the factorization of d input data flows T'_i by collecting them under an output vector form [T'_1...T'_d] (Fig. 20a with p'_0 and p'_6 coming from the d-counter, and graphic simplified representation on Fig. 20b). d changes of input data

values in the Attach operator correspond to one change of the output data values. In a FDFPN, the operator Separate corresponds to the factorized equivalent of Compose defined in 4.1.1.1.

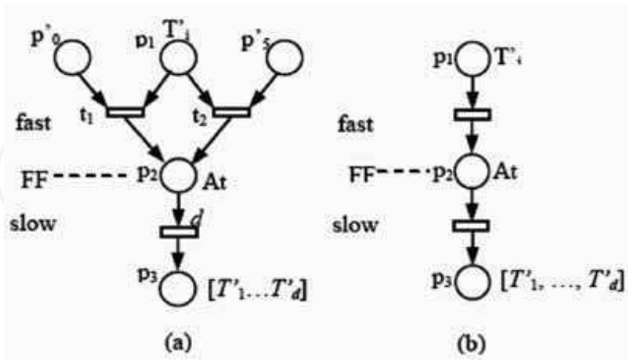


Fig. 20. Attach operator

4.1.3.3 Iterate

It is identified by It and it proceeds to the iteration of a subnet which has s as input and e as output. The operator provides the specification of connexions between repetitive subnets, and appears in the FDFPN as a cycle through the "It" operator. On Fig. 21a, p'_0 and p'_6 come from the previously described *d*-counter, produced by a control operator which will be defined in section 4. (Fig. 21b being the simplified representation of the operator). in : initializing step ; fi : final step (counting completed)

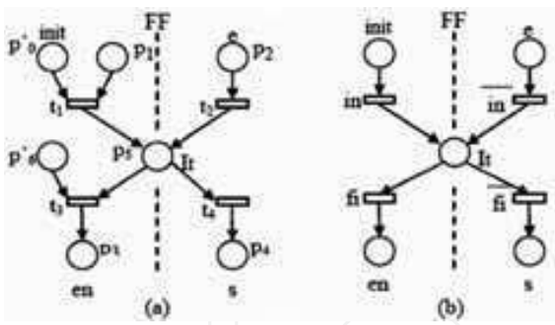


Fig. 21. Iterate operator

4.1.3.4 Diffuse

This operator provides *d* times in output the repetition of an input data. Diffuse (Di) is a factorized equivalent to the Duplicate function defined in 3.2.3.3. (Fig. 22).

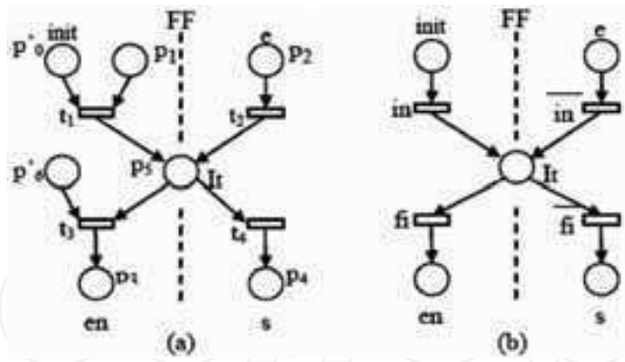


Fig. 22. Diffuse operator

4.1.4 Example of a Matrix Vector Product

From the example of previous MVP, the corresponding FDFPN is given on Fig. 23a. Factorization enables to limit the number of operators in the architecture - and therefore the number of logic elements required – since data are processed sequentially. As for the validation places that enables to fire the net transitions, they come from a Control Flow Petri Nets (CFPN), which is described in the next paragraph (Fig. 23b).

Given the algorithm specification, i.e. the FDFPN, control generation of its implementation is deduced from data production and consumption relations, and neighborhood relation between all FF. Hence the generation of control signals equations that can be modelled with Petri Nets, by connecting control units related to each FF. Control synthesis of a hardware implementation consists in producing of validation and initialization signals for needed counters. Control generation of hardware implementation corresponding to the algorithm specification described by its FDFPN is thus modelled by CFPN.

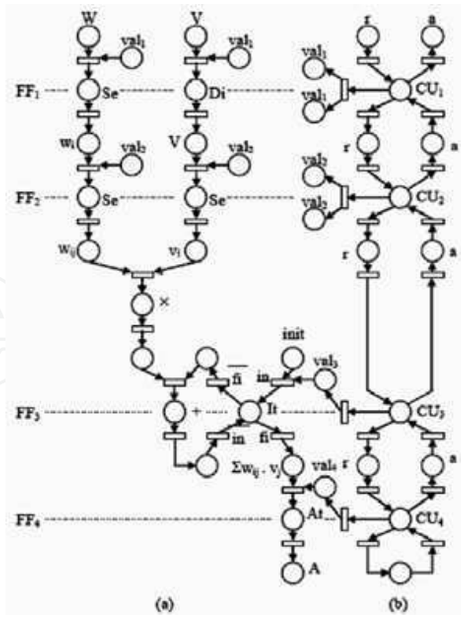


Fig. 23. FDFPN description of a MVP

4.1.5 Definition of Control Flow Petri Nets

A CFPN is a 3-tuple (*R*, *F*, *Pc*) in which : *R* is a 2-part places Petri Net, *F* is a set of factorization frontiers, *Pc* is a set of control places.

4.1.5.1 Control synthesis

Five steps are necessary :
- Design of a FDFPN.
- Design of the PN representing neighborhood relations between frontiers.
- Definition of neighborhood, production and consumption relations using this Petri Net.
- Generation of signal control equations.
- Modelling using CFPN by connecting unit controls related to each FF.

4.1.5.2 Control units

In a sequential circuit containing registers, each FF has relations on its both sides (slow and fast). Relations between request and acknowledgment signals, up and down, for both slow and fast sides, provide the design of the control unit. It is composed of a d-counter and additional logic which generate communication protocols, cpt (counter value) and val (validation signal) for transitions firing.

Functions rules : If the control unit (CU) receives aupper request (*ur*= 1) and the down acknowledge is finished (*da*=0), it validates the data transfer (*ua*=1) and sends a request to the next operator (*dr*=1) (Fig. 24). If a new request is presented while da is not yet activated, then CU does not validate a new data transfer which is left pending. CU controls bidirectional data flow.

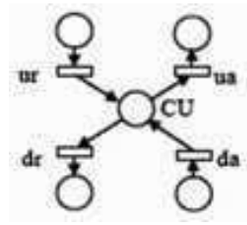


Fig. 24. Control Unit representation

4.2 Example of the Matrix Product

Once these operators have been defined, they can now be used in the Petri Net description of a systolic array, as it is developped in the following example. Be *C* = *A*.*B* a processing to perform, with *A*, *B* and *C* squared matrixes of the same size (n=2 to simplify). Processings to perform are :

$$c_{i,j}=\mathrm{Sum}(a_{i,k}.b_{k,j})_{k=1...2} \quad (10)$$

which require eight operators for multiplication and to propagate a_{ik}, b_{kj} and c_{ij} (Fig. 25).

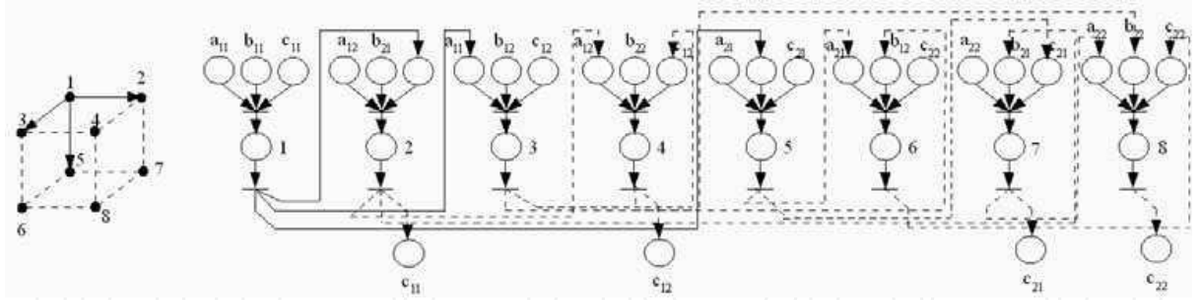


Fig. 25. First step of data propagation

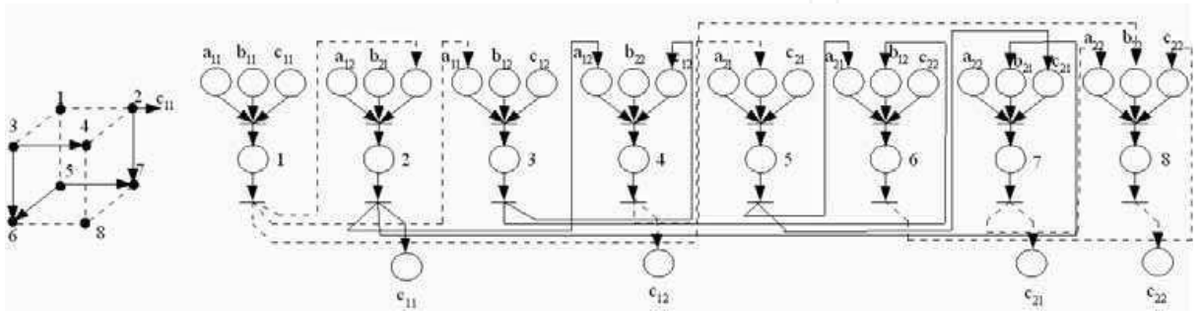


Fig. 26. Second step of data propagation

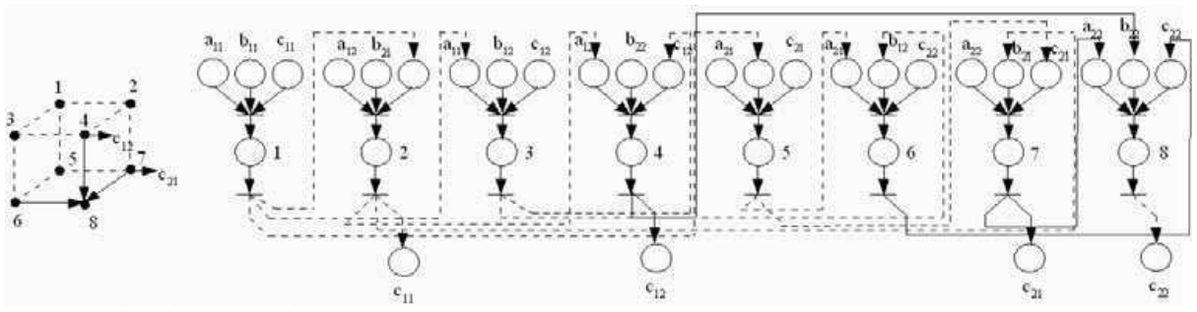


Fig. 27. Third step of data propagation

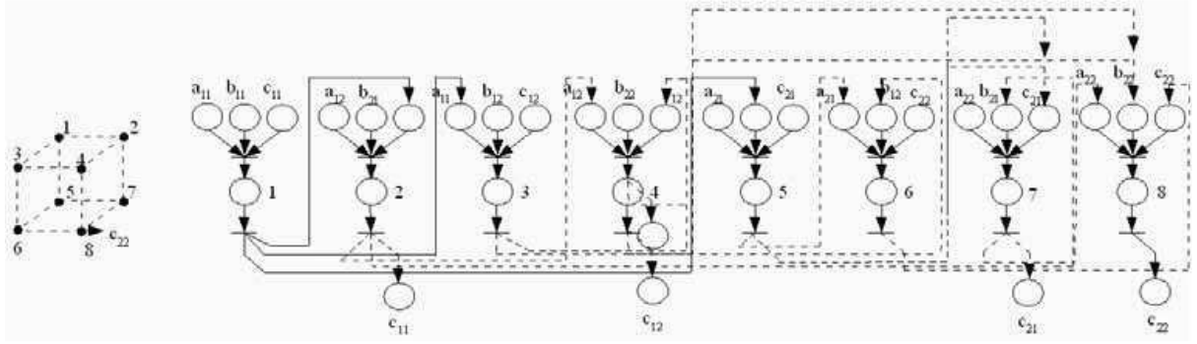


Fig. 28. Fourth step of data propagation

In the first step (Fig. 25), operator 1 receives a_{11}, b_{11} and c_{11}. It performs $c_{11}=a_{11}.b_{11}$ and propagates the three data to operators 3, 5 and 2. In the second step (Fig. 26), operator 2 receives a_{12} et b_{21}, operator 3 receives b_{12} and c_{12} and operator 5 receives a_{21} and c_{21}. Operator 2 performs : $c_{11} = a_{11}.b_{11} + a_{12}.b_{21}$. Operator 3 performs $a_{11}.b_{12}$ and operator 5 processes

$a_{21}.b_{11}$. These operators are respectively connected to operators 4 and 7 on the one hand, 6 and 7 on the other hand.
In the third step (Fig. 27), operator 4 receives b_{22}, operator 6 receives c_{22} and operator 7 receives a_{22}. These 3 operators are linked to operator 8. They perform : $c_{12} = a_{11}.b_{12} + a_{12}.b_{22}$ and $c_{21} = a_{21}.b_{11} + a_{22}.b_{21}$. In the final step (Fig. 28), operator 8 performs $c_{22} = a_{21}.b_{12} + a_{22}.b_{22}$.
By propagating data in the 3 directions, the processing domain becomes totally defined :

$$D = \{(i,j,k) \mid 1 \le i \le N, 1 \le j \le N, 1 \le k \le N\}$$

Classic projections are :
– Π = (1,1,0) or (1,0,1) or (0,1,1) which results in the linear network in Fig. 1a.
– Π = (0,0,1) or (0,1,0) or (1,0,0) which results in the squared network in Fig. 1b.
– Π = (1,1,1) which results in the hexagonal network in Fig. 1c.
For example, with the first solution, the result is as in Fig. 1. Each cell is made of a multiplier/adder with accumulation (Fig. 29).

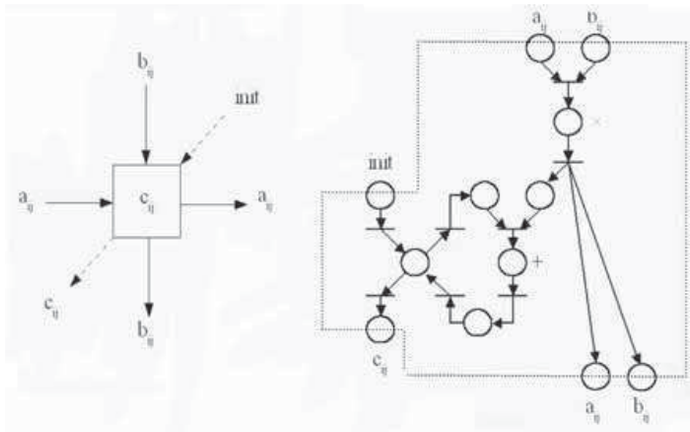
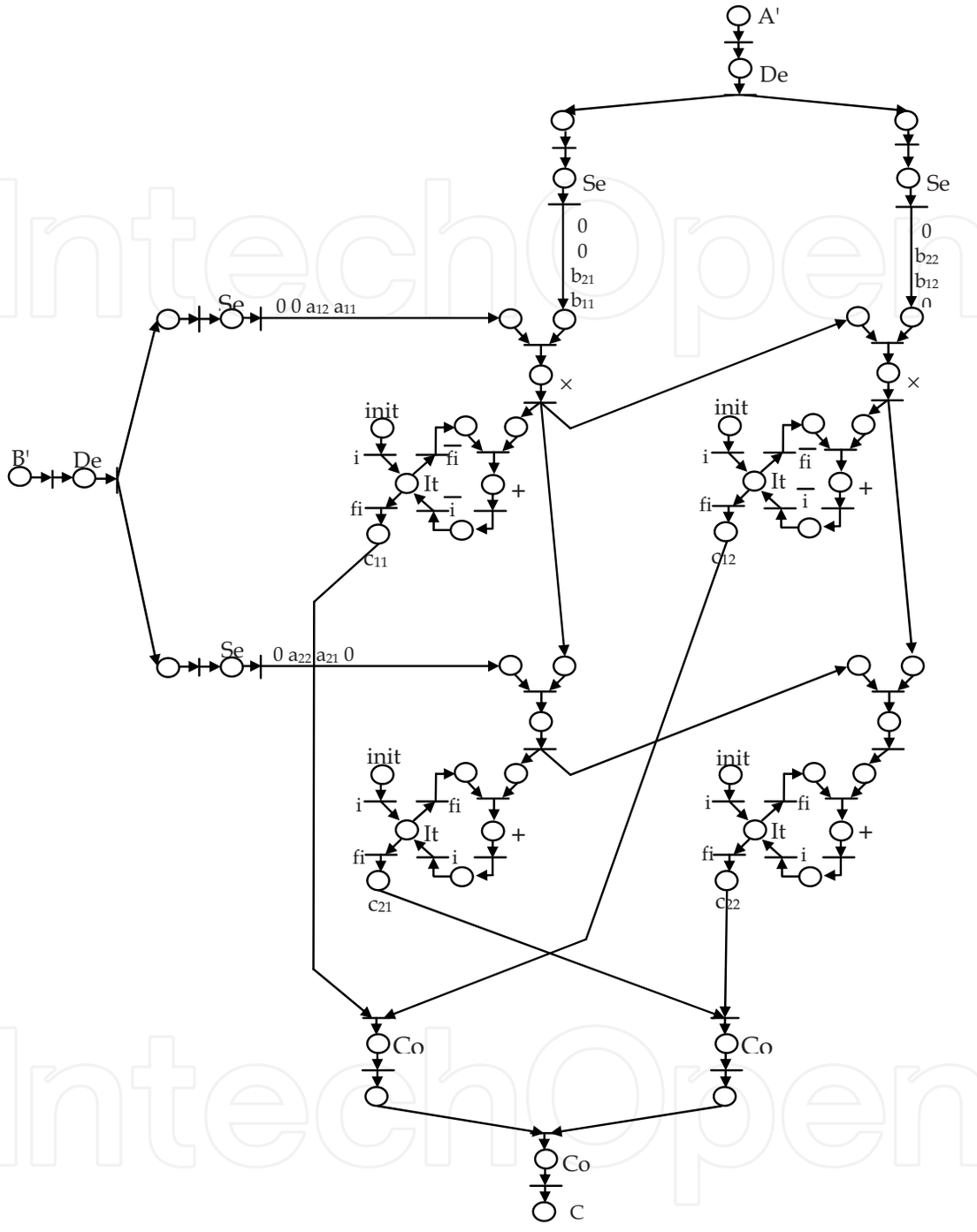


Fig. 29. Squared network of matrix product *C=A.B*

The Architectural Petri Net defining the complete systolic network is obtained by adding Decompose and Compose operators in input and output so as to perform the interface with the environment (Fig. 30). In order to be free from the related hardware problems that can occur to retrieve results in the cells, the hexagonal structure can also be used. In this type of network, a, b and c circulate in 3 directions (Fig. 31). For instance, with a 3×3 matrix product, the network operating cycle is as following :
1 - Network is reset. a_{11}, b_{11} and c_{11} come in input respectively of operators o_5, o_9 and o_1.
2 - a_{11}, b_{11} and c_{11} are propagated to o_{15}, o_{17} and o_{13}.
3 - a_{11}, b_{11} and c_{11} come as input of o_{19} in which $c_{11} = a_{11}.b_{11}$ is done. a_{12}, a_{21}, b_{12}, b_{21}, c_{12} and c_{21} come in input respectively of operators o_4, o_6, o_8, o_{10}, o_2 and o_{12}.
4 - c_{11}, a_{12} and b_{21} come as input of o_6 at the same time. $c_{11} = a_{11}.b_{11}+a_{12}.b_{21}$ is done. Other data are propagated.
5 - c_{11}, a_{13} and b_{31} come as input of o_7 at the same time. $c_{11}= a_{11}.b_{11}+a_{12}.b_{21}+a_{13}.b_{31}$ Other data are propagated.
Processings are done similarly for other terms until the matrix product has been completed.



$$A=\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \rightarrow A'=\begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ 0 & a_{21} & a_{22} & 0 \end{pmatrix} \quad B=\begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \rightarrow B'=\begin{pmatrix} b_{11} & 0 \\ b_{21} & b_{12} \\ 0 & b_{22} \\ 0 & 0 \end{pmatrix} \quad C=\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

Fig. 30. Petri Net of the systolic network for the matrix product

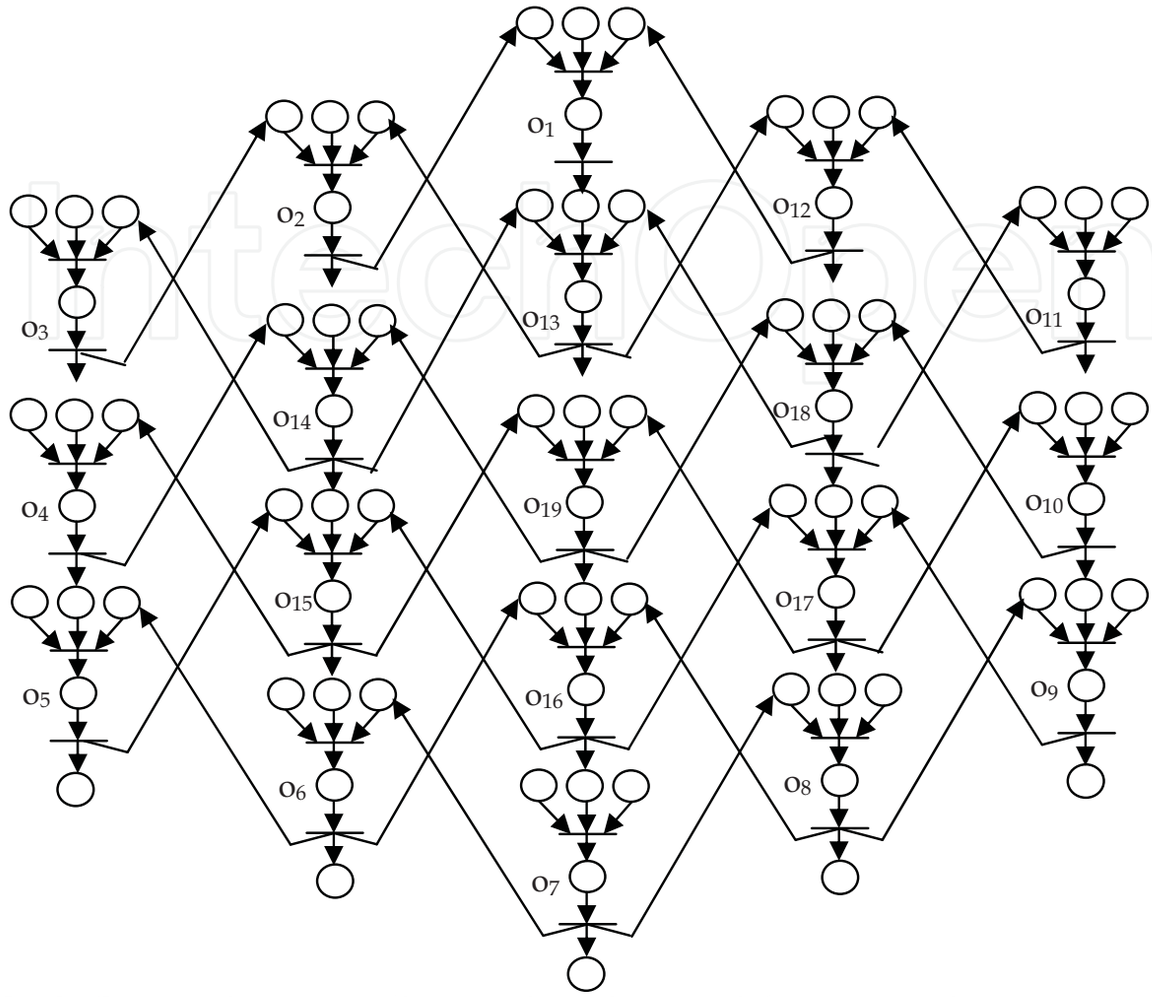


Fig. 31. Petri Net description of hexagonal systolic network for matrix product

5. Conclusion

The main characteristics of currently available integrated circuits give the possibility to make massively parallel systems, as long as the processings « volume » are given priority to data transfer. Systolic model is a powerful tool for conceiving specialized networks, using identical elementary cells locally interconnected. Each cell receives data coming from neighbourhing cells, performs a simple processing, then transmits the results to neighbourhing cells after a time cycle. Only cells on the network frontier communicate with the environment. Their conception is often based on methods using recurrent equations, or on sequential algorithms or fluency graphs. It can be efficiently developped thanks to a tool completely formalized, lying on a strong mathematical basis, i.e. Petri Nets, and their Architectural extension. Moreover, this model enables to do their synthesis and to ease their implementation on reprogrammable components.

6. References

Abellard, A., (2005). Architectural Petri Nets : Basics concepts, methodology and examples of application, *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, pp. 2037-2042, Waikoloa, HI, USA, October 2005, IEEE.

Abellard, A.; Abellard, P. & Gorce, P. (2007). Architectural Petri Nets : Basics concepts, methodology and examples of application, *Proceedings of IEEE/ASME AIM International Conference on Advanced Intelligent Mechatronics*, Zurich, Switzerland, September 2007, IEEE.

Abellard, A. & Abellard, P. (2008). A Design Methodology of Systolic Architectures Based on a Petri Net Extension. Application to a Stereovision Hardware/Software Processing Improvement, *Proceedings of ICSEA - International Conference on Software Engineering Advances*, pp. 77-82, Sliema, Malta, October 2008, IARIA.

Almhana, J. (1983). *Modélisation par réseaux de Petri à flux de données. Application à la synthèse de l'opérateur de Riccati rapide*. PhD Thesis, Université d'Aix-Marseille III, France.

Barreto, R. ; Maciel, P. ; Tavares, E. ; Oliveira, M. & Lima R. (2008), A time Petri Net-based method for embedded hard real-time software synthesis, *Design Automation for Embedded Systems*, Vol. 12, pp. 31-62, ISSN 0929-5585 (Print) 1572-8080 (Online), Springer.

Blume, H. ; von Sydow, T. & Noll, T.G. (2006), A case study for the application of deterministic and stochastic Petri Nets in the SoC communication domain, *Journal of VLSI Signal Processing*, Vol. 43, pp. 223-233, ISSN 0922-5773, Springer.

Cortés, L.A. ; Eles, P. & Peng, Z. (2003), Modeling and formal verification of embedded systems based on a Petri net representation, *Journal of Systems Architecture*, Vol. 49, pp. 571–598, ISSN 1383-7621, Elsevier.

Eles, P. ; Kuchcinski, K. & Peng, Z. (1996), Synthesis of systems specified as interacting VHDL processes, *Integration-The VLSI Journal*, Vol. 21, No. 1-2, pp. 113-138, ISSN 0167-9260, Elsevier.

Gomes, L. ; Barros, J.P. & Costa, A. (2005), Structuring Mechanisms in Petri Net Models: From specification to FPGA based implementations. In: Adamski, M. ; Karatkevich, A. & Wegrzyn, M. (Eds.), *Design of embedded control systems*, pp. 153-166, ISBN 978-0-387-23630-8, Springer.

Ghavami, B. & Pedram H. (2009), High performance asynchronous design flow using a novel static performance analysis method, *Computers and Electrical Engineering*, in press, Elsevier.

Hsiung, P.A. & Gau, C.H. (2002), Formal synthesis of real-time embedded software by time-memory scheduling of colored time Petri Nets, *Electronic Notes in Theoretical Computer Science*, Vol. 65, No. 6, pp. 140-159, Elsevier.

Hsiung, P.A. ; Lin, C.Y. & Lee, T.Y. (2004), Quasi-dynamic scheduling for the synthesis of real-time embedded software with local and global deadlines, *Lecture Notes in Computer Science*, Vol. 2968, pp. 229–243, ISBN 3-540-21974-9, Springer-Verlag.

Huang, C.C. & Liang W.Y. (2003), Object-oriented development of the embedded system based on Petri-nets, *Computer Standards & Interfaces*, Vol. 26, pp. 187–203, Elsevier.

Maciel, P. ; Barros, E. & Rosenstiel, W. (1999), A Petri Net model for hardware/software codesign, *Design Automation for Embedded Systems*, Vol. 4, No. 4, pp. 243-310, Springer.

Oliveira, M. ; Maciel, P. ; Barreto, S. & Carvalho, F. (2004), Towards a software power cost analysis framework using colored Petri Net, *Lecture Notes in Computer Science*, pp. 362–371, ISBN 3-540-23095-5, Springer-Verlag.

Sgroi, M. ; Lavagno, L. ; Watanabe, Y. & Sangiovanni-Vincentelli, A. (1999) Synthesis of embedded software Using free-choice Petri Nets, *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pp. 805-810, ISBN 1-58133-109-7, New Orleans, LA, USA, June 1999, IEEE.

Strbac, P. ; Tuba, M. & Simian, D. (2009) Hierarchical model of a systolic array for solving differential equations implemented as an upgraded Petri Net, *WSEAS Transactions on Systems*, Vol. 8, No. 1, pp. 12-21, WSEAS.

Capello, P.R. & Steiglitz, K. (1983). Unifying VLSI array designs with geometric transformations, *Proceedings of International Conference on Parallel Processing*, pp. 448-457, Bellaire, USA.

Castro-Pareja, C.R ; Sagadeesh, J.M. : Venugopal, R. & Shekha, R. (2004), FPGA based 3D median filtering using word-parallel systolic arrays, *Proceedings of IEEE ISCAS International Symposium on Circuits and Systems*, Vol. 3, pp. 157-160, Vancouver, Canada, May 2004, IEEE.

Gannon, D. (1982). Pipelining array computation for MIMD parallelism. *Proceedings of International Conference on Parallel Processing*, Columbus, OH, USA, 1982.

Jackson, P.A. ; Chan, C.P. ; Scalera, J.E. ; Rader, C.M. & Vai, M.M. (2004). A Systolic FFT Architecture for Real Time FPGA Systems, *Proceedings of HPEC - Eighth Annual Workshop on High Performance Embedded Computing,* Lexington, MA, USA, 2004.

Johnson, K.T. & Hurson, A.R. (1993). General purpose systolic arrays. *Computer*, Vol.26, No.1, pp. 20-31.

Kung, H.T. (1982). Why systolic architectures ? *Computer*, Vol.15, pp.37-46.

Kung, H.T. (1988), Systolic communications, *Proceedings of International Symposium on Computer Architectures,* San Diego, CA, USA, 1988, IEEE.

Kung, S.Y. (1984), On supercomputing with systolic/wavefront array processors, *Proceedings of the IEEE,* Vol.72, pp. 867-884.

Lee, J.J. & Song, G.Y (2002), Implementation of the Systolic Array for Dynamic Programming, *Proceedings of ICITA International Conference on Information Technology and Applications,* ISBN 1-86467-114-9, Bathurst, Australia, 2002, IEEE.

Lee, J.J. & Song, G.Y (2003), Implementation of the super systolic array for convolution, *Proceedings of ASP-DAC Asia and South Pacific Design Automation Conference,* pp. 491-494, ISBN 0-7803-7659-5, Kitakyushu, Japan, January 2003, IEEE.

Lee, J.J. & Song, G.Y (2004), Implementation of a bit-level super systolic FIR filter, *Proceedings of IEEE AP-ASIC - Asia Pacific Conference on Advanced Systems Integrated Circuit,* pp. 206-209, ISBN 0-7803-8637-X, Fukuoka, Japan, August 2004, IEEE.

Leiserson, C.E. & Saxe, J.B (1983), Optimizing synchronous circuitry by retiming, *Proceedings of 3D CalTech Conference on VLSI,* pp. 87-116, 1983.

Li, G.J. & Wah, B.W (1984), The design of optimal systolic arrays, *IEEE Transactions on Computers,* Vol.33, No.10, 1984.

Lim, H. & Swartzlander, E.E (1996a), Multidimensional systolic arrays for multidimensional DFTs, *Proceedings of the IEEE International Conference on Acoustic, Speech and Signal Processing,* Vol.6, pp. 3276-3279, Atlanta, GA, USA, May 1996, IEEE.

Lim, H. & Swartzlander, E.E (1996b), Efficient systolic arrays for FFT algorithms, *Proceedings of the 29th Asilomar Conf. Signals, Systems and Computers,* Vol.1, pp. 141-145, ISBN 0-8186-7646-9, Pacific Grove, CA, USA, 1996, IEEE.

Lim, H. & Swartzlander, E.E (1999), Multidimensional systolic arrays for the implementation of Discrete Fourier Transform, *IEEE Transactions on Signal Processing,* Vol.47, No.5, pp. 1359-1370.

Mihu, I.Z. ; Brad, R. & Breazu, M. (2001), Specifications and FPGA implementation of a systolic Hopfield-type associative memory, *Proceedings of the International Conference on Neural Networks,* Vol.1, pp. 228-233, Washington, DC, USA, 2001.

Moldovan, D.I. (1982), On the analysis of synthesis VLSI algorithms, *IEEE Transactions on Computer,* Vol.31, No.11, pp. 1121-1126.

Mongenet, C. (1985). *Une méthode de conception d'algorithmes systoliques, résultats théoriques et réalisation*. PhD Thesis, Université de Nancy, France.

Nash, J.G. (2002), Automatic latency optimal design of FPGA based systolic arrays, *Proceedings of the 10th IEEE Symposium on Field Programmable Custom Computing Machines*, pp. 299-300, Napa, CA, USA.

Nash, J.G. (2005), Computationally Efficient Systolic Architecture for Computing the Discrete Fourier Transform, *IEEE Transactions on Signal Processing*, pp. 4640-4651, Vol. 53, No. 12, ISSN 1053587X.

Quinton, P. (1983). The systematic design of systolic arrays, *Report IRISA 193*, Rennes, France.

Quinton, P. & Robert, Y. (1991). *Systolic algorithms & architectures*, Ed. Prentice Hall, ISBN 0138807906, London.

Sousa, L.A. (1998), Bidirectional systolic arrays for digital recursive filters, *Proceedings of the International Conference on Electronics, Circuits and Systems*, Vol.3, pp. 451-502, ISBN 0-7803-5008-1, Lisboa, Portugal, September 1998, IEEE.

Yang, Y. ; Zhao, W. & Inoue, Y. (2005), High performance systolic arrays for band matrix multiplication, *Proceedings of the IEEE International Symposium on Circuits and Systems*, Vol.2, pp. 1130-1133, ISBN 0-7803-8834-8, Kobe, Japan, May 2005, IEEE.