Automatic Design of Systolic Arrays

Laura Szakacs
Babes-Bolyai University Cluj
laura@cs.ubbcluj.ro

lszakacs@risc.uni-linz.ac.at

December, 2002

Abstract

The first examples of systolic designs that appeared in the early eighties were conceived in an "ad hoc" manner, requiring a great amount of creativity and inspiration of its inventors. Then systematic design methodologies have been proposed to overcome the limitations of heuristics and designer intuition and to provide a formal framework for mapping algorithms to systolic architectures. The paper provides a survey of the available systolic array design methods outlining their advantages and drawbacks.

Three different approaches to the automatic systolic array design are discussed in detail, with representative examples and useful observations added.

The long-term purpose of this research is to automate the generation and verification of systolic designs with the help of automated theorem proving systems combined with computer algebra systems (in particular Theorema, see www.theorema.org).

Contents

1 Introduction

The first examples of systolic designs that appeared in the early eighties were conceived in an "ad hoc" manner, requiring enormous amount of creativity and inspiration of its inventors. Then systematic design methodologies have been proposed to overcome the limitations of heuristics and designer intuition and to provide a formal framework for mapping algorithms to systolic architectures.

1.1 Design Methodologies

Among the first systematic design methodologies two seemingly distinct methods, called **data dependency method** and **parameter method** ware proposed.

The essence of the *data dependency method* is the representation of the dependency structure of an algorithm in concise, matrix form. The technique for mapping algorithms into systolic arrays is based on the mathematical transformations of index sets and the so called *data dependence vectors*.
This method has been extensively studied by Fortes and Moldovan [Mol83], [MF86], S.Y. Kung [Kun87] and others. The procedure based on this method and presented by Moldovan in [Mol83] was also implemented at the University of Southern California in a software package called ADVIS (automatic design of VLSI[1] systems).

A procedure for the derivation of optimal systolic design was first developed in the *parameter method* [LW85]. This methodology presents the optimal mapping of algorithms that are represented as *linear recurrences* onto systolic arrays. Three sets of parameters are used to characterize systolic arrays:

- velocities of data flow
- data distribution
- periods of computation

The relationship between these parameters are represented as constraint equations that control the correctness of the design, thus the design is formulated in an optimization problem.

The two design methodologies, the parameter method and the data dependency method are compared by O'Keefe, Fortes and Wah [OFW91], describing the relationship between them. It is shown that the parameter method applies to a subclass of the algorithms that can be processed by the dependency method. Explicit mathematical relations are established between the parameters, equations and constraints of the two important methods for the design of systolic arrays.

The mentioned works, same as others like that of Quinton, Robert, Van Dongen [Qui84], [Qui87], [QR90], [QD89], Delosme and Ipsen [DI86], Nelis

[1] Very Large Scale of Integration

and Deprettere [ND88], Huang and Lengauer [HL87] contributed to the foundation respectively the evolution of a unifying approach to the design of VLSI algorithms usually referred to as the **space-time transformation methodology**. In a certain way, all the proposed methods by the above mentioned researchers vary very little, differing slightly in the degree of formalism and the way to approach the problem. In the ultimate instance, they all use the concept of transformation of dependencies. A review of the main ideas involved in these systolic algorithm synthesis methods is presented by Song in [Son94].

The task is the transformation of a sequential algorithm, expressed in the form of up to three nested loops, or in the form of *uniform recurrence equations*, into a systolic algorithm applying a set of mathematical transformations on the *dependence vectors*.
The computations involved in an algorithm, as well as the dependences between computations, are transformed by a *time function* and a *space allocation function*.

- The *time function* maps each computation of the algorithm to a positive integer that represents the time at which the computation is executed.

- The *allocation function*, on the other hand, obtains the position of the processing element to perform the computation involved.

Such a space-time mapping must satisfy several constraints to be valid. Basically two conditions has to hold:

- if a statement depends on the result of another statement, it should be executed after the result itself is available,

- two statements allocated to the same processor should not be scheduled at the same instant of time

If we restrict ourselves to the case when the transformations applied are linear functions, it appears that the above conditions can be equivalently expressed as a linear programming problem, when one considers uniform recurrences. Linear transformations have proved to be very effective in order to design locally connected architectures such as systolic arrays. These are the main reasons why research in this domain has mainly focused on uniform recurrences, and linear transformations.
A more detailed presentation of the space-time transformation method is provided in **section 2**.

In most cases, the initial specification is expressed as a recurrence equation. However, the majority of the above mentioned works have worked with uniform recurrence equations (URE), a subclass of linear recurrence equations (LRE).

The *uniformization problem*, that is to say the problem of transforming linear recurrence equations into uniform recurrence equations is tackled by Quinton and Dongen in [QD89], Fortes and Moldovan [FM84] and others.

Such a transformation is mandatory when the target architecture is locally connected, as for example, a systolic array.
A more general review about the various transformations that can be applied to a system of recurrence equations is presented by Lavenier, Quinton and Rajopadhye in [LQR99].

The main formal manipulation techniques of SREs are summarized and exemplified:

- Serialization of Reductions
- Alignment
- Localization
- Change of basis

Many algorithms already were parallelized using the efficient technique of time-space transformations. However, this methodology also has some drawbacks. As space-time allocation is an operational notion, it could be not adequate to base a method on this notion. Early introduction of operational notions often leads to exclude possible solutions of a problem. Time and space boundary conditions often constitute another problem in the design of systolic arrays.
Various attempts have been made to overcome the drawbacks of the space-time transformation method. A remarkable alternate approach represents the methods based on viewing systolic design as program design.

Gribomont and Dongen presents another methodology for the mapping of systolic arrays based on the concept of **generic systolic array** introduced and illustrated in [GD92].
This approach represents a program-oriented methodology, being more closed to the program design. The technique takes in account two points, namely that:

- the architecture is often chosen before the real beginning of the development
- the basic operations to be executed by individual cells are partially known at the beginning

Thus the development starts from a *generic systolic array*, whose parameters have to be instantiated. A more detailed description of the method is given in **section 3**.

The class of problems that can be expressed in terms of URE is restricted and expressing most problems using URE is quite tedious. The time-space transformation method depends heavily on finding an affine timing function. The problem with finding affine timing function is that one needs to solve a

system of linear recurrence equations. The method proposed by Kazerouni, Rajan and Shyamasundar in [KRS95], [KRS96] is not restricted to affine timing functions.
They propose a general method for mapping a system of linear recurrence equations (SLRE) onto specific systolic architectures. It generates solutions suitable for existing target architectures rather than designing new ones.

The method essentially consist of mapping *normalized linear recurrence equations*, a subclass of linear recurrence equations, - which properly includes the class of uniform recurrence equations - onto a generic architecture called *basic systolic architecture* and then applying correctness preserving transformations to adopt this intermediate solution onto specific target architectures (also presented in [KRS94]). **Section 4** provides a more detailed presentation of a part of the method (applicable to URE systems) and it is related to the space-time transformation methodology.

1.2 Partitioning

The synthesis method for systolic arrays based upon space and time transformations can be extended to generate systolic implementations on a fixed number of processors. The main idea of all these extensions is to merge many cells into a single processor, so as to compress the array. This step is called *partitioning*. Algorithm partitioning is essential when the size of the computational problem is larger than the size of the systolic array intended for that problem.
It can take two different forms:

- The locally parallel globally sequential (**LPGS**) form, presented by Moldovan in [MF86].
- The locally sequential globally parallel (**LSGP**) form, studied by Darte, Delosme [DD90], [Dar91], Bu, Deprettere and Dewilde [BDD90].

The LPGS approach is to partition the array into blocks whose size is the number of available processors (p), and to compute each block one after the other. The different computation points in the current block are allocated and scheduled in the p processors, in accordance to the dependence constraints. This method requires small local memories, but a large external buffer is needed to store the data used by the next block.
In the LSGP approach the array obtained by the mapping procedure is partitioned into p blocks of virtual processors, each block being allocated to one physical processor.

Of course, the different points allocated to the same processor have to be computed at different times in the array in such a way that they can be sequentially executed by the physical processor. This method permits to synthesize systolic arrays with a fixed number of cells and, as a particular case, permits to

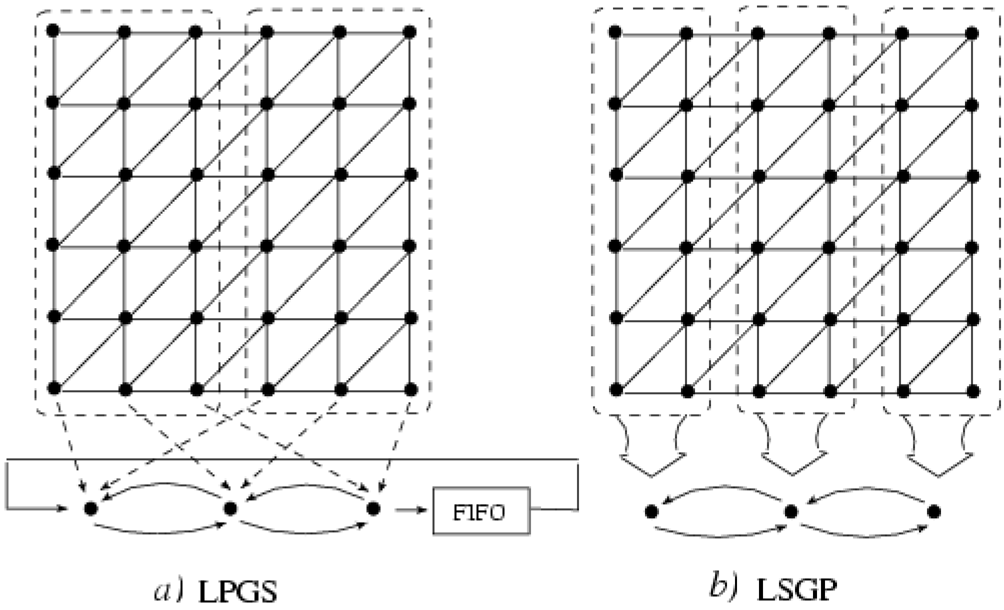

Figure 1: LPGS and LSGP method

improve the efficiency of the cells in a systolic array obtained by the projection method.
The methods are illustrated in Figure 1.

a) In the LPGS scheme, the block size is chosen to match the array size, i.e. one block can be mapped to one array. All nodes within one block are processed concurrently, i.e. locally parallel. One block after another block of node data is loaded into the array and processed in a sequential manner, i.e. globally sequential.

b) In the LSGP scheme, one block is mapped to one processing element (PE). Each PE sequentially processes the nodes of the corresponding block. The number of blocks is equal to the number of PEs in the array.

In the LPGS method, a general rule is to select a (global) scheduling that does not violate the data dependencies.
As to the scheduling scheme for the LSGP, after processor allocation, an acceptable (i.e., suficiently slow) schedule is chosen so that at any instant there is at most one active PE in each block.
Note that the LPGS design has the advantage that blocks can be executed one after another in a natural order. However, this simple ordering is valid only when there is no reverse data dependence for the chosen blocks.

A unified partitioning and scheduling scheme is proposed for LPGS and LSGP in [HH95]. The main contribution includes a unified partitioning model and a systematic two level scheduling scheme. The unified partitioning model can support LPGS and LSGP design in the same manner. The systematic two level scheduling scheme can specify the intra-processor schedule and interprocessor schedule independently. Thus, more interprocessor parallelism can be effectively explored. The design methodology presented in [CK98] also proposes such a unified approach to the partitioning problem.

2 The space-time transformation methodology

The **space-time transformation methodology** is a unifying approach to the automatic design of systolic algorithms based on the concept of transformation of dependencies.

2.1 Definitions

A common way of describing an algorithm is that of using abstract specifications such as *recurrence equations.*

Let D be the set of points of integer coordinates belonging to a convex polyhedral domain of Euclidean n-space R^n

$$D_n = \left\{ z \in Z^n | \mathcal{B}z \leq b \right\}$$

where $\mathcal{B}$ is a m x n matrix and b an m x 1 vector over Z. In the following we consider limited convex polyhedral domains.

Also note that a *vertex* z of a polyhedron D is any point $z \in D$ which cannot be expressed as a convex combination of any other distinct points in D.
$(\forall z_0, z_1, \ldots, z_{q-1} \in D, \nexists \mu_0, \mu_1, \ldots, \mu_{q-1} \in R$ so that $\mu_i \geq 0,$
$i \in \left\{0, 1, \ldots, q-1\right\}, \sum_{i=0}^{q-1} \mu_i = 1$ and $z = \sum_{i=0}^{q-1} \mu_i z_i)$

Definition 2.1. (**System of Linear Recurrence Equations**)
A *system of linear recurrence equations* (SLRE) is a collection of $s \in N$ equations of the following form:

$$V_i(z) = f_i(V_1(\delta_{1_i}(z)), \ldots, V_s(\delta_{s_i}(z))), \quad 0 \leq i < s$$

where

1. $V_i : D_{n_i} \rightarrow R$. $V_i, i \in \left\{0, \ldots s\right\}$ are variable names belonging to a finite set V. Each variable is indexed with an integral index, whose dimension (called the *index dimension* of the variable) is constant for a given variable.

2. $z \in D_{n_i}$

3. δ_{j_i} is an affine function from D_{n_i} to D_{n_j} (also called *affine mapping function*) of the form
$\delta_{j_i}(z) = A_{j_i} z + b_{j_i}$, where
A_{j_i} is a constant n_j x n_i matrix over Z
b_{j_i} is a constant n_j x 1 matrix (vector) over Z

4. $V_i(z)$ does not appear on the r.h.s. of the equation

5. $f_i : R^s \rightarrow R$

Definition 2.2. (**Normal Linear Recurrence Equation**)
A linear recurrence equations (LRE) is called *normal* iff all the transformations δ_{j_i} are from D_n to D_n for a fixed n.

Note: Such a constraint guarantees that all the nodes in the dependency graph (see Definition 2.5) have the same out-degree and further induces a coarse regularity on the connection pattern in the systolic implementation. In the case of normal recurrence equations $\forall j, i$, the matrix A_{j_i} is square matrix of order n.

Definition 2.3. (**Uniform Recurrence Equation**)
A linear recurrence equation is said to be *uniform* (URE) iff it is normal and the transformations δ_{j_i} preserve differences, i.e $z - z' = \delta_{j_i}(z) - \delta_{j_i}(z')$

Note: In this case not only will all the nodes in the dependency graph have the same out-degree, but also the connection pattern induced by the URE's on the systolic implementation will be regular. In the case of uniform recurrences the matrix A_{j_i} is the identity matrix.

A **System of Uniform Recurrence Equations** (SURE) consists of equations of the following form:

$$V_i(z) = f_i(V_1(z - \theta_{i_1}), \dots, V_k(z - \theta_{i_k}(z))), \quad 0 \leq i < s$$

where $z \in D$, $\theta_{i_1}, \dots, \theta_{i_k}$ are vectors of Z^n.

Unless it is uniform, a system of recurrence equations cannot be mapped directly on a systolic array. Indeed, as the dependence vectors are not constants, processors may have to communicate with an arbitrary large number of other processors, and this is not possible in a systolic array, where the processors are only connected to their neighborhood.

For that reason the problem of uniformization, that is to say, the transformation of linear recurrences into uniform ones was studied in many works ([QD89] and others).

Definition 2.4. (**Dependence vectors**)
Let $\Theta = \{\theta_0, \theta_1, \dots, \theta_l\}$ the set of vectors θ_i of a SURE. The θ_i are called *dependence vectors* of a system.

Consider points $z, z' \in D$. We say that z is dependent on z' by θ_i, if $\exists\ \theta_i \in \Theta$, such that

$$z = z' + \theta_i.$$

The dependences can be represented by a directed graph called *dependence graph* of the SURE. It abstracts the dependency relations among the variables in the SURE.

Definition 2.5. (**Dependence graph**)
The dependence graph of a SURE is a directed graph $G(V, E)$, (also denoted by (D, Θ)), where
the set of nodes $V = D$, i.e. the nodes are points of D, and
the set of edges $E = \left\{\varepsilon_{zz'} | \exists \theta_i \in \Theta, z = z' + \theta_i\right\}$, i.e. if z is dependent on z' by θ_i, there exists an edge from node z' to node z.

2.2 Time function

Given a SURE, we want to obtain a time function τ that schedules the computations associated to the points of domain D. For a given $z \in D$ we assume that the computations $V_i(z)$ are performed in parallel, and take a unit time.

A natural requirement is that in order to perform the computations of $V_i(z)$, its arguments should have been computed before. The time function τ associates to each point $z \in D$ the time instant it is computed. If such a function exists, then we say that the SURE is *computable*.

Definition 2.6. (**Timing function**)

Let $x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix} \in D.$

Let $\gamma \in Z$, and $\lambda = \begin{pmatrix} \lambda_0 \\ \lambda_1 \\ \vdots \\ \lambda_{n-1} \end{pmatrix}$, $\lambda_i \in Z$, $i = \overline{0, n-1}$.

The *time function* is the following linear function τ, $(\tau : D \to N)$:

$$\begin{aligned} \tau(x) &= \lambda_0 x_0 + \lambda_1 x_1 + \ldots \lambda_{n-1} x_{n-1} + \gamma \\ &= \lambda^t x + \gamma \end{aligned}$$

that satisfies the two conditions:

1. $\tau(x) \geq 0, \forall x \in D.$
2. If z depends on z', then $\tau(z) > \tau(z')$

The first condition is a matter of convenience, since τ is interpreted as time. The second condition makes scheduling of dependent computations possible.

Theorem 2.1. (Quinton and Roberts [QR90])
For a given SURE, with dependence graph $(D; \Theta)$ and set V_D of **vertices** of D, the parameters $\lambda_0, \lambda_1, \ldots, \lambda_{n-1}$ and γ define a timing function τ iff

1. $\forall v_i \in V_D, \lambda^t v_i + \gamma \geq 0$ and
2. $\forall \theta_i \in \Theta, \lambda^t \theta_i > 0$, or $\lambda^t \theta_i \geq 1$.

Proof:
Condition 1 is equivalent to having τ non negative over D, or

$$\forall v_i \in V_D, \lambda^t v_i + \gamma \geq 0 \text{ is equivalent to } \forall z \in D, \tau(z) \geq 0.$$

In fact, if τ is always non negative over D, in particular $\tau(v_i) \geq 0$, where v_i are vertices of D. Therefore,

$$\tau(v_i) = \lambda^t v_i + \gamma \geq 0.$$

On the other hand, $\lambda^T v_i + \gamma \geq 0$, $\forall v_i \in V_D$, let us show that $\tau(z) \geq 0$, $\forall z \in D$. Any point $z \in D$ can be expressed as a convex combination of vertices of D. So,

$$z = \sum_{i=0}^{\nu-1} \mu_i v_i, \;\; \mu_i \geq 0, \sum_{i=0}^{\nu-1} \mu_i = 1.$$

We have then

$$\tau(z) = \lambda^t z + \gamma = \sum_{i=0}^{\nu-1} \mu_i \lambda^t v_i + \gamma = \sum_{i=0}^{\nu-1} \mu_i (\lambda^t v_i + \gamma) \geq 0.$$

Condition 2 is equivalent to requiring $\tau(z) > \tau(z')$ if z depends on z'. In fact, if z depends on z', then

$$\exists \theta_i \in \Theta, \;\; z - z' = \theta_i.$$

We can write

$$\tau(z) - \tau(z') = \lambda^t z + \gamma - \lambda^t z' - \gamma = \lambda^t \theta_i.$$

Thus,

$$\tau(z) > \tau(z') \text{ is equivalent to } \lambda^t \theta_i > 0.$$

qed.

Note that if z is dependent on z' by θ_i, or $z = z' + \theta_i$, then

$$\lambda^t \theta_i = \lambda^t (z - z') = \tau(z) - \tau(z').$$

Thus $\lambda^t \theta_i$ expresses the *delay* between the computation of z and the computation of z'. Theorem 2.1 requires this delay to be greather than 0 (at least 1).

Theorem 2.1. gives conditions for the obtention of a time function. Another question is that of optimality of time functions. Several criteria for optimality can be adopted. Consider a SURE with dependence graph $(D; \Theta)$. One criterium is to minimize the delay between dependent computations according to a dependence $\theta_k \in \Theta$. Thus we want to minimize $\lambda^t \theta_k$, subject to

$$\lambda^t \theta_i \geq 1, \forall \theta_i \in \Theta.$$

2.3 Allocation function

The *allocation function* $\alpha : Z^n \rightarrow Z^{n-1}$ maps to each point of D a position $\alpha(z)$ where the computation associated to that point is performed. Point $\alpha(z)$ can be interpreted as the position of a processing element of a certain computing system.
We assume that each processing element is capable of realizing the computation associated to a point of D in a unit time.
The allocation function α must not map to the same processing element different points of D if the associated computations are executed at the same time.

Definition 2.7. (**Allocation function**)
The allocation function $\alpha : Z^n \rightarrow Z^{n-1}$ is a function of the form

$$\alpha(x) = (\alpha_0(x), \alpha_1(x), \ldots \alpha_{n-2}(x))$$

where each α_i is a linear function $\alpha_i : Z^n \rightarrow Z$, such that

$$\forall x, y \in D, x \neq y \text{ and } \alpha(x) = \alpha(y) \Rightarrow \tau(x) \neq \tau(y).$$

Theorem 2.2. (Quinton and Roberts [QR90])
Let u be a non-null vector of Z^n such that $\lambda^t u \neq 0$.
Let $u_j \neq 0$ be a component of u. The following application defines an allocation function
$\alpha(x) = (\alpha_0(x), \alpha_1(x), \ldots, \alpha_{n-2}(x))$, where

$$\begin{array}{rcll} \alpha_k(x) & = & u_j x_k - u_k x_j & \text{if } 0 \leq k < j \\ \alpha_k(x) & = & u_j x_{k+1} - u_{k+1} x_j & \text{if } j \leq k < n-1. \end{array}$$

Proof:
The allocation function $\alpha(x)$ can be written as $\alpha(x) = Sx$, where S is the following $n-1$ x n matrix.

$$S = \begin{pmatrix} u_j & & & -u_0 & & & \\ & \ddots & & \vdots & & 0 & \\ & & u_j & -u_{j-1} & & & \\ & & & -u_{j+1} & u_j & & \\ & 0 & & \vdots & & \ddots & \\ & & & -u_{n-1} & & & u_j \end{pmatrix}$$

We can write that $\alpha(u) = 0$.
Consider $x, y \in D$, $x \neq y$ and $\alpha(x) = \alpha(y)$. Then there exists a nonzero scalar μ such that $x - y = \mu u$. In fact, let $z = x - y$. We have $\alpha(z) = 0$, which implies

$$\begin{array}{rcll} \alpha_k(z) & = & u_j z_k - u_k z_j = 0, & 0 \leq k < j \\ \alpha_k(z) & = & u_j z_{k+1} - u_{k+1} z_j = 0, & j \leq k < n-1. \end{array}$$

Thus, $z_k = (z_j/u_j)u_k = \mu u_k$ and $z = \mu u$ or $x - y = \mu u$.
As $\lambda^t u \neq 0$, we have

$$\tau(x) - \tau(y) = \lambda^t(x - y) = \mu\lambda^t u \neq 0.$$

qed.
From now on we use also the following notation to express the allocation function α:

$$\alpha(x) = Sx = \begin{pmatrix} s_{00} & s_{01} & \dots & s_{0\ n-1} \\ \vdots & \vdots & \ddots & \vdots \\ s_{n-2\ 0} & s_{n-2\ 1} & \dots & s_{n-2\ n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}.$$

2.4 Example: matrix multiplication

Consider the multiplication $C = AB$ of two NxN matrices, where $A = (a_{ij})$, $B = (b_{ij})$, and $C = (c_{ij})$, $c_{ij} = \sum_{k=1}^{N} a_{ik}b_{kj}$. The following obvious sequential algorithm is:

for $k := 1$ to N do
  for $i := 1$ to N do
    for $j := 1$ to N do
      $c_{ij} := c_{ij} + a_{ik}b_{kj}$

Comment: we assume that C is already initialized with $c_{ij} = 0,\ i, j = \overline{1, N}$.
We rewrite now this algorithm in another equivalent form, with the following restrictions:

1. Every variable is indexed by all the indices k, i, j.
2. Every variable is computed explicitly and appears once on the left side of an assignment command.
3. No variable is used on both sides of the assignment operator.

Thus, we have the following equivalent sequential algorithm:

for $k := 1$ to N do
  for $i := 1$ to N do
    for $j := 1$ to N do
      begin
        $a_{ijk} := a_{i\ j-1\ k}$
        $b_{ijk} := b_{i-1\ j\ k}$
        $c_{ijk} := c_{i\ j\ k-1} + a_{ijk}b_{ijk}$
      end

The following initial values are assumed, for $1 \leq i, j, k \leq N$:

$$\begin{aligned} a_{i\ 0\ k} &= a_{ik} \\ b_{0\ j\ k} &= b_{kj} \\ c_{i\ j\ 0} &= 0 \end{aligned}$$

and the result is given by $c_{ij} = c_{i\ j\ N}$.

The same algorithm represented as a SURE (System of Unified Recurrence Equations):

$1 \le i \le N$, $1 \le j \le N$,$1 \le k \le N$,

$$
\begin{aligned}
C(i,j,k) &= C(i,j,k-1) + A(i,j-1,k)B(i-1,j,k) && (1)\\
A(i,j,k) &= A(i,j-1,k) && (2)\\
B(i,j,k) &= B(i-1,j,k) && (3)
\end{aligned}
$$

The input values $A(i,0,k)$ and $B(0,j,k)$ are defined as a_{ik} and b_{kj}, respectively. The initial values of $C(i,j,0)$ are 0. The output values $C(i,j,N)$, that appear only on the left side of the equations, will be the results of c_{ij}.

D is the convex polyhedral domain defined by

$$\left\{ z \in Z^n | \mathcal{B}z \le b \right\}$$

where $n = 3$, $z = \begin{pmatrix} i \\ j \\ k \end{pmatrix}$

$$\mathcal{B} = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \; b = \begin{pmatrix} 1 \\ 1 \\ 1 \\ N \\ N \\ N \end{pmatrix}$$

We obtain the dependence vectors, expressed in the base (i,j,k) as follows: In (1) the value of $C(i,j,k)$ is computed using $C(i,j,k-1)$, $A(i,j-1,k)$ and $B(i-1,j,k)$. Thus we obtain the following three dependence vectors:

$$\theta_1 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \quad \theta_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \quad \theta_3 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Note that from (2) we obtain θ_2, and from (3) we obtain θ_3. It follows that

$$\Theta = \left\{ \theta_1, \theta_2, \theta_3 \right\}.$$

The dependence vectors of the algorithm form the so-called *dependence matrix*

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}.$$

Figure 2 illustrates the dependence graph for $N = 3$.

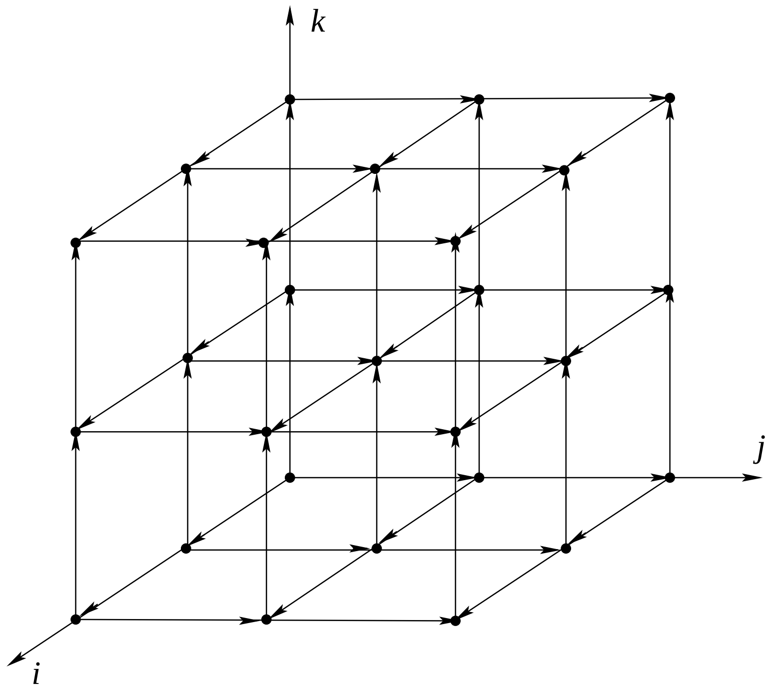

Figure 2: Dependence graph

The next step is to find an adequate *timing function* $\tau(x) = \lambda_0 i + \lambda_1 j + \lambda_2 k + \gamma$, where $x = (i, j, k)^t$ that satisfies the two conditions of Theorem 2.1.
For condition 1 ($\forall v_i \in V_D, \lambda^t v_i + \gamma \geq 0$) it suffices to consider vertex (1, 1, 1):

$$\lambda_0 + \lambda_1 + \lambda_2 + \gamma \geq 0.$$

For condition 2 ($\forall \theta_i \in \Theta, \lambda^t \theta_i > 0$) we should have

$$\begin{array}{rcl} \lambda_0 & \geq & 1, \\ \lambda_1 & \geq & 1, \\ \lambda_2 & \geq & 1. \end{array}$$

Thus we can have the following timing function

$$\tau(x) = \lambda^t x = i + j + k.$$

where

$$\lambda = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \gamma = 0$$

As a next step we are looking for an *allocation function*.
We apply Theorem 2.2, using

$$u = \begin{pmatrix} -1 \\ -1 \\ 1 \end{pmatrix}$$

We have $\lambda^t u \neq 0$. Let us choose $u_j = u_2 = 1$. We have

$$\begin{array}{rcccl} \alpha_0(x) & = & u_2 i - u_0 k & = & i + k \\ \alpha_1(x) & = & u_2 j - u_1 k & = & j + k \end{array}$$

Therefore

$$\alpha(x) = Sx = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} i+k \\ j+k \end{pmatrix}.$$

With this α allocation function the points of D are mapped to the processing elements with the layout of Figure 3.

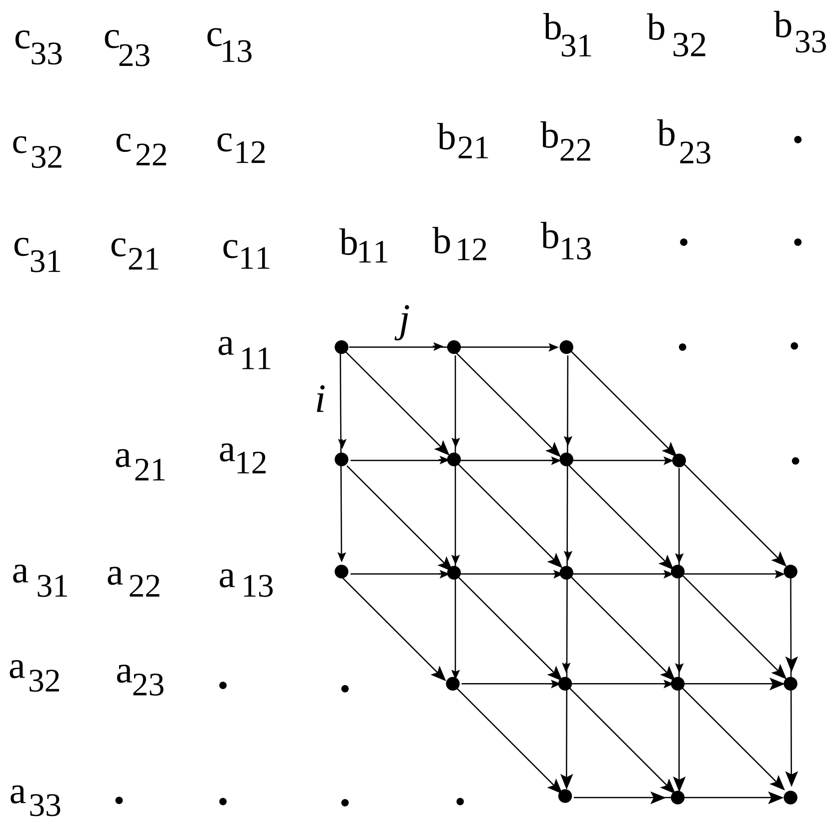

Figure 3: Systolic array for matrix multiplication (solution Weiser-Davis)

Figure 3 illustrates the computing system that carries out the computations of the given SURE. The communication between processing elements is indicated by the arrows of the figure. The communication directions are obtained by applying function α to the dependence vectors of the system.

Thus,

$$\alpha(\theta_1) = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\alpha(\theta_2) = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{4}$$

$$\alpha(\theta_3) = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

As we obtained θ_1 from the dependence between $C(i, j, k)$ and $C(i, j, k-1)$ in equation (1), it follows that $\alpha(\theta_1)$ indicates the direction according to which the C-values are moving. Similarly $\alpha(\theta_2)$ can be associated to the A-values and $\alpha(\theta_3)$ to the B-values.
Figure 4 illustrates the computations that take place in one typical cell of the array, where A_I, B_I, C_I are input values to the respective cell, while A_Q, B_Q, C_Q are the output values. We also specify the computations performed in one time unit as follows: from (1) we obtain $C_Q = C_I + A_I * B_I$, from (2) $A_Q = A_I$, and from(3) we have $B_Q = B_I$.

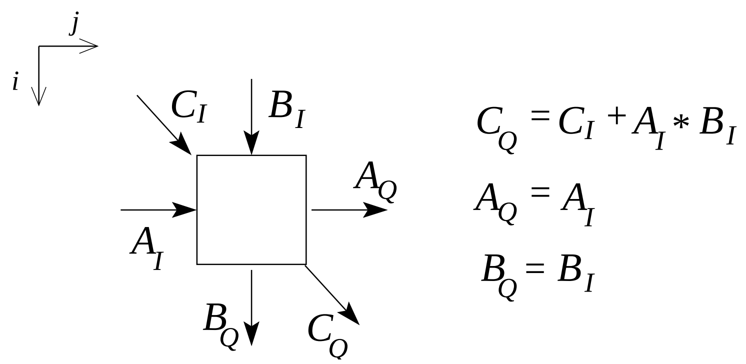

Figure 4: The scheme of computation in a cell of the array

According to the specified time function and allocation function the values of $C(i, j, k)$, $A(i, j, k)$ and $B(i, j, k)$ are computed in cell $(i + k, j + k)^t$ of the systolic array at time instant $i + j + k$.
One can easily verify whether the given systolic algorithm computes indeed correctly the results by checking that all the arguments needed for the computation in an arbitrary cell are available at the right time instant. Take for example the $C(i, j, k-1)$ argument needed to compute $C(i, j, k)$. $C(i, j, k-1)$ was computed at time instant $\tau((i, j, k-1)^t) = i + j + k - 1$, i.e. the previous time-step, at the cell $\alpha((i, j, k-1)^t) = (i + k - 1, j + k - 1)^t$. According to

(4) the C values are moving in direction $\alpha(\theta_1) = (1,1)^t$, which means that at the next time-step the computed $C(i,j,k-1)$ value will move towards the cell $(i+k-1, j+k-1)^t + (1,1)^t = (i+k, j+k)^t$, which is exactly the cell where the value is needed. Similarly can be shown that the other arguments are also available in the right moment at the right cell. As concerning the placement of the input data at the beginning of the computations, it can also be identified with the help of the calculated time function and allocation function. Let us take for example the components of C. For $k = 1$ $C(i,j,0) = c_{ij}$ input data is needed at time instant $i+j+1$ at the cell $(i+1, j+1)^t$. The computations begin at the time instant $t_\circ = min\{i+j+k|(i,j,k)^t \in D\} = 3$. As $C(i,j,0)$ has to be available at time instant $i+j+1$ at the cell $(i+1, j+1)^t$, one can easily count that the position of the input data $C(i,j,0)$ at the beginning of the computations (i.e. time instant $t_\circ - 1$) has to be $(i+1, j+1)^t - (i+j+1-t_\circ+1)\alpha(\theta_1) = (2-j, 2-i)^t$.

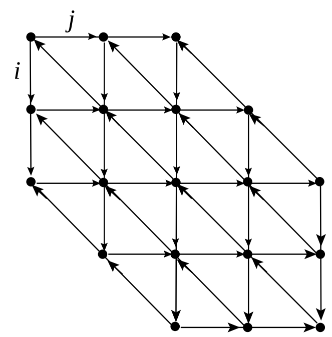

Figure 5: Systolic array for matrix multiplication (solution Kung-Leiserson)

Note: If we choose another u vector

$$u = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

to project the domain D, we obtain the allocation function

$$\alpha(x) = Sx = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{pmatrix} i-k \\ j-k \end{pmatrix}.$$

leading us to another systolic array with the layout in Figure 5.

3 Generic Systolic array

This approach, introduced and illustrated in [GD92], represents a program-oriented
methodology, being more closed to the program design.

3.1 Description of the linear generic systolic array

The *linear generic systolic array* (linear **gsa**) is determined by the following elements:

- The *topology of the network* (see Figure 6)

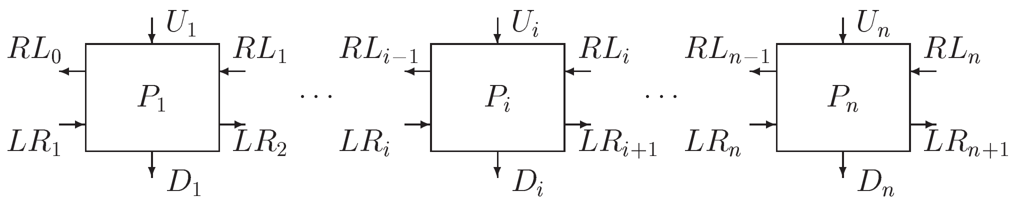

Figure 6: The linear generic systolic array

- The *scheme of communication* consists of two parts:

  communication ; computation

- The *internal memory* of cell P_i contains six *communication registers*, denoted $A[i]$, $B[i]$, $C[i]$, $E[i]$, $F[i]$ and $G[i]$. The remaining part of the memory is denoted $M[i]$. Its size is independent from the size n of the network. (see Figure 7)

- The *communication phase* for cell P_i consists in the concurrent execution of the following statements:

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

  (Note 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 $x := e$.)

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

An instance of the linear gsa is obtained by replacing Φ by a total function. 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 linear gsa comprises the following four logical components:

- 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.

The input channels are $LR_1, U_1, \ldots, U_n$ and RL_n,
the output channels are $RL_0, D_1, \ldots, D_n$ and LR_{n+1}.

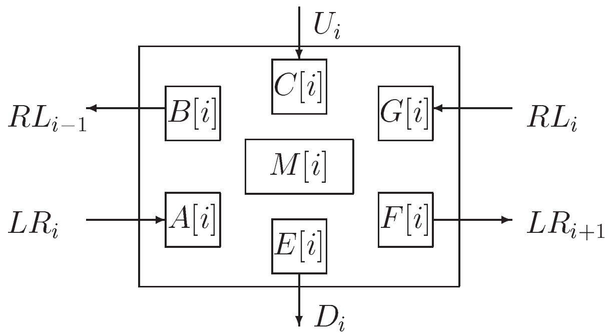

Figure 7: A typical cell of the linear generic systolic array

The structure of the program executed by the array is

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

Description of the transmission phase:

$$LR_1?A[1]; i = \overline{2,n}\ A[i] := F[i-1]; LR_{n+1}!F[n] \qquad \text{Left} - \text{to} - \text{right},$$

$$k = \overline{1,n}\ U_k?C[k]; k = \overline{1,n}\ D_k!E[k] \qquad \text{Up} - \text{to} - \text{down},$$

$$RL_0!B[1]; i = \overline{1,n-1}\ G[i] := B[i+1]; RL_n?G[n] \qquad \text{Right} - \text{to} - \text{left}$$

The computation phase is modeled by:

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

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) : $i \in N$) 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 following recurrence system describes the effect of the execution of one step of the program executed by the array.

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

$\forall r \in \{1, \ldots, n\}$, and $i > 0$
Additional convention:

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

where $X_i[r]$ (the symbol X stands for F, E, B or M) denotes the value of a register X of cell r, after the ith iteration. $dU_r(i)$ is the ith element of the input stream dU_r.

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

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

3.2 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. (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.)

2. The behaviour of the array and its environment is formally stated as a set of recurrence equations. (This step is mechanical: the recurrence system corresponding to the array - and its environment - is obtained by mere instantiation of the **generic recurrence system**)

3. The recurrence system is solved, or an adequate invariant is found. (From the practical point of view, in several cases the discovery of an adequate invariant can be easier than the discovery of the solution of the recurrence system.)

4. The data and result streams of the array are interpreted as data and results of the problem.
   (This 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.)

3.3 Example: matrix-vector multiplication

The product Y of a p x p matrix A and a p dimensional vector X is obtained by executing the assignments $y_k := y_k + a_{kj}x_j, \forall k, j \in \{1, \ldots, p\}$. The initial value of every y_k is 0. Assignments on y_k and $y_{k'}$ may be performed concurrently iff $k \neq k'$. Since we want to use a linear gsa for the mapping of the algorithm onto a systolic array, the communication scheme is already fixed. Each cell will perform the assignment $y := y + a * x$, for some y, a and x. We only have 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 a-flow (matrix components) 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 y (components of Y) and RL is assigned to x (components of X). The storage logical unit is not used. This leads to the following functions:

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

The corresponding recurrence system is:

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

We are interested in the calculation of the result stream rR, corresponding to the y_k values of the product Y. It is obtained as follows:

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

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 $i - 2l + 1 > 0$, $\forall l \in \{1, \ldots, n\}$. These conditions reduce to $i \geq 2n$. (For $0 < i < 2n$, the value $rR(i)$ depends

on the initial values of cells. As no condition is required about the initial values, this part of the result stream cannot be used.)

The allocation implemented by the array and even the relation between the size n of the systolic array and the dimension p of the matrix and of the vector are fixed in the fourth and last step of the design procedure.

The components of the vector Y, the values $y_k = \sum_{l=1}^{n} a_{kl}x_l$ must be extracted from the stream rR for $k = \overline{1,n}$. The first useful value of the result stream is $rR(2n)$, so 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 $k = \overline{1,n}$ with the identity

$$rR(2n-1+k) = dL(n-1+k) + \sum_{l=1}^{n} dU_{n-l+1}(2n-1+k-l) * dR(2n+k-2l). \quad (5)$$

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 $dR(j) = x_{j\ mod\ n+1}$ ("+1" is introduced because the range of $j\ mod\ n$ is $\{0,\ldots,n-1\}$, while the components of X are indexed in $\{1,\ldots,n\}$). This choice leads to:

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

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 $A_{k,(2n+k-2l) mod\ n+1}$. This allows the determination of $dU_i(j)$. The results are summarized below:

$$\begin{array}{rcll} dL(j) & = & 0 & j \in \{n,\ldots,2n-1\}, \\ dU_i(j) & = & A_{j-i-n+2,(j+i-n)\ mod\ n+1} & i \in \{1,\ldots,n\}, \\ & & & j \in \{n+i-1,\ldots,2n+i-2\}, \\ dR(j) & = & x_{j\ mod\ n+1} & j \in \{1,\ldots,3n-2\}. \end{array} \quad (6)$$

Similar results can be obtained for even n.
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, $rD_i(j+1) = dU_i(j)$.

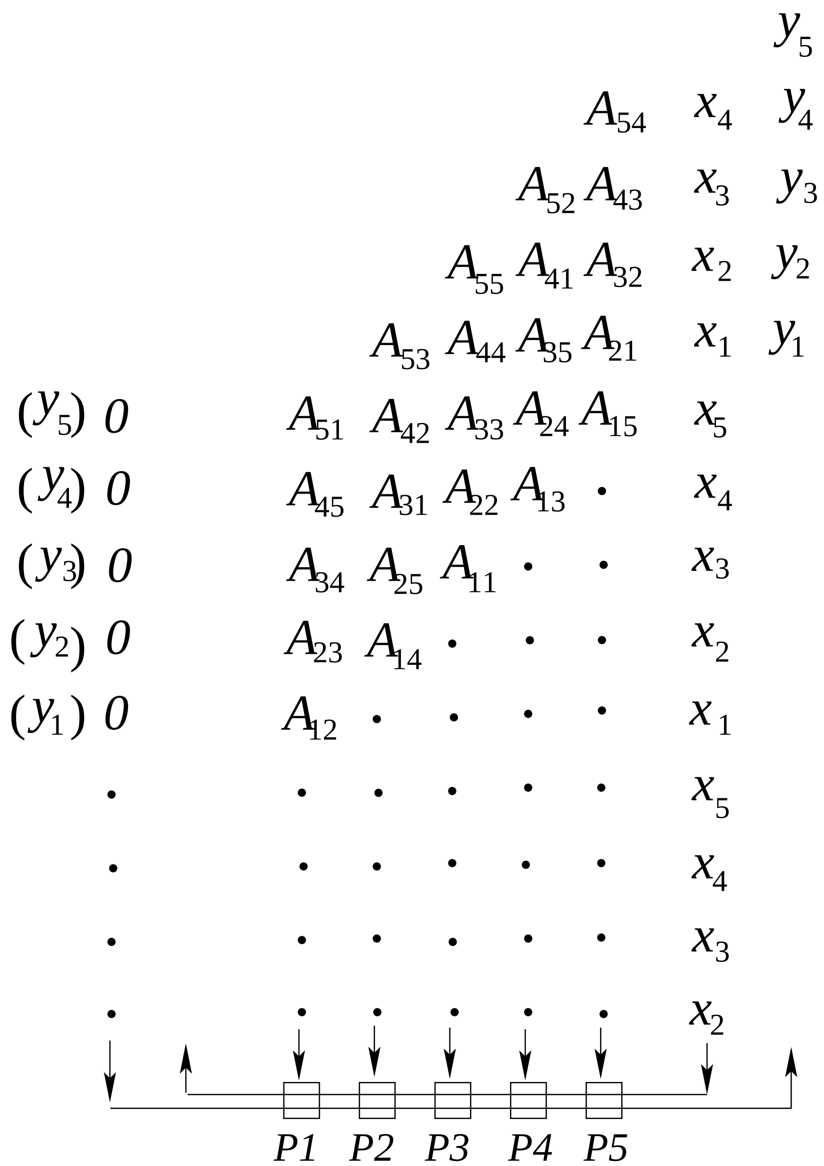

Figure 8: Systolic array for matrix-vector multiplication

The execution is completed after $3n - 1$ steps. The initial contents of the registers are arbitrary, so are the members of the input streams which do not occur in formulas (6).
Figure 8 presents the graphical representation of the array for $n = 5$.
As an example let us take the first useful value of the result stream, $rR(2n) = rR(10)$, corresponding to the value of y_1 that leaves the rightmost processor ($P5$) at time-step 10. From (5), (6) with $n = 5$ and $k = 1$ we obtain $y_1 = A_{15}x_5 + A_{13}x_3 + A_{11}x_1 + A_{14}x_4 + A_{12}x_2$. More exactly $dU_1(5) * dR(1) = A_{12}x_2$ is computed in processor $P1$ at time-step 5. At the next step $A_{14}x_4$ is added to it in processor $P2$, and so on $A_{11}x_1$, then $A_{13}x_3$ and finally $A_{15}x_5$.

4 BSA-method

In [KRS96] another method is developed for deriving systolic programs for a large class of linear recurrence equations referred to as *directed recurrence equations* (DRE) which properly includes the class of uniform recurrence equations. This approach uses as starting point a graphical abstraction of the DRE also called *dependency graph* but defined in another way than in the case of the space-time transformation method. Applying a set of rules it is transformed to a graph called *modified dependency graph* (**MDG**) where the level of each node yields the time steps for the actions.

The advantage of such an approach is that one need not solve a system of linear inequalities for finding appropriate timing functions. Using the MDG, a systolic program for a very general architecture referred to as *basic systolic architecture* (**BSA**) can be designed.

Finally the BSA is mapped to a given systolic architecture using a set of semanticspreserving transformations. Here the mapping of channels and timing functions is considered unlike the allocation function of the space-time transformation method.

Definition 4.1. (**Directed Linear Recurrence Equation**)
Directed linear recurrence equations (DRE) are equations of the form:

$$\begin{aligned} a_k(p_k^i) &= v_k^i,\ i \in \{1, \ldots, l_k\} \\ a_k(p) &= f_k(a_1(\delta_{1_k}(p)), \ldots, a_s(\delta_{s_k}(p))) \end{aligned}$$

where

1. $a_i : D_n \to R$,
2. δ_{i_k} is an affine transformation from D_n to D_n, such that $\delta_{i_k}(p) = A_{i_k}p + b_{i_k} = A_{i_k}(p + b'_{i_k})$ and $dim(kernel(A_{i_k})) \leq 1$. When $dim(kernel(A_{i_k})) = 0$, $b'_{i_k} = A_{i_k}^{-1} b_{i_k}$ and when $dim(kernel(A_{i_k})) = 1$, $A_{i_k}(b'_{i_k}) = \overline{0}$ and $b'_{i_k} \neq \overline{0}$.

   A_{i_k} is a constant n x n matrix over Z
   b_{i_k}, b'_{i_k} are constant n x 1 matrices (vectors) over Z

3. $p \in D_n,\ v_k^i$ is a scalar constant (input)
4. $a_k(p)$ does not appear on the r.h.s. of the equation
5. f_k is a single-value function strictly dependent of its arguments.

Note: $dim(kernel(A_{i_k}))$ refers to the dimension of the kernel (or the null space) of A_{i_k}. If $\forall i \in \{1, \ldots, s\}$, δ_i is identity, then we obtain uniform recurrence equation (see Definition 2.3).

As mentioned, the method described in [KRS96] is applicable to a larger class of recurrence equations, yet in this paper we are interested only in relating it to the method described in Section 2 and for the sake of simplicity we will use it only for URE systems.

Definition 4.2. (**Dependency Graph for a DRE**)
The dependency graph for a directed recurrence equation is a directed labeled graph $G(V, E, \mathcal{L})$, as defined below:

1. Let $V = \{(a_k, p) | p \in D_n\}$ be the set of vertices,
2. E be the set of edges, where
$E = \{e^p_{ki_k} | e^p_{ki_k}$ is an edge joining $(a_i, \delta_{i_k}(p))$ and $(a_k, p) \forall i, k = \overline{1, s}$, $p \in D_n\}$ and
3. $\mathcal{L}$ is a map from E to V defined by $\mathcal{L} : e^p_{ki_k} \mapsto (a_i, \delta_{i_k}(p))$.

Note: In a DG all the nodes are distinct and all the outgoing edges from a node are labeled with the same label.

Proposition 4.1. The dependency graph represents the directed recurrence equation.

Proof: For every variable in the DRE there exists a vertex in the dependency graph and for every dependency between two variables in DRE there is a directed edge between the corresponding vertices in the DG.

4.1 Derivation of a Timing Function

Before describing the derivation of a timing function some other definitions and notations has to be introduced.

Definition 4.3. (**Direct dependence**)
The computation of (a_k, p) is said to depend directly on (a_i, q)(denoted by $(a_i, q) \to (a_k, p)$) iff there is an edge in $G(V, E, \mathcal{L})$ directed from (a_i, q) to (a_k, p).

Notations:

- Let $child_b(a_i, q) = \{(a_k, p) | (a_k, p)$ is a child of (a_i, q) on an edge labeled $b = (a_i, q)$ in the DG$\}$.
- Let $child_bp((a_i, p), (a_j, q))$ *iff* $(a_j, q) \in child_b(a_i, p)$.
- Let $level(a_i, p)$ be the maximal depth of (a_i, p) in the DG.
- The transformation introduces extra nodes referred to as *dummy nodes*. We say $dummyp(d, p) \equiv true$ *iff* (d, p) is a dummy node.

Definition 4.4. (**Child-set**) The child-set of (a_i, p) under transformation A_{i_j}, denoted by $child\, i_j(a_i, p)$, is the set of all $(a_j, q)'s$, $p = A_{i_j} q + b_{i_j}$ that depend directly on (a_i, p) i.e.

$$child\, i_j(a_i, p) = \{(a_j, q) | child_b p((a_i, p), (a_j, q)) \equiv true \wedge p = A_{i_j} q + b_{i_j}\},$$

$$\text{where } b = (a_i, p).$$

4.2 Algorithm for transformation of DG into MDG

The timing function is obtained by transforming the DG into another graph referred to as the Modified Dependency Graph (MDG).

Steps of the transformation:

I. **Preprocessing Steps**

1. All children of a given node are partitioned into classes of child-sets (see Definition 4.4).
2. The edges leading into classes of child-sets are relabeled such that all edges leading to child nodes in the same partition have identical labels, while those leading to child nodes in different partitions have distinct labels.

II. **Iterative Rewrite Rule(s)**

The procedure is expressed in terms of rewrite rules of the form $\alpha \rightsquigarrow \beta$ with the interpretation that the pattern in α is replaced by the pattern in β. The first three rules mentioned in [KRS96] deal with the nodes belonging to a child-set defining an order and rerooting data through these nodes. As in the case of a SURE all the child-sets have only one element we will only have to use rule 4 which ensures that a node appears at the same depth in all the *chains* (see Definition 4.6) of the the MDG that contain the node:

rule 4

Apply this rule till it is no longer applicable.

$$level((c_i, p) + 1 \leq level(c_j, q)) \wedge child_b p((c_i, p), (c_j, q))$$

$$\rightsquigarrow child_b p((c_i, p), (d_{c_j}, \circ)) \wedge \neg child_b p((c_i, p), (c_j, q))$$
$$\wedge child_b p((d_{c_j}, \circ), (c_j, q))$$

Note: The graphical illustration of the rule is shown by Figure 9.

Definition 4.5. (**Dependence**)

(a_k, p) is said to *depend on* (a_i, q), denoted by $(a_i, q) \overset{*}{\rightarrow} (a_k, p)$, iff there exists a path from (a_i, q) to (a_k, p) labeled (a_i, q) in MDG.

We also define

$child^*_b(a_i, q) = \{(a_k, p) | (a_k, p)$ is a child of (a_i, q) on a path labeled $b\}$.

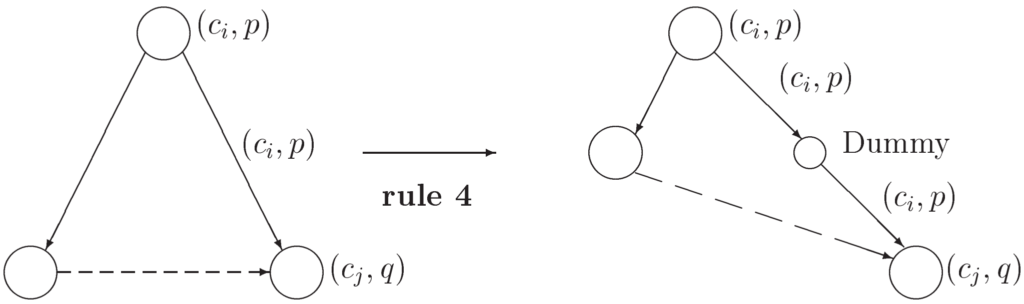

Figure 9: Illustration of rule4

Proposition 4.2. The transformational step preserves the dependency relation.

Proof: We consider only rule 4 which introduces new nodes. The deleted *edge* is replaced by a *path*. Let (a_j, q) depend directly on (a_i, p) in MDG^i. Now, in MDG^{i+1} either the dependency is maintained or there could be a dummy node, say (d_{a_j}, q) between (a_j, q) and (a_i, p) and directed edges (a_i, p) to (d_{a_j}, q) and (d_{a_j}, q) to (a_j, q). Thus, it immediately follows that the dependency between (a_j, q) and (a_i, p) is maintained.

Definition 4.6. (**Chain**)
A chain is a maximal directed path all of whose edges have the same label. A chain is identified by the edge label.

Proposition 4.3. The MDG is a set of distinct total order chains. [KRS96]

Proposition 4.4. The MDG is computationally equivalent to the DG.

Proof:

1. For every node in the DG, there exists at least one node in the MDG. (The transformation step never delete a node.)

2. For every directed edge in the DG, there exists an equivalent directed edge or path in the MDG. This follows from the fact that rule 4 at most changes an edge into a path.

3. All the arguments needed for a computation in any given node are available at the time of its computation. Rule 4 ensures that a node appears at the same depth in all the chains of the MDG that contain the node. This combined with Proposition 4.2 implies the result.

Definition 4.7. (**Timing function**)
A timing function, t, is a mapping from V to N, where V is the set of vertices in the dependency graph, such that if (a_i, p) depends on (a_j, q), then $t(a_i, p) > t(a_j, q)$, where p, $q \in D_n$.

Proposition 4.5. (a_k, q) can be computed at $t = level(a_k, q)$. [KRS96]

4.3 Basic Systolic Architecture (BSA)

The MDG is a set of linear chains of computations. The labels repeated (in different chains) correspond to the same computation. To obtain a systolic architecture, we have to embed all these chains into a lattice structure Z^k for some k. The process is based on classifying the chains into parallel, and coplanar (but intersecting) chains. The classification is achieved by the relations defined below.

Definition 4.8. (**Coplanar**)
Define a coplanar relation C on the chains of MDG using the following rules,

1. If U and V intersect, then they are coplanar.
2. If U and V are parallel, then they are coplanar.
3. If U, V are coplanar and W intersects U at i and V at j, where i, j are distinct, U, V, W are coplanar.
4. If U, V, W are coplanar and X, U, V are coplanar then so are X, U, V, W.

Definition 4.9. (**Parallel**)
Define a parallel relation P on the chains of MDG using the following rules:

1. If U, V, W, X are coplanar such that U and V intersect at i, V and W intersect at j, W and X intersect at k, X and U intersect at l where i, j, k, l are distinct and
$$(diff - level(i,l) = diff - level(k,j)) \wedge (diff - level(i,j)$$
$$= diff - level(k,l))$$
where $diff - level(i,j) = abs(level(i) - level(j))$ then U is parallel to W and V is parallel to X.
2. If U is parallel to V and V is parallel to W then U is parallel to W, where U, V, W are chains.

Method to Obtain the BSA from a MDG

Step 1: Obtain the coplanar and parallel classes using $\mathcal{C}$ and $\mathcal{P}$ respectively. Let us denote the respective partitions by $\mathcal{C} = \{C_1, \ldots, C_r\}$ and $\mathcal{P} = \{P_1, \ldots, P_s\}$.
Step 2: Embedding the planes:

a) Choosing the initial plane and the basis chains for the planes.

1. Find a, b such that
   $((a,b \in C_i) \wedge (a \in P_k \wedge b \notin P_k) \wedge (\nexists (c,d) \ni ((c,d \in C_i) \wedge (c \in P_j \wedge d \notin P_j) \wedge (num-of-dummies(a,b) > num-of-dummies(c,d)) \wedge ((a,b) \neq, d)))))$
   `/*` a`,` b `are coplanar but not parallel and no other such pair of chains exists such that the nr. of dummies is less in them */`
   where a, b are chains, $P_j, P_k \in \mathcal{P}$ and $C_i \in \mathcal{C}$ and $num-of-dummies(a,b)$ is a function that returns the number of dummy nodes in the pair of chains a, b.
2. Choose plane, say C_i, with the least $num-of-dummies()$ as the initial plane for embedding in Z^k.

b) Order of Embedding the planes in Z^k.
Let C_i denote the plane chosen as in a), `planes_embedded :=` $\emptyset$ and `planes_not_embedded :=` $\mathcal{C}$.
`repeat`
`begin`

1. Choose a pair of basis chains for C_i that enables us to embed all the chains in C_i onto a plane in Z^k. (In fact, for the initial plane, we have already chosen a basis while choosing the initial plane.) Let us denote the basis chains by a_i and b_i. Let us choose a coordinate plane in Z^k for C_i. We map a_i and b_i to an orthogonal basis z_{i1} and z_{i2} in C_i.
2. `planes_embedded := planes_embedded` $\bigcup\{C_i\}$.
   `planes_not_embedded := planes_not_embedded` $-$ $\{C_i\}$.
   Choose another plane, say C_k, such that one of the basis chains of C_k also belongs to some plane in `planes_embedded`.
   Let $C_i := C_k$.
   `end`

`until planes_not_embedded =` $\emptyset$

Step 3: Embedding chains in each plane
`repeat`
`begin`

1. Choose a plane, say C_m embedded in Z^k, in the reverse order to that in Step 2 above. Let a_i and b_i be its basis chains mapped onto z_{i1} and z_{i2} respectively in C_m.
2. Layout all chains in the same parallel class as a_i and b_i parallel to z_{i1} and z_{i2} respectively.

`end`
`until all planes in` $\mathcal{C}$ `have been considered.`

Lemma 4.1. The MDG is embedded in the BSA. [KRS96]

Theorem 4.1. BSA exactly computes the given DRE/LRE.

Proof: The proof follows by just putting the following observations together:

1. The MDG is embedded in the BSA. (From Lemma 4.1)
2. The MDG is computationally equivalent to the DG. (From Proposition 4.4)
3. The dependency graph represents the directed recurrence equation. (From Proposition 4.1)

4.4 Mapping onto Systolic Architectures

A mapping from a BSA to another network G' is a triple $M = (e, d, g)$ where e, d and g are defined as below:

- e is a projection scheme. By a projection scheme we mean *'projection along an axis'* or *'translation along an axis'* defined as follows:
    1. *Projection along an axis.* For any point $q = (x_1, \ldots, x_n) \in$ a BSA we define the projection along x_i- axis as follows: $e : (x_1, \ldots, x_n) \mapsto (x'_1, \ldots, x'_{n-1}) \ni (\forall\ 1 \leq j < i,\ x'_j = x_j) \wedge (\forall i \leq j < n, x'_j = x_{j+1})$ where point $(x_1, \ldots, x_n)$ associates to processor $l = (x'_1, \ldots, x'_{n-1})$ in G' .
    2. *translation along an axis.* For any point $q = (x_1, \ldots, x_n) \in$ a BSA, we define the translation along x_i- axis by $e : (x_1, \ldots, x_n) \mapsto (x'_1, \ldots, x'_n) \ni (\forall\ 1 \leq j < i,\ x'_j = x_j) \wedge (\forall i < j \leq n,\ x'_j = x_j) \wedge (x'_i = x_i + x_k) \wedge (1 \leq k \leq n) \wedge (k \neq i)$ where point $(x_1, \ldots, x_n)$ associates to processor $l = (x'_1, \ldots, x'_n)$ in G' .
- d is a function that maps a channel in a BSA to a channel in G'. For any channel $(p, q) \in \{\{\text{inputchannels}\} \cup \{\text{outputchannels}\}\}$, d is defined as follows: $d : (p, q) \mapsto (e(p), e(q))$,
- The timing function g maps the execution time associated with each computation in a BSA to the execution time associated with each computation in G'. (We would have to modify the time function obtained for the BSA only in the case when after the mapping multiple functions to be computed at the same time instant are assigned to the same node)

4.5 Example

Let us return to the example of matrix-multiplication presented in Subsection 2.4. As variables $A(i, j, k)$, $B(i, j, k)$ and $C(i, j, k)$ are computed at the same time, nodes $A(i, j, k)$, $B(i, j, k)$ and $C(i, j, k)$ can be merged, thus we obtain 12 distinct chains in the MDG for $N = 2$ (see Figure 10). For the sake of simplicity the labels are not displayed (each chain is labeled with the name of its first node). For $N = 3$ we would have 27 chains.

The level of a node indicates the time instant when the computations take place. Thus $A(1, 1, 1)$, $B(1, 1, 1)$ and $C(1, 1, 1)$ are computed at time instant 1, $C(1, 1, 2)$, $C(1, 2, 1)$, $C(2, 1, 1)$ and the coresponding A and B values at time instant 2 and so on.

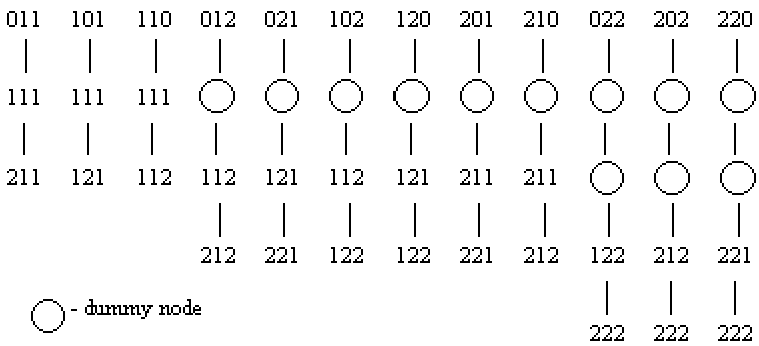

Figure 10: Modified Dependence Graph for $N = 2$

To obtain a systolic architecture, we have to embed all these chains into a lattice structure Z^k for some k. For the sake of commodity let us number the chains from 1 to 12 (we will refere to chain 011 as chain 1 and so on). Following the procedure described in Subsection 4.3 in Step 1, according to Definition 4.8 and Definition 4.9 we obtain 24 distinct coplanar classes, each of them containing two elements and no parallel classes. If in Definition 4.8 we also consider an additional condition:

5. If U and V are coplanar, V and W are coplanar, W and X are coplanar, X and U are coplanar then U, V, W, X are coplanar chains.

we obtain the following Coplanar and Parallel classes:
$\mathcal{C} =$
$= \{\{1, 2, 5, 8\}, \{1, 3, 4, 9\}, \{8, 9, 11, 12\}, \{5, 7, 10, 12\}, \{2, 3, 6, 7\}, \{4, 6, 10, 11\}\}$,
$\mathcal{P} = \{\{1, 4, 5, 10\}, \{2, 6, 8, 11\}, \{3, 7, 9, 12\}\}$.
In Step 2 we can choose C_1 as initial plane with chains 1 and 2 as basis chains

for the embedding. Following Step 2 and Step 3, one can make the observation that the structure of the BSA obtained corresponds to that of the dependence graph from Section 2 (see Figure 2).

Note that the nodes that have the same time instant associated belong to the same hyperplane.

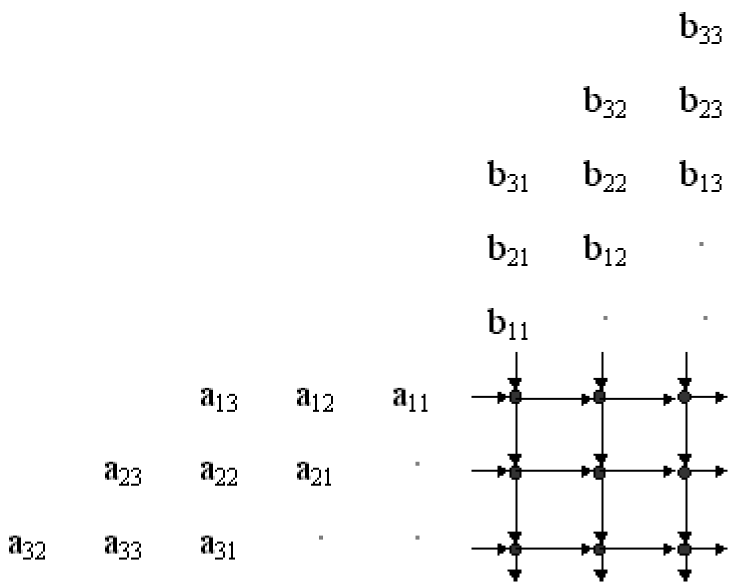

Figure 11: Systolic array for matrix multiplication (Solution Okuda-Song)

We can apply different mappings to the BSA of our example. Projection along $k-$ axis gives us a mesh-connected network (see Figure 11). In this case $e : (i, j, k) \mapsto (i, j)$. Translation along $i-$axis followed by translation along $j-$axis and projection along $k-$axis gives us a hexagonal network corresponding exactly to that of Example 2.4, where $e : (i, j, k) \mapsto (i + k, j + k)$ (see Figure 3).

The scheme of computation in a cell of the array can be deduces in the same way as in the case of the space-time methodology. Note that in the case of projection along $k-$ axis we obtain locally connected input and output channels for fixed i and j $(d : ((i, j, k), (i, j, k + 1)) \mapsto ((i, j), (i, j)))$, thus the communication can be replaced by a simple memory assignment. As this channel corresponds to the C values, it follows that in the corresponding systolic array (Figure 11) the values of the resulting matrix will be evaluated in the local memories of the cells of the array.

The timing function obtained for the BSA does not have to be modified because in the case of both mappings there are no multiple functions to be computed at the same time instant in the same processing element. The function g can be defined as a function taking three parameters a time t_j, a node p and computation elements $a_{t_j,i}$: $g(t_j, i, a_{t_j,i}) = t_j$.

5 Conclusions

The paper provides a survey of the available systolic array design methods touching also other related problems like array partitioning, the uniformization and optimization problem. Advantages and drawbacks of each of them are also shown.

Three different approaches to the automatic systolic array design are discussed in detail, with representative examples and useful observations added.

The further purpose of this work is to automate the generation and verification of systolic designs with the help of automated theorem proving systems combined with computer algebra systems (in particular Theorema, see www.theorema.org).

From this point of view we have found most useful the ideas presented in section 4 concerning the BSA-method, - as it works with symbolic transformations on the dependence graph and can be used for a larger set of problems than the other presented methods -, but also taking into account its connection with the space-time transformation methodology (presented in section 2). The method presented in section 3, starting from generic systolic array comprises an interesting idea, though it is not completely automated. The introduced formal notations and definitions could be still useful.

Concerning the automatic systolic design problem one can actually consider two cases. Both of them are remarcable:

- The case when we are interested in designing systolic arrays for a given architecture.

- Finding out new, interesting systolic solutions for given problems, which could be optimized from one or another point of view.

References

[BDD90] J. Bu, E. F. Deprettere, and P. Dewilde. A design methodology for fixed-size systolic arrays. In S. Y. Kung and E. E. Swartzlander, editors, *Application Specific Array Processors*, pages pages 591–602. IEEE Computer Society, 1990.

[CK98] Y.-K. Chen and S. Y. Kung. A systolic design methodology with application to full-search block-matching architectures. *Journal of VLSI Signal Processing Systems*, 19(1):51–77, June 1998.

[Dar91] Alain Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *The VLSI Jounal*, 12(3):293–304, Dec 1991.

[DD90] A. Darte and J.M. Delosme. Partitioning for array processors. Technical Report 90-23, LIP, ENS-Lyon, 46 allee d'Italie, 69364 Lyon cedex 07, France, October 1990.

[DI86] Jean-Marc Delosme and Ilse C. F. Ipsen. Systolic array synthesis: computability and time cones. In *Parallel algorithms & architectures (Luminy, 1986)*, pages 295–312, Amsterdam; New York, 1986. North-Holland.

[FM84] J.A.B. Fortes and D.I. Moldovan. Data broadcasting in linearly scheduled array processors. In *11th Annual Symp. on Computer Architecture*, pages 224–231, 1984.

[GD92] Pascal Gribomont and Vincent Van Dongen. Generic systolic arrays: A methodology for systolic design. *Lecture Notes in Computer Science*, 668:746–761, 1992.

[HH95] Y. T. Hwang and Y. H. Hu. A unified partitioning and scheduling scheme for mapping multi-stage regular iterative algorithms onto processor arrays. *J. of VLSI Signal Processing*, 11(1&2):133–150, 1995.

[HL87] C.-H. Huang and C. Lengauer. The derivation of systolic implementations of programs. *Acta Informatica*, 24(6), Nov 1987.

[KRS94] Ladan Kazerouni, Basant Rajan, and R. K. Shyamasundar. Derivation of systolic programs. In *International Conference on Parallel Processing (3)*, pages 69–73, 1994.

[KRS95] Ladan Kazerouni, Basant Rajan, and R. K. Shyamasundar. Mapping linear recurrences onto systolic arrays. In *IPPS: 10th International Parallel Processing Symposium*. IEEE Computer Society Press, 1995.

[KRS96] Ladan Kazerouni, Basant Rajan, and R. K. Shyamasundar. Mapping linear recurrence equations onto systolic architectures. *International Journal of High Speed Computing (IJHSC)*, 8(3):229–270, 1996.

[Kun87] Sun-Yuan Kung. *VLSI Array Processors.* NJ: Prentice Hall, Englewood Cliffs, 1987.

[LQR99] Dominique Lavenier, Patrice Quinton, and Sanjay Rajopadhy. Advanced systolic design. In Marcel Dekker, editor, *Digital Signal Processing for Multimedia Systems*, Signal Processing, chapter 23, pages 657–692. 1999.

[LW85] G.-J. Li and B. W. Wah. The design of optimal systolic arrays. *IEEE Trans. on Computers*, C-34(1):66–77, Jan. 1985.

[MF86] Dan I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. Computers*, 35(1):1–12, 1986.

[Mol83] Dan I. Moldovan. On the design of algorithms for VLSI systolic array. *Proc. IEEE*, 71(1):113–120, 1983.

[ND88] H. W. Nelis and E. F. Deprettere. Automatic design and partitioning of systolic/wavefront arrays for VLSI. *Circuits systems signal process*, 7(2), 1988.

[OFW91] M. T. O'Keefe, J. A. B. Fortes, and B. W. Wah. On the relationship between systolic array design methodologies. *IEEE Trans. on Computers*, 41(12):1589–1593, 1991.

[QD89] Patrice Quinton and Vincent Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1(2):95–113, Oct 1989.

[QR90] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures.* Prentice-Hall, 1990.

[Qui84] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrence equations. In *11th Ann. Int. Symp. on Computer Architecture*, pages 208–214. IEEE Computer Society Press, 1984.

[Qui87] P. Quinton. The systematic design of systolic arrays. In Francoise Fogelman Soulie, Yves Robert, and Maurice Tchuente, editors, *Automata Networks in Computer Science*, chapter 9, pages 229–260. Manchester University Press, 1987.

[Son94] Siang W. Song. *Systolic algorithms: concepts, synthesis, and evolution.* Temuco, CIMPA School of Parallel Computing, Chile, 1994.