Partitioning for array processors*

Alain Darte
Laboratoire LIP-IMAG
Ecole Normale Supérieure de Lyon
69364 LYON

Jean-Marc Delosme
Yale University
Department of Electrical Engineering
New Haven, CT 06520

august 1990

1 Introduction

Many scientific algorithms, consisting of repetitive computations can be expressed as uniform recurrence equations. Several methods have been proposed to synthesize systolic architectures from these equations. Usually, a dependence graph and a description of a domain of computation given by linear constraints are extracted from the equations. This domain of computation (a set of computational points) is then projected along a well chosen direction onto a plane. Each point is assigned to a processor. However, this method must be refined if we want to map an arbitrary size problem onto a fixed number of processors. This is the case, for example, if we want to conceive a compiler which does this automatically. This paper proposes a systematic method for mapping algorithms into fixed size array processors, in using the usual projection followed by a partioning.

The paper is organised as following: section 2 presents the general principles of the partitioning used and the qualities of a good partitioning. Sections 3 and 4 introduce the mathematical tools needed to understand the next sections which present a method for mapping a n-dimensional domain

*This work has been supported by the ESPRIT Basic Reseach Action 3280 "Novel Algorithms for New Real-time VLSI Architectures" (NANA)

of computation onto a $(n-1)$-dimensional array (section 5) and a method for mapping onto a p dimensional array for $p < n-1$, where usually $p = 1$ or 2 (section 6). Section 7 gives two examples to illustrate the previous sections. Finally, Section 8 discusses briefly the qualities and the weaknesses of our method.

2 Partitioning

Partitioning is an unavoidable step for mapping computations of large size problems onto an array processor of fixed size. There are several methods for solving this problem depending on what we want to minimize: for example, the amount of external communications or the size of the local memory. One possibility is to partition the domain of computation in blocks of size the number of available processors and to compute all the points of one block before computing another one. This approach is discussed in [MF86]. For array processors that have a sufficiently large local memory, another partitioning technique is to project the domain of computation onto a plane, to subdivide this plane in blocks, and to map each block onto a processor. In this case, every point in a block must be computed at a different time. The local memory size needed is proportional to the volume of the block (with a multiplicative constant depending on the data dependencies and the number of variables of the algorithm). However, such a partitioning has to meet certain criteria if we want to use it in a compiler, as is described in [Omt90]. First, it should provide fast algorithms, second the space-time decription obtained has to be practical, i.e. the control overhead should be small. Each processor has just to execute single operations at each time step: we have to make sure that the complexity of the implementation depends primarely on these operations and not on the control. Furthermore, in the perspective of a compiler, it is not conceivable to enumerate every point of the domain to generate code. Therefore, it is very desirable that all processors have the same program, because then it will be sufficient to look at only one. Also, this regularity is important, if we want to program a SIMD machine.

This regularity already exists in the domain of computation because of the use of uniform recurrence equations. Indeed, the domain of computation can be seen as a convex subset of Z^n, i.e. a space generated by a set of vectors with integer components, with an inherent regularity. To preserve this important property, we will express the points of the domain with their components in the basis of Z^n, studied in Section 4. In particular, we will use the Hermite form and the Smith form, valid in any principal ideal ring. The following section briefly presents these forms when the ring used is Z. For more details, the reader is refered to [New72].

3 Hermite Form and Smith Form

Definition 1 *A matrix is said unimodular if its components are in Z and if its determinant is equal to* 1 *or* -1.

3.1 Hermite Form

For every matrix A in Z_n, there exist a unimodular matrix Q and a matrix H such as:

- H is an upper triangular matrix with integer, non negative components.
- Each non diagonal component is strictly less than the diagonal component of the same column.
- $A = QH$.

3.2 Smith Form

For every matrix A in Z_n, there exist two unimodular matrices Q_1 and Q_2 and a matrix S such as:

- S is diagonal:
  $$S = (s_1, \ldots, s_r, 0, \ldots, 0)$$
  where r is the rank of A and each s_i strictly positive.
- $\forall i, 1 \leq i \leq r-1$, s_i divides s_{i+1}.
- $A = Q_1 S Q_2$.

Note that if $d_i(A)$ is the greatest common divisor (gcd) of the $\begin{pmatrix} n \\ i \end{pmatrix}^2$ $i * i$ determinantal minors of A and if we put $d_0(A) = 1$ for convenience then $s_i = d_i(A)/d_{i-1}(A)$.

4 Construction of Basis

Definition 2 *Given $F = (\vec{u}_1, \ldots, \vec{u}_p)$, a set of p linearly independant vectors in Z^n, the* lattice *generated by F is the set of points $O + \vec{k}$ where $\vec{k}$ is a linear integer combination of F and O the origin. It is noted $L(F)$.*

Definition 3 *A set of vectors $(\vec{v}_1, \ldots, \vec{v}_p)$ is a basis of the lattice $L(F)$ if every point in the lattice can be expressed in uniquely with integer components in this set.*

In the following, we will identify the set F and the matrix with these vectors as columns. Let G be the n-dimensional canonical lattice in Z^n (in fact Z^n itself). First, we characterize all the basis of G, i.e. all the isomorphisms of the Z-module Z^n.

Lemma 1 *Let P be a $n * n$ unimodular matrix. Then, the column vectors of P form a basis of G.*

proof: If X represents the coordinates in the initial basis and X' the ones in the basis formed by the column vectors of P, then $X = PX'$ and thus $X' = P^{-1}X$. But since P is unimodular, its inverse has integer components. Hence, every point of the lattice can be expressed with integer components X' in the new set of vectors.

□

Theorem 1 *Let P be a $n * n$ matrix, then the column vectors of P form a basis of G if and only if P is unimodular.*

proof: Decompose P in Smith form $P = Q_1 S Q_2$ where Q_1 and Q_2 are unimodular and S diagonal. For the column vectors of P to form a basis, P (and thus S) must be nonsingular. We have $X = Q_1 S Q_2 X'$. Put $Y = Q_1^{-1} X$ and $Y' = Q_2 X'$. Then $Y = SY'$. By the previous lemma, $L(Q_1) = G$ and $L(Q_2^{-1}) = G$, therefore, we can continue the proof for the matrix S and its inverse. All the Y and Y' are simultaneously integer if and only if the diagonal elements of S are 1 or -1, which implies that S is unimodular, just as P.

This proof can be extended for a p-dimensional lattice in a n-dimensional space in using the Smith form for rectangular matrices. The condition on the Smith form gives a constraint on the $n * p$ matrix describing the set of vectors which is the following: the gcd of all the $p * p$ determinantal minors of P must be 1.

□

Note that if F is a basis of G, every subset of F is a basis of the intersection of the subspace (in R^n) that it generates, with G. Now, given a p-dimensional vectorial subspace, how can we find a basis of the intersection of this subspace with G? Just take a vectorial basis of this subspace, represented by a matrix P of rank p with integer components. Decompose it in Hermite form,

$$P = Q \begin{pmatrix} H \\ 0 \end{pmatrix}$$

with Q unimodular. The first p column vectors of Q form the desired basis. Furthermore, if the gcd of all the $p * p$ determinantal minors of P is 1 then H is the identity and the first p column vectors of Q are the column vectors of P. This is an automatic method to complete for example a vector with relatively prime components into a basis of Z^n.

5 Mapping a n-dimensional Problem onto a $(n-1)$-dimensional Array followed by Partitioning

Start from a projection vector, say $\vec{s}$, and a scheduling vector, say $\vec{\tau}$. Each point $\vec{p}$ of the domain of computation will be executed at time $\vec{\tau}.\vec{p}$. Furthermore, we want, after projection, compress the array by a factor c, i.e. merging c virtual processors in a same physical processor as is explained in [DI89]. Of course, these c points must be computed each at a different time.

For not "missing" any computational points in our description and since the domain of computation in a subset of the n-dimensional lattice G, let us describe our transformations only with unimodular matrices. The projections are not necessary orthogonal but oblique, for the deep reason that Z is a ring and not a field. However, to use non unimodular matrices gives nothing but complications because then we have to manipulate rational numbers.

$\vec{s}$ and $\vec{\tau}$ are taken each with relatively prime components to satisfy the conditions given by theorem 1. Complete vector $\vec{s}$ to a basis of G and we obtain a matrix we denote S. Choose $\vec{\tau}$ to have $\vec{\tau}.\vec{s} = c$ which means that the points projected along $\vec{s}$ into the same virtual processor are active every c time steps. Complete the same way vector $\vec{\tau}$ to obtain a unimodular matrix with first *row* vector the vector $\vec{\tau}$. Call its inverse T.

The interpretation of these matrices is as follows: the first column vector of S is $\vec{s}$, the $(n-1)$ following ones form a basis of the hyperplane on which we project, this is the virtual processor array that we want to partition. The $(n-1)$ last column vectors of T are all orthogonal to $\vec{\tau}$ and form a basis of the hyperplane of all the points computed at time 0. The first column vector of T represents the translation vector $\vec{t}$ to go from the hyperplane of the points computed at time t to the one of the points computed at time $(t+1)$, this since $\vec{\tau}.\vec{t}=1$. Naturally, the first component of the product $T^{-1}S$ is c while the matrix $S^{-1}T$ represents the coordinates of the column vectors of T in the basis S. In particular let us focus on the lower right submatrix of size $(n-1)*(n-1)$ of this product that we will denote by M from now on. It represents the projection of the time basis on the virtual array. This matrix gives the position of the points computed at the same time modulo c. For example, if we have:

$$S^{-1}T = \begin{pmatrix} . & . & . \\ . & 2 & 1 \\ . & 1 & 2 \end{pmatrix}$$

then the position of the points computed at time 0 modulo c is the following:

$$\begin{matrix} 0 & . & . \\ . & . & 0 \\ . & 0 & . \end{matrix}$$

First, note that:

- The first row of $T^{-1}S$ gives the time interval along each space vector. In particular, since this row is the first one of a unimodular matrix, its components are relatively prime. This implies that we can find every time, modulo c, on the hyperplan onto which we project, because the $(n-1)$ last components generate Z/cZ. Intuitively, this was obvious.

- The determinant of M is the first component of the inverse of $S^{-1}T$, i.e. c, which means that we slow down the algorithm by a factor c along the projection vector but compress by the same factor.

It is then nice to be able to partition the hyperplan onto which we project with "blocks" ($(n-1)$-dimensional rectangles) of c points to make a compression as regular as possible. The following lemma gives a sufficient condition to allow a covering with given size blocks.

Lemma 2 *If M is upper triangular with diagonal $(d_1, \ldots, d_{(n-1)})$ then we can merge all the points in a box with edges parallel to the axis and of size $(d_1 * \cdots * d_{(n-1)})$.*

proof: Let t be an integer between 0 and $(c-1)$. By the first remark above, there exists a point p_1 projected on the virtual array and computed at time t (modulo c). In moving along the last column vector of T (we add a suitable multiple of the last column of M to p_1), we can find a point p_2 computed at the same time and with last component strictly less than $d_{(n-1)}$. In the same way, we construct p_i from $p_{(i-1)}$ until obtaining a point $p_{(n-1)}$ computed at time t and contained in the box with edges parallel to the axis and of size $(d_1 * \cdots * d_{(n-1)})$. Since, by the second remark, the box contains c points $((d_1 * \cdots * d_{(n-1)}) = c)$, and because we can find in it a point computed at time t for each of the c different times t, all the points in the box are computed at a different time.

$\square$

Then the next question is to know which size of blocks are allowed in the covering given $\vec{s}$, $\vec{\tau}$ and c. The following theorem gives the answer.

Theorem 2 *Given $\vec{s}$, $\vec{\tau}$, c and $(d_1 * \cdots * d_{(n-1)}) = c$ any decomposition of c in integer factors, it is always possible to find a basis of the hyperplane of projection so that all the points in a box with edges parallel to the axis and of size $(d_1 * \cdots * d_{(n-1)})$ are computed at a different time. This implies that we can always compress with the size of boxes we want.*

proof: The gcd of the $(n-2) * (n-2)$ determinantal minors of the matrix M divides the determinant of the matrix $S^{-1}T$ which is 1. Thus, the Smith form of the matrix M is:

$$\begin{pmatrix} Id & 0 \\ 0 & c \end{pmatrix}$$

which proves incidentally that we can always compress along a line. Then, to have:

$$M = \begin{pmatrix} d_1 & 1 & 0 & \ldots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & 1 \\ 0 & \ldots & \ldots & 0 & d_{(n-1)} \end{pmatrix}$$

carry out the following operations on the matrix M:

$x = c$
$for(k = (n-1); k > 1; k--)$
$\{$
  $column(k) \leftarrow -column(k)$
  $column(k) \leftarrow column(k) - x * column(k-1)$
  exchange $column(k)$ and $column(k-1)$
  $row(k) \leftarrow row(k) - d_k * row(k-1)$
  $x = x/d_k$
$\}$

Applying the previous lemma completes the proof, since in carrying out all these transformations, we have just multiplied S and T on the right by unimodular matrices of the form

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

thus without modifying neither $\vec{s}$ nor $\vec{r}$.

$\square$

However, these unimodular transformations are not unique. It is important to use an algorithm preserving "small" values for S^{-1} so that the shape of the domain is not too elongated. Furthermore, since an upper triangular matrix is sufficient, we can still multiply S^{-1} on the left by a unimodular upper triangular matrix to reduce its norm. A good heuristic is to reduce the norm

$$\sup_i \sum_j |M_{i,j}|$$

which is associated to the norm

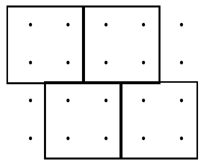
$$\sup_j x_j$$

Be that as it may, even if we don't find the matrix which gives the simplest domain, the method gives in general good results, specially when the dimension of the problem is too large for solving the problem by hand (for $n = 4$, the visualisation of the problem is already difficult). We can also, instead of using the Smith form, use just a Hermite form for the matrix M. In this case, we are sure, if the vector $\vec{s}$ is well chosen, to obtain a "nice" shape. But then, we can't use any size of boxes: it is given by the Hermite form (the following section discuss this problem).

The partitioning is then done by juxtaposing such blocks to cover the whole domain. It is astute to dispose the boxes along the column vectors of M so that every processor has the same program, which simplifies the code generation. For example, if the matrix is equal to:

$$M = \begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix}$$

it will be good to distribute the boxes as following:



Furthermore, the closer the shape of the boxes is to a n-dimensional cube, the smaller is the amount of external communications. Indeed, the number of external communications is proportional to the perimeter of the box whereas its surface area is given by c.

In this section, we have modified the basis of the hyperplane of projection which gives sometimes too elongated domains: for a given space basis, the shape of the boxes we can use depends on the choice of the vector $\vec{\tau}$ and on the arithmetic properties of its components with the choosen space basis. Furthermore, machines such as the Intel iWarp are 2-dimensional arrays, hence it is desirable to be able to project any n-dimensional domain onto a 2-dimensional domain for example, which we will partition. This is the subject of the following section.

6 Projection of a n-dimensional problem onto a p-dimensional Domain followed by Partitioning

Until now, we have considered that all the points computed at the same time belonged to a same hyperplane. However, if we want to project a hyperplane onto a smaller dimensional space, many points will be computed at the same time and in the same physical processor. Thus, it seems to be better to consider the space of the points computed at the same time to be a space of the same dimension as the one onto which we want to project. A larger dimension would lead to collisions, a smaller one to a loss of efficiency. Such a p-dimensional space in a n-dimensional space can be defined as the intersection of $(n-p)$ hyperplanes, represented by their normal vector. To project onto a plane, we will use $n-2$ vectors. For $n=3$, we find again the notion of scheduling vector (the vector $\vec{\tau}$).

Apply all the constraints given by the dependence vectors to the first vector, $\vec{\tau}_1$. Then, in the hyperplane with $\vec{\tau}_1$ as normal vector, all the points could be computed at the same time. Hence, there is no constraint on the other time vectors. For a point $\vec{p}$, we can see the quantities $\vec{\tau}_i.\vec{p}$ as time as if we were using a clock with $(n-p)$ hands, $\vec{\tau}_1$ being the smallest (the hours for example, the others representing minutes, then seconds, ...). The problem is now similar to the problem of the previous section: complete $\vec{\tau}_1$ such that the matrix M (of size $p * p$) has a determinant equal to c so that we can compress by this factor. However, nothing guarantees that the points, assigned to the same physical processor, are computed at different times. We have to choose meticulously the vectors $\vec{\tau}_i$ for $2 \leq i \leq n-p$ and we are going to use the following lemma to construct them.

Lemma 3 *Let a, b, c be three relatively prime integers. Then there exists an integer λ such that $a + \lambda b$ and c are relatively prime.*

proof: c can be decomposed in prime factors:

$$c = \prod_{i=1}^{n} p_i^{s_i}$$

where the p_i are prime integers and the s_i positive integers. Let J be the set of indices i such that p_i divides a and let

$$\lambda = \prod_{i \notin J} p_i^{s_i}$$

If one of the p_i divides $a + \lambda b$ then:

- If p_i divides a then p_i divides λb. But p_i and λ are relatively prime by construction thus p_i divides b. That is not possible for a, b and c are relatively prime.

- If p_i doesn't divide a then p_i divides λ and thus λb. It follows that p_i divides a, which is not possible either.

Hence, $a+\lambda b$ and c are relatively prime.

$\square$

Note that if $\vec{s}_1,\ldots,\vec{s}_n$ is the set of vectors giving the projection (we project along $\vec{s}_1,\ldots,\vec{s}_{n-p}$ onto the space $\vec{s}_{n-p+1},\ldots,\vec{s}_n$), we can add to these p last vectors any linear integer combinations of the first ones without changing the shape of the domain after projection. Let $\vec{p}$ be the first row vector of $T^{-1}S$, i.e. $\vec{\tau}.\vec{s}_1,\ldots,\vec{\tau}.\vec{s}_n$. All the components of $\vec{p}$ are relatively prime. In taking $c = gcd(\vec{\tau}.\vec{s}_1, \vec{\tau}.\vec{s}_{n-p+1},\ldots,\vec{\tau}.\vec{s}_{n-1})$, $b = gcd(\vec{\tau}.\vec{s}_2,\ldots,\vec{\tau}.\vec{s}_{n-p})$ and $a = \vec{\tau}.\vec{s}_n$ in the previous lemma, we can transform S (without changing the shape of the domain after projection) so that the p last components of $\vec{p}$ and the first one (which is c) are relatively prime.

Now construct P of size $(n-p)*n$ as following:

$$P = \begin{pmatrix} \vec{\tau}_1.\vec{s}_1 & \ldots & \vec{\tau}_1.\vec{s}_{n-p+1} & \ldots & \vec{\tau}_1.\vec{s}_n \\ 0 & Id & 0 & \ldots & 0 \end{pmatrix}$$

By construction, the gcd of the $(n-p)*(n-p)$ determinantal minors is equal to 1, so we can complete this matrix P into a unimodular square matrix P'. Then define $T^{-1} = P'S^{-1}$. Let us focus, as in the previous section, on the inverse of P'. The submatrix M (of size $p*p$ this time) has a determinant equal to the $(n-p)*(n-p)$ upper left determinantal minor of P, i.e. c. Furthermore, its determinantal minors of smaller size are 1 since we have completed P with the identity and some zeros. We are again in the same situation as in the previous section. In transforming the matrix M in its Hermite form (in manipulating the p last column vectors of T), we obtain an upper triangular matrix M which gives the way to partition the domain. Furthermore, all the points in these boxes are computed at different times. Indeed, take two points $\vec{p}$ and $\vec{q}$, in the same box which are computed at the same time:

$$\vec{p} = \sum_{i=1}^{n} \lambda_i \vec{s}_i$$

and

$$\vec{q} = \sum_{i=1}^{n} \beta_i \vec{s}_i$$

These two points are computed at the same second, at the same minute, etc, ... In using the matrix P, each equality gives a condition on the components of these vectors. In particular,

$$\forall i, 2 \leq i \leq n-p, \lambda_i = \beta_i$$

Since $\vec{\tau}.\vec{s}_1$, $\vec{\tau}.\vec{s}_{n-p+1}$, $\ldots$, $\vec{\tau}.\vec{s}_n$ are relatively prime and since the determinant of M is $\vec{\tau}.\vec{s}_1 = c$, we can use the same proof as in the previous section to prove that all the points in the same box (this time computed at the same time except for the hand corresponding to $\vec{\tau}_1$) are computed (modulo c) at different times. This implies that $\vec{p} = \vec{q}$.

The choice of the vector $\vec{\tau}$ determines the size of a box. We have to choose the vector $\vec{\tau}$ in a polytope, described by the constraints given by the inner products of $\vec{\tau}$ with the dependence vectors. We have already added the constraint $\vec{\tau}.\vec{s}_1 = c$. Is it possible to add a small constraint to foresee the size of boxes allowed ? The case $p = 1$ doesn't offer the choice (the boxes are of size c in the linear array), so look at the case $p = 2$ (for an 2-dimensional array processor):

Lemma 4 *Let d be the gcd of c and $\vec{\tau}_1.\vec{s}_{n-1}$. Then the first component of the Hermite form of M is c/d.*

proof: If we represent the times of computation (modulo c) on the virtual processors along $\vec{s}_{n-1}$, from a point computed at time 0, we find 0 then $\vec{\tau}_1.\vec{s}_{n-1}$, then $2\vec{\tau}_1.\vec{s}_{n-1}$, etc .... More precisely, modulo c, these times form a cyclic subgroup of Z/cZ. Its cardinal is c/d. With c/d movings along $\vec{s}_{n-1}$, we find again zero. This implies that one of the basis vector of the plane of the points computed at the same time is projected along $\vec{s}_1$ onto the vector $(c/d)\vec{s}_{n-1}$, i.e. the first component of the Hermite form of M is c/d.

$\square$

So it is sufficient to add a constraint on $\vec{\tau}_1.\vec{s}_{n-1}$ to obtain the desired size of boxes. Usually, this constraint is small. For the band matrix multiplication for example, we choose:

$$\vec{s}_1 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

which we complete into:

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

If we want a latency greater than 3, for example, we can take the vector $\vec{\tau} = \begin{pmatrix} 10 \\ 3 \\ 3 \end{pmatrix}$ but the vector $\vec{\tau} = \begin{pmatrix} 9 \\ 4 \\ 3 \end{pmatrix}$ will be as good. It will allow us to use $4 * 4$ boxes for partitioning the domain and preserves a rectangular domain.

7 Examples

Let us look at simple example: the multiplication of two full matrices of size $n * n$, $C = A * B$ on a 2-dimensional array, given by the equations:

$$\begin{array}{l} a(i,j,k) = a(i,j-1,k) \\ b(i,j,k) = b(i-1,j,k) \\ c(i,j,k) = c(i,j,k-1) + a(i,j,k) * b(i,j,k) \end{array}$$

Choose to keep the matrix C within the array and compress by a factor $c = d^2$ to simplify. Take:

$$S = \begin{pmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad S^{-1} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & -1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

Our method gives:

$$T^{-1} = \begin{pmatrix} 1 & d & c \\ 0 & 0 & 1 \\ 0 & -1 & -d \end{pmatrix} \quad T = \begin{pmatrix} 1 & 0 & d \\ 0 & -d & -1 \\ 0 & 1 & 0 \end{pmatrix}$$

which gives:

$$S^{-1}T = \begin{pmatrix} 0 & 1 & 1 \\ 0 & d & 1 \\ 1 & 0 & d \end{pmatrix}$$

Now we have to choose a way to cover the domain of the virtual processors (a $n * n$ square) with the boxes (of size $d * d$). If we want to use an SIMD machine to program this algorithm, we will shift the boxes as explained in section 5, such that every processor computes and communicates at the same time and with the same neighboor. If the machine is a rectangular array, we might use more processors than we need, but this loss is not very important. Actually, we will use an array of width n/d and length

$n/d + n/c$, i.e $(n^2/c)(1 + 1/d)$ processors which is a small loss with regard to n^2/c. Each processor will use c units of memory for each variable and a program formed by a loop of c steps (because all the operations are identical for the same time modulo c). The execution time is of the order of $(1 + d + c)n$. However, if we have a MIMD machine, we are not obliged to shift the boxes. The processors of a same line will have the same program (a cycle of c steps), th! e program of each line being diff

Look at the case of a compression factor equal to 4. The algorithm will be executed in $7n^2$ time steps with $n/4$ cells, i.e. with an efficiency equal to 4/7. With regard to the usual algorithm which requires $3n^2$ time steps with n cells, the four following cells (figure 1) are replaced by the cell in figure 2. The intersection of the wires at the entrance of the multiplier are multiplexers which let the left data or the right data through (according to a bit of control coming with the data for example). At any time, only one of this data is meaningful. The data of the matrices A and B come in the cells as follows:

$$\ldots \quad - \quad - \quad a_{1,1} \quad a_{0,1} \quad - \quad - \quad a_{1,0} \quad a_{0,0} \quad \rightarrow$$

on the first row and

$$\ldots \quad b_{1,1} \quad - \quad b_{1,0} \quad - \quad b_{0,1} \quad - \quad b_{0,0} \quad \rightarrow$$

on the first column.

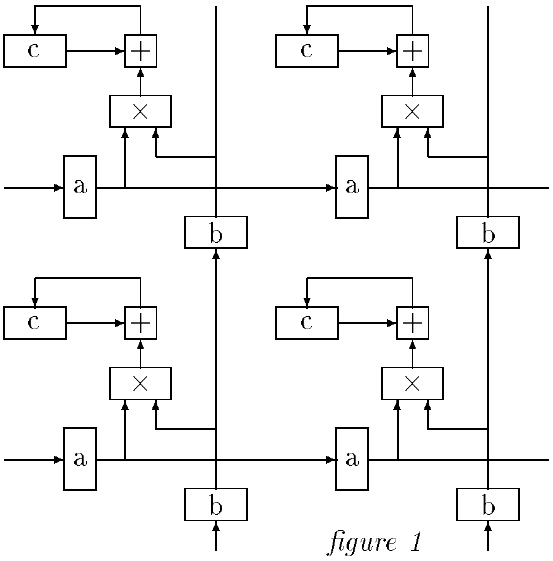


figure 1

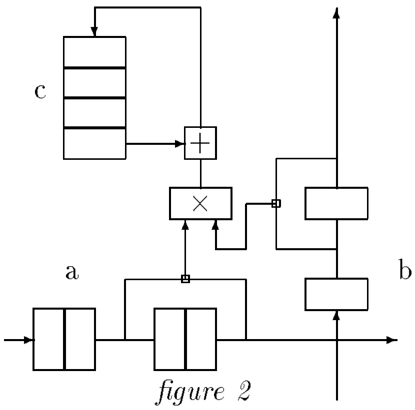


figure 2

Let us now look at an example using the results of section 6, the projection of a n-dimensional domain of computation onto a domain of dimension strictly less than $(n-1)$. To simplify, although we have 2-dimensional arrays, project the domain of the multiplication of matrices onto a one dimensional space, to give an idea of what a projection of a 4-dimensional domain onto a 2-dimensional domain will be. For the data a, b and c to circulate from the left to the right on the linear array, take the last row of S^{-1} equal to $\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$. Complete this vector into a unimodular matrix and we obtain:

$$S = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & -1 & 1 \end{pmatrix} \quad S^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$$

Take a time vector $\vec{\tau}$ satisfying:

$$\begin{aligned} \vec{\tau}_1 &> 0 \\ \vec{\tau}_2 &> 0 \\ \vec{\tau}_3 &> 0 \\ \vec{\tau}.\vec{s}_1 &= c \end{aligned}$$

for example $\vec{\tau} = \begin{pmatrix} c+1 & 1 & 1 \end{pmatrix}$ The method gives:

$$T^{-1} = \begin{pmatrix} c+1 & 1 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix} \quad T = \begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & -1 & c+1 \end{pmatrix}$$

which gives

$$S^{-1}T = \begin{pmatrix} 0 & 0 & -1 \\ 0 & 1 & 0 \\ 1 & 0 & c \end{pmatrix}$$

The vector representing the second hand of the clock $\vec{\tau} = \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$ shows that we can consider there are n secondes in a minute. The execution time is then of the order of $(c+2)n^2$ for a number of cells of the order of $3n/c$. We have also to put the data at the right place at time 0 which requires $2n$ time steps, and the same for the data c to down load from the array after the last computation.

8 Conclusion

The method presented in this paper was built such that it can be used in a compiler for programmable array processors, with other tools manipulating the convex domains of computations. In particular, it does not look at the search for the "good" projection vector, but uses it as an input data. The search for an optimal scheduling vector is also not adressed, but can be found through linear programming, for example.

However, the way, given in this paper, of interpreting the matrices S, T, $S^{-1}T$ and their inverses allow us to construct automatically matrices of projection with integer components and "basis" of partitioning. Furthermore, all the useful information for the description of the algorithm is readily available in these matrices. Finally, the regularity of the times of computation in the boxes will permit to generate the code, automatically, which is needed to implement the algorithms on array processors with a relatively large memory, in a time proportional to the volume of a box, i.e. to the factor of compression. This is fairly easy to do when we project n-dimensional domains onto $(n-1)$-dimensional array processors but seems to be a bit more complicated for smaller dimensional array processors.

Finally, note that the method proposed in section 6 (when we use several time vectors) is not optimal for two reasons:

First, all the constraints given by the dependence vectors are applied to the first time vector, which is a strong restriction. For example, for the multiplication presented previously, we can substract the second row of T^{-1} to the first one and obtain an algorithm which is faster and satisfies the constraints on the dependence vectors. But how to find this automatically and efficiently (without enumeration) is not solved, yet. A solution might beM to use a heuristic similar to the one proposed in section 5.

Second, if the width of the domain along the second time vector gives the number of seconds in a minute, this is not optimal either. We should take the maximal width (along the second time vector) of the intersection of the domain with an hyperplane orthogonal to the first time vector.

However, in spite of these defects which should be examined more closely, the method gives, in general, encouraging results.

□

I would like to thank Theo Omtzigt for his help and the time he lost to answer my questions

References

[DI89] Jean-Marc Delosme and Ilse C.F. Ipsen. Parallel computation of algorithms with uniform dependences. *SIAM*, 1989. Yale University, Department of Computer Science.

[MF86] D.I. Moldovan and J.A.B Fortes. Partitioning and mapping of algorithms into fixed size systolic arrays. In *IEEE Trans. on Computers*, volume 35, pages 1–12, January 1986.

[New72] Morris Newman. *Integral Matrices.* Academic Press, 1972.

[Omt90] E.T.L. Omtzigt. Domain flow and streaming architectures. *ASAP*, 1990. Yale University, Department of Electrical Engineering.