Systematic Design of Regular VLSI Processor Arrays

Jichun Bu

Delft University of Technology

May 1990

Systematic Design of Regular VLSI Processor Arrays

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft, op gezag van de
Rector Magnificus Prof. drs. P.A. Schenck,
in het openbaar te verdedigen ten overstaan van een
commissie aangewezen door het College van Dekanen
op dinsdag 22 Mei 1990 te 14.00 uur

door

Jichun Bu

geboren te Changchun
electrotechnisch ingenieur

Dit proefschrift is goedgekeurd door de promotor
Prof. dr. ir. P. Dewilde

Dr. ir. Ed F. Deprettere heeft als begeleider in hoge mate bijgedragen aan het totstandkomen van het proefschrift

To ChongYi
and to my parents

Contents

Summary

The appearance of low cost, high-density, and fast very-large-scale integrated (VLSI) devices makes it becomes increasingly attractive and practical to design efficient application-specific super-computing systems. Such systems will offer rapid solutions of large computational problems in science and engineering. Foremost among them are regular VLSI processor arrays, especially systolic/wavefront arrays, which are shown to be extremely useful for the real-time solution of large problems in, e.g., modern digital signal processing, numerical algebra, graph theory, computer graphics, and many others.

In this thesis we present a systematic way to design regular VLSI processor arrays in particular systolic arrays. The following themes are addressed: (a) mapping regular algorithms onto fixed-size systolic arrays, (b) converting imperative nested loop programs into regular algorithms, and (c) designing a VLSI parallel system for computer image synthesis.

This thesis is subdivided in five chapters, of which the first is introductory, and the last consists of some concluding remarks.

Chapter 2 deals with the problem of the systematic design of fixed-size arrays for regular algorithms (RAs). Usually, systolic arrays are designed to solve problems in a one-pass computation; the size of an array is adjusted to suit the size of the problems to be solved. (Such arrays are therefore also called full-size arrays.) Although a full-size systolic array can be extended modularly to include more processors when the size of problems grows, the size of a physical array cannot be arbitrarily large since it is constrained by many factors, such as the cost of the array, the number of I/O pins that the array may have, and other technological factors. A systematic method

for designing fixed-size systolic arrays for RAs is presented in chapter 2. It unifies and generalizes various array reduction techniques developed in the literature. The method consists of the following steps: (i) the conceptual derivation of a full-size array for a given RA, (ii) the partitioning of the index space of the RA into subsets such that the subsets can be computed serially using only a portion of the full-size array, which we call a "reduced-size" array, and (iii) the reduction of the size of the reduced-size array using "processor clustering" techniques. We present two processor-clustering techniques, which can be used to achieve design objectives such as transforming inefficient arrays into efficient arrays, reducing the size/dimension of an array, and balancing the local memory and external communication of processors.

Chapter 3 deals with the problem of systematic conversion of imperative nested loop programs into RAs. In RAs, data dependences are given explicitly in a localized form (each computation in an RA depends only on some neighboring computations). However, in nested loop programs, data dependences are generally hidden, not explicit. Therefore, a nested loop program can only be transformed into a regular algorithm if (1) data dependences in the program can be extracted, and expressed explicitly, and (2) the extracted data dependences can be localized. To convert a nested loop program into an RA, we first transform the program into a single-assignment form, in which all data dependences are expressed explicitly in the form of algebraic expressions. Then we show when and how the data dependences can be localized in order to obtain an RA version of the program. The method provides also a systematic way to analyze data dependences in imperative nested loop programs. We believe that our approach is complementary to those used in parallelizing/vectorizing compilers; instead of searching for independent iterations or statements (i.e., parallelism), we detect iterations that are strictly dependent on each other. It is important to know which computations depend on which in mapping algorithms onto systolic arrays since they have only limited inter-processor communication capability.

Chapter 4 presents a parallel VLSI architecture for the fast synthesis of high-quality realistic computer images, using an algorithm called the "radiosity method" in computer graphics. We focus on the reformulation of the algorithm in order to

make it more suitable for VLSI array processing. In the radiosity method, a fairly time-consuming step is the generation of a matrix of "form-factors" which is used to describe the (light) energy balance within a closed environment. A ray-tracing algorithm for the computation of the form-factors is presented. Since rays can be traced independently, a high degree of parallelism can be achieved. An architecture for the ray-tracing algorithm using pipelined Cordic processors and an architecture for the generation of matrix of form-factors are presented.

Chapter 1

Introduction

1.1 VLSI array processing

High-performance computers are increasingly in demand in many areas of science and engineering applications, e.g., digital signal and image processing, robotics, radar, numerical analysis, structural analysis, weather forecasting, computer graphics, simulations, petroleum exploration. Current advanced computer architectures are centered around the concept of parallel and pipelined processing. State-of-the-art parallel computing system can be characterized into three structural classes [HB85]: pipelined (or vector) computers, multiprocessor system, and array processors. These super-power computers provide a cost-effective means to achieve high system performance.

When large computations have to be performed in real-time, it becomes increasingly practical and more advantageous to use a special-purpose parallel computer system dedicated to one specific application thanks to the rapid advent of VLSI technology. Such application-specific systems can achieve (nearly) optimal utilization of available resources by considering both the algorithm and the hardware design at the same time.

To be suitable for VLSI implementation, a special-purpose system should have the following structural properties [Kun82]:

Simple and regular design. Cost-effectiveness has always been a chief concern in

designing special-purpose systems; their cost must be low enough to justify their limited applicability. The cost of special-purpose systems is dominated by their design costs. Simple and regular design will yield cost-effective special-purpose system.

Concurrency and communication. There are essentially two ways to build a fast computer system. One is to use fast components, and the other is to use concurrency. The technological trend clearly shows a diminishing growth rate for component speed, and major improvement in computation speed must come from the concurrent use of many processing elements. The degree of concurrency in a special-purpose system is largely determined by the underlying algorithm, and massive parallelism can be achieved when the algorithm is designed to introduce a high degree of pipelining and multiprocessing. The issue here is to design algorithms that support a high degree of concurrency, and in the meantime employ simple, regular communication and control.

Balancing computation with I/O. A special-purpose system will typically receive data and output results through an attached host. Therefore, I/O considerations influence overall performance. The ultimate performance goal of a special-purpose system is—and should be no more than—a computation rate that balances the available I/O bandwidth with the host. The design of a special-purpose system should be modular so that its structure can be easily adjusted to match a variety of available I/O bandwidths.

To meet these challenges, systolic array architecture has been introduced for high-speed VLSI implementation of application-specific and computation-intensive algorithms [KL80], [Kun79], [Kun80b], [Kun82], [Kun84].

1.2 What is a systolic array ?

There are a number of quantitative and qualitative definitions of systolic arrays in the literature, [KL80], [Kun82], [LS83], [Kun84], [Ull84], [Hel85]. Perhaps the first qualitative definition is given by Kung and Leiserson in [KL80]:

A systolic system is a network of processors which rhythmically compute and pass

data through the system. Physiologists use the word "systole" to refer to the rhythmically recurrent contraction of the heart and arteries which pulses blood through the body. In a systolic computing system, the function of a processor is analogous to that of the heart. Every processor regularly pumps data in and out, each time performing some short computation, so that a regular flow of data is kept up in the network.

Or more precisely [Kun88]:

A systolic system is a computing network possessing the following features:

- *Synchrony.* The data are rhythmically computed (timed by a global clock) and passed through the network.
- *Modularity and regularity.* The array consists of modular processing units with homogeneous interconnections. Moreover, the computing network may be extended indefinitely.
- *Spatial locality and temporal locality.* The array manifests a locally communicative interconnection structure, i.e., spatial locality. There is at least one unit delay allotted so that signal transactions from one processing unit to the next can be completed, i.e., temporal locality.
- *Pipelinability.* The array exhibits a linear rate pipelinability, i.e., it should achieve an O(N) speedup in terms of processing rate, where N is the number of processing elements.

In this thesis, we use the following definition of a systolic array, which is consistent with the definition given in [Rao85] (see also [RK86], [RK88a]):

A systolic array is a network of processors in which processors can be placed at integer points of a subset $\tilde{I}$ (called processor space) of $\mathbf{Z}^n$, where $\mathbf{Z}$ is the set of all integers, so that

- if there is a direct link from (to) the processor at location I to (from) the processor at location $I+d$, $d \in \mathbf{Z}^n$, then

(a) there is such a link for every processor $I \in \tilde{\mathcal{I}}$, and

(b) d is independent of the size of the problem to be solved.

If $I+d$ falls outside the processor space, then it is an output (input) to (from) the external environment.

- The number of distinct displacement vectors d's is independent of the size of the processor space $\tilde{\mathcal{I}}$.
- Each processor executes repeatedly a set of instructions with a period Δ which is independent of the size of array.
- Each processor has m inputs IN_i, $i=1,...,m$, and m outputs OUT_i, $i=1,...,m$. If a processor receives a value at time t at an input IN_i, then it places an output value at time $t+\Delta$ at output OUT_i, and at same time, IN_i receives the next input value (if any).

1.3 How to design systolic arrays?

In the literature, various design concepts and methods can be found for the systematic design and analysis of systolic arrays, including [Kun80a], [Qui84], [CS84], [Mol83], [MW84], [MF86], [Che86], [DI86], [Kun84], [Kun88]. Detailed discussion and comparison of approaches for systolic array design can be found in [FFW85], [RK86], [RK88a], [Kun88]. Based on the pioneering work of Karp, Miller, and Winograd [KMW67], Rao *et al.* have found that algorithms which are implementable on systolic arrays belong to a subclass of algorithms called "regular iterative algorithms" (RIAs) [RK86], [Rao85], [RK88a]. All these results have pointed out that for an algorithm to be implementable on a systolic array it must possess regular data dependences. We will refer to such algorithms as *regular algorithms* (RAs).

We define an RA as a triple $< \mathcal{I}, F, \mathcal{D} >$, where

- $\mathcal{I}$ is called the index space of the algorithm. It contains all the points with integer coordinates enclosed in a convex subset of the n-dimensional Euclidean space $\mathbf{R}^n$, where $\mathbf{R}$ is the set of reals.

- F is an input-output mapping (possibly depending on I):

$$F : (in_1(I), ..., in_m(I)) \rightarrow (out_1(I), ..., out_m(I)).$$

- $\mathcal{D} = \{d_i | i = 1, ..., m\}$ is a set of n-vectors of integers (integral vectors), representing dependences between computations; they are usually called dependence vectors.

- Each index point $I \in \mathcal{I}$ represents a computation, and each computation I requires m inputs, denoted by $in_1(I), ..., in_m(I)$, and generates m outputs, denoted by $out_1(I), ..., out_m(I)$. The dependences between computations are regular, meaning that each computation $I \in \mathcal{I}$ depends on the m computations $I - d_i$, $i = 1, ..., m$. They are defined as follows.

$$\text{for each } I \text{ in } \mathcal{I}: \quad \begin{cases} in_1(I) & = & out_1(I - d_1); \\ \ldots & = & \ldots; \\ in_m(I) & = & out_m(I - d_m); \end{cases}$$

  We assume that $in_i(I)$ is an input of the algorithm if $I - d_i$ is not in the index space $\mathcal{I}$. Similarly, we assume that $out_i(I - d_i)$ is an output of the algorithm if I is not in the index space $\mathcal{I}$.

Note that we define F merely as an input-output mapping, and its actual implementation is immaterial.

Mapping an RA onto systolic arrays

Assume that each computation $I \in \mathcal{I}$ takes one unit of time (one clock cycle). We can use the result of [Mol83], [Rao85], [RK88a], [MF86] to map an RA onto full-size systolic arrays (when possible) as follows (see Chapter 2 for more details).

First, determine a coprime vector $\lambda \in \mathbf{Z}^n$ such that $\lambda^t d_i \geq 1$ for all $i = 1, ..., m$, where the superscript t denotes the vector transposition. (A nonzero integral vector is called a coprime vector if the greatest common divisor of its components is equal to 1.)

Next, determine a coprime vector $u \in \mathbf{Z}^n$ such that $\lambda^t u \neq 0$, and an $(n-1) \times n$ matrix of integers (integral matrix) P of rank $n-1$ such that $Pu = 0$.

In terms of λ, u, and P, a full-size systolic array for the algorithm can be derived as follows.

- The set of processors of the array is given by the set $\tilde{\mathcal{I}} = \{p | p = PI, I \in \mathcal{I}\}$, $\tilde{\mathcal{I}} \subset \mathbf{Z}^{n-1}$. The efficiency of each processor is defined as $1/|\lambda^t u|$.
- The functionality of the processors is defined by the input-output mapping F.
- The processors are regularly interconnected, and the set of interconnection primitives is given by $\tilde{\mathcal{C}} = \{(\lambda^t d_i, Pd_i) | i = 1, ..., m\}$.

See chapter 2 (section 2.2) for more details.

In summary, we can follow the following trajectory to design systolic arrays:

Problems $\Longrightarrow$ Algorithms $\Longrightarrow$ RAs $\Longrightarrow$ Systolic arrays

The systolic arrays obtained are usually referred to as full-size arrays because their sizes depend generally on the size of the index space of the RAs.

The following questions arise: What if the efficiency of the processors are not 100%? What if the size of the array is too large? What if we only have a fixed number of processors? Generally, the size of the full-size array will depend on some parameters. How to design a systolic array with fixed number of processors to solve all the instances (where the value of the parameters is fixed) of the problem? These questions are answered in chapter 2.

Moreover, what if an algorithm is not given in RA form? An imperative nested loop program is a typical construct that is widely used in the area of scientific computing; there are a large number of algorithms in this form. How to convert such a program (e.g., a do-loop program in Fortran) into an RA? This problem is treated in chapter 3.

1.4 An overview of the thesis

Chapter 2 presents a systematic method to design *fixed-size* systolic arrays for RAs. It unifies and generalizes various array reduction techniques developed in the literature. The method consists of the following steps: (i) deriving conceptually a full-size array for a given RA, (ii) partitioning the index space of the RA into subsets such that the subsets can be computed serially using only a portion of the full-size array, which we call a "reduced-size" array, and (iii) reducing the size of the reduced-size array using "processor clustering" techniques. We present two processor-clustering techniques, which can be used to achieve design objectives such as transforming inefficient arrays into efficient arrays, reducing the size/dimension of an array, and balancing the local memory and external communication of processors.

Chapter 3 is devoted to the conversion of imperative nested loop programs into regular algorithms. A systematic procedure is described. To convert a nested loop program into an RA, we first transform the program into a single-assignment form, in which all data dependences are expressed explicitly in the form of algebraic expressions. Then we discuss when and how the data dependences can be localized in order to obtain an RA version of the program. The method provides a systematic way to analyze data dependences in imperative nested loop programs.

Chapter 4 is about an application. We present a highly parallel VLSI architecture for the fast synthesis of high-quality realistic computer images, using an algorithm called "radiosity method" in computer graphics. We focus on the reformulation of the algorithm in order to make it suitable for VLSI array processing. In the radiosity method, a fairly time-consuming step is the generation of a matrix of "form-factors" which is used to describe the (light) energy balance within a closed environment. A ray-tracing algorithm is presented for the computation of the form-factors. Since rays can be traced independently, a high degree of parallelism can be achieved. An architecture for the fast implementation of the ray-tracing algorithm using pipelined Cordic processors and an architecture of the generation of matrix of form-factors are presented.

Chapter 2

Design of Fixed-Size Systolic Arrays

2.1 Introduction

In this chapter we present a methodology to design systolic arrays that are suitable for solving problems of arbitrary sizes. Usually, systolic arrays are designed to solve problems in a one-pass computation; the size of an array is adjusted to suit the size of the problems to be solved. Such arrays are called full-size arrays. They can be extended modularly to include more processors when the size of problems grows. However, the size of an (physical) array cannot be arbitrarily large in practice. It is constrained by many factors, such as the cost of the array, the number of I/O pins that the array may have, and some other technological factors [MF86]. From this a problem arises: how to design a systolic array using a fixed number of processors (called a fixed-size array) to solve problems of different sizes? This problem is often referred to as the partitioning problem in the systolic array design. Solving this problem is not trivial since the following factors must be taken into account [Jai86], [MF86], [Kun88]: (1) the minimum overall computation time, (2) the minimum control overhead, and (3) a balanced tradeoff between external communication and the amount of local memory of processors.

We present in this chapter a methodology to design fixed-size arrays for iterative algorithms with regular data dependences (we will refer to such algorithms as *regular*

algorithms (RAs), see section 2.2). The method consists essentially of the following steps. For a given RA, we first determine a full-size array using the space-time partitioning method as described in, e.g., [MF86], [RK88a]. Since one such an algorithm solves a class of problems which differ only in their sizes, the size of the full-size array may also vary depending on the size of the problems to be solved. Then we partition the algorithm's index space into subsets such that the subsets can be computed serially using only a portion of the full-size array. This portion, which we call a reduced-size array, can be derived directly from the partitioning of the index space. When necessary, for instance, the size of the reduced-size array is still problem-size dependent, we use "processor-clustering" techniques, which maps essentially several processors into one processor, to reduce the size of the array to a constant.

We present two processor-clustering techniques, which can be used to achieve the following design objectives: (1) to transform inefficient arrays into efficient arrays, (2) to reduce the size of an array (by an arbitrary factor), (3) to reduce the dimension of an array, and (4) to balance the local memory and external communication of processors. Moreover, a technique is described to cluster processors in such a way that the number of I/O pins of the resulting processor is independent of the number of processors that are clustered. (The number of registers in a processor after the clustering will, however, increase with the number of processors that are clustered.)

The question of how to solve problems of arbitrary sizes on systolic arrays with a limited number of processors has been addressed by several authors. Particular solutions have been described for problems such as matrix LU factorization, matrix inversion [HC81], matrix QR factorization [Joh82], [Hel85], [MWF84], and a few other problems in the areas of linear algebra and digital signal processing [SK85], [NLV87]. However, techniques used to derive these solutions do not provide a general methodology to solve the partitioning problem.

To solve the partitioning problem in general, two methods called LPGS (locally parallel and globally sequential) and LSGP (locally sequential and globally parallel) are proposed in [Jai86]. These two methods have been expanded later in [ND88], [Jai89]. However, only partial solutions were obtained for when the index space of an

algorithm is of dimension two or three. In [HNS87], a method is described to design systolic arrays under the constraint of the number of processors. It can be shown that this method is equivalent to the "clustering" method described in [ND88] in three-dimensional index spaces [Jai89]. In [MF86], a partitioning technique, which is similar in nature to the LPGS partitioning strategy, is described. Using this method, a fixed-size array can be obtained by dividing the index space of an algorithm into bands and then mapping the bands to a processor space. See also [Kun88] for a similar approach. However, these methods are limited to when there is no counter data flow between the bands. Our method is a unification and generalization of the above methods.

The rest of this chapter is organized as follows. In section 2.2 we introduce the class of regular algorithms we deal with, and we briefly describe a procedure for the mapping of these algorithms onto full-size systolic arrays. In section 2.3 we describe two processor clustering techniques called "passive-clustering" and "active-clustering" respectively. The passive-clustering can be used to transform an inefficient array into an efficient array. The active-clustering can be used to reduce the size and/or dimension of an array. We show that the active-clustering can, moreover, be used to balance the amount of the local memory and I/O operations of processors. However, when the active-clustering is used only to design a fixed-size systolic array, the processors will have large amount of local memory in general, depending on the size of the algorithm's index space. This problem can be solved (partially) by first partitioning the algorithm's index space into subsets so that they can be executed serially on a reduced-size array, and then mapping the reduced-size array to a fixed-size array. The partitioning of index space and its properties are discussed in section 2.4. In section 2.5, we summarize our discussion in a design procedure.

2.2 Mapping RAs onto full-size systolic arrays

A regular algorithm (RA) can be characterized by a triple $< \mathcal{I}, F, \mathcal{D} >$, where $\mathcal{I}$ is called the index space of the algorithm. It contains all the points with integer

coordinates enclosed in a convex subset of the n-dimensional Euclidean space $\mathbf{R}^n$, where $\mathbf{R}$ is the set of reals. F is an input-output mapping (possibly depending on I):

$$F : (in_1(I), ..., in_m(I)) \rightarrow (out_1(I), ..., out_m(I)).$$

$\mathcal{D} = \{d_i | i = 1, ..., m\}$ is a set of n-vectors of integers (integral vectors), representing dependences between computations; they are usually called dependence vectors.

Each index point $I \in \mathcal{I}$ represents a computation, and each computation I requires m inputs, denoted by $in_1(I), ..., in_m(I)$, and generates m outputs, denoted by $out_1(I), ..., out_m(I)$. The dependences between computations are regular, which means that each computation $I \in \mathcal{I}$ depends on the m computations $I - d_i$, $i = 1, ..., m$. They are defined as

$$\begin{cases} in_1(I) &= out_1(I - d_1); \\ \ldots &= \ldots; \\ in_m(I) &= out_m(I - d_m); \end{cases}$$

for each I in $\mathcal{I}$. We assume that $in_i(I)$ is an input of the algorithm if $I - d_i$ is not in the index space $\mathcal{I}$. Similarly, we assume that $out_i(I - d_i)$ is an output of the algorithm if I is not in the index space $\mathcal{I}$.

Note that when the mapping F is defined by a set of equations, then such an algorithm represents in fact a set of uniform recurrent equations [KMW67]; it is also called an RIA (regular iterative algorithm) in [Rao85], [RK88a]. However, F can also be defined procedurally as a computer (do-loop) program, which is the case considered in [Mol83], [MF86].

Assume that each computation $I \in \mathcal{I}$ takes one unit of time (one clock cycle). Then we can use the theory of RIAs [Rao85], [RK88a], to map an RA $< \mathcal{I}, F, \mathcal{D} >$ to a systolic array (when possible) as follows.

First, determine a coprime vector $\lambda \in \mathbf{Z}^n$ (called a scheduling vector) such that $\lambda^t d_i \geq 1$ for all $i = 1, ..., m$, where the superscript t denotes the vector transposition. (A nonzero integral vector is called a coprime vector if the greatest common divisor of its components is equal to 1.) In terms of λ, a computation $I \in \mathcal{I}$ will be scheduled at time $t = \lambda^t I$. The execution of the algorithm will start at time $t_{min} = min_{I \in \mathcal{I}} \lambda^t I$,

and terminate at time $t_{max} = max_{I \in \mathcal{I}} \lambda^t I$. All the index points on a (hyper-)plane $\lambda^t I = \alpha$, $t_{min} \leq \alpha \leq t_{max}$, are scheduled at the same time $t = \alpha$; they can be executed in parallel.

Next, determine a coprime vector $u \in \mathbf{Z}^n$ (called a projecting vector or an iteration vector) such that $\lambda^t u \neq 0$, and an $(n-1) \times n$ matrix of integers (integral matrix) P of rank $n-1$ such that $Pu = 0$.

In terms of λ, u, and P, a full-size systolic array of the algorithm can be derived as follows. The set of processors (otherwise called the processor space) of the array is given by the set $\tilde{\mathcal{I}} = \{p | p = PI, I \in \mathcal{I}\}$, $\tilde{\mathcal{I}} \subset \mathbf{Z}^{n-1}$. Each computation $I \in \mathcal{I}$ will be executed by (mapped to) a processor $p = PI$ at $t = \lambda^t I$. The functionality of the processors is defined by the input-output mapping F. Each processor p is assumed to have m inputs and m outputs; each input of p corresponds to a unique input of F, and each output of p corresponds to a unique output of F. We denote the inputs and outputs of a processor p by $in_1(p), ..., in_m(p)$ and $out_1(p), ..., out_m(p)$ respectively, and we assume that they correspond to the inputs and outputs of F of the same names. The processors are regularly interconnected, using a set of interconnection primitives $\tilde{\mathcal{C}} = \{(\lambda^t d_i, Pd_i) | i = 1, ..., m\}$; for each $p \in \tilde{\mathcal{I}}$, there is an interconnection from $out_i(p)$ to $in_i(p + Pd_i)$ with $\lambda^t d_i$ delays, for all $i = 1, ..., m$.

From now on we denote such a full-size array by a pair $< \tilde{\mathcal{I}}, \tilde{\mathcal{C}} >$.

The efficiency of each processor is defined as $1/|\lambda^t u|$ [Rao85]. The reason is the following. Let $I \in \mathcal{I}$ be an arbitrary index point, and let $p = PI$. Then processor p will compute I at time $t = \lambda^t I$, and it will be ready to take the next computation at $t = \lambda^t I + 1$. However, the next computation that will be executed by p is either $I + u$ (if $\lambda^t u > 0$) or $I - u$ (if $\lambda^t u < 0$), which is scheduled at $t = \lambda^t I + |\lambda^t u|$. Therefore, the processor p will only be active once in $|\lambda^t u|$ clock cycles. For this reason the efficiency of p is defined as $1/|\lambda^t u|$.

Remark 2.1 Since $Pu = 0$ and P has rank $n-1$, two computations $I, I' \in \mathcal{I}$ will be carried out by the same processor $PI = PI'$ if and only if $I - I' = ku$ for some $k \in \mathbf{Z}$.

Remark 2.2 The projecting vector u can also be defined indirectly in terms of P. Let P be an $(n-1) \times n$ integral matrix of rank $n-1$ such that the matrix $T = \begin{bmatrix} \lambda^t \\ P \end{bmatrix}$

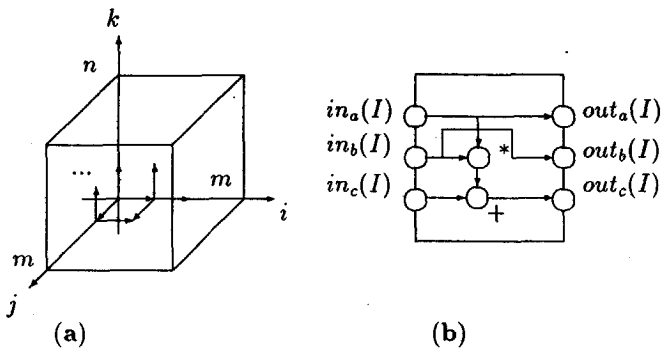


Figure 2.1: (a) The index space of the algorithm. (b) A computation $I \in \mathcal{I}$

is nonsingular. Then the projecting vector u is given by a coprime right null vector of P (i.e., $Pu = 0$). The transformation T is usually referred to as a space-time transformation; a transformed algorithm $< T\mathcal{I}, F, T\mathcal{D} >$ is usually referred to as a space-time representation of the algorithm $< \mathcal{I}, F, \mathcal{D} >$.

Example 2.I: Mapping a matrix multiplication algorithm to a full-size systolic array.

Let $A = \{a_{ij}\} \in \mathbf{R}^{m \times n}$, and $B = \{b_{ij}\} \in \mathbf{R}^{n \times m}$ be two matrices. Then the computation of the product $C = AB$ can be described by the following recurrent equations:

$$\text{For all } I = [i,j,k]^t \in \mathcal{I} = \{[i,j,k]^t | 1 \le i \le m, 1 \le j \le m, 1 \le k \le n\}:$$
$$\begin{cases} a[i,j,k] = a[i,j-1,k]; \\ b[i,j,k] = b[i-1,j,k]; \\ c[i,j,k] = c[i,j,k-1] + a[i,j,k] * b[i,j,k]; \end{cases}$$

or equivalently,

$$\begin{cases} a[I] = a[I - (0,1,0)^t]; \\ b[I] = b[I - (1,0,0)^t]; \\ c[I] = c[I - (0,0,1)^t] + a[I] * b[I]; \end{cases}$$

The input of the algorithm (variable initialization) is given by

$$a[i,0,k] = a_{ik} \text{ for all } 1 \leq i \leq m, 1 \leq k \leq n;$$
$$b[0,j,k] = b_{kj} \text{ for all } 1 \leq j \leq m, 1 \leq k \leq n;$$
$$c[i,j,0] = 0 \text{ for all } 1 \leq i \leq m, 1 \leq j \leq m;$$

and the output of the algorithm is given by

$$c_{ij} = c[i,j,n] \text{ for all } 1 \leq i \leq m, 1 \leq j \leq m;$$

The index space $\mathcal{I}$ of the algorithm is shown in Fig. 2.1a. The I/O mapping

$$F : (in_a(I), in_b(I), in_c(I)) \rightarrow (out_a(I), out_b(I), out_c(I))$$

is defined by

$$\begin{cases} out_a(I) = in_a(I); \\ out_b(I) = in_b(I); \\ out_c(I) = in_c(I) + in_a(I) * in_b(I); \end{cases}$$

for each I in $\mathcal{I}$, see Fig. 2.1b.
And the dependences between computations are defined by

For all $I \in \mathcal{I}$:
$$\begin{cases} in_a(I) = out_a(I - d_a); \\ in_b(I) = out_b(I - d_b); \\ in_c(I) = out_c(I - d_c); \end{cases}$$

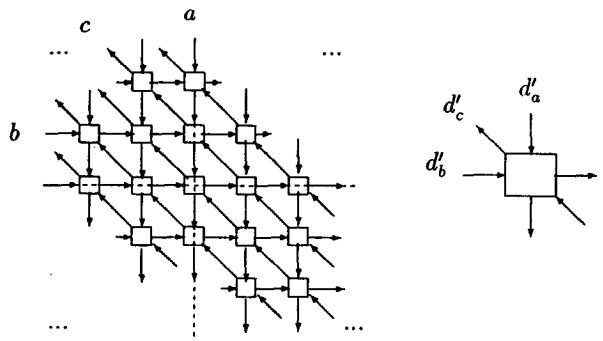
where $d_a = (0,1,0)^t$, $d_b = (1,0,0)^t$, and $d_c = (0,0,1)^t$. The set of dependence vectors of the algorithm is therefore given by $\mathcal{D} = \{d_a, d_b, d_c\}$.

It is shown in [Rao85] that the systolic array for matrix- matrix multiplications proposed by Kung and Leiserson [KL80] can be derived systematically from this algorithm by choosing

(a) a scheduling vector $\lambda = (1,1,1)^t$,

(b) a projecting vector $u = (1,1,1)^t$, and

(c) a rank 2 matrix $P = \begin{bmatrix} 1 & 0 & -1 \\ 0 & -1 & 1 \end{bmatrix}$.



(c) (d)

Figure 2.1: (**c**) A portion of the systolic array according to Kung and Leiserson. (**d**) A processor

It can be easily verified that $\lambda^t[d_a\ d_b\ d_c] \geq (1,1,1)$, $Pu = 0$, and the matrix $\begin{bmatrix} \lambda^t \\ P \end{bmatrix}$ is nonsingular. The full-size array is then given by $\{\tilde{\mathcal{I}}, \tilde{\mathcal{C}}\}$, where $\tilde{\mathcal{I}} = \{p | p = PI, I \in \mathcal{I}\}$, and $\tilde{\mathcal{C}} = \{(\lambda^t d_a, Pd_a), (\lambda^t d_b, Pd_b), (\lambda^t d_c, Pd_c)\}$.

A portion of the array is shown in Fig. 2.1c, and a processor element is shown in Fig. 2.1d, where $d'_a = Pd_a = (0,-1)^t$, $d'_b = Pd_b = (1,0)^t$, and $d'_c = Pd_c = (-1,1)^t$. The functionality of each processor is defined by the I/O mapping F. The efficiency of each processor is equal to $1/|\lambda^t u| = 1/3$.

□

2.3 Clustering of processors

In this section we first show that when $|\lambda^t u| > 1$, then we can always replace clusters of $|\lambda^t u|$ inefficient processors by efficient processors. (We will call this the clustering of passive processors, or passive-clustering for short.) The resulting array will remain regularly interconnected, and will have the same latency and the same throughput as the original array. Then we extend this clustering method to cluster efficient processors (we will call this method the active-clustering in contrast to the passive-

clustering). We show that this active-clustering technique can be used (1) to reduce the size of an array by an arbitrary factor, (2) to reduce the dimension of an array, and (3) to balance the number of I/O operations and the amount of local memory (registers) of processors. Moreover, we describe a method to cluster processors in such a way that the number of I/O pins of a processor after the clustering is independent of the number of processors that are clustered. The control circuitry introduced in a processor after the clustering is simple (it consists of a counter, multiplexers, demultiplexers, and FIFOs or first-in-first-out buffers), and it can be generated automatically.

2.3.1 The passive clustering

In the discussion we assume without loss of generality that $\lambda^t u$ is positive, and is greater than 1.

Let the index space $\mathcal{I}$ be partitioned into bands $\mathcal{I} = \cup_{j=j_{min}}^{j_{max}} \mathcal{B}_j$, where

$$\mathcal{B}_j = \{I | I \in \mathcal{I}, j\lambda^t u \leq \lambda^t I < (j+1)\lambda^t u\}.$$

Without loss of generality, we assume that $j_{min} = 0$. We will ignore the finiteness of the algorithm's index space: if I is an index point in one band, then we assume that there is an index point I' in each of the other bands such that $I' = I + ku$, where k is an integer. (Note that I' is unique in each band according to the definition of bands.)

Each band $\mathcal{B}_j$ contains $\lambda^t u$ (hyper-)planes, given by $\lambda^t I = j\lambda^t u + \alpha$, for $\alpha = 0, \ldots, \lambda^t u - 1$. (Recall that all the index points on a scheduling plane $\lambda^t I = t$ are scheduled at time t.)

Proposition 2.1 *For each j, $0 \leq j \leq j_{max}$, (1) $\tilde{\mathcal{I}} = \{p | p = PI, I \in \mathcal{B}_j\}$; (2) for any $I, I' \in \mathcal{B}_j$, if $I \neq I'$, then I and I' will be mapped to different processors.*

Proof: (1) This is a direct consequence of the assumption that if I is an index point in one band then there is an I' in each of the other bands such that $I' = I + ku$.

(2) Let $I, I' \in \mathcal{B}_j$, and $I \neq I'$. Assume that I and I' are mapped to the same processor, that is, $PI = PI'$. Then we will have $I - I' = ku$ for some integer $k \neq 0$.

It follows then that $\lambda^t I = \lambda^t I' + k\lambda^t u$, and therefore, I and I' cannot be in the same band, a contradiction.

□

In other words, for each j, $0 \le j \le j_{max}$, and for each processor $p \in \tilde{\mathcal{I}}$, there is a unique index point $I \in \mathcal{B}_j$ such that $p = PI$; and for each $I \in \mathcal{B}_j$ there is a unique $p \in \tilde{\mathcal{I}}$ such that $p = PI$.

It is therefore sufficient to consider only one of the bands $\mathcal{B}_j$ of the index space, we let it be $\mathcal{B}_0 = \{I | I \in \mathcal{I}, 0 \le \lambda^t I < \lambda^t u\}$. Since $\lambda^t u > 1$, each processor will only be active once in $\lambda^t u$ clock cycles. Below we show that clusters of $\lambda^t u$ such inefficient processors can be identified in the processor array, and each of them can be replaced by an efficient processor.

Let J be an arbitrary point on the plane $\lambda^t I = 0$, and let $v_1, \ldots, v_{\lambda^t u - 1}$, be $\lambda^t u - 1$ integral vectors such that $J + v_i$ is on the plane $\lambda^t I = i$. (We will introduce a dummy index point in $\mathcal{I}$ if $J + v_i \notin \mathcal{I}$.) Moreover, let J be mapped to processor p_0 and let $J + v_i$ be mapped to processor p_i, that is, $p_0 = PJ$ and $p_i = P(J + v_i)$ for $i = 1, \ldots, \lambda^t u - 1$. (We will introduce a dummy processor in $\tilde{\mathcal{I}}$ if $p_0 + Pv_i \notin \tilde{\mathcal{I}}$.)

Proposition 2.2 *The $\lambda^t u$ computation tasks of processors p_0, $p_1, \ldots, p_{\lambda^t u - 1}$ can be carried out serially using one processor.*

Proof: Since these processors are scheduled serially in each of $\lambda^t u$ clock cycles: processor p_i, $i = 0, 1, \ldots, \lambda^t u - 1$, will only be active at $t = j\lambda^t u + i$, for all $j = 0, \ldots, j_{max}$. Therefore we can also execute their tasks using only one processor.

□

We can therefore eliminate processors p_i, for $i = 1, \ldots, \lambda^t u - 1$, and assign their tasks to processor p_0 (we say that these processors are "clustered" into one processor p_0, and we call the resulting processor a *clustered processor*). After this, the clustered processor p_0 will be active in each clock cycle, hence it has efficiency 100%. Such clustering of processors is always possible since the vectors $v_1, \ldots, v_{\lambda^t u - 1}$ always exist. (Obviously, there is a great number of choices of such sets of vectors.)

The clustering of processors can be equivalently specified by $\lambda^t u - 1$ vectors in the processor space. Let $v_i' = Pv_i$, $i = 1, \ldots, \lambda^t u - 1$, we call these vectors *clustering*

vectors. Then p_0 and p_0+v_i', $i=1,\ldots,\lambda^t u-1$ can be clustered into one processor p_0.

Let $\tilde{\mathcal{P}}=\{p|p=PI, I\in\mathcal{B}_0, \lambda^t I=0\}$. Then we can cluster all the processors $p+v_i'$ for $i=1,...,\lambda^t u-1$ into one processor p for each $p\in\tilde{\mathcal{P}}$. The set of processors of the array after the clustering (called clustered array) will be given by $\tilde{\mathcal{P}}$. It contains $|\tilde{\mathcal{P}}|$ clustered processors, where $||$ denotes the cardinality of a set. The size of the array $\tilde{\mathcal{I}}$ is reduced by a factor of $\lambda^t u$ after clustering, that is, $|\tilde{\mathcal{P}}|=\frac{1}{\lambda^t u}|\tilde{\mathcal{I}}|$.

Proposition 2.3 *Given $\tilde{\mathcal{P}}$, and v_i', $i=1,...,\lambda^t u-1$, defined as above. Then for each $p\in\tilde{\mathcal{I}}$ such that $p\notin\tilde{\mathcal{P}}$, there is a unique processor $p'\in\tilde{\mathcal{P}}$ and a unique clustering vector v_i', with $1\le i\le\lambda^t u-1$, such that $p=p'+v_i'$.*

Proof: Let $p\in\tilde{\mathcal{I}}$ and $p\notin\tilde{\mathcal{P}}$. Then there is a unique J on plane $\alpha=i$, with $1\le i\le\lambda^t u-1$, such that $p=PJ$. For all the points $I=J-v_j$, $j=1,...,\lambda^t u-1$, $I=J-v_i$ is the only index point on the plane $\alpha=0$. Therefore, we have a unique $p'=PI$ and $v_i'=Pv_i$ such that $p'=p+v_i'$.

□

In other words, the clustering of processors is uniquely defined when the set of clustering vectors is given.

Proposition 2.4 *(1) The clustered array will have the same latency and throughput as the array before the clustering. (2) The clustered array will be regularly interconnected.*

Proof: (1) In terms of latency and throughput, it is equivalent to execute $\lambda^t u$ tasks which are scheduled serially at $0,1,\ldots,\lambda^t u-1$ using one processor, or using $\lambda^t u$ processors.

(2) The clustered array will be regularly interconnected since there will be no difference in the interconnection patterns for different processors in $\tilde{\mathcal{P}}$.

□

Theorem 2.1 *If $\lambda^t u>1$, then it is always possible to replace clusters of $\lambda^t u$ inefficient processors by efficient processors without changing the throughput and latency of the array. The clustered array will be regularly interconnected.*

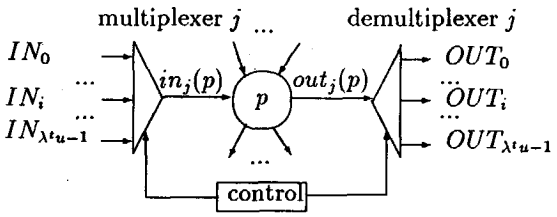


Figure 2.2: A clustered processor

Proof: This is a direct consequence of proposition 2.2 and proposition 2.4.

□

A weak version of this theorem is proved in [ND88] and [Jai89] for two- or three-dimensional index spaces.

Note that we can also use different sets of clustering vectors to cluster processors in different parts of the array $\tilde{\mathcal{I}}$. If the number of different sets of clustering vectors is independent of the size of the array, then the number of different interconnection patterns obtained in the clustered array will also be independent of the size of the array. Such an array is said to be piecewise regularly interconnected [Thi89a].

The structure of the clustered array

We examine now the structure of the clustered processors and the structure of the clustered array. Recall that before the clustering, each processor $p \in \tilde{\mathcal{I}}$ has m inputs $in_j(p)$, $j = 1, ..., m$, and m outputs $out_j(p)$, $j = 1, ..., m$; the output $out_j(p)$ of p is connected to the input $in_j(p+d'_j)$ of $p + d'_j$, $d'_j = Pd_j$, for $j = 1, ..., m$, where $d_1, ..., d_m$ are the dependence vectors of the algorithm.

In the clustered array, a processor $p \in \tilde{\mathcal{P}}$ will execute the task of processor $p+v'_i$ ($v'_0 = 0$) of the original array at $t \bmod \lambda^t u = i$ for $i = 0, ..., \lambda^t u - 1$, where mod denotes the modulo operator. Therefore, the inputs and outputs of the clustered processor p will be multiplexed in general, as shown in Fig. 2.2 (where only the circuitry related to the input $in_j(p)$ and output $out_j(p)$ of processor p is shown in detail). In Fig. 2.2, the input IN_i of the multiplexer corresponds to the input $in_j(p+v'_i)$ of processor $p+v'_i$

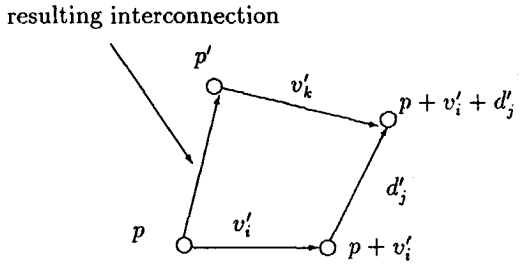


Figure 2.3: Interconnection after clustering of processors

of the original array. Similarly, the output OUT_i of the demultiplexer corresponds to the output $out_j(p+v'_i)$ of processor $p+v'_i$ of the original array. The control circuit, a $\lambda^t u$-ring counter, will select the input IN_i and the output OUT_i of the multiplexer and demultiplexer respectively, at clock cycle $t \bmod \lambda^t u = i$ for $i = 0, ..., \lambda^t u - 1$.

Note that a demultiplexer can be simplified (or removed) if a part (all) of its outputs are connected to the same processor, these outputs and the corresponding interconnections can be replaced by a single output and a single interconnection because data are time multiplexed; similarly for the multiplexers.

As the output $out_j(p+v'_i)$ of processor $p+v'_i$ goes to the input $in_j(p+v'_i+d'_j)$ of processor $p+v'_i+d'_j$ in the original array, it will be re-routed after the clustering to the ith input IN_i of the jth selector of a clustered processor p', if $p+v'_i+d'_j$ is clustered to p' (p' is unique according to proposition 2.3). The processor p' can be easily determined: if $p+v'_i+d'_j$ is active at clock cycle k, $0 \leq k \leq \lambda^t u - 1$, in the original array, then $p' = p+v'_i+d'_j-v'_k$. See Fig. 2.3. We will obtain the interconnection structure of the clustered array if we repeat this process for all d'_j, $j = 1, ...m$, and for all v'_i, $i = 0, ..., \lambda^t u - 1$.

It follows that this passive-clustering of processors will not introduce additional memory in the clustered processors. However, the number of I/O pins of the clustered processors may possibly increase by a factor of $\lambda^t u$ (assuming that the multiplexers and demultiplexers are implemented within the clustered processors). Therefore, minimizing the number of I/O pins of the clustered processors will be

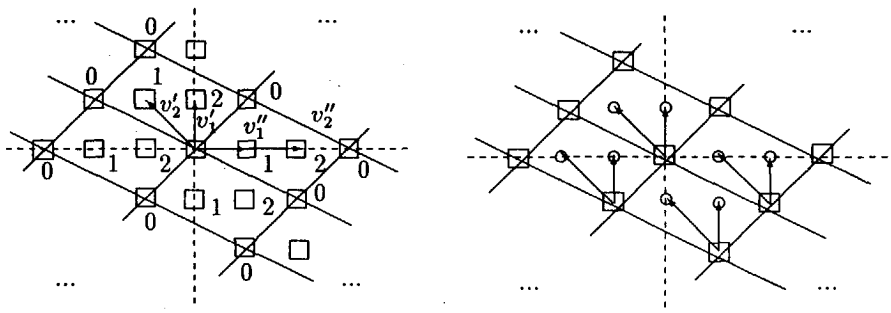


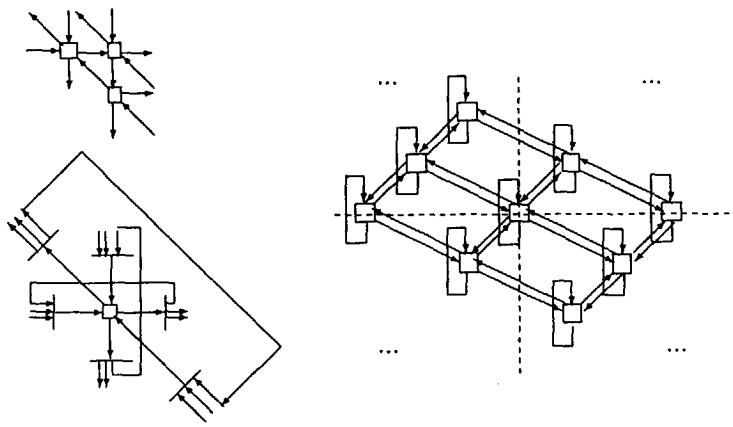
Figure 2.4: (a) Clustering vectors. (b) Processor clustering

an important criterion for the selection of the clustering vectors. One solution is to cluster processors in such a way that self-loops are created and to implement the self-loops within the clustered processors. Moreover, we consider the following criteria for the selection of the clustering vectors to be practical: (1) the interconnection structure of the resulting clustered array is simple; and/or (2) the processor interconnections in the resulting clustered array can be implemented using a given set of interconnection primitives.

Remark 2.3 We can also describe the clustering method in the algorithm's index space $\mathcal{I} = \cup_{j=0}^{j_{max}} \mathcal{B}_j$, as follows. First, in each band $\mathcal{B}_j$, we cluster for each point I on the plane $\alpha = j\lambda^t u$ the index points $I + v_i$, $i = 1, ..., \lambda^t u - 1$ to that point. After this, only the points on the plane $\lambda^t I = j\lambda^t u$ will remain. Then, we project these points to the processor space, which will give us the clustered array $\tilde{\mathcal{P}}$.

Example 2.II: Let us consider the systolic array of Example 2.I. Recall that $\lambda^t u = 3$ in Example 2.I, and the efficiency of processors is 1/3.

In Fig. 2.4a, a portion of the systolic array of Example 2.I (see Fig. 2.1c) is shown, however, without interconnections. For each processor p in Fig. 2.4a, let



(c) (d)

Figure 2.4: (c) Processor(s) before and after clustering. (d) Interconnection of the clustered array

$p = PI$ for some $I \in \mathcal{I}$, then the value of $\lambda^t I \bmod \lambda^t u$ of p is also shown in Fig. 2.4a; it can be either 0,1, or 2. (the λ, u, P, and $\mathcal{I}$ are defined in Example 2.1.)

Moreover, in Fig. 2.4a, two sets of clustering vectors are shown. They are $\{v_1' = (0,1)^t, v_2' = (-1,1)^t\}$ and $\{v_1'' = (1,0)^t, v_2'' = (2,0)^t\}$ respectively.

To make processors efficient, let us cluster the processors using the first set of clustering vectors, i.e., $\{v_1' = (0,1)^t,\ v_2' = (-1,1)^t\}$. Let the clustered processors be represented by those squares which are labeled with 0 in Fig. 2.4a. See also Fig. 2.4b, where the processors which will disappear after the clustering are indicated by small circles. Then, using the clustering technique described in this section, it follows that the (internal) structure of a clustered processor will be as shown in Fig. 2.4c, where all the multiplexers and demultiplexers (represented by the bars in the figure) are controlled by a simple 0-1-2 ring counter (which is not shown in the figure). The interconnection structure of the clustered array will be as shown in Fig. 2.4d.

□

2.3.2 The active clustering

In this section we extend the passive-clustering technique described in the previous section to cluster processors that are active at the same time. It has, among the others, two important applications: (1) it can be used (partially) to reduce the size and/or dimension of an array, and (2) it can be used to balance the I/O operations and local storage of processors.

We first discuss the case when all the processors in $\tilde{\mathcal{I}}$ are active in each clock cycle, that is, $\lambda^t u = 1$. Then we give an extension to when $\lambda^t u > 1$.

Let $\eta > 1$ be an integer, and let $v'_1, ..., v'_{\eta-1}$ be $\eta - 1$ integral vectors. Assume that the set of processors $\tilde{\mathcal{I}}$ can be partitioned into η subsets $\tilde{\mathcal{I}}_0, \tilde{\mathcal{I}}_1, ..., \tilde{\mathcal{I}}_{\eta-1}$ in such a way that for each $p_0 \in \tilde{\mathcal{I}}_0$, $p_i = p_0 + v'_i \in \tilde{\mathcal{I}}_i$, for $i = 1, ..., \eta - 1$ (dummy processors will be added when necessary). We show below how to map (cluster) processors $p_0, p_1, \ldots, p_{\eta-1}$ into one processor p_0. When processors $p_0, p_1, \ldots, p_{\eta-1}$ are clustered, the resulting (clustered) processor p_0 has to execute continually η tasks using η clock cycles. The ordering of the execution of these tasks can be arbitrary since they can be executed in parallel before clustering, that is, their executions can be performed independently of each other. However, the latency of the computation (i.e., total computation time) will be increased by a factor of η since η parallel tasks are executed serially on one processor. In the following discussion we use the processors in $\tilde{\mathcal{I}}_0$ to represent the clustered processors.

The active clustering

Assume without loss of generality that the execution ordering of the tasks of p_0, p_1,...,$p_{\eta-1}$ on clustered processor p_0 is given by $0, 1, ..., \eta - 1$. In order to determine the clustered array, we first perform the following "slow-down" transformation:

a. insert $\eta - i - 1$ additional delays at each output of processor p_i, for $i = 0, ..., \eta - 1$, and

b. insert i additional delays at each input of processor p_i, for $i = 0, ..., \eta - 1$.

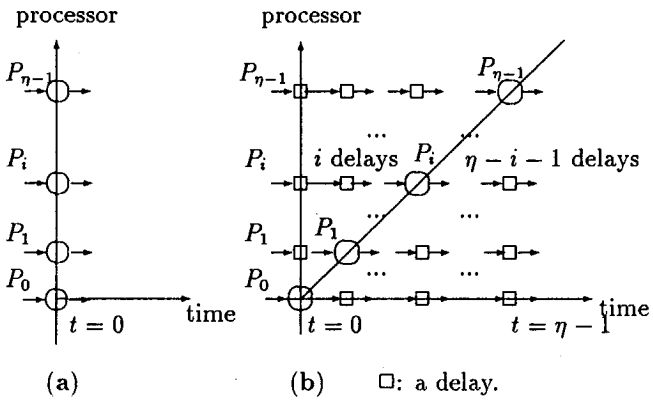


Figure 2.5: (a) The execution of P_0,...,$P_{\eta-1}$ before clustering. (b) The execution of the processors after the slow-down transformation

The execution of $p_0, ..., p_{\eta-1}$ before the slow-down transformation is shown in Fig. 2.5a, where all the η processors will receive their input data at the same time, say $t = 0$, and will produce output data at the same time. The execution of these processors after the slow-down transformation is shown in Fig. 2.5b. Assume that, similarly to Fig. 2.5a, all the input data to the processors are available at $t = 0$. Then the delays inserted at the input of each p_i, $0 \leq i \leq \eta - 1$, will ensure that p_i will receive the same data when it is executed at $t = i$ as when it is executed at $t = 0$. Similarly, the delays inserted at the outputs of the processors will ensure that their output data will become available at the same time as if they were executed in parallel (however, using η clock cycles instead of one clock cycle). The array obtained after this slow-down transformation will, therefore, be functionally equivalent to the original array. Note that this slow-down transformation decreases in fact the efficiency of each processor to $1/\eta$.

The above transformation ensures that we can cluster the processors in the same way as described in the previous section. The clustered processors will have the same structure as shown in Fig. 2.2. Note that in the clustered array delays on an interconnection can usually be combined into a first-in-first-out buffer of length $\eta-1$.

When the clustered processors are pipelined, higher throughput will be obtained.

Since before the clustering each processor has m inputs and m outputs, the number of additional registers introduced in each clustered processor will be given by $m(\eta-1)$. In general, we can replace the inserted delays at an input/output of a processor by one register with appropriate control circuitry. Such replacement will be especially useful when the value of η is large.

This active-clustering technique can be used to balance the processor I/Os and the amount of local memory of the clustered processors. What we have do is to create (more) self-loops by clustering processors along processor interconnections, and to implement the self-loops inside the processors. These self-loops will then not require any I/O operation, of course, their data has to be stored in the processor's local memory.

A regular partition of a processor array

Although there are many ways to cluster processors, we consider the following approach to be especially practical, since it will result in clustered processors with a constant number of I/O pins, independent of the value of η. What we do is first partition the processor space $\tilde{\mathcal{I}}$ into regular sections (partitions), and then cluster the processors in each partition into one processor. One way to partition the processor space $\tilde{\mathcal{I}}$ into regular sections is as follows.

Let u_i, $i = 1, ..., n-1$, be a set of linearly independent integral vectors in the processor space $\tilde{\mathcal{I}}$. (Recall that $\tilde{\mathcal{I}} \subset \mathbf{Z}^{n-1}$.) Then we define a partition of the array as

$$\tilde{\mathcal{G}}_p = \{p' | p' \in \tilde{\mathcal{I}},\ p' = p + \textstyle\sum_{i=1}^{n-1} a_i u_i,\ 0 \le a_i < 1\},$$

where $p \in \tilde{\mathcal{I}}$ is used as a reference point of $\tilde{\mathcal{G}}_p$. It contains all the processors enclosed in the parallelepiped, which is characterized by a vertex point p and a set of $n-1$ vectors $\{u_i\}$. (See Fig. 2.6a for a 2-dimensional example.)

Now let $\tilde{\mathcal{J}}$ denote the set

$$\tilde{\mathcal{J}} = \{q | q \in \tilde{\mathcal{I}},\ q = q_0 + \textstyle\sum_{i=1}^{n-1} b_i u_i,\ b_i \in \mathbf{Z}\},$$

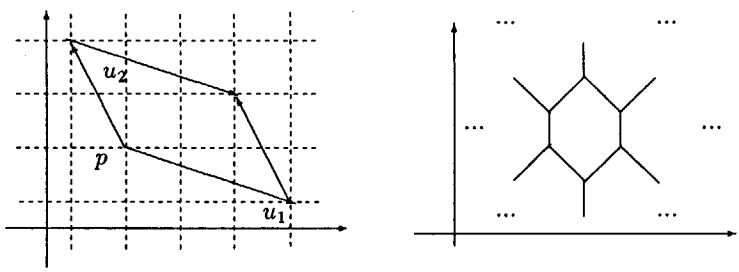


Figure 2.6: (a) A partition $\tilde{\mathcal{G}}_p = \{p' | p' = \sum_{i=1}^{2} a_i u_i, 1 \leq a_i \leq 1\}$ of processor space $\tilde{\mathcal{I}}$. (b) An alternative way to define $\tilde{\mathcal{G}}_p$

where $q_0 \in \mathbf{Z}^{n-1}$ is used as a reference point; without loss of generality, we take $q_0 = 0$ from now on. The processor space can be then written as

$$\tilde{\mathcal{I}} = \cup_{p \in \tilde{\mathcal{J}}} \tilde{\mathcal{G}}_p.$$

Also here we ignore the boundary of the processor space by assuming that all the $\tilde{\mathcal{G}}_p$'s are identical (isomorphic), and they contain the same number of processors. Note that there are many other ways to partition the processor space into regular sections; see Fig. 2.6b for an alternative.

Properties of the processor clustering

Now let us choose the u_i's to be large enough so that for each $p' \in \tilde{\mathcal{G}}_p$, $p' + d'_j$ is either in $\tilde{\mathcal{G}}_p$ itself, or in a neighboring $\tilde{\mathcal{G}}_q$ for all $i = 1, ..., m$. (Recall that d_j, $j = 1, ..., m$ are the dependence vectors of the algorithm, and $d'_j = P d_j$.) We say that $\tilde{\mathcal{G}}_q$, $q \in \tilde{\mathcal{J}}$ is a neighbor of $\tilde{\mathcal{G}}_p$ if $q - p = \sum_{i=1}^{n-1} \delta_i u_i \neq 0$, with $\delta_i = \pm 1$, or 0.

Proposition 2.5 *Let the vectors u_i, $i = 1, ..., n-1$, be selected in such a way that for each $p' \in \tilde{\mathcal{G}}_p$, $p' + d'_j$ is either in $\tilde{\mathcal{G}}_p$ itself, or in a neighboring $\tilde{\mathcal{G}}_q$ for all $i = 1, ..., m$. If for each $p \in \tilde{\mathcal{J}}$ all the processors in $\tilde{\mathcal{G}}_p$ are clustered into one processor, represented by p, then the number of I/O pins of the clustered processor p is independent of the number of processors in $\tilde{\mathcal{G}}_p$.*

Proof: The number of the neighbors of $\tilde{\mathcal{G}}_q$ of $\tilde{\mathcal{G}}_p$ is independent of the number of processors in $\tilde{\mathcal{G}}_p$. After the clustering, each clustered processor will only communicate with its neighboring processors. Moreover, assume that the clustered processor p will communicate with a neighboring clustered processor q, then p will need at most one interconnection to the processor q for each of its inputs and outputs since the data can be time multiplexed.

□

In other words, when we ensure the u_i's are large enough with respect to d'_j's, then the number of I/O pins of the clustered processors is independent of the size of $\tilde{\mathcal{G}}_p$s.

Since each clustered processor p will only communicate with its neighboring processors q, and the neighborhood relationship is fully defined by the chosen vector u_i's, when u_i's are large enough, then we can map a large class of systolic arrays onto one array with fixed interconnections. In particular, we can design an (systolic) array with programmable processing elements, where each processor is able to communicate with its neighboring (as defined above) processing elements, to simulate/emulate a class of systolic arrays.

The I/O bandwidth of a clustered processor can be determined as follows. Let η be the number of processors in $\tilde{\mathcal{G}}_p$, and let η_j be the number of processors in $\tilde{\mathcal{G}}_p$ whose jth outputs (i.e., $out_j(p')$, $p' \in \tilde{\mathcal{G}}_p$), $1 \leq j \leq m$, are interconnected with some processors in a neighboring partition $\tilde{\mathcal{G}}_q$. Then after the clustering, there will be an interconnection from the clustered processor p (derived from $\tilde{\mathcal{G}}_p$) to the clustered processor q (derived from $\tilde{\mathcal{G}}_q$), which will be used to transfer η_j data items in η clock cycles.

Let η'_j be the number of processors in $\tilde{\mathcal{G}}_p$ whose jth outputs are interconnected with some processors within $\tilde{\mathcal{G}}_p$ itself. Then these processors will create η'_j self-loops (local memory) in the clustered processor $\tilde{\mathcal{G}}_p$. We will obtain the I/O bandwidth of the clustered processor if we repeat this procedure for each output $out_j(p')$ (or each input $in_j(p')$) of $p' \in \tilde{\mathcal{G}}_p$, and for all $p' \in \tilde{\mathcal{G}}_p$.

After the clustering, the set of processors in the clustered array will be given by $\tilde{\mathcal{J}}$. It can be equivalently represented by the set of coefficients $\tilde{\mathcal{I}}_b$ of u_i's:

$$\tilde{\mathcal{I}}_b = \{(b_1, ..., b_{n-1})^t | \sum_{i=1}^{n-1} b_i u_i \in \tilde{\mathcal{J}}, b_i \in \mathbf{Z}\}.$$

The size of the array $\tilde{\mathcal{I}}$ is reduced by a factor of $\eta = |\tilde{\mathcal{G}}_p|$ after the clustering.

Until now we have assumed that the efficiency of each processor in $\tilde{\mathcal{I}}$ is 100%, that is, $\lambda^t u = 1$. When $\lambda^t u > 1$, we can proceed as follows. First, we cluster the processors using the passive-clustering technique described in the previous section to obtain a 100% efficient clustered array $\tilde{\mathcal{P}}$. Then, we cluster this efficient array using the active-clustering technique described above. Note that each processor in the array $\tilde{\mathcal{P}}$ will have $\lambda^t u$ "micro" execution cycles after the passive-clustering. When we map more such processors, say p_0,...,p_η, into one processor, say p_0, by the active-clustering, care must be taken in scheduling the execution of p_0,...,p_η on the clustered processor p_0. Generally, we have to execute the ith micro execution cycle of a processor p_k, for $0 \leq k \leq \eta$, after all the $(i-1)$th micro execution cycles of p_0,...,p_η are finished, for all $i = 1, ..., \lambda^t u$.

Reducing the size/dimension of an array by active-clustering

The clustering technique described above can also be used to map a class of full-size arrays of different sizes to a fixed-size array. Indeed, what we have to do is to choose an appropriate length for each vector u_i, $i = 1, ..., n-1$, such that the set of clustered processors $\tilde{\mathcal{I}}_b$ will remain the same for each of the full-size arrays of the class. However, since the full-size array $\tilde{\mathcal{I}}$ contains $O(N^{n-1})$ processors in general, where N is the largest extent of the algorithm's index space $\mathcal{I}$, each $\tilde{\mathcal{G}}_p$ will also contain $O(N^{n-1})$ processors since the number of $\tilde{\mathcal{G}}_p$'s is a constant. Therefore, the amount of local memory introduced in the clustered processors (and the execution time of these processors) will generally also be of order $O(N^{n-1})$. In the next section, we will describe a method to reduce the amount of local memory in the clustered processors (and their execution time).

An alternatively way of mapping full-size arrays to a fixed-size array is first to choose the size of $\tilde{\mathcal{G}}_p$ appropriately so that it matches with the size of the desired fixed-size array. Then, for each $p' \in \tilde{\mathcal{G}}_p$, cluster all the processors $p' + \sum_{i=1}^{n-1} b_i u_i \in \tilde{\mathcal{I}}$ into the processor p'. It can be shown that such clustering of processors will only change

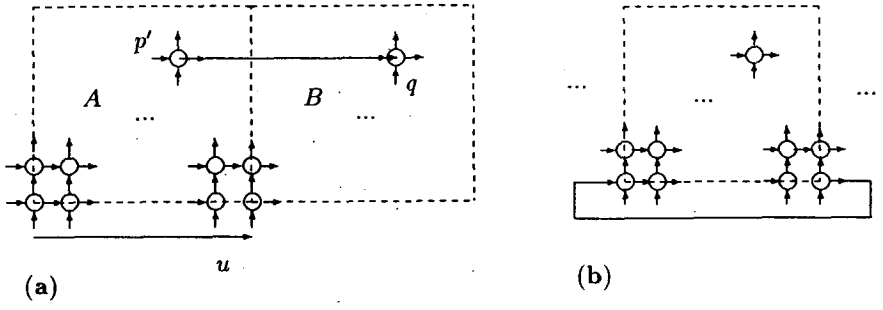


Figure 2.7: An alternative way to cluster processors: (**a**) q is clustered to p'. (**b**) The clustered array obtained

the interconnection structure of the *boundary* processors. We say that a processor is a boundary processor if it is connected with processors in other partitions $\tilde{\mathcal{G}}_q$. We can illustrate this using Fig. 2.7a, where a processor $q=p'+u$, in partition B is clustered to processor p' in partition A. The clustered array is shown in Fig. 2.7b, which has "wrap-around" interconnections. We can, in fact, design a large set of operations which map essentially several processors to one processor for the reduction of the size of an array, to a constant when necessary.

The clustering technique can also be used to reduce the dimension of a full-size (or a clustered) array $\tilde{\mathcal{I}}$. Let P' be an $(n-2)\times(n-1)$ integral matrix of rank $n-2$ such that $P'u_i=0$, where u_i can be chosen arbitrarily in the set $\{u_1,...,u_{n-1}\}$. Then we can map the processor space $\tilde{\mathcal{I}}\subset\mathbf{Z}^{n-1}$ to a new processor space $\tilde{\mathcal{I}}'=\{p'|p'=P'p, p\in\tilde{\mathcal{I}}\}\subset\mathbf{Z}^{n-2}$. The processors in $\tilde{\mathcal{I}}$ that are mapped to the same processor in $\tilde{\mathcal{I}}'$ must be clustered using the active-clustering technique described in this section. Moreover, let $\tilde{\mathcal{G}}'_p=\{p'|p'=P'p, p\in\tilde{\mathcal{G}}_p\}$. Then we can cluster the processors in each $\tilde{\mathcal{G}}'_p$ into one processor so that the resulting clustered processors will have a constant number of I/O pins. This procedure can be used repetitively to map an $(n-1)$-dimensional array to an $(n-k)$-dimensional array, $1\leq k<n-1$. It gives a general, yet simple way of implementing the "multi-projection" operation [Kun88] (section 3.4.3) for the reduction of the dimension of systolic arrays. Moreover, it turns out that this

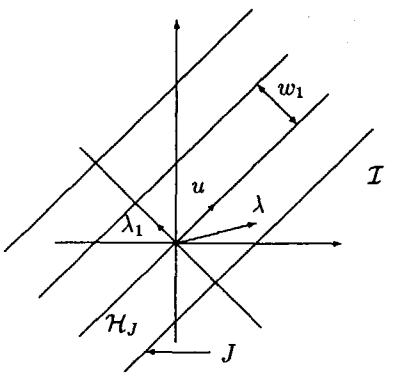


Figure 2.8: Partitioning of the index space $\mathcal{I}$ into bands $\mathcal{H}_J$

technique is a generalization of that used in [US89], [LK88] for the design of linear systolic arrays.

2.4 Partitioning the index space and full-size array

In this section we describe a method of partitioning the algorithm's index space $\mathcal{I}$ in such a way that the partitions of the index space can be executed serially using a reduced-size array. When necessary, this reduced-size array can be mapped to a fixed-size array using the clustering techniques described in the previous section.

To simplify the discussion, we start out with a given full-size array $< \tilde{\mathcal{I}}, \tilde{\mathcal{C}} >$ derived from the algorithm, the vectors λ and u, and the matrix P are therefore known. Moreover, without loss of generality, we assume in the discussion that the efficiency of the processors is 100%, that is, $\lambda^t u = 1$; we can always cluster the processors if this is not the case.

Let λ_1 be a unit vector in $\mathbf{R}^n$ such that (1) λ and λ_1 are linearly independent, (2) $\lambda_1^t u = 0$, and (3) $\lambda_1^t d \geq 0$ for all $d \in \mathcal{D}$ (recall that $\mathcal{D}$ is the set of dependence vectors of the algorithm). Then we can partition the index space $\mathcal{I}$ into bands using a set of parallel, equidistant (hyper-)planes which have λ_1 as their normals. Let w_1, an

integer constant, be the distance between two adjacent planes. Then we can write a band as

$$\mathcal{H}_J = \{I | I \in \mathcal{I}, \lambda_1^t J \leq \lambda_1^t I < \lambda_1^t J + w_1\},$$

where J belongs to a subset of index points which are used as reference points, one for each band. See Fig. 2.8 for a 2-dimensional example.

Each band $\mathcal{H}_J$ of the index space can be mapped to a band of the processor space $\tilde{\mathcal{I}}$ (using the given matrix P). Since $\lambda_1^t u = 0$, two index points in different bands of the index space will be mapped to two processors in different bands of the processors space. As a direct consequence, different bands $\mathcal{H}_J$ of the index space will be executed on different parts of the full-size array.

Let us ignore the finiteness of the index space $\mathcal{I}$: we assume from now on that all the bands $\mathcal{H}_J$ of the index space are identical (isomorphic), independent of the choice of λ_1 and J. Then, all the bands of the processor space are also identical (isomorphic). Let $\mathcal{H}_J$ be mapped to a band $\tilde{\mathcal{H}}_p$ of the processor space, where $p = PJ$, then $\tilde{\mathcal{H}}_p$ can be written as $\tilde{\mathcal{H}}_p = \{p' | p' = PI, I \in \mathcal{H}_J\}$.

Since $\lambda_1^t d \geq 0$ for all $d \in \mathcal{D}$, there is no counter-data flow through the partitioning planes of the index space. Therefore, the execution of all the bands of the index space can be serialized. All the bands can be executed on one band, say $\tilde{\mathcal{H}}_p$, of the processor space. The execution of the index points within each band $\mathcal{H}_J$ of the index space is scheduled by the given scheduling vector λ.

We call such a band $\tilde{\mathcal{H}}_p$ a *reduced-size array* (with respect to the full-size array $\tilde{\mathcal{I}}$). It has a fixed width (which corresponds to the distance w_1 between the bands of the index space) in one dimension. Recall that the dimension of $\tilde{\mathcal{I}}$ is $n-1$. The number of processors in this reduced-size array will be of order $O(N^{n-2})$ in general (N is the largest extent of the index space $\mathcal{I}$). As we have shown in the previous section, mapping this reduced-size array to a fixed-size array will introduce in each processor in the fixed-size array $O(N^{n-2})$ local memory.

Remark 2.4 Note that if $\lambda_1^t d = 0$ for all $d \in \mathcal{D}$, then there are no data dependences between the bands $\mathcal{H}_J$ of index space $\mathcal{I}$. The algorithm is actually subdivided into a

set of independent sub-algorithms. In this case we will consider the sub-algorithms individually.

We can obviously repeat the above process to partition each band of the index space again into sub-bands by determining a unit vector λ_2 which satisfies the following conditions: (1) λ, λ_1, and λ_2 are linearly independent, (2) $\lambda_2^t u = 0$, and (3) $\lambda_2^t d \geq 0$ for all $d \in \mathcal{D}$. Since the index space $\mathcal{I}$ is of dimension n, there exists at most $n-1$ such vectors.

In general, let $\{\lambda_1, ..., \lambda_k\}$, $k \leq n-1$, be a set of k linearly independent unit vectors such that (1) $\lambda, \lambda_1, ..., \lambda_k$ are linearly independent, (2) $\lambda_i^t u = 0$ for all $i = 1, ..., k$, and (3) $\lambda_i^t d \geq 0$ for all $d \in \mathcal{D}$ and for all $i = 1, ..., k$. Let $w_1, ..., w_k$ be k integer constants. Then we can partition the index space $\mathcal{I}$ into bands using k sets of parallel equidistant (hyper-)planes. Each pair (λ_i, w_i), $1 \leq i \leq k$, defines one set of parallel planes with λ_i as their normals and w_i as the distance between two adjacent planes in the set. A band of the index space is then given by as

$$\mathcal{H}'_J = \cap_{i=1}^{k} \{I | I \in \mathcal{I}, \lambda_i^t J \leq \lambda_i^t I < \lambda_i^t J + w_i\},$$

where $J \in \mathcal{I}$ is a reference point. All the bands $\mathcal{H}'_J$ of the index space can be executed serially using a reduced-size array which is defined as $\tilde{\mathcal{H}}' = \{p' | p' = PI, I \in \mathcal{H}'_J\}$. The execution of the index points within each band $\mathcal{H}'_J$ is scheduled according to the scheduling vector λ, which is given.

The reduced-size array $\tilde{\mathcal{H}}'$ will contain $O(N^{n-k-1})$ processors in general, and mapping this array to a fixed-size array will introduce $O(N^{n-k-1})$ local memory in each processor. In order to reduce this amount of local memory introduced in the processors in the fixed-size array, we have to choose $k \leq n-1$ to be as large as possible. Obviously, when $k = n-1$, the number of processors in the reduced-size array $\tilde{\mathcal{H}}'$ will be independent of the size of the index space. The partitioning method described in [MF86] is similar to the case when $k = n-1$. See also [Kre89].

A method which is more suitable for the interactive determination of the λ_i's and size of the fixed-size array to be designed is as follows. Let $s_1, ..., s_{n-1}$ be linearly independent vectors in $\mathbf{Z}^n$ on a scheduling plane, that is, $\lambda^t s_i = 0$ for all $i = 1, ..., n-1$. And let $\mathcal{G}_J$ be a box (parallelepiped) in the index space which is defined as

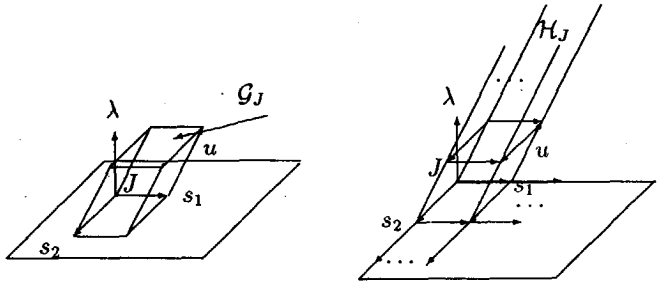


Figure 2.9: (a) A box $\mathcal{G}_J$ of the index space $\mathcal{I}$. (b) Partitioning of $\mathcal{I}$ into $\mathcal{H}_J$s

$$\mathcal{G}_J = \{I | I \in \mathcal{I}, I = J + a_0 u + \textstyle\sum_{i=1}^{n-1} a_i s_i,\ 0 \le a_i < 1\}$$

where $J \in \mathcal{I}$ is a reference point. See Fig. 2.9 for a 3-dimensional example. Then we can view the index space $\mathcal{I}$ as being constructed by the stacking of such boxes in the directions of $\pm s_1,...,\pm s_{n-1}$, and $\pm u$. If we extend $\mathcal{G}_J$ in the directions of $\pm u$ only, see Fig. 2.9b, then we obtain a box of the index space which is defined similarly to $\mathcal{H}'_J$. However, $\mathcal{H}_J$ is defined in this case by $n-1$ sets of parallel planes, where λ_i (the unit normal of the planes in the ith set) and w_i (the distance between the planes in the ith set) are defined as follows: (1) λ_i is a unit vector which is perpendicular to u and all the s_j's except s_i, that is, $\lambda_i^t u = 0$ and $\lambda_i^t s_j = 0$ for $j = 1,...,n-1$ and $j \neq i$; and (2) w_i is given by $\lambda_i^t s_i$. We can then select from these λ_i's those which satisfy the condition $\lambda_i^t d \ge 0\ \forall d \in \mathcal{D}$ to partition the index space and to determine the reduced-size array.

Note that $\mathcal{G}_J$ defines also a box (parallelepiped) $\tilde{\mathcal{G}}_p$ in the processor space $\tilde{\mathcal{I}}$, where $\tilde{\mathcal{G}}_p = \{p' | p' = PI, I \in \mathcal{G}_J\}$, and $p = PJ$. Let $u_i = Ps_i$ for $i = 1,...,n-1$, then $\tilde{\mathcal{G}}_p$ can equivalently be written as $\tilde{\mathcal{G}}_p = \{p' | p' = p + \sum_{i=1}^{n-1} a_i u_i,\ 0 \le a_i < 1\}$. We can also view the processor space as being constructed by the stacking of such $\tilde{\mathcal{G}}_p$'s in the directions of $\pm u_1,...,\pm u_{n-1}$. It can be shown that if λ_i is selected for the partitioning of the index space, then the reduced-size array will have a width $|u_i|$ in the direction of u_i.

2.5 Design of fixed-size arrays: a procedure

In this section we give a procedure to summarize our fixed-size systolic array design methodology.

Procedure: For a given regular algorithm $< \mathcal{I}, F, \mathcal{D} >$,

1. Determine a scheduling vector λ. λ must satisfy the condition $\lambda^t d \geq 1$ for all $d \in \mathcal{D}$.

   If there are more such scheduling vectors, then we select the scheduling vector λ which gives an optimal execution time, that is, the λ which minimizes

   $$t = max_{I \in \mathcal{I}} \lambda^t I - min_{J \in \mathcal{I}} \lambda^t J.$$

2. Determine $n-1$ coprime vectors λ_i, $i = 1, ..., n-1$, such that:

   (2a) the vectors $\lambda, \lambda_1, ..., \lambda_{n-1}$ are linearly independent, and

   (2b) the number of λ_i's, $1 \leq i \leq n-1$, which satisfy the condition $\forall d \in \mathcal{D}$: $\lambda_i^t d \geq 0$ is maximal.

3. Determine the size parameters of the fixed-size array $w_i > 0$, $i = 1, ..., n-1$. The size reduction effect of the processor-clustering operations (both active and passive) should be taken into account in the determination of the w_i's. (See step 5.)

4. Choose appropriate u and P, determine (conceptually) a full-size array, and then a reduced-size array.

5. Cluster the processors when necessary. We cluster the processors in the following cases:

   (5a) The efficiency of the processors is less than 1.

   (5b) The number of λ_i's that satisfies condition (2b) is fewer than $n-1$.

Note that we can still cluster active processors to balance the processor's I/Os and local memories when the number of λ_i's that satisfies condition (2b) is equal to $n-1$.

Remark 2.5 In practice, it is desirable to determine the λ_i's in such a way that the I/O processors of the array will be on (or near) the boundary of the array. For this topic see [Rao85], [Kun88] (section 3.4.2) for more details.

Remark 2.6 Since the size of the index space $\mathcal{I}$ may depend on a set of parameters, the λ that minimize the function $max_{I\in\mathcal{I}}\lambda^t I - min_{J\in\mathcal{I}}\lambda^t J$ may not exist in step 1 of the procedure. We will then select a λ according to some other criteria that are relevant to the application. Details are omitted here, they are beyond the scope of this chapter.

Note that we can use this procedure to design an optimal full-size array for an RA which has the shortest execution time, 100% processor-efficiency, and using a minimum number of processors. What we have to do is the following:

1. determine an optimal scheduling vector λ which gives the shortest execution time of the RA (see step 1 of the procedure);

2. choose the projecting vector u (and P) in such a way that the number of processors in the clustered array after the passive-clustering is minimal. That is, to choose u (and P) that minimizes $|\tilde{\mathcal{P}}| = \frac{1}{\lambda^t u}|\tilde{\mathcal{I}}|$ (see section 2.3 for the definitions of $\tilde{\mathcal{P}}$ and $\tilde{\mathcal{I}}$).

As an example, let us consider the Warshall-Floyd algorithm [AHU74] for the computation of the transitive closure (and the shortest path) problem in graph theory. It is shown in [KLL87] (see also [Kun88]) that this algorithm can be converted into a regular algorithm, which has $\lambda = (1,1,3)^t$ as an optimal scheduling vector. The index space of the algorithm is given by $\mathcal{I} = \{(i,j,k)^t | 1 \leq i,j,k \leq N\}$. As described in [KLL87], an optimal systolic array for this algorithm can be obtained by choosing $u = (1,0,0)^t$, and $P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. It contains N^2 processors, and the processors are 100% efficient.

However, we can also choose $u=(0,0,1)^t$, and $P=\begin{bmatrix}1&0&0\\0&1&0\end{bmatrix}$, which will give us a systolic array containing $N^2/3$ processors after the passive-clustering of processors (note that $\lambda^t u = 3$). It is shown recently [SC90] that this number of processors is minimal for the time-optimal systolic array implementation of the re-indexed Warshall-Floyd algorithm described in [KLL87].

2.6 Concluding remarks

We have present in this chapter a general technique for the clustering and partitioning of systolic arrays. A systematic procedure for designing fixed-size systolic arrays is given. Our method is a generalization of the methods described in [Jai86], [MF86], [HNS87], [ND88], [Jai89], and [Kun88].

The processor clustering techniques described in section 2.3 can also be used to design systolic arrays under the constraint of the number of processors [HNS87]. They can further be used to design systolic arrays with an optimal (i.e., minimum) execution time and high processor utilization (100%); only one of them can be optimized without the (passive) clustering of processors, see [Mol83], [Rao85], [RK88a].

In the discussion, we have ignored the finiteness of index space and processor space by introducing dummy index points and processors. When more information about the algorithm's index space is available, such dummy computations and processors should be kept to a minimum.

The described partitioning technique can also be used in the design of wavefront arrays and the parallel processing of uniform recurrence equations on multiprocessors.

A computer-aided design tool is in development to support this systolic array design procedure.

Chapter 3

Systolic Array Implementation of Nested Loop Programs

3.1 Introduction

In this chapter we deal with the problem of the systematic design of systolic arrays for algorithms that are given in the form of imperative nested *loop* programs, e.g., do-loops in Fortran or for-loops in Pascal-like programming languages.

Since Kung and Leiserson introduced a class of regular and locally connected processor arrays, known as systolic arrays, for high-speed VLSI implementation of application-specific and computation-intensive algorithms [KL80], [Kun79], [Kun80b], [Kun82], [Kun84] systolic array architecture has received great attention. A large number of such arrays have been proposed for solving problems from a wide spectrum of applications, including numerical linear algebraic problems [Kun79], [Kun80a], [Kun80b], [HC80], [Qui84], [Rao85], combinatorial problems [GKT79], [Che86], graph-theoretic problems [KLL87], and many others.

In systolic array design, important progress made is the development of systematic systolic array design methodology. Various design concepts and methods are proposed for the systematic design and analysis of systolic arrays, including [Kun80a], [Qui84], [CS84], [Mol83], [MW84], [MF86], [Che86], [DI86], [Kun84], [Kun88]. Based on the pioneering work of Karp, Miller and Winograd [KMW67], Rao *et al.* [Rao85], [RK86], [RK88a] have found that algorithms which are imple-

mentable on systolic arrays belong to a subclass of algorithms called regular iterative algorithms (RIAs).

These results have pointed out that an algorithm must possess regular data dependences in order to be implementable on a systolic array; we will refer to such algorithms as regular algorithms (RAs). An RA can be characterized by an index space $\mathcal{I} \subset \mathbf{Z}^n$, an input-output mapping F, and a set $\mathcal{D}$ of dependence vectors. Each point in $\mathcal{I}$ represents a computation defined by F, and the computations depend regularly on each other: for each I in $\mathcal{I}$, I depends on $I-d$ for all d in $\mathcal{D}$ (if $I-d$ is not in $\mathcal{I}$ then it is considered to be an input of the RA).

However, the number of algorithms which are available in RA form is rather limited. Many algorithms which have been shown to be implementable on systolic arrays were not initially available in this form, and the conversion of an algorithm into an RA is generally not an easy task. Usually, heuristic methods are used to derive RA(s) from an initial algorithm specification, see e.g., [Rao85], [FGQ86], [Mol83], [LK88], [KD89]. These methods rely on the designer's inspections, and they are, therefore, error prone. Moreover, using such heuristic methods for the construction of high-level design specifications will seriously degrade the reliability of a design.

Recently, several formal techniques have been proposed to solve this algorithm conversion (otherwise called localization) problem for algorithms which are expressed in (a) affine recurrent equations (AREs) [DI85], [Raj86], [DI86], [RF87], [WD88], [vDQ88], or (b) weak single-assignment codes (WSAs) [RTRK88], [Thi89a], or (c) imperative nested loop programs (NLPs) [BD88b], [BD88a], [BD89a]. A brief overview and comparison of these techniques is given in section 3.2; see also [BD89a].

In this chapter we present a formal and systematic method, based on our previous work [BD88b], [BD88a], [BD89a], to convert (when possible) a class of imperative nested loop programs into RAs. Once an RA version of such a loop program is obtained, the technique described in the previous chapter can be used to pursue the systolic array design. The reason why we consider these loop programs is that there are many algorithms from a wide spectrum of applications available in this form (design and use of such programs will certainly continue in many domains of

interest). They are easy to test and to modify in practice since they can be directly compiled and executed on a computer. Moreover, it is advantageous to start a design with an executable algorithm description with desired numerical properties.

Since the intended execution of a loop program is serial, that is, the iterations of the program will be executed one by one according to the ordering specified by the program loops (also the statements within each iteration will be executed serially as they appear in the body of the loop), data dependences in such programs are hidden, not explicit in general. In contrast to these loop programs, data dependences in regular algorithms are given explicitly in a localized form (i.e., each computation in an RA depends only on some neighboring computations). Therefore, a loop program can only be transformed into a regular algorithm if (1) data dependences in the program can be extracted, and expressed explicitly, and (2) the extracted data dependences can be localized. Both these tasks are not trivial, see Example 3.I.

To avoid drifting into too much detail, we only examine data dependences between different iterations of a nested loop program. This is done by transforming the body of a program into a functional mapping, and ignoring the computation details within each iteration. By doing so, each nested loop program in the class we consider can be represented by a dependence graph whose structure is independent of the value of the variables of the program. Characteristic properties of the dependence graph of the nested loop programs are given. In terms of dependence graphs, we first describe a procedure to transform a nested loop program into a single-assignment form, in which each variable will be assigned to a unique value during its execution, and dependences are expressed explicitly in the form of algebraic expressions. Then we discuss the computation and the localization of the data dependences. To illustrate the method, we will apply our procedure to convert the nested loop programs that describe matrix-matrix multiplication, matrix LU decomposition, and matrix QR decomposition, respectively, into regular algorithms.

The rest of the chapter is organized as follows. In the next section we discuss related work. In section 3.3 we define and characterize the class of nested loop programs we consider. In section 3.4 we define dependence graphs for the nested loop programs, and we discuss their characteristic properties. In section 3.5 we

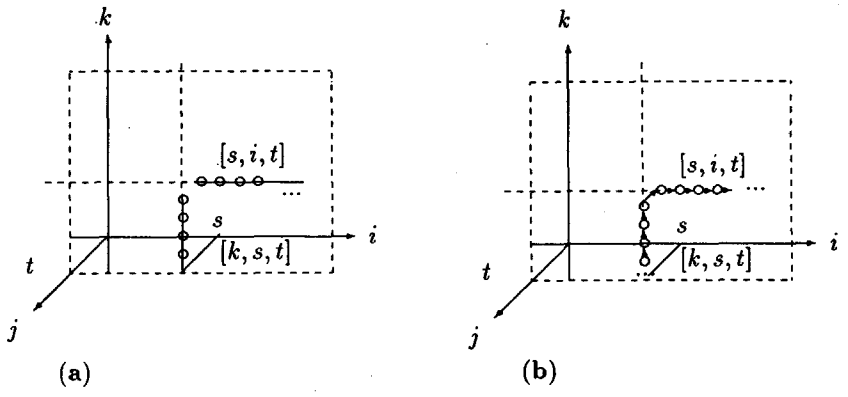


Figure 3.1: (a) The set $\mathcal{A}$. (b) Dependences between iterations in $\mathcal{A}$

discuss the computation of data dependences of nested loop programs and their transformation into single-assignment programs. In section 3.6 we present a strategy to localize broadcast data dependences in the nested loop programs. A necessary condition for a nested loop program to be systolic is given. Finally, in section 3.7 we give several examples to illustrate the method presented in this chapter.

Example 3.I:

```
for k = 1 to N − 1
    for i = k + 1 to N
        for j = k to N
            (a[k, j], a[i, j]) := F(a[k, j], a[i, j]);
```

In this program, each iteration will take two elements a_{kj} and a_{ij} from a matrix $A = [a_{ij}]$, perform some arithmetic operation (the actual functionality of F is immaterial here), and store them back in matrix A. ($a[k, j]$ and $a[i, j]$ of the program will address a_{kj} and a_{ij} of A respectively). Note that each iteration can be represented uniquely by a tuple $[k, i, j]$ or a point (index point) in 3-dimensional space. We denote the set of iterations of this program by $\mathcal{I}$, and call it the iteration space of the program.

It is not obvious at all in this program which iterations depend on which. Mental execution of the program is usually carried out in order to determine the data dependences in the program. Let s, t be fixed, and consider the iterations in the set $\mathcal{A}_{st} = \{[s,i,t] | [s,i,t] \in \mathcal{I}\} \cup \{[k,s,t] | [k,s,t] \in \mathcal{I}\}$, see Fig. 3.1a, where the dashed part of the lines does not belong to the iteration space. Then it can be shown that an iteration I in $\mathcal{A}_{st}$ will depend on the iteration $I-[1,0,0]$ if $k<s$; I will depend on $I-[1,1,0]$ if $k=s$ and $i=s+1$; and I will depend on $I-[0,1,0]$ otherwise (see Fig. 3.1b). We will show in section 3.7 how to derive such dependences, using the formal methods presented in the paper, for algorithms such as LDU and QR decomposition of matrices.

The set $\mathcal{A}_{st}$ is essentially defined by the indices $[k,j]$ and $[i,j]$ of the variables $a[k,j]$ and $a[i,j]$ respectively. Altering the index of variables may result in totally different dependency relations. For instance, let us consider an array element, say $a[5]$, then the dependence structure of program P-1 below is shown schematically in Fig. 3.1c, and that of P-2 in Fig. 3.1d.

```
for i = -N to N                                          (P-1)
    for j = -M to M
        (a[j], a[j + d]) := F(a[j], a[j + d]);

for i = -N to N                                          (P-2)
    for j = -M to M
        (a[i - j], a[i + j]) := F(a[i - j], a[i + j]);
```

These examples illustrate that the sequence of repeatedly reading, updating and writing the value of a particular variable instance may produce a complicated trajectory in the program's iteration space.

□

3.2 Related work

Methods for mapping imperative nested loop programs to systolic arrays have been proposed by several authors. An overview of these methods can be found in

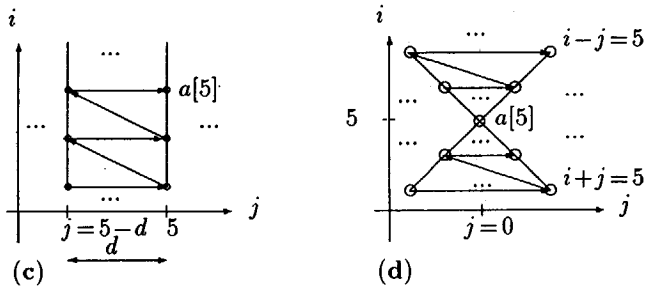


Figure 3.1: (c) Dependences in P-1. (d) Dependences in P-2

[FFW85]. A typical approach can be found in [Mol82], [Mol83] (see also [MW84]). As described in [Mol83], all variables in a loop program must first be "pipelined" (otherwise called "localized"), and a set of dependence vectors must be found. The program can then be transformed into a systolic array using affine (linear) transformations. However, these methods derive dependence vectors in a heuristic way. Therefore, they do not provide a satisfactory answer to the problem addressed in this chapter. It is our objective to determine and localize data dependences of a loop program in a systematic way.

Nested loop programs, especially Fortran do-loops, have been an important source of parallelism in numerical programs. They have been studied for a long time in compiler techniques. Various compiler techniques have been developed to detect (statement-level) parallelism and concurrency of operations in nested loop programs in order to recompile them to accommodate vector, concurrent, and multiprocessor computers. See [Kuc78], [PKL80], [PW86] for an overview of these techniques. Some ideas encountered in systolic array design for loop programs are very close to those used in compiler techniques. For instance, the systolic array design techniques described in [Mol82], [Mol83], [MW84] are similar to a compiler technique called the wavefront method [KMC72], [BCKT79], [PKL80] (also known as hyperplane method [Lam74], or loop reindexing method [Kuh80]) for parallel execution of the iterations of a nested loop program. In fact, the systolic array design method described in [Mol83], [MF86] is an extension of that of [Kuh80], see [FFW85].

However, these compiler techniques do not determine the data dependence relations in loop programs in detail. Data exchanges between concurrent operations are not made explicitly, instead, they are controlled by some synchronization techniques [Kuc78], [PKL80], [PW86] during the program execution. Since systolic arrays have only limited inter-processor communication capability, it requires a closed match between data dependences in a loop program and inter-processor communication links in a systolic array. For this reason, the use of these compiler techniques in mapping nested loop programs to systolic arrays is rather limited.

In our method, we examine in a formal and systematic way the data dependences in loop programs in detail. We determine which iterations depend strictly on which in a nested loop program, instead of determining which operations are independent (i.e., parallelism) as in the case of parallel/vectorizing compilers. Moreover, we modify (localize) data dependences of these programs, without changing their functionality and numerical properties, to make them suitable for systolic array implementation.

The problem of transforming an algorithm specification others than nested loop program into a regular algorithm for systolic array implementation has also been addressed in the literature, including [Raj86], [RF87], [WD88], [RTRK88], [Thi89a]. A frequently used model for the initial algorithm specification is called affine recurrent equations (AREs). An ARE is an equation of the form

$$\forall I \in \mathcal{I}: x[I] = f(..., y[AI+b], ...),$$

where $\mathcal{I} \subset \mathbf{Z}^n$, $\mathbf{Z}$ is the set of all integers, $n \geq 1$ is an integer, A is an $n \times n$ matrix, and b is an $n \times 1$ vector. A key feature of AREs is that the ordering of the evaluations of the function f over all the index points I in $\mathcal{I}$ is implied by the data dependences, not pre-specified as in the case of nested loop programs. Also, data dependences in AREs can be expressed explicitly as functions of I, like $I-AI-b$ in the above ARE. In order to transform AREs into RAs, data dependences have to be localized. Various techniques have been proposed for the localization of such data dependences, see e.g., [Raj86], [RF87], [vDQ88], [WD88]. However, AREs have a serious shortcoming in that their computability is difficult (if not impossible) to determine. Hence the

correctness of an algorithm specified in this form is hard to verify. This is also one of our motivations to look at an alternative model, in this case, imperative nested loop programs (NLPs), for algorithm specification.

In [RTRK88], [Thi89a], a general technique is described for the localization of data dependences in weak single-assignment codes (WSAs). A WSA contains equations of the form

$$\forall I \in \mathcal{I} : x[BI + c] = F(..., y[AI + b], ...).$$

WSAs are a generalization of AREs. In order to guarantee the uniqueness of the results, F has to contain commutative and associative operators.

Comparing nested loop programs with these models, NLPs do not have the disadvantages and the limitations of AREs and WSAs. However, the extraction of data dependences in NLPs is, in general, a difficult problem. Also, the ordering of executions in NLPs, an additional constraint imposed by program loops, has to be taken into account in the localization of data dependences.

The localization problem has also been addressed at the architectural level in parallel and array processing [Kuc78], [NS82], [FM84]. In [FM84] a method is described for determining whether and how data broadcasts in (regular) processor arrays can be eliminated or reduced by choosing an adequate linear schedule. In [LRS83], [LS83], a localization technique called system-retiming is described. In this method, broadcasting variables are first propagated along a (breadth first) spanning tree of the connection graph of the system, and then delays are inserted (when necessary) via system-retiming. However, these methods assume that the target system architecture is already known.

3.3 Nested loop programs

In this section we define and characterize the class of nested loop programs we consider.

Nested loop programs

The nested loop programs (NLPs) we consider are of the following form:

For $i_1 = f_{l,1}$ step 1 to $f_{u,1}$
  ..., ...
  For $i_n = f_{l,n}$ step 1 to $f_{u,n}$
  begin
    S_1;
    ...
    S_K;
  end

where i_j, $j = 1, \ldots, n$, are referred to as index variables or loop indices, and $f_{l,j}$, $f_{u,j}$ are referred to as boundary functions which are integer-valued functions (expressions), possibly involving loop indices $i_1, \ldots, i_{j-1}$. Without loss of generality, each loop is assumed to have a unit increment (see also [PW86], [Ban88]).

The body of the loop consists of a sequence of assignment statements $S_1; \ldots; S_K$. In each iteration, these statements will be evaluated in the given ordering.

Let S be a statement in the loopbody, then we use $L(S)$ to denote the set of left-hand-side variables of S (usually, S has only one left-hand-side variable), and use $R(S)$ to denote the set of right-hand-side variables of S. We say that a variable is *updated* in S if it is in $L(S)$, and a variable is *used* in S if it is in $R(S)$; we simply say that a variable is *in* S (or in the loopbody) if it is used and/or updated in S (or a statement). For instance, let S_1 be the following assignment

$$a[i,j] := a[i,j] - c * a[k,j]/b[k]; \qquad (S_1)$$

then we have $L(S_1) = \{a[i,j]\}$, $R(S_1) = \{a[i,j], c, a[k,j], b[k]\}$. The variables in S_1 are $a[i,j]$, $a[k,j]$, $b[k]$, and c.

We assume that all the variables in the loopbody are of the form $v[f(I)]$, where v is the name of the variable, $f(I)$ is the indexing function (or just the index) of the variable, and $I = [i_1, \ldots, i_n]^t$ is called an iteration vector; the superscript t denotes the vector transposition. All the indexing functions of variables are assumed to

be affine functions of I, that is, $f(I)$ is of the form $AI + b$, where $A \in \mathbf{Z}^{m \times n}$ (A is referred to as the indexing matrix of the variable), $\mathbf{Z}$ denotes the set of all integers, $b \in \mathbf{Z}^m$, and $m \leq n$. (This assumption is commonly made in the study of parallelism in nested loop programs, see e.g. [Lam74],[BCKT79],[FM84].) For instance, the name of the variable $a[i,j]$ in statment S_1 above is "a", and assume $I = [i,j,k]^t$, then the indexing function of $a[i,j]$ is $f(I) = (i,j)^t = AI$, where indexing matrix $A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$. In this way, we treat $a[i,j]$ and $a[(i,j)^t]$ (and $a[f(I)]$, and $a[AI]$) as equivalent notations for the same variable. Scalar variables (i.e., variables without indices) are treated as variables with zero indexing functions. For instance, the scalar variable c in S_1 will also be written as $c[0]$, with an indexing function $f(I) = 0I = 0$.

We assume that the set of variables in the loopbody is *consistent*, meaning that the variables with identical names have the same number of indices.

Note that we can have the following interpretations of a variable: (a) as an element of an array, like $a[i,j]$, where i and j may assume some values, and the name of the array is given by the name of the variable; (b) as a symbol, that is, as it appeared in the loopbody, like $a[i,j]$, $a[k,j]$, $b[k]$, etc. (here we mean: "$a[i,j]$", "$a[k,j]$", "$b[k]$", etc.). Since we use both interpretations in our discussion, we will use the word variable in its usual sense to avoid any confusion, that is, as in case (a), and we will refer to a variable in case (b) as a *defining* variable. A defining variable is uniquely determined by its name v and the formal definition of its indexing function.

Variables in the sets $L(S)$ and $R(S)$ are treated as defining variables. In the discussion, we also denote a defining variable $v[f(I)]$ by $v[f]$; $v[f]$ will address the array element $v[f](I) \equiv v[f(I)]$ at iteration I. We will omit the qualifier "defining" if it is clear from the context.

We assume that arrays with different names are disjoint, that is, two arrays will have no element in common if they have different names. This excludes the possibility of the overlapping of arrays with different names, such as using both $x[i]$ and $y[i]$ to address the same element of an array.

The body of the program

Let S_i, $i=1,...,K$, be the statements in the loopbody, and define

$$\mathcal{U}=\cup_{i=1}^{K} L(S_i), \text{ and}$$
$$\mathcal{W}=\cup_{i=1}^{K}[R(S_i) \ominus (\cup_{j=0}^{i-1} L(S_j))],$$

where S_0 is a dummy statement with $L(S_0)=\emptyset$, the empty set. ($A \cup B$ denotes the union of sets A and B, and $A \ominus B$ denotes the difference of sets A and B: $A \ominus B=\{x|x \in A, x \notin B\}$.)
The set $\mathcal{U}$ contains all the variables that are updated in the loopbody, and the set $\mathcal{W}$ contains all the variables whose first occurrence in the loopbody is at the right-hand-side of a statement. In this way, the loopbody can be viewed as a mapping $\mathcal{W}(I) \mapsto \mathcal{U}(I)$ [KMC72], where $\mathcal{U}(I)$ and $\mathcal{W}(I)$ denote the array elements addressed by the defining variables in $\mathcal{U}$ and $\mathcal{W}$ in iteration I respectively.

Example 3.II: Let us consider the following loop program:

For $i = 1$ step 1 to N
begin
$a[i] := b[i+3] * a[i] + (c[i])^2;$ (S_1)
$d[i] := a[i] - e * b[i] + c[i];$ (S_2)
end

For this program we have

$$L(S_1) = \{a[i]\},$$
$$L(S_2) = \{d[i]\},$$
$$R(S_1) = \{b[i+3], a[i], c[i]\}, \text{ and}$$
$$R(S_2) = \{a[i], e, b[i], c[i]\}.$$

Therefore,

$$\mathcal{U} = L(S_1) \cup L(S_2)$$
$$= \{a[i], d[i]\};$$

$$\begin{aligned}\mathcal{W} &= [R(S_1) \ominus L(S_0)] \cup [R(S_2) \ominus (L(S_1) \cup L(S_0))] \\ &= R(S_1) \cup (R(S_2) \ominus L(S_1)) \\ &= \{b[i+3], a[i], c[i], e, b[i]\}.\end{aligned}$$

The body of the loop in this example can then be viewed as a mapping

$$(b[i+3], a[i], c[i], e, b[i]) \mapsto (a[i], d[i]),$$

which is implemented by the two statements S_1 and S_2 of the program.

□

The functionality of the loopbody. In the discussion, we will consider the body of the loop as a (multi-valued) *function* which takes the variables in $\mathcal{W}(I)$ as its arguments and assigns the result of each evaluation (iteration) to the variables in $\mathcal{U}(I)$, i.e., as $\mathcal{U}(I) := F(\mathcal{W}(I))$. An important property of F (as a function) is that its functionality is independent of its formal arguments. For instance, $F(x, y) = x + y$ defines the same function as $F(x, z) = x + z$.

However, the functionality of the loopbody, as it is given (i.e., a sequence of assignments), does not necessarily possess this property. It may depend implicitly on both the index point I and the indexing function of the variables, and replacing a variable in the loopbody by a new variable will eventually change its behavior. To see this, let us consider the following loopbody:

$$\begin{aligned}&a[f_1(I)] := ...; && (S_1)\\ &\quad\cdots \qquad\qquad \cdots \\ &a[f_2(I)] := H_1(..., a[f_2(I)], ...); && (S_i)\\ &\quad\cdots \qquad\qquad \cdots \\ &b[g(I)] := H_2(..., a[f_1(I)], ..., a[f_2(I)], ..., a[f_3(I)], ...); && (S_j)\end{aligned}$$

The dependence of its functionality and the indexing function, say, f_2, and index point I, is not explicit in this description. However, if we replace in each statement variable $a[f_2(I)]$ by a new variable, say $a[h(I)]$, with $h \neq f_2$, then the functionality of the loopbody will be changed. This is because in the original loopbody, when a new value is assigned to $a[f_2(I)]$ in iteration I, say, after the execution of the statement

S_i, if $f_2(I) = f_1(I)$ in this iteration, then this new value of $a[f_2(I)]$ will be assigned implicitly to $a[f_1(I)]$; similarly to $a[f_3(I)]$. Such implicit assignment of the value of variables will be (partially) lost after $a[f_2(I)]$ is replaced by $a[h(I)]$.

We can avoid such problems by making all the implicit assignments of the value of variables in the loopbody explicit. In this example, we can do this by inserting the following statements immediately after S_1:

$$\begin{cases} \text{if } f_2(I) = f_1(I) \text{ then } a[f_2(I)] := a[f_1(I)]; \\ \text{if } f_3(I) = f_1(I) \text{ then } a[f_3(I)] := a[f_1(I)]; \end{cases}$$

and insert the following statement immediately after S_i:

$$\begin{cases} \text{if } f_1(I) = f_2(I) \text{ then } a[f_1(I)] := a[f_2(I)]; \\ \text{if } f_3(I) = f_2(I) \text{ then } a[f_3(I)] := a[f_2(I)]; \end{cases}$$

before the variable substitution.

In general, let S_1,...,S_K be the loopbody of an NLP, where each statement is of the form

$$v[f(I)] := ...; \qquad (S)$$

Let us insert a new statement

$$\text{if } g(I) = f(I) \text{ then } v[g(I)] := v[f(I)];$$

for each variable $v[g(I)]$, $g \neq f$, of the program immediately after each statement S. (Clearly, we need only to do this if the equation $f(I) = g(I)$ has a solution in $\mathcal{I}$.) Then it is easy to show that (a) the functionality of the loopbody remains unchanged, and (b) the functionality of the loopbody is independent of the variables used in the statements. Note that the inserted statements now becomes a part of the loopbody. From the resulting loopbody a functional description can be extracted in a straightforward way.

From now on we will consider the loopbody as a function $\mathcal{W}(I) := F(\mathcal{U}(I))$, and ignore the actual functionality of F and the data dependences within F.

Remark 3.1 There are compiler techniques available to detect statement-level data dependences, and to determine whether the loopbody can be split up into smaller ones. See, e.g., [Kuc78], [PKL80], [PW86] for more details. For instance, the loop program of Example 3.II can be split into the following two programs:

For $i = 1$ step 1 to N
    $a[i] := b[i+3] * a[i] + (c[i])^2$; (S_1)
For $i = 1$ step 1 to N
    $d[i] := a[i] - e * b[i] + c[i]$; (S_2)

We will not consider such loop programs in our discussion. However, our results can be extended to handle programs like this.

The iteration space of the program

From the loop specification of the program, we can easily deduce an iteration space (otherwise called an index space) $\mathcal{I}$ of the program:

$$\mathcal{I} = \{I \mid I = [i_1, \ldots, i_n]^t, f_{l,1} \leq i_1 \leq f_{u,1}, \ldots, f_{l,n} \leq i_n \leq f_{u,n}\}.$$

When the program is executed, the body of the loop will be evaluated $|\mathcal{I}|$ times, $|\ |$ denotes the cardinality of a set, once for each $I \in \mathcal{I}$ in the ordering defined by the program loops. Therefore, the program loop specification defines an ordering on $\mathcal{I}$ in addition to an iteration space $\mathcal{I}$: for any $I, J \in \mathcal{I}$, $I \prec J$ if the iteration I is evaluated before the iteration J. We denote this ordering by $\prec$, and we write $I \preceq J$ if $I \prec J$ or $I = J$. (Note that the ordering $\prec$ is consistent with the lexicographical ordering).

Proposition 3.1 *Let $I, J \in \mathcal{I}$, then, $I \prec J$ if and only if the first nonzero component of $J - I$ is positive.*

Proof: Trivial, since all the loops have positive increment 1 by assumption.

□

In this way, the program loop specification can be characterized by a completely ordered set $<\mathcal{I}, \prec>$. To exclude trivial programs, we assume that $\mathcal{I}$ is not empty.

Remark 3.2 The iteration space $\mathcal{I}$ will be partially ordered if some of the loops are specified by "do-in-parallel" like constructions. We assume in our discussion that no such constructions are involved in the program loops. The results can however be generalized to handle partially ordered iteration spaces.

Note that it is not always possible to generate (enumerate) the points in the iteration space at compile time. For instance, some boundary functions may involve parameters which are unknown at compile time. In this case, the program specifies, in fact, a class of algorithms that differ only in the size of their iteration spaces. We assume in our discussion that the iteration space of an NLP is parameterized, and that it cannot be enumerated at compile time.

Moreover, we assume in the discussion that the iteration space $\mathcal{I}$ is convex. That is, it contains all the points with integer coordinates (or integral points/vectors) in a convex subset of $\mathbf{R}^n$, with $\mathbf{R}$ the set of reals.

Recall that the intersection of a finite number of half-spaces is called a convex polytope, or simply polytope, when it is bounded and nonempty [PS82].

Proposition 3.2 *If the boundary functions $f_{l,j}$, $f_{u,j}$, $j = 1, \ldots, n$, are affine functions of $i_1, \ldots, i_{j-1}$, then the iteration space $\mathcal{I}$ is convex, containing all integral vectors in a convex polytope.*

Proof: Let for all $j = 1, \ldots, n$,

$$f_{l,j} = \underbrace{[*, ..., *}_{j-1}, 0, ..., 0][i_1, ..., i_n]^t + b_{l,j} = A_{l,j}I + b_{l,j}, \text{ and}$$
$$f_{u,j} = \underbrace{[*, ..., *}_{j-1}, 0, ..., 0][i_1, ..., i_n]^t + b_{u,j} = A_{u,j}I + b_{u,j},$$

where $*$ represents an integer. Then the jth loop $f_{l,j} \leq i_j \leq f_{u,j}$ can be equivalently written as

$$A_{l,j}I + b_{l,j} \leq e_j^t I \leq A_{u,j}I + b_{u,j},$$

or equivalently,

$$(A_{l,j} - e_j^t)I \leq -b_{l,j}, \text{ and } (e_j^t - A_{u,j})I \leq b_{u,j},$$

where e_j^t is a column vector with zero components except for a 1 at the jth position.

Therefore, we have the convex polytope

$$\mathcal{I} = \{I \mid (A_l - I_{n\times n})I \leq -b_l, \ (I_{n\times n} - A_u)I \leq b_u\},$$

where A_l and A_u are strictly lower triangular matrices of integers (or integral matrices), b_l and b_u are integral column vectors, and $I_{n\times n}$ is the $n\times n$ identity matrix. □

The convexity of the iteration space is usually assumed in systolic array design since various optimization problems can be formulated as (integer) linear programming problems.

A summary

The nested loop programs (NLPs) we consider can be characterized by a 5-tuple $<\mathcal{I}, \prec, F, \mathcal{W}, \mathcal{U}>$, where

- $\mathcal{I}$ is the iteration space of the program; $\mathcal{I} \subset \mathbf{Z}^n$ is convex.
- For any $I, J \in \mathcal{I}$, $I \prec J$ if the first nonzero element of $J-I$ is positive.
- F is a (multi-valued) function, possibly depending on I.
- $\mathcal{U}$ and $\mathcal{W}$ are two sets of defining variables; variables in these sets are of the form $v[f]$, with $f(I)=AI+b$, $A \in \mathbf{Z}^{m\times n}$, $b \in \mathbf{Z}^m$, and $m \leq n$.
- The body of the loop is a mapping defined by $\mathcal{U}(I) := F(\mathcal{W}(I))$, and it will be evaluated for all I in $\mathcal{I}$ serially according to $\prec$.

When the functionality of the loopbody F is immaterial to the discussion; we will write the program as a 4-tuple $<\mathcal{I}, \prec, \mathcal{W}, \mathcal{U}>$ for short.

3.4 Dependence graph of a nested loop program

In this section we first show that each NLP is associated with a dependence graph (DG) whose structure is independent of the value of the variables. (We also call it an iteration-level dependence graph since it only displays data dependences between different iterations.) Then we define an embedding of the DG in the program's iteration space, and by doing so, dependences between iterations, say J depending on I, can be expressed by an integral vector $J-I$, called a dependence vector. A

characterization of the dependence vectors is given. Moreover, we show that the DG can be decomposed into subgraphs, which allows us to investigate the dependence structure of the DG by considering the dependence structure of its subgraphs.

3.4.1 The dependence graph

Dependence between iterations

Let $< \mathcal{I}, \prec, \mathcal{W}, \mathcal{U} >$ be an NLP, and let $\mathcal{I} = \{I_1, I_2, \ldots, I_M\}$ with $M = |\mathcal{I}|$ and $I_1 \prec I_2 \prec, \ldots, \prec I_M$. Then, the execution of the NLP will proceed according to the following sequence (II) of iterations:

$$\begin{array}{ll} I_1\text{:} & \mathcal{W}(I_1) \mapsto \mathcal{U}(I_1); \\ I_2\text{:} & \mathcal{W}(I_2) \mapsto \mathcal{U}(I_2); \\ & \ldots; \\ I_M\text{:} & \mathcal{W}(I_M) \mapsto \mathcal{U}(I_M); \end{array} \qquad \text{(II)}$$

Using this sequence, we define a *direct dependence* between two different iterations as follows:

An iteration J depends *directly* on an iteration I, via a variable, say $v[\xi]$, if

(Dependence conditions)

- There is a variable $v[\xi] \in \mathcal{U}(I) \cap \mathcal{W}(J)$; (a)
- $I \prec J$; (b)
- $v[\xi] \notin \mathcal{U}(I')$ for all $I' \in \mathcal{I}$ such that $I \prec I' \prec J$. (c)

This is because the value of $v[\xi]$ updated in I will be used in the evaluation of the iteration J (conditions (a) and (b)), and the value of $v[\xi]$ will not be changed in any of the iterations between I and J (condition (c)). We will denote such a dependence by $I \xrightarrow{v[\xi]} J$, or simply $I \rightarrow J$. If $I \rightarrow J$, then the iteration J has to be evaluated after the iteration I.

Proposition 3.3 *If $I \xrightarrow{v[\xi]} J$, then there is a variable $v[f] \in \mathcal{W}$ and a variable $v[g] \in \mathcal{U}$ (f and g are not necessarily distinct), such that $f(J) = g(I)$.*

Proof: This is a direct consequence of the dependence condition (a).

□

This proposition gives a necessary condition for an iteration J to be dependent on an iteration I. Note that direct dependences can only be defined by variables with equal names.

Example 3.III: Let us consider the following nested loop program for the computation of the convolution sum $y_i = \sum_{k=0}^{N} w_k x_{i-k}$ for $0 \leq i \leq N$:

For $i = 0$ step 1 to N
  For $k = 0$ step 1 to N
    $y[i] = y[i] + w[k] * x[i-k]$;

This algorithm can be written as $<\mathcal{I}, \prec, \mathcal{W}, \mathcal{U}>$, with

$$\mathcal{I} = \{[i,k]^t | 0 \leq i \leq N, 0 \leq k \leq N\};$$
$$\mathcal{U} = \{y[i]\};$$
$$\mathcal{W} = \{y[i], w[k], x[i-k]\}.$$

The execution of this program will proceed according to the following sequence of iterations:

$$\begin{array}{lll}
[0,0]^t: & \{y[0], w[0], x[0]\} & \mapsto \{y[0]\}; \\
[0,1]^t: & \{y[0], w[1], x[-1]\} & \mapsto \{y[0]\}; \\
[0,2]^t: & \{y[0], w[2], x[-2]\} & \mapsto \{y[0]\}; \\
 & \ldots & \ldots \\
[0,N]^t: & \{y[0], w[N], x[-N]\} & \mapsto \{y[0]\}; \\
 & & \\
[1,0]^t: & \{y[1], w[0], x[1]\} & \mapsto \{y[1]\}; \\
[1,1]^t: & \{y[1], w[1], x[0]\} & \mapsto \{y[1]\}; \\
[1,2]^t: & \{y[1], w[2], x[-1]\} & \mapsto \{y[1]\}; \\
 & \ldots & \ldots \\
[1,N]^t: & \{y[1], w[N], x[1-N]\} & \mapsto \{y[1]\}; \\
 & \ldots & \ldots \\
[N,0]^t: & \{y[N], w[0], x[N]\} & \mapsto \{y[N]\}; \\
[N,1]^t: & \{y[N], w[1], x[N-1]\} & \mapsto \{y[N]\}; \\
[N,2]^t: & \{y[N], w[2], x[N-2]\} & \mapsto \{y[N]\}; \\
 & \ldots & \ldots \\
[N,N]^t: & \{y[N], w[N], x[0]\} & \mapsto \{y[N]\};
\end{array}$$

From this sequence we can deduce that the iteration $[0,1]^t$ depends directly on the iteration $[0,0]^t$ via $y[0]$, and the iteration $[N,2]^t$ depends directly on $[N,1]^t$ via $y[N]$, etc.

□

Using the sequence Π, we define the program's inputs and (possible) outputs as follows.

- If $v[\xi] \in \mathcal{W}(J)$, and $v[\xi] \notin \mathcal{U}(I)$ for all $I : I \prec J$, then $v[\xi]$ in the iteration J is an input of the program.
- If $v[\xi] \in \mathcal{U}(I)$, and $v[\xi] \notin \mathcal{U}(J)$ for all $J : I \prec J$, then the value of $v[\xi]$ in the iteration I will not be changed, and it is an (possible) output of the program.

For instance, in Example 3.III, $y[1]$ in the iteration $[1,0]^t$ is an input of the program. The value of $y[1]$ should be initialized before the iteration $[1,0]^t$.

Remark 3.3 We can conclude from the above discussion that when there is a scalar variable in the set $\mathcal{W} \cap \mathcal{U}$, then all the iterations will have to be executed serially according to $\prec$. To see this, let x be a scalar variable, and assume that $x \in \mathcal{W}$ and $x \in \mathcal{U}$. Then $x \in \mathcal{W}(I)$ and $x \in \mathcal{U}(I)$ for all $I \in \mathcal{I}$. Therefore $x \in \mathcal{U}(I_i) \cap \mathcal{W}(I_{i+1})$. Hence, $I_1 \rightarrow I_2 \rightarrow, \ldots, \rightarrow I_M$. For this reason, scalar variables in $\mathcal{W} \cap \mathcal{U}$ should be eliminated when possible in order to increase the degree of parallelism of the program. Note that, however, we have not taken the properties of the operations performed in the loopbody into account here. Obviously, strict serial evaluation of the iterations of a program with loopbody $x := x \odot a[f(I)]$, where $a[f(I)]$ is an arbitrary variable, is not necessary if the operator $\odot$ is associative and commutative. In fact, loop programs like this belong to a class of algorithms called weak single-assignment codes (WSAs). Techniques for dealing with WSAs can be found in [RTRK88], [Thi88], [Thi89b] etc.

The dependence graph of the NLP

Referring to the sequence Π, we define (an unfolded) dependence graph (DG) of the NLP as follows.

- There are $|\mathcal{I}|$ nodes in the DG, each node represents one unique iteration in the sequence Π. The node representing the iteration I will be labeled I.

- The edges of the DG correspond to the direct dependences of the program: $(I, J, v[\xi])$ is an edge of the DG if and only if $I \xrightarrow{v[\xi]} J$.

  We also denote an edge $(I, J, v[\xi])$ by (I, J) when its label $v[\xi]$ is immaterial or is clear from the context.

For instance, the DG of the program in Example 3.III can be drawn as shown in Fig. 3.2 (inputs and outputs of the program are not shown).

Proposition 3.4 *The structure of the DG of a nested loop program is independent of the value of the variables of the program.*

Proof: Since the sequence Π and direct dependences are defined independently of the value of the variables of the program.

□

We also say that the DG is defined by the structure $< \mathcal{I}, \prec, \mathcal{W}, \mathcal{U} >$ in the discussion. Obviously, any structure $< \mathcal{I}, \prec, \mathcal{W}', \mathcal{U}' >$ ($\mathcal{W}' \subseteq \mathcal{W}$ and $\mathcal{U}' \subseteq \mathcal{U}$) defines a DG.

It is clear from the above discussion that the dependence graph is uniquely defined by the execution sequence Π, which is in turn defined by $< \mathcal{I}, \prec, \mathcal{W}, \mathcal{U} >$. However, since we have assumed that the iteration space $\mathcal{I}$ of the program is parameterized, it is not possible to generate the points in $\mathcal{I}$ at compile time. Therefore, we cannot explicitly generate the sequence Π, nor the DG.

Embedding the dependence graph in the iteration space

A geometrical interpretation of the dependence graph DG can be obtained by assigning each node of the graph to a unique point in an s-dimensional Euclidean space $\mathbf{R}^s$. We say that the dependence graph is embedded in $\mathbf{R}^s$ by an embedding function $e : \mathbf{Z}^n \mapsto \mathbf{R}^s$, which assigns each node $I \in \mathcal{I}$ to a unique point $e(I)$ in $\mathbf{R}^s$.

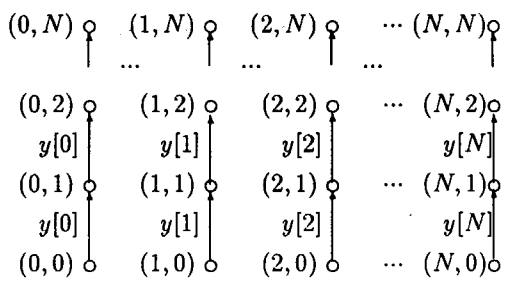


Figure 3.2: *DG* of the program in Example 3.III

An edge (I, J) in the dependence graph will then correspond to a vector $e(J) - e(I)$ in $\mathbf{R}^s$, called a dependence vector.

We assume from now on that the nodes of a *DG* are embedded in $\mathbf{Z}^n$ by an identity mapping $\mathbf{Z}^n \mapsto \mathbf{Z}^n$. That is, each node I of the *DG* is assigned to the point $I \in \mathbf{Z}^n$. Therefore, the set of nodes of the *DG* is equal to $\mathcal{I}$, and the dependence vector associated with an edge (I, J) is simply given by an integral vector $J - I$.

Proposition 3.5 *If J depends on I (i.e, $I \rightarrow J$), then the dependence vector $J-I$ is an affine function of I. That is, $J-I$ can be expressed in the form of $QI+p$, with $Q \in \mathbf{Q}^{n \times n}$ and $p \in \mathbf{Q}^n$, where $\mathbf{Q}$ is the set of rationals.*

Proof (sketch): Let $I \rightarrow J$. Then, according to proposition 3.3, there is a variable $v[f] \in \mathcal{W}$ and a variable $v[g] \in \mathcal{U}$ (f and g are not necessarily different indexing functions), such that $f(J) = g(I)$. Let $f(I) = AI + b$, and $g(I) = BI + c$. Then we have $f(J) = g(I) \Leftrightarrow AJ + b = BI + c \Leftrightarrow AJ = BI + c'$, with $c' = c - b$. It follows that J can be expressed as $J = Q'I + p$, with $Q' \in \mathbf{Q}^{n \times n}$, and $p \in \mathbf{Q}^n$ (see appendix 3.A for a constructive way to compute Q' and p in integer arithmetic). Hence $J - I = Q'I - I + p$, or equivalently, $J - I = QI + p$, with $Q = Q' - I_{n \times n}$, and $I_{n \times n}$ the $n \times n$ identity matrix. □

Note that in this proposition Q and p may depend on I and J since the variables $v[f] \in \mathcal{W}$ and $v[g] \in \mathcal{U}$ may be different in different iterations I and J.

3.4.2 A decomposition of the dependence graph

For each variable $v[f] \in \mathcal{W} \cup \mathcal{U}$ of the program we define a *variable dependence graph*, denoted by $DG^{v[f]}$, as follows:

- The $DG^{v[f]}$ has $|\mathcal{I}|$ nodes, and each node is labeled with one $I \in \mathcal{I}$. The node I of the graph represents the variable $v[f(I)]$ in the iteration I.
- The edges in the $DG^{v[f]}$ are defined as follows: $(I, J, v[\xi])$ is an edge of the $DG^{v[f]}$ if and only if (i) $(I, J, v[\xi])$ is an edge of the DG, and (ii) $f(I) = f(J) = \xi$.

Moreover, let $v[f], v[g] \in \mathcal{W} \cup \mathcal{U}$, and $f \neq g$, then we define

- $(I, J, v[\xi])$ is an edge from node I of $DG^{v[f]}$ to node J of $DG^{v[g]}$ if and only if (i) $(I, J, v[\xi])$ is an edge of DG, and (ii) $f(I) = g(J) = \xi$.

The set of variable dependence graphs $DG^{v[f]}$, $v[f] \in \mathcal{W} \cup \mathcal{U}$, gives us a more detailed representation of the data dependences between iterations of the NLP.

Let us partition the sets $\mathcal{W}$ and $\mathcal{U}$ into subsets which contain variables with equal names. That is, $\mathcal{W} = \cup_v \mathcal{W}^v$, and $\mathcal{U} = \cup_v \mathcal{U}^v$, where $\mathcal{W}^v$ and $\mathcal{U}^v$ denote the subsets of $\mathcal{W}$ and $\mathcal{U}$ containing the variables with name v respectively; $\mathcal{W}^v = \emptyset$ if the name v appears only in $\mathcal{U}^v$, and $\mathcal{U}^v = \emptyset$ if the name v appears only in $\mathcal{W}^v$. For instance, in Example 3.III we have $\mathcal{U} = \{y[i]\}$, $\mathcal{W} = \{y[i], w[k], x[i-k]\}$; here we have three variable names x, y, and w. And we have $\mathcal{W}^x = \{x[i-k]\}$, $\mathcal{U}^x = \emptyset$, $\mathcal{W}^y = \{y[i]\}$, $\mathcal{U}^y = \{y[i]\}$, and $\mathcal{W}^w = \{w[k]\}$, $\mathcal{U}^w = \emptyset$.

Let us define for each pair $<\mathcal{W}^v, \mathcal{U}^v>$ an abstract structure $<\mathcal{I}, \prec, \mathcal{W}^v, \mathcal{U}^v>$, and denote the dependence graph of such a structure by DG^v. Moreover, let $<V, E>$ be a graph, with V the set of nodes and E the set of edges, and let the union of two graphs $<V_1, E_1>$ and $<V_2, E_2>$ be defined as $<V_1 \cup V_2, E_1 \cup E_2>$.

Proposition 3.6 *The dependence graph DG is equal to the union of the DG^vs, that is, $DG = \cup_v DG^v$.*

Proof: Since (1) the set of nodes of DG and all the DG^vs are identical, given by $\mathcal{I}$; and (2) if $(I, J, v[\xi])$ is an edge of DG, then $(I, J, v[\xi])$ is an edge of DG^v defined by $<\mathcal{I}, \prec, \mathcal{W}^v, \mathcal{U}^v>$, and vice versa.

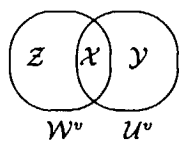


Figure 3.3: Partition of variable sets $\mathcal{W}$ and $\mathcal{U}$

□

Therefore, in the definition of $DG^{v[f]}$, we can replace the condition (i) "if $(I, J, v[\xi])$ is an edge of DG ..." by the condition "if $(I, J, v[\xi])$ is an edge of DG^v ...", and we can consider the $DG^{v[f]}$s of variables with different names separately.

The structure of DG^v and $DG^{v[f]}$

For each distinct name v of the variables we can only have one of the following cases:
case I: $\mathcal{W}^v \neq \emptyset$ and $\mathcal{U}^v = \emptyset$,
case II: $\mathcal{W}^v = \emptyset$ and $\mathcal{U}^v \neq \emptyset$, and
case III: $\mathcal{W}^v \neq \emptyset$ and $\mathcal{U}^v \neq \emptyset$.
Hence there will be only three types of DG^vs, each corresponding to one of these three cases.

In case I, since $\mathcal{U}^v = \emptyset$, variables in $\mathcal{W}^v$ will not define any dependences between iterations. Therefore, none of the $DG^{v[f]}$s will have (internal) edges. Moreover, there will be no edges between different $DG^{v[f]}$s. Each node I in $DG^{v[f]}$ represents an input of the program whose value has to be initialized in iteration I.

Case II is similar to case I. Since $\mathcal{W}^v = \emptyset$ in case II, none of the $DG^{v[f]}$s will have (internal) edges, and there will be no edges between different $DG^{v[f]}$s. However, the variables need not to be initialized in this case (some of them are possible outputs of the algorithm).

Dependences between iterations can therefore be defined only by variables in case III. For case III, let us partition the sets $\mathcal{W}^v$, and $\mathcal{U}^v$ into

$$\mathcal{W}^v = \mathcal{X} \cup \mathcal{Z}, \text{ and } \mathcal{U}^v = \mathcal{X} \cup \mathcal{Y},$$

respectively, with $\mathcal{X} = \mathcal{W}^v \cap \mathcal{U}^v$, $\mathcal{Y} = \mathcal{U}^v \ominus \mathcal{W}^v$, and $\mathcal{Z} = \mathcal{W}^v \ominus \mathcal{U}^v$ (see Fig. 3.3).

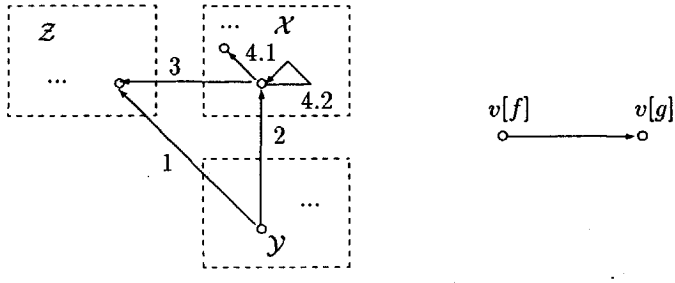


Figure 3.4: Possible dependences

Proposition 3.7 *Let $v[f], v[g] \in \mathcal{W}^v \cup \mathcal{U}^v$. If (I, J) is an edge from node I of $DG^{v[f]}$ to node J of $DG^{v[g]}$, then $v[f]$ and $v[g]$ will belong to one of the following cases (see Fig. 3.4):*

(1) $v[f] \in \mathcal{Y}$, $v[g] \in \mathcal{Z}$;

(2) $v[f] \in \mathcal{Y}$, $v[g] \in \mathcal{X}$;

(3) $v[f] \in \mathcal{X}$, $v[g] \in \mathcal{Z}$;

(4.1) $v[f] \in \mathcal{X}$, $v[g] \in \mathcal{X}$, *and* $f \neq g$;

(4.2) $v[f] \in \mathcal{X}$, $v[g] \in \mathcal{X}$, *and* $f = g$;

Proof: Cases (1)-(4) are direct consequences of the fact that if (I, J) is an edge from node I of $DG^{v[f]}$ to node J of $DG^{v[g]}$, then $v[f]$ must be in $\mathcal{U}^v = \mathcal{X} \cup \mathcal{Y}$, and $v[g]$ must be in $\mathcal{W}^v = \mathcal{X} \cup \mathcal{Z}$. Moreover, in case (4), there are two possibilities: (4.1) $f = g$, and (4.2) $f \neq g$. (Note that in case (1)-(3) $f \neq g$ by definition.)

□

Due to proposition 3.7, $DG^{v[f]}$ will have no internal edges if $v[f]$ is in $\mathcal{Y}$ or in $\mathcal{Z}$. Moreover, the in-degree of each node in $DG^{v[f]}$ is zero if $v[f]$ is in $\mathcal{Y}$, and the out-degree of each node in $DG^{v[f]}$ is zero if $v[f]$ is in $\mathcal{Z}$.

Note that if $v[f] \in \mathcal{Z}$ and none of the nodes in $DG^{v[f]}$ has in-coming edges, then we can handle $DG^{v[f]}$ in the same way as in case I. Similarly, if $v[f] \in \mathcal{Y}$ and none of the nodes in $DG^{v[f]}$ has out-going edges, then we can handle $DG^{v[f]}$ in the same way as in case II.

3.5 Converting NLPs into single-assignment programs

In this section we present a procedure for transforming an NLP into a *single-assignment* program, which has the property of that to each variable of the program a unique value will be assigned during the program execution. In this single-assignment program, dependence vectors will be expressed explicitly, in the form of algebraic expressions.

An NLP $<\mathcal{I}, \prec, F, \mathcal{W}, \mathcal{U}>$ can be transformed into a single-assignment form in terms of the variable dependence graphs ($DG^{v[f]}$s), as follows.

step 1. For each $DG^{v[f]}$ of the NLP, we assign to each node I of the $DG^{v[f]}$ a unique variable.

This corresponds to a simple variable substitution in the body of the NLP: we substitute for each variable $v[f] \in \mathcal{W} \cup \mathcal{U}$ a new and unique variable whose indexing function is an identity function. Since $v[f]$ is uniquely identified by its name v and its indexing function f, we will denote the new variable that is used to substitute $v[f]$ by $\phi_{v,f}[I]$.

After this, we obtain a new program; let us call it NLP-1. Moreover, we remove the execution ordering $\prec$ from the iteration space of NLP-1. Therefore NLP-1 is of the form $\forall I \in \mathcal{I}$: $\mathcal{U}' := F(\mathcal{W}')$, with all the variables in $\mathcal{U}' \cup \mathcal{W}'$ of the form $\phi_{v,f}[I]$.

step 2. Then we have to ensure that each variable $\phi_{v,f}[I] \in \mathcal{W}'$ of NLP-1 has at the beginning of each iteration I the same value as $v[f(I)]$ at the beginning of the iteration I of the original program NLP. This can be accomplished as follows.

For each $I \in \mathcal{I}$:

- in case I, we simply need to set the value of $\phi_{v,f}[I]$ to that of $v[f(I)]$ since the value of $v[f(I)]$ will not be changed during the execution of NLP;
- in case II, we do not need to set the value of $\phi_{v,f}[I]$ since the value of $v[f(I)]$ will not be used in the iteration I of NLP;

- in case III, we have to (i) set the value of $\phi_{v,f}[I]$ to that of $v[f(I)]$ if the node I of $DG^{v[f]}$ represents an input of NLP; or (ii) set the value of $\phi_{v,f}[I]$ to that of $\phi_{v,f}[I']$ if (I', I) is an edge of $DG^{v[f]}$; or (iii) set the value of $\phi_{v,f}[I]$ to that of $\phi_{v,g}[I']$ if (I', I) is an edge from node I' of $DG^{v[g]}$ to node I of $DG^{v[f]}$.

Step 2 of this transformation guarantees that $\phi_{v,f}[I]$ of NLP-1 will have the same value after iteration I as $v[f]$ of NLP after iteration I, for all $I \in \mathcal{I}$. Hence NLP-1 is input-output equivalent to the original program NLP.

Step 1 of the above transformation is straightforward, so we focus below on the second step. Only one pair $< \mathcal{W}^v, \mathcal{U}^v >$ will be considered since all the others can be handled in a similar way.

Expressing dependence vectors explicitly

Case I: $\mathcal{W}^v \neq \emptyset$ and $\mathcal{U}^v = \emptyset$.

As we said, for each $v[f] \in \mathcal{W}^v$, we have only to set the value of $\phi_{v,f}[I]$ to that of $v[f(I)]$ for all $I \in \mathcal{I}$ in this case. This can be expressed by the following (single-assignment) statement:

$$\forall I \in \mathcal{I}: \quad \phi_{v,f}[I] := v[f(I)]; \tag{I.1}$$

Let A be an integral matrix, and let $Nul(A)$ denote the set of integral vectors in the right null space of A, that is, $Nul(A) = \{I \mid AI = 0,\ I \in \mathbf{Z}^n\}$.

Proposition 3.8 *Let $f(I) = AI + b$ be the indexing function of $v[f]$. Then the same value will be assigned to $\phi_{v,f}[I]$ and $\phi_{v,f}[J]$, $I, J \in \mathcal{I}$ and $I \neq J$, if $I - J \in Nul(A)$.*

Proof: The value of $v[f(I)]$ will be assigned to $\phi_{v,f}[I]$, and the value of $v[f(J)]$ will be assigned to $\phi_{v,f}[J]$. If $I - J \in Nul(A)$, then and $f(I) - f(J) = A(I - J) = 0$. That is, $v[f(I)]$ and $v[f(J)]$ refer to the same array element.

□

Corollary 3.1 *Let $\mathcal{D}_{J,A}$, $J \in \mathcal{I}$, denote the set $\{J+J'|J' \in Nul(A)\} \cap \mathcal{I}$. Then the value $v[f(J)]$ will be assigned to $\phi_{v,f}[I]$, for all $I \in \mathcal{D}_{J,A}$. That is, $\forall I \in \mathcal{D}_{J,A}$: $\phi_{v,f}[I] := v[f(J)]$.*

□

Note that the assignment $\forall I \in \mathcal{D}_{J,A}$: $\phi_{v,f}[I] := v[f(J)]$ in this corollary (hence the statement I.1) can be implemented in many different ways, for instance, as follows:

(a) put $\phi_{v,f}[J] := v[f(J)]$, and then

(b) introduce a new statment $\forall I \in \mathcal{D}_{J,A}$, $I \neq J$: $\phi_{v,f}[I] := \phi_{v,f}[J]$;

Statement (b) is usually referred to as a *broadcast* statement since one value $\phi_{v,f}[J]$ has to be sent to many variables when $dim(Nul(A)) \geq 1$, where $dim(Nul(A))$ denotes the dimension of the null space of A. Such a broadcast statement defines a broadcast data dependence, which can be characterized by a set of dependence vectors $\{I-J|I \in \mathcal{D}_{J,A}\}$.

Some notation. Now we introduce some notation before we proceed. As we know, a defining variable $v[f]$ will address a specific array element $v[\xi]$ at iteration I if $f(I)=\xi$.

- We denote the set of array elements that are addressed by $v[f]$ for all $I \in \mathcal{I}$ by $Ins(v,f) = \{v[f(I)]|I \in \mathcal{I}\}$.
- We denote the subset of $\mathcal{I}$ in which a particular $v[\xi]$ is addressed by a defining variable $v[f]$ by $Def(v,f,\xi)$, called the *definition space* of $v[\xi]$ defined by $v[f]$, $Def(v,f,\xi) = \{I|v[f(I)] = v[\xi], I \in \mathcal{I}\}$. Obviously, if $f(I) = AI+b$, then $Def(v,f,f(I)) = \{I+I'|I' \in Nul(A)\} \cap \mathcal{I}$.
- Let $\mathcal{F}$ be a set of indexing functions. Then we define $Ins(v,\mathcal{F}) = \cup_{g \in \mathcal{F}} Ins(v,g)$, and $Def(v,\mathcal{F},\xi) = \cup_{g \in \mathcal{F}} Def(v,g,\xi)$. We will omit the name v from the sets $Ins(v,f)$, $Ins(v,\mathcal{F})$, $Def(v,f,\xi)$, and $Def(v,\mathcal{F},\xi)$ when it is clear from the context.
- Let $\mathcal{I}' \neq \emptyset$ be a subset of $\mathcal{I}$. Then we denote the $\prec$-ordered list of $\mathcal{I}'$ by $\rho(\mathcal{I}')$, the minimum of $\mathcal{I}'$ (i.e., the first element in $\rho(\mathcal{I}')$) by $min_{\prec}(\mathcal{I}')$, and the maximum of $\mathcal{I}'$ (i.e., the last element in $\rho(\mathcal{I}')$) by $max_{\prec}(\mathcal{I}')$.

- Let $I \in \mathcal{I}'$, then we denote the successor of I in $\rho(\mathcal{I}')$ by $next(I, \mathcal{I}')$. (Clearly, if $J = next(I, \mathcal{I}')$, then there is no $K \in \mathcal{I}'$ such that $I \prec K \prec J$.) In case $I = max_{\prec}(\mathcal{I}')$ or $I \notin \mathcal{I}'$, we define $next(I, \mathcal{I}') = \infty$, with $\infty \notin \mathcal{I}$, and $I \prec \infty \ \forall I \in \mathcal{I}$. If $J = next(I, \mathcal{I}')$, then we define $I = pred(J, \mathcal{I}')$.

Case II: $\mathcal{W}^v = \emptyset$ and $\mathcal{U}^v \neq \emptyset$.

Let $\mathcal{F}$ be a set of indexing functions, and let $\mathcal{U}^v$ be given by $\{v[f] | f \in \mathcal{F}\}$ in this case. Let $v[f]$ be a variable in $\mathcal{U}^v$. Then, as we know, the value of $v[f(I)]$ will only be updated, but not used in iteration I of NLP, for all $I \in \mathcal{I}$. Therefore, no value need to be assigned to $\phi_{v,f}[I]$ of NLP-1 at the beginning of each iteration.

However, since $v[f(I)]$ of NLP is possibly an output of the program, we have to know how to find the value of $v[f(I)]$, for all $I \in \mathcal{I}$, after the execution of NLP-1.

Proposition 3.9 *Let $v[f], v[g] \in \mathcal{U}^v = \{v[f] | f \in \mathcal{F}\}$. Then after the execution of NLP-1, the final value of $v[f(I)]$, $I \in \mathcal{I}$, will be given by $\phi_{v,g}[J]$ if $g(J) = f(I)$, with $J = max_{\prec}(Def(\mathcal{F}, f(I)))$.*

Proof: Let $I \in \mathcal{I}$ be an arbitrary iteration. As we know, in iteration I of NLP, $v[f]$ will address a particular element $v[f(I)]$, and $v[f(I)]$ will be updated in all the iterations in $Def(\mathcal{F}, f(I)) = \cup_{g \in \mathcal{F}} Def(g, f(I))$ during the program execution. The last updating of $v[f(I)]$ is obviously in the iteration $J = max_{\prec}(Def(\mathcal{F}, f(I)))$. In this iteration, $v[f(I)]$ will be represented by $\phi_{v,g}[J]$ if $g(J) = f(I)$ in the NLP-1.

□

Case III: $\mathcal{W}^v \neq \emptyset$ and $\mathcal{U}^v \neq \emptyset$.

In this case we write the sets $\mathcal{W}^v$ and $\mathcal{U}^v$ as $\mathcal{W}^v = \mathcal{X} \cup \mathcal{Z}$ and $\mathcal{U}^v = \mathcal{X} \cup \mathcal{Y}$ respectively, where $\mathcal{X} = \mathcal{W}^v \cap \mathcal{U}^v$, $\mathcal{Y} = \mathcal{U}^v \ominus \mathcal{W}^v$, and $\mathcal{Z} = \mathcal{W}^v \ominus \mathcal{U}^v$.

We first discuss the case $\mathcal{Y} = \emptyset$ and $\mathcal{X}$ contains only one variable $v[f]$, with $f(I) = AI + b$. Extension to the other cases will be give at the end of this section. We assume without loss of generality that $\mathcal{Z}$ is not empty (the case $\mathcal{Z} = \emptyset$ is covered implicitly in the discussion). Moreover, since $DG^{v[g]}$ and $DG^{v[g']}$ for $v[g], v[g'] \in \mathcal{Z}$ and $g \neq g'$ are disconnected, and they can be handled independently in a similar way,

we concentrate only on one variable $v[g]$ in $\mathcal{Z}$ below.

Recall that
(a) $(I, J, v[\xi])$ is an edge in $DG^{v[f]}$ if $I \rightarrow J$ and $f(I) = f(J) = \xi$.
(b) $(I, J, v[\xi])$ is an edge from node I of $DG^{v[f]}$ to node J of $DG^{v[g]}$ if $I \rightarrow J$ and $f(I) = g(J) = \xi$.
(c) A node I of $DG^{v[f]}$ represents an input of the program if I has no in-coming edge, that is, there is no $J \prec I$ such that $f(I) = f(J)$, and a node I of $DG^{v[g]}$ represents an input of the program if it has no in-coming edge from $DG^{v[f]}$, that is, there is no $I' \prec I$ such that $g(I) = f(I')$ ($DG^{v[g]}$ contains no internal edges, and out-degree of each node of $DG^{v[g]}$ is zero).

The outputs are defined similarly to those of case II, hence they are omitted below.

It turns out that the dependences (edges) in $DG^{v[f]}$, and between $DG^{v[f]}$ and $DG^{v[g]}$, hence also how the values of $\phi_{v,f}[I]$ and $\phi_{v,g}[\dot{I}]$ must be set, are largely determined by the dimension of the null space of A. Below we discuss the following three possible cases: $dim(Nul(A)) = 0$, $dim(Nul(A)) = 1$, and $dim(Nul(A)) > 1$.

A: $dim(Nul(A)) = 0$. If $dim(Nul(A)) = 0$, then $DG^{v[f]}$ contains no edges since $f(I) = f(J)$ will have no solution for $I \neq J$ in $\mathcal{I}$. Therefore, the value of $\phi_{v,f}[I]$ must be set to that of $v[f(I)]$ for all $I \in \mathcal{I}$.

For $DG^{v[g]}$, we have for each $I \in \mathcal{I}$:

- If $g(I) = f(I')$ has no solution for I' in $\mathcal{I}$, then node I of $DG^{v[g]}$ will have no in-coming edges, hence it represents an input of the program, and the value of $\phi_{v,g}[I]$ must be set to that of $v[g(I)]$.

- If $g(I) = f(I')$ has a solution for I' in $\mathcal{I}$ (notice that I' is then unique), then node I of $DG^{v[g]}$ represents an input of the program if $I \prec I'$; (I', I) will be an edge from node I' of $DG^{v[f]}$ to node I of $DG^{v[g]}$ if $I' \prec I$. Therefore, the value of $\phi_{v,g}[I]$ must be set to that of $v[g(I)]$ if $I \prec I'$, and to that of $\phi_{v,f}[I']$ if $I' \prec I$.

In summary, when $dim(Nul(A)) = 0$, then the value of $\phi_{v,f}[I]$ and the value of

$\phi_{v,g}[I]$, $I \in \mathcal{I}$, must be set as follows.

$$\forall I \in \mathcal{I}: \quad \phi_{v,f}[I] := v[f(I)]; \tag{III.a.1}$$

$$\forall I \in \mathcal{I}: \quad \phi_{v,g}[I] := \begin{cases} \phi_{v,f}[I'] & \text{if } g(I) = f(I'),\ I' \in \mathcal{I}, \text{ and } I' \prec I \\ v[g(I)] & \text{otherwise} \end{cases}; \tag{III.a.2}$$

Clearly, statement III.a.2 defines the following set of dependence vectors: $\{I - I' | I, I' \in \mathcal{I}, I' \prec I, f(I') = g(I)\}$.

B: $dim(Nul(A)) = 1$. In this case, the edges in $DG^{v[f]}$ are given by the following proposition.

Proposition 3.10 *Assume $dim(Nul(A)) = 1$. Let $c \succ 0$ (i.e, the first nonzero element of c is positive) be a coprime vector in $Nul(A)$. Then, for each node $I \in \mathcal{I}$, $(I, I+c)$ is the only edge in $DG^{v[f]}$ if $I + c \in \mathcal{I}$.*

Proof: Let (I, J) be an edge in $DG^{v[f]}$, that is $I \rightarrow J$. Then according to the definition of $I \rightarrow J$, I and J must satisfy the following conditions: (a) $f(I) = f(J)$, (b) $I \prec J$, and (c) there is no I': $I \prec I' \prec J$ such that $f(I') = f(I)$. From (a) it follows $J - I \in Nul(A)$ or $J - I = kc$, with $k \in \mathbf{Z}$, from (b) it follows that k must be positive, and from (c) it follows that k must be equal to 1.

□

In other words, when $dim(Nul(A)) = 1$, then $DG^{v[f]}$ is a regular dependence graph, and it contains only c as dependence vectors. Clearly, a node I in $DG^{v[f]}$ represents an input of the program if $I - c \notin \mathcal{I}$. Therefore, the value of $\phi_{v,f}[I]$ must be set in this case as follows.

$$\forall I \in \mathcal{I}: \quad \phi_{v,f}[I] := \begin{cases} \phi_{v,f}[I-c] & \text{if } I - c \in \mathcal{I} \\ v[f(I)] & \text{otherwise} \end{cases}; \tag{III.b.1}$$

Proposition 3.11 *Let $I', I \in \mathcal{I}$. Then (I', I) will be an edge from node I' of $DG^{v[f]}$ to node I of $DG^{v[g]}$ if $f(I') = g(I)$ and $I' \prec I \preceq I' + c$.*

Proof: This is a direct consequence of the fact that (I', I) is an edge from node I' of $DG^{v[f]}$ to node I of $DG^{v[g]}$ if $I' \rightarrow I$ and $f(I') = g(I)$.

□

Therefore, the value of $\phi_{v,g}[I]$ must be set as:

$$\forall I \in \mathcal{I}:$$
$$\phi_{v,g}[I] := \begin{cases} \phi_{v,f}[I'] & \text{if } g(I) = f(I'),\ I' \in \mathcal{I}, \text{ and} \\ & I' \prec I \preceq I' + c \\ v[g(I)] & \text{otherwise} \end{cases} ; \qquad \text{(III.b.2)}$$

The set of dependence vectors defined by this statement is given by $\{I - I' | I, I' \in \mathcal{I}, I' \prec I \preceq I' + c, f(I') = g(I)\}$.

Example 3.IV: Before we continue, we illustrate the above discussion using the program NLP of Example 3.III. The program is given by $< \mathcal{I}, \prec, F, \mathcal{W}, \mathcal{U} >$, where

$$\mathcal{I} = \{[i,k]^t | 0 \leq i \leq N, 0 \leq k \leq N\},$$
$$\mathcal{U} = \{y[f_y]\},$$
$$\mathcal{W} = \{x[f_x], y[f_y], w[f_w]\},$$

with

$$f_x(I) = A_x I = [1, -1][i,k]^t,$$
$$f_y(I) = A_y I = [1, 0][i,k]^t, \text{ and}$$
$$f_w(I) = A_w I = [0, 1][i,k]^t.$$

Moreover, we have $\mathcal{U} = \mathcal{U}^x \cup \mathcal{U}^y \cup \mathcal{U}^w$, and $\mathcal{W} = \mathcal{W}^x \cup \mathcal{W}^y \cup \mathcal{W}^w$, with

$$\mathcal{W}^x = \{x[f_x]\}, \mathcal{U}^x = \emptyset,$$
$$\mathcal{W}^w = \{w[f_w]\}, \mathcal{U}^w = \emptyset, \text{ and}$$
$$\mathcal{W}^y = \{y[f_y]\}, \mathcal{U}^y = \{y[f_y]\}, \text{ or equivalently, } \mathcal{X} = \{y[f_y]\}, \mathcal{Z} = \mathcal{Y} = \emptyset.$$

Let $\phi_x[I]$, $\phi_y[I]$, and $\phi_y[I]$ be three new variables. Then we obtain a new program NLP-1, given by, $< \mathcal{I}, \prec, F, \mathcal{W}', \mathcal{U}' >$, after $x[f_x]$, $y[f_y]$, and $w[f_w]$ are substituted by $\phi_x[I]$, $\phi_y[I]$, and $\phi_y[I]$, respectively. This new program NLP-1 becomes input-output equivalent to the original program NLP after the following statements are inserted at the beginning of the loopbody.

(a) Since $\mathcal{W}^x = \{x[f_x]\}, \mathcal{U}^x = \emptyset$, according to the discussion of case I, we insert the statement

$$\forall I \in \mathcal{I}: \quad \phi_x[I] := x[f_x(I)];$$

Similarly for $\phi_w[I]$, for which we insert the statement

$\forall I \in \mathcal{I}$: $\phi_w[I] := w[f_w(I)]$;

(b) Since $\mathcal{X} = \{y[f_y]\}$, $\mathcal{Z} = \mathcal{Y} = \emptyset$, and $f_y(I) = A_y I = [1,0][i,k]^t$ with $dim(Nul(A_y)) = 1$, according to the discussion of case III, we insert the statement

$$\forall I \in \mathcal{I}: \quad \phi_y[I] := \begin{cases} \phi_y[I-c] & \text{if } I-c \in \mathcal{I} \\ y[f_y(I)] & \text{otherwise} \end{cases} ; \text{ where } c = [0,1]^t \in Nul(A_y).$$

As result, we have transformed the given program into the following single-assignment program:

$\forall I \in \mathcal{I}$:

$$\begin{aligned} &\phi_x[I] := x[f_x(I)]; \\ &\phi_w[I] := w[f_w(I)]; \\ &\phi_y[I] := \begin{cases} \phi_y[I-c] & \text{if } I-c \in \mathcal{I} \\ y[f_y(I)] & \text{otherwise} \end{cases} ; \\ &\mathcal{U}'(I) := F(\mathcal{W}'(I)); \end{aligned}$$

□

Note that we can also express the values of $\phi_{v,f}[I]$ and $\phi_{v,g}[I]$ in terms of $\mathcal{D} = Def(v, f, f(I))$, and $next(I, \mathcal{D})$. From the definitions of $Def(v, f, f(I))$ and $next(I, \mathcal{D})$, it follows that (I', I), $I', I \in \mathcal{I}$, will be an edge of $DG^{v[f]}$ if $I' \prec I$ and $I = next(I', \mathcal{D})$ (or equivalently, $I' = pred(I, \mathcal{D})$); and (I', I) will be an edge from node I' of $DG^{v[f]}$ to node I of $DG^{v[g]}$ if $g(I) = f(I')$ and $I' \prec I \preceq next(I', \mathcal{D})$. Therefore, we will have the following general expressions for the values of $\phi_{v,f}[I]$ and $\phi_{v,g}[I]$:

$\forall I \in \mathcal{I}$:

$$\phi_{v,f}[I] := \begin{cases} \phi_{v,f}[I'] & \text{if } I' = pred(I, \mathcal{D}), \text{ and } I' \in \mathcal{I}, \\ & \text{where } \mathcal{D} = Def(f, f(I)) \\ v[f(I)] & \text{otherwise} \end{cases} ; \qquad \text{(III.1)}$$

$\forall I \in \mathcal{I}$:

$$\phi_{v,g}[I] := \begin{cases} \phi_{v,f}[I'] & \text{if } g(I) = f(I'),\ I' \in \mathcal{I}, \text{ and} \\ & I' \prec I \preceq next(I', \mathcal{D}), \\ & \text{where } \mathcal{D} = Def(f, f(I')) \\ v[g(I)] & \text{otherwise} \end{cases} ; \qquad \text{(III.2)}$$

When $dim(Nul(A))=0$, then $pred(I,\mathcal{D})$ in III.1 will not exist, and $next(I',\mathcal{D})$ in III.2 will be equal to ∞, hence statements III.a.1 and III.a.2 will follow. When $dim(Nul(A))=1$, then I' in III.1 will be equal to $I-c$, if $I-c\in\mathcal{I}$, with c defined in proposition 3.10, and $next(I,\mathcal{D})$ in III.2 will be equal to $I+c$, if $I+c\in\mathcal{I}$, from which statements III.b.1 and statement III.b.2 will follow.
The main advantage of using these general expressions is that it gives a quite clear statement of the problem: to compute $pred(I)$ in III.1 and $next(I)$ in III.2.

Below we discuss the computation of $next(I,\mathcal{D})$ for the case $dim(Nul(A))=k>1$, where $f(I)=AI+b$, and $\mathcal{D}=Def(f,f(I))$. The computation of $pred(I,\mathcal{D})$ is similar, therefore omitted.

C: $dim(Nul(A))=k>1$.

Let $Def(f,f(I))=\{I+I'|I'\in Nul(A)\}\cap\mathcal{I}$. Let c_i, $i=1,...,k$, be k linearly independent vectors in $Nul(A)$, then each $I'\in Nul(A)$ can be written as $I'=\sum_{i=1}^{k}\alpha_i c_i$, with $\alpha_i\in\mathbf{R}$ for $i=1,...,k$. The problem of finding $next(I,\mathcal{D})$ can be formulated as: find an $I'=\sum_{i=1}^{k}\alpha_i c_i\in\mathcal{I}$ such that there is no other $I''\in\mathcal{D}$ which satisfies $I\prec I''\prec I+I'$.

Proposition 3.12 *Let $\mathcal{D}=Def(f,f(I))$, with $f(I)=AI+b$. If $dim(Nul(A))>1$, then there is a constant coprime vector $c\in Nul(A)$, with $c\succ 0$, such that $next(I,\mathcal{D})=I+c$ so long as $I+c\in\mathcal{I}$.*

Proof: Let $\mathcal{T}=\{I\mid I\in Nul(A), I\neq 0\}$. Since the vectors in $\mathcal{T}$ are nonzero n-vectors, we can partition $\mathcal{T}$ into at most n subsets $\mathcal{T}_i$, $i=0,\ldots,n-1$, with a vector in $\mathcal{T}_i$ if and only if it has i leading zero components.

Let $\mathcal{T}_j$, $0\leq j\leq n-1$, be the nonempty set which contains vectors with the most leading zeros, that is, each vector x in this set is of the form

$$x=[0,...,0,x_{j+1},*,...,*]^t,$$

with $x_{j+1}\neq 0$. Let $c\succ 0$ be the smallest coprime vector in $\mathcal{T}_j$ in terms of $\prec$. Then there will be no other $c'\succ 0$ in $\mathcal{T}_j$ such that $I+c'\prec I+c$. Hence we have $next(I,\mathcal{D})=I+c$, if $I+c\in\mathcal{I}$.

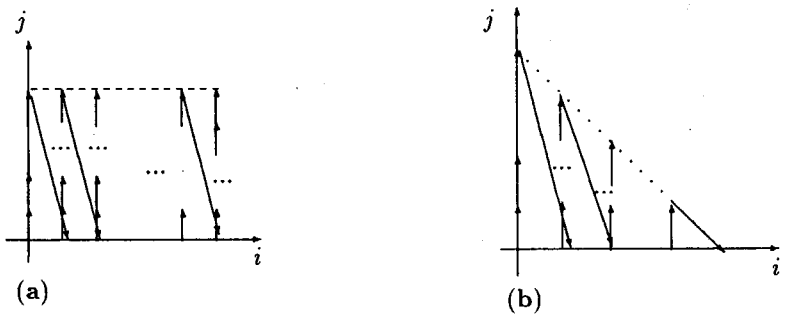


Figure 3.5: Dependence of $next(I, \mathcal{D})$ on the topology of the iteration space $\mathcal{I}$, $I = [i, j]^t$

□

Now let d be a constant vector in $\mathbf{Z}^n$. We define the set of *boundary* points with respect to (w.r.t) d as $Bp(d, \mathcal{I}) = \{I | I \in \mathcal{I}, I + d \notin \mathcal{I}\}$.

Then proposition 3.12 says that $next(I, \mathcal{D})$ will be a constant vector if I is not a boundary point in $Bp(c, \mathcal{I})$. However, when $I \in Bp(c, \mathcal{I})$, then $next(I)$ will generally depend on the boundary of the iteration space $\mathcal{I}$, and we have to obtain more information about $\mathcal{I}$ to find an explicit expression for $next(I)$, if possible. See Fig. 3.5 for two examples.

Extensions

• ($\mathcal{Y} = \emptyset$, and $|\mathcal{X}| > 1$). Up to now we have assumed that $\mathcal{Y} = \emptyset$ and $\mathcal{X}$ contains only one variable $v[f]$.

When $\mathcal{X}$ contains more variables, let $\mathcal{F}$ be a set of indexing functions, and assume $\mathcal{X} = \{v[f] | f \in \mathcal{F}\}$. Then, for each $v[f] \in \mathcal{X}$, and for each node I of $DG^{v[f]}$, there are the following possibilities:

(i) I has no in-coming edges, that is, I represents an input;

(ii) I has an in-coming edge from node I' of the $DG^{v[h]}$, $h \in \mathcal{F}$, and it can be either $h = f$ or $h \neq f$. It is easy to show that in this case the node I' is given by

$I' = pred(I, \mathcal{D})$, where $\mathcal{D} = Def(\mathcal{F}, f(I))$.

In this case we have therefore to change the statement III.1 into

$$\forall I \in \mathcal{I}: \quad \phi_{v,f}[I] := \begin{cases} \phi_{v,h}[I'] & \text{if } h(I') = f(I),\ I' \in \mathcal{I}, \text{ and} \\ & I' = pred(I, \mathcal{D}), \text{ where} \\ & h \in \mathcal{F}, \text{ and } \mathcal{D} = Def(\mathcal{F}, f(I)) \\ v[f(I)] & \text{otherwise} \end{cases}; \qquad \text{(III.3)}$$

and we have to do this for each $\phi_{v,f}[I]$ corresponding to $v[f] \in \mathcal{X}$.

Similarly, since from each node I' of $DG^{v[h]}$, $v[h] \in \mathcal{X}$, there is possibly an edge to node I of $DG^{v[g]}$, we have to change the statement III.2 into

$$\forall I \in \mathcal{I}: \quad \phi_{v,g}[I] := \begin{cases} \phi_{v,h}[I'] & \text{if } g(I) = h(I'),\ I' \in \mathcal{I}, \text{ and} \\ & I' \prec I \preceq next(I', \mathcal{D}), \text{ where} \\ & h \in \mathcal{F}, \text{ and } \mathcal{D} = Def(\mathcal{F}, h(I')) \\ v[g(I)] & \text{otherwise} \end{cases}; \qquad \text{(III.4)}$$

Let $\mathcal{D} = Def(\mathcal{F}, f(I))$, then the $next(I, \mathcal{D})$ is given by

$$next(I, \mathcal{D}) = min_{\prec}\{next(I, \mathcal{D}') | \mathcal{D}' = Def(g, f(I)), g \in \mathcal{F}\}.$$

The set of dependence vectors defined by the statement III.3 is given by

$$\{I - I' | I, I' \in \mathcal{I}, I' = pred(I, \mathcal{D}), f(I) = h(I'), h \in \mathcal{F}, \mathcal{D} = Def(\mathcal{F}, f(I))\}.$$

Since if $I' = pred(I, \mathcal{D}) \in \mathcal{D}$, then there is at least one $h \in \mathcal{F}$ such that $f(I) = h(I')$, therefore, we can also write this set of dependence vectors as $\{I - I' | I, I' \in \mathcal{I}, I' = pred(I, \mathcal{D})\}$, with $\mathcal{D} = Def(\mathcal{F}, f(I))\}$.

The set of dependence vectors defined by statement III.4 is given by

$$\{I - I' | I, I' \in \mathcal{I}, I' \prec I \preceq next(I, \mathcal{D}), g(I) = h(I'), \mathcal{D} = Def(\mathcal{F}, h(I'))\},$$

with $h \in \mathcal{F}$, and g the indexing function of a variable in $\mathcal{Z}$.

- ($\mathcal{Y} \neq \emptyset$, and $\mathcal{X} = \emptyset$). The discussion of this case is much like the case $\mathcal{Y} = \emptyset$ and $\mathcal{X} \neq \emptyset$. Since none of the nodes of $DG^{v[f]}$, $v[f] \in \mathcal{Y}$ will have an in-coming edge,

the statement III.3 will be ignored in this case. The statement III.4 will remain unchanged, however, under the assumption that $\mathcal{F}$ is the set of indexing functions of variables in $\mathcal{Y}$.

- ($\mathcal{Y} \neq \emptyset$, and $\mathcal{X} \neq \emptyset$). The discussion of this case is, in fact, a combination of the case $\mathcal{Y} = \emptyset$, and $\mathcal{X} \neq \emptyset$, and the case $\mathcal{Y} \neq \emptyset$, and $\mathcal{X} = \emptyset$. We omit the details here.

For most programs we have examined, the set $\mathcal{Y}$ is usually empty, and the cardinality of the set $\mathcal{X}$ is usually small, typically equal to 1 or 2; it is in our opinion that it is unlikely that the set $\mathcal{X}$ will contain many variables. When the set $\mathcal{Y}$ is not empty, the variables in this set are frequently used as *auxiliary* variables, and they are not essential for the dependence analysis of the program.

To simplify the discussion, we will handle from now on variables in $\mathcal{Y}$ in the same way as the variables in $\mathcal{X}$ by introducing dummy edges (dependences) in their variable dependence graphs.

A conclusion

A nested loop program $<\mathcal{I}, \prec, F, \mathcal{W}, \mathcal{U}>$ can be transformed into single-assignment by

- replacing each variable $v[f]$ in $\mathcal{W}$ and $\mathcal{U}$ by a new variable $\phi_{v,f}[I]$;
- introducing a set of new statements of type I.1, III.3, and III.4.

Dependence vectors in these statements can be expressed explicitly in the form of algebraic expressions.

3.6 Converting nested loop programs into RAs

In this section we discuss when, and how, an NLP can be converted into an RA for systolic array implementation. We start out from the single-assignment form derived from the NLP described in the previous section. First, we give a necessary condition for an NLP to be *systolic*. (An NLP which can be mapped to a systolic

array is usually referred to as a systolic NLP.) Then we show how to *localize* the dependence vectors, that is, how to make each iteration I depending only on its "neighboring" iterations, i.e., iterations in a set $\{I-d|d\in\mathcal{P}\}$, with the cardinality of $\mathcal{P}$ independent of the size of the iteration space.

3.6.1 Systolic NLPs

We assume that the iteration space $\mathcal{I}$ of the NLP satisfies the following condition in addition to that $\mathcal{I}$ is convex:

$$max_{I\in\mathcal{I}}\beta^t I - min_{J\in\mathcal{I}}\beta^t J = O(N),$$

where $\beta\in\mathbf{Z}^n$ is an arbitrary constant vector, and N is the maximum extent of the iteration space $\mathcal{I}$. In other words, we assume that the maximum "diameter" of $\mathcal{I}$ is $O(N)$.

It is shown in [Rao85], [RK86] that every RIA with a strongly separating hyperplane can be mapped onto systolic arrays, and conversely, every systolic array executes such an RIA. Moreover, it is shown in [RK88b], [RK88c] that a necessary condition for an algorithm to be systolic is that its input-output latency (execution time) is (no larger than) $O(N)$, where N is the maximum extent of the algorithm's index space. Therefore, a necessary condition for an NLP to be systolic is that its maximum I/O latency is $O(N)$, with N the maximum extent of the program's iteration space.

Proposition 3.13 *Let $\mathcal{W}=\cup_v\mathcal{W}^v$, and $\mathcal{U}=\cup_v\mathcal{U}^v$. Assume that the set $\mathcal{W}^v\cap\mathcal{U}^v=\{v[f]|f\in\mathcal{F}\}$ is not empty for a variable name v. Let $f\in\mathcal{F}$, $f(I)=AI+b$. Then, the I/O latency of the NLP is at least $O(N^k)$ if $dim(Nul(A))=k$.*

Proof: Let I be an arbitrary point in $\mathcal{I}$. Since $v[f]\in\mathcal{W}^v\cap\mathcal{U}^v$, the variable $v[f(I)]$ will be used and updated in all the iterations in $Def(\mathcal{F},f(I))$. Therefore, the iterations in $Def(\mathcal{F},f(I))$ have to be evaluated serially according to the ordering $\prec$ defined by the program's loops. Since $Def(\mathcal{F},f(I)) = Def(f,f(I))\cup[\cup_{\substack{g\in\mathcal{F}\\ g\neq f}}Def(g,f(I))]$, the I/O latency of the NLP is at least $|Def(f,f(I))|$, with

$Def(f, f(I)) = \{I + I' | I' \in Nul(A)\} \cap \mathcal{I}$ by definition. It follows then from the assumption that the diameter of $\mathcal{I}$ is $O(N)$ that $|Def(f, f(I))| = O(N^k)$.

□

Theorem 3.1 *An NLP is non-systolic if there is a variable $v[f]$ in $\mathcal{W} \cap \mathcal{U}$, with $f(I) = AI + b$, such that $dim(Nul(A)) = k > 1$.*

Proof: This is a direct consequence of proposition 3.13.

□

From now on, we assume that the indexing function $f(I) = AI + b$ of each variable $v[f]$ in $\mathcal{W} \cap \mathcal{U}$ satisfies the condition that $dim(Nul(A)) \leq 1$.

3.6.2 Localization of dependence vectors

Let the NLP be transformed into a single-assignment program NLP-1 as described in the previous section. As we know, NLP-1 can be characterized by $< \mathcal{I}, \prec, F, \mathcal{W}', \mathcal{U}' >$, where $\mathcal{W}'$ and $\mathcal{U}'$ are obtained after the substitution of each variable $v[f]$ in $\mathcal{W}$ and $\mathcal{U}$ by a new variable $\phi_{v,f}[I]$. Moreover, in addition to F, NLP-1 has three types of new statements. They are:

(s1) $\forall I \in \mathcal{I}$: $\phi_{v,f}[I] := v[f(I)]$;

For each pair $< \mathcal{W}^v, \mathcal{U}^v >$ of NLP, if $\mathcal{U}^v = \emptyset$, then each variable $v[f] \in \mathcal{W}^v$ corresponds to such a statement in NLP-1. This statement has a property that the same value will be assigned to $\phi_{v,f}[I]$ and $\phi_{v,f}[J]$, $I, J \in \mathcal{I}$ and $I \neq J$, if $I - J \in Nul(A)$, where A is the indexing matrix of f. (See proposition 3.8.)

(s2) $\forall I \in \mathcal{I}$: $\phi_{v,f}[I] := \begin{cases} \phi_{v,h}[I'] & \text{if } h(I') = f(I),\ I' \in \mathcal{I}, \text{ and } I' = pred(I, \mathcal{D}), \\ & \text{where } h \in \mathcal{F}, \text{ and } \mathcal{D} = Def(\mathcal{F}, f(I)) \\ v[f(I)] & \text{otherwise} \end{cases}$;

For each pair $< \mathcal{W}^v, \mathcal{U}^v >$ of NLP, if $\mathcal{W}^v \neq \emptyset$, $\mathcal{U}^v \neq \emptyset$, then each variable $v[f]$ in $\mathcal{W}^v \cap \mathcal{U}^v = \{v[f] | f \in \mathcal{F}\} \neq \emptyset$ corresponds to such a statement in NLP-1.

Such a statement defines a set of dependence vectors $\{I - I' | I, I' \in \mathcal{I}, I' = pred(I, \mathcal{D})\}$, where $\mathcal{D} = Def(\mathcal{F}, f(I))\}$.

Moreover, each variable $v[g] \in \mathcal{W}^v \ominus \mathcal{U}^v$ corresponds to a statement as follows in NLP-1:

$$\text{(s3)}\ \forall I \in \mathcal{I}: \quad \phi_{v,g}[I] := \begin{cases} \phi_{v,h}[I'] & \text{if } g(I)=h(I'),\ I' \in \mathcal{I}, \text{ and } I' \prec I \preceq next(I', \mathcal{D}), \\ & \text{where } h \in \mathcal{F}, \text{ and } \mathcal{D} = Def(\mathcal{F}, h(I')) \\ v[g(I)] & \text{otherwise} \end{cases} ;$$

Such a statement defines a set of dependence vectors $\{I - I' | I, I' \in \mathcal{I}, I' \prec I \preceq next(I', \mathcal{D}), g(I) = h(I')\}$, where $h \in \mathcal{F}$, and $\mathcal{D} = Def(\mathcal{F}, h(I'))\}$.

The objective of localization is to transform each statement of the above three types in NLP-1 into a (set of) functionally equivalent *propagation* statement(s) of the following form:

$$\phi_{v,f}[I] := \phi_{v,f}[I - c],$$

using a fixed number (independence of size of $\mathcal{I}$) of *propagation vectors* c. (Additional variables may be introduced to assist the localization.) The propagation statements must satisfy the following conditions:

Forward propagation. All the propagation vectors c must satisfy $c \succ 0$.

Regular data dependences. The propagation vector c in statement $\phi_{v,f}[I] := \phi_{v,f}[I - c]$ must be a constant vector when I is not a boundary point of the program's iteration space $\mathcal{I}$.

Memory conflict free. Different values may not be assigned (propagated) to the same variable during data propagation.

Finite number of additional variables. When a variable (value) is propagated to an iteration in which the variable is neither used nor updated, then an additional variable (memory location) has to be introduced in that iteration to temporarily store the propagated value. This condition says that the total number of such additional variables introduced in one iteration must be independent of the size of $\mathcal{I}$.

Converting statement s1 into propagations

Proposition 3.14 *Let $f(I) = AI + b$, with $dim(Nul(A)) \geq 1$. Then a statement of the form $\forall I \in \mathcal{I}$: $\phi_{v,f}[I] := v[f(I)]$ can always be replaced by a functionally equivalent propagation statement.*

Proof: Let $c \succ 0$ be a coprime vector in $Nul(A)$. Then the statment $\forall I \in \mathcal{I}$: $\phi_{v,f}[I] := v[f(I)]$ is functionally equivalent to the following propagation statement:

$$\phi_{v,f}[I] := \begin{cases} v[f(I)] & \text{if } I \in Bp(-c,\mathcal{I}) \\ \phi_{v,f}[I-c] & \text{otherwise} \end{cases} ; \qquad \text{(P.1)}$$

where $Bp(-c,\mathcal{I})$ is the set of boundary points of $\mathcal{I}$ defined with respect to $-c$, since in both statements the value of $v[f(I)]$ will be assigned to $\phi_{v,f}[J]$ for all $J \in Def(f, f(I))$.

□

Note that when $dim(Nul(A)) = 0$, then no two $\phi_{v,f}[I]$, $\phi_{v,f}[J]$, $I, J \in \mathcal{I}$, will address to the same $v[f(I)]$. Therefore, propagation through a boundary of $\mathcal{I}$ is in this case not possible. When $dim(Nul(A)) > 1$, however, we can use more than one propagation vector to propagate the value of $v[f(I)]$ to $\phi_{v,f}[I]$. In particular, when $v[f(I)]$ is a scalar variable, then $Nul(A) = \mathbf{Z}^n$, and we will have a large number of choices for the propagation vector c. Such freedom can be exploited to optimize other aspects of a design, for instance, to optimize the parallel scheduling that can be found for the execution of the algorithm.

In general, it is not advantageous to use more propagation vectors than necessary. However, it is sometimes possible to use more propagation vectors to reduce the number of initialization points. See Fig. 3.6 for a 2-dimensional illustration.

Remark 3.4 In proposition 3.14, only vectors in the null space of indexing matrix are used as propagation vectors. Such a propagation is (therefore) usually called a null space propagation. The idea of null space propagations is also used in [Raj86] (see also [RPF86], [RF87]) for the conversion of AREs (affine recurrent equations) into RIAs (RAs).

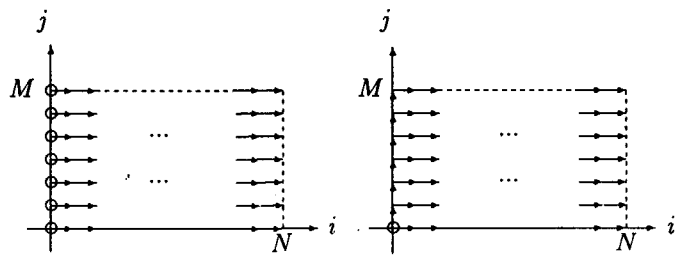


Figure 3.6: Propagation of the value of x to all the $\phi_{v,f}[i,j]$, where $0 \leq i \leq N$ and $0 \leq j \leq M$: (**a**) Using one propagation vector $[1,0]^t$ with initialization $\phi_{v,f}[0,j] = x$ for all $0 \leq j \leq M$. (**b**) Using two propagation vectors $[1,0]^t$ and $[0,1]^t$ with initialization $\phi_{v,f}[0,0] = x$

Converting statements s2 and s3 into propagations

Assume that $\mathcal{X} = \mathcal{W}^v \cap \mathcal{U}^v = \{v[f] | f \in \mathcal{F}\} \neq \emptyset$, then each variable $v[f]$ in $\mathcal{X}$ corresponds to a statement s2 in NLP-1, which defines a set of dependence vectors

$$\mathcal{V}_{s2} = \{I - I' | I, I' \in \mathcal{I}, I' = pred(I, \mathcal{D})\},$$

with $\mathcal{D} = Def(\mathcal{F}, f(I))\} = \cup_{h \in \mathcal{F}} Def(h, f(I))$. Similarly, each variable $v[g] \in \mathcal{W}^v \ominus \mathcal{U}^v$ corresponds to a statement s3, which defines a set of dependence vectors

$$\mathcal{V}_{s3} = \{I - I' | I, I' \in \mathcal{I}, I' \prec I \preceq next(I', \mathcal{D}), g(I) = h(I')\},$$

with $h \in \mathcal{F}$, and $\mathcal{D} = Def(\mathcal{F}, h(I'))\}$.

In order to convert statement s2 into a functionally equivalent propagation statement(s), we have to decompose each dependence vector $d \in \mathcal{V}_{s2}$ into a positive linear combination of a set of propagation vectors. That is, to find a set $\{c_1, ..., c_k\}$ of propagation vectors, with k an integer constant, such that for all $d \in \mathcal{V}_{s2}$, $d = \sum_{i=1}^{k} a_i c_i$, with a_i positive integers. The same applies to statement s3. However, such a decomposition will depend largely on the topology of the iteration space $\mathcal{I}$ and the indexing functions involved.

Note that when $\mathcal{F}$ contains only one indexing function f, with $f(I) = AI + b$ and $dim(Nul(A)) = 1$, then we will have in statement s2 $I' = pred(I, \mathcal{D}) = I - c$, where $c \succ 0$ is the coprime vector in $Nul(A)$. Hence this statement is a propagation statement already. When $dim(Nul(A)) = 0$, then this statement becomes an s1 statement, however, it cannot be converted into propagations.

In our opinion, it is unlikely that the $\mathcal{F}$ will contain many indexing functions for a realistic program. When $\mathcal{F}$ contains in addition to f another indexing function h, then s2 will become the following propagation statment if $max_{\prec}(Def(h, f(I)) \preceq min_{\prec}(Def(f, f(I))$. Note that $I = min_{\prec}(Def(f, f(I))$ if $I - c \notin \mathcal{I}$.

$$\phi_{v,f}[I] := \begin{cases} \phi_{v,f}[I-c] & \text{if } I-c \in \mathcal{I} \\ \phi_{v,h}[I-c'] & \text{otherwise} \end{cases};$$

where $c' = min_{\prec}(Def(f, f(I)) - max_{\prec}(Def(h, f(I))$. Similarly for when $\mathcal{F}$ contains more indexing functions.

In statement s3, let $g(I) = BI + e$, and let $c \in Nul(B)$. Then the value of $\phi_{v,h}[I']$ will be assigned to both $\phi_{v,g}[I]$ and $\phi_{v,g}[I+c]$ if the iteration $I+c$ satisfies the condition $I' \prec I + c \preceq next(I', \mathcal{D})$. Therefore, the null space propagation (see proposition 3.14) may also be used in this case.

3.7 Examples

In this section we give some examples to illustrate the conversion of nested loop programs into single-assignment programs and (then) into RAs.

3.7.1 Convolution

Nested loop program A nested loop program for the computation of the convolution sum $y_i = \sum_{k=0}^{N} w_k x_{i-k}$, $0 \leq i \leq N$ can be written as follows.

```
for i = 0 to N
    for k = 0 to N
        y[i] := y[i] + w[k] · x[i − k];
```

In our notation, we write this program as $< \mathcal{I}, \prec, F, \mathcal{W}, \mathcal{U} >$, where

$$\mathcal{I} = \{[i,k]^t | 0 \le i \le N, 0 \le k \le N\};$$
$$F(a,b,c) = a + b \cdot c;$$
$$\mathcal{U} = \{y[f_y]\}; \text{ and}$$
$$\mathcal{W} = \{y[f_y], w[f_w], x[f_x]\};$$

with $f_x(I) = A_x I = [1,-1][i,k]^t$, $f_y(I) = A_y I = [1,0][i,k]^t$, and $f_w(I) = A_w I = [0,1][i,k]^t$.

Single assignment program In Example 3.IV we have shown that the above program can be transformed into the following single-assignment program.

$$\begin{array}{l}
\forall I \in \mathcal{I}: \\
\quad \phi_x[I] := x[f_x(I)]; \\
\quad \phi_w[I] := w[f_w(I)]; \\
\quad \phi_y[I] := \left\{ \begin{array}{ll} \phi_y[I-c] & \text{if } I-c \in \mathcal{I} \\ y[f_y(I)] & \text{otherwise} \end{array} \right. ; \text{ where } c = [0,1]^t \\
\quad \mathcal{U}'(I) := F(\mathcal{W}'(I));
\end{array}$$

Regular algorithm (RA) In the above single-assignment program, the statement

$$\forall I \in \mathcal{I}: \ \phi_x[I] := x[f_x(I)];$$

can be transformed into the following propagation statement by propagating the value of $x[f_x(I)]$ in the null space of A_x:

$$\forall I \in \mathcal{I}: \ \phi_x[I] := \left\{ \begin{array}{ll} \phi_x[I-c_x] & \text{if } I-c_x \in \mathcal{I} \\ x[f_x(I)] & \text{otherwise} \end{array} \right. ;$$

where $c_x = [1,1]^t \in Nul(A_x)$. Equivalently, this statement can be written as

$$\forall I \in \mathcal{I}: \ \phi_x[I] := \left\{ \begin{array}{ll} \phi_x[I-c_x] & \text{if } i \ge 1 \text{ and } k \ge 1 \\ x[f_x(I)] & \text{otherwise} \end{array} \right.$$

Similarly, the statement $\forall I \in \mathcal{I}: \ \phi_w[I] := w[f_w(I)]$ can be transformed into the following propagation statement by propagating the value of $w[f_w(I)]$ in the null space of A_w:

$$\forall I \in \mathcal{I}: \ \phi_w[I] := \left\{ \begin{array}{ll} \phi_w[I-c_w] & \text{if } i \ge 1 \text{ and } k \ge 0 \\ w[f_w(I)] & \text{otherwise} \end{array} \right. ;$$

where $c_w = [1,0]^t \in Nul(A_w)$.

As result, we will obtain the following RA for the convolution program:

$$
\begin{array}{l}
\forall I \in \mathcal{I}: \\
\quad \phi_x[I] := \begin{cases} \phi_x[I-c_x] & \text{if } i \geq 1 \text{ and } k \geq 1 \\ x[f_x(I)] & \text{otherwise} \end{cases}; \\
\quad \phi_w[I] := \begin{cases} \phi_w[I-c_w] & \text{if } i \geq 1 \text{ and } k \geq 0 \\ w[f_w(I)] & \text{otherwise} \end{cases}; \\
\quad \phi_y[I] := \begin{cases} \phi_y[I-c] & \text{if } k \geq 1 \text{ and } i \geq 0 \\ y[f_y(I)] & \text{otherwise} \end{cases}; \\
\quad \phi_y[I] := F(\phi_y[I], \phi_w[I], \phi_x[I]);
\end{array}
$$

with $c_x \in [1,1]^t$, $c_w = [1,0]^t$, and $c = [0,1]^t$.
A systolic array implementation of this algorithm is described in [Qui84].

Remark 3.5 It can easily be shown that the following NLP for the multiplication of two matrices $[a_{ij}]$ and $[b_{ij}]$ (i.e., $c_{ij} = \sum_{k=1}^{N} a_{ik} b_{kj}$, $1 \leq i \leq N$, $1 \leq j \leq N$),

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            c[i,j] := c[i,j] + a[i,k] · b[k,j];
```

has the same "structure" as the above NLP for the convolution (except that its iteration space is of dimension three instead of two). That is, the sets $\mathcal{U}$ and $\mathcal{W}$ defined for the variables $c[i,j]$, $a[i,k]$ and $b[k,j]$ are similar to that of $y[i]$, $w[k]$ and $x[i-k]$ respectively:

$$
\begin{array}{l}
\mathcal{U}^c = \mathcal{W}^c = \{c[i,j]\}, \text{ or } \mathcal{X}^c = \{c[i,j]\}, \mathcal{Y}^c = \emptyset \text{ and } \mathcal{Z}^c = \emptyset; \\
\mathcal{U}^a = \emptyset, \mathcal{W}^a = \{a[i,k]\}; \text{ and} \\
\mathcal{U}^b = \emptyset, \mathcal{W}^b = \{b[k,j]\}.
\end{array}
$$

Moreover, the variables $c[i,j]$, $a[i,k]$ and $b[k,j]$ can be written as $c[A_c I]$, $a[A_a I]$ and $b[A_b I]$ respectively, where

$$
A_c = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \quad A_a = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \text{and} \quad A_b = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix},
$$

then the null spaces of all the three indexing matrices A_c, A_a and A_b are of dimension one.

The transformation of this program into an RA is fairly similar to that of the above convolution program, details are omitted.

3.7.2 Matrix LDU factorization

Nested loop program An NLP for the computation of the LDU factorization of a matrix $[a_{ij}]$ can be written as follows:

```
for k = 1 to N − 1
    for j = k + 1 to N
        for i = k + 1 to N
            a[i, j] := a[i, j] − a[i, k] · a[k, j]/a[k, k];
```

The iteration space $\mathcal{I}$ of this program is given by

$$\mathcal{I} = \{I=[k,j,i]^t | 1 \leq k \leq N-1, k+1 \leq j \leq N, k+1 \leq i \leq N\}.$$

The variables in the loopbody are:

1. $a[i,j]$, or $a[f_1(I)]$, $f_1(I) = A_1 I$;
2. $a[i,k]$, or $a[f_2(I)]$, $f_2(I) = A_2 I$;
3. $a[k,j]$, or $a[f_3(I)]$, $f_3(I) = A_3 I$; and
4. $a[k,k]$, or $a[f_4(I)]$, $f_4(I) = A_4 I$.

where

$$A_1 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}, A_2 = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}, A_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}, \text{ and } A_4 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

The null space of the indexing matrices A_1, A_2 and A_3 are all of dimension one, and propagation vectors in $Nul(A_1)$, $Nul(A_2)$ and $Nul(A_3)$ are

1. $c_1 = [1,0,0]^t$,

2. $c_2 = [0,1,0]^t$ and

3. $c_3 = [0,0,1]^t$ respectively.

   The null space of A_4 is of dimension two, and the propagation vectors in $Nul(A_4)$ are

4. $c_{4,1} = [0,1,0]^t$ and $c_{4,2} = [0,0,1]^t$.

Moreover, we have for this program

$$\begin{aligned}&\mathcal{U} = \{a[f_1]\},\\ &\mathcal{W} = \{a[f_1], a[f_2], a[f_3], a[f_4]\}, \text{ and}\\ &\mathcal{X} = \{a[f_1]\},\ \mathcal{Y} = \emptyset, \text{ and } \mathcal{Z} = \{a[f_2], a[f_3], a[f_4]\}.\end{aligned}$$

The body of the loop can be written as $\mathcal{U}(I) := F(\mathcal{W}(I))$, with F defined as

$$F(x_1, x_2, x_3, x_4) = x_1 - x_2 \cdot x_3 / x_4;$$

Single assignment program Let $\phi_1[I]$, $\phi_2[I]$, $\phi_3[I]$ and $\phi_4[I]$ be four new variables which are used to replace $a[i,j]$, $a[i,k]$, $a[k,j]$ and $a[k,k]$ in the loopbody respectively. Then, as described in section 3.5, the above algorithm can be converted into the following single-assignment program.

$$\begin{aligned}&\forall I = [k,j,i]^t \in \mathcal{I}:\\ &\phi_2[I] := \begin{cases}\phi_1[I'] & \text{if } A_1 I' = A_2 I,\ I' \in \mathcal{I},\ I' \prec I \preceq I' + c_1\\ a[f_2(I)] & \text{otherwise}\end{cases}; \quad \text{(s3.1)}\\ &\phi_3[I] := \begin{cases}\phi_1[I'] & \text{if } A_1 I' = A_3 I,\ I' \in \mathcal{I},\ I' \prec I \preceq I' + c_1\\ a[f_3(I)] & \text{otherwise}\end{cases}; \quad \text{(s3.2)}\\ &\phi_4[I] := \begin{cases}\phi_1[I'] & \text{if } A_1 I' = A_4 I,\ I' \in \mathcal{I},\ I' \prec I \preceq I' + c_1\\ a[f_4(I)] & \text{otherwise}\end{cases}; \quad \text{(s3.3)}\\ &\phi_1[I] := \begin{cases}\phi_1[I - c_1] & \text{if } I - c_1 \in \mathcal{I}\\ a[f_1(I)] & \text{otherwise}\end{cases}; \quad \text{(s2.1)}\\ &\phi_1[I] := F(\phi_1[I], \phi_2[I], \phi_3[I], \phi_4[I]);\end{aligned}$$

Localization Here we have three s3-statements, s3.1, s3.2 and s3.3.

The localization of s3.1 proceeds as follows. First, we express I as a function of I' under the constraints $A_1 I' = A_2 I$ and $I' \prec I \preceq I' + c_1$.

Let $I = [k, j, i]^t$ and $I' = [k', j', i']$. Then

(a) From $A_1 I' = A_2 I \Leftrightarrow [i', j']^t = [i, k]^t$ it follows that $i = i'$ and $k = j'$, hence I can be written as $I = [j', j, i']^t$.

(b) To satisfy the condition $I' \prec I \preceq I' + c_1 \Leftrightarrow [k', j', i']^t \prec [j', j, i']^t \preceq [k'+1, j', i']^t$, we must have

$[j' - k', j - j', 0]^t \succ 0$ and $[k' + 1 - j', j' - j, 0]^t \succ 0 \Rightarrow j' - k' > 0$ and $k' + 1 - j' \geq 0$.

Hence we have $k' < j' \leq k' + 1 \Rightarrow j' = k' + 1$. So we have $I = [j', j, i']^t = [k' + 1, j, i']^t$, and $I' = [k', j', i']^t$, with $j' = k' + 1$.

(c) Since I is in $\mathcal{I}$, according to the program loops, it follows that $j \geq (k' + 1) + 1$.

Therefore, the dependence vector $I - I'$ can be decomposed into

$$I - I' = [k' + 1, j, i']^t - [k', k' + 1, i']^t = [1, 1, 0]^t + p[0, 1, 0]^t = [1, 1, 0]^t + pc_2,$$

with $p = j - k' - 2$. Since $c_2 \in Nul(A_2)$, the value of $\phi_1[I']$ will be assigned to all the $\phi_2[I - pc_2]$, for all $p = j - k' - 2, ..., 0$. Therefore, the statement s3.1 can be converted into the following propagation statement:

$$\forall I = [k, j, i]^t \in \mathcal{I}:$$
$$\phi_2[I] := \begin{cases} \phi_2[I - c_2] & \text{if } I - c_2 \in \mathcal{I} \\ \phi_1[I - [1, 1, 0]^t] & \text{if } I - c_2 \notin \mathcal{I} \text{ and } I - [1, 1, 0]^t \in \mathcal{I} \\ a[f_2(I)] & \text{otherwise} \end{cases};$$

which is equivalent to

$$\forall I = [k, j, i]^t \in \mathcal{I}:$$
$$\phi_2[I] := \begin{cases} \phi_2[I - c_2] & \text{if } j > k + 1 \\ \phi_1[I - [1, 1, 0]^t] & \text{if } j = k + 1 \text{ and } k > 1 \\ a[f_2(I)] & \text{otherwise} \end{cases};$$

In a similar way, s3.2 can be converted into

$$\forall I = [k,j,i]^t \in \mathcal{I}:$$
$$\phi_3[I] := \begin{cases} \phi_3[I - c_3] & \text{if } i > k+1 \\ \phi_1[I - [1,0,1]^t] & \text{if } i = k+1 \text{ and } k > 1 \\ a[f_3(I)] & \text{otherwise} \end{cases};$$

and s3.3 can be converted into

$$\forall I = [k,j,i]^t \in \mathcal{I}:$$
$$\phi_4[I] := \begin{cases} \phi_4[I - c_{4,1}] & \text{if } j > k+1 \\ \phi_4[I - c_{4,2}] & \text{if } i > k+1 \\ \phi_1[I - [1,1,1]^t] & \text{if } i = j = k+1 \\ a[f_4(I)] & \text{otherwise} \end{cases};$$

As result, we will have the following RA for the matrix LDU decomposition:

$$\forall I = [k,j,i]^t \in \mathcal{I}:$$
$$\phi_2[I] := \begin{cases} \phi_2[I - c_2] & \text{if } j > k+1 \\ \phi_1[I - [1,1,0]^t] & \text{if } j = k+1 \text{ and } k > 1 \\ a[f_2(I)] & \text{otherwise} \end{cases};$$
$$\phi_3[I] := \begin{cases} \phi_3[I - c_3] & \text{if } i > k+1 \\ \phi_1[I - [1,0,1]^t] & \text{if } i = k+1 \text{ and } k > 1 \\ a[f_3(I)] & \text{otherwise} \end{cases};$$
$$\phi_4[I] := \begin{cases} \phi_4[I - c_{4,1}] & \text{if } j > k+1 \\ \phi_4[I - c_{4,2}] & \text{if } i > k+1 \\ \phi_1[I - [1,1,1]^t] & \text{if } i = j = k+1 \\ a[f_4(I)] & \text{otherwise} \end{cases};$$
$$\phi_1[I] := \begin{cases} \phi_1[I - c_1] & \text{if } j > k-1 \text{ and } i > k-1 \\ a[f_1(I)] & \text{otherwise} \end{cases};$$
$$\phi_1[I] := F(\phi_1[I], \phi_2[I], \phi_3[I], \phi_4[I]);$$

Note that the propagation of the variable $a[k,k]$ can be avoided by reformulating the body of the loop as

$$x[i,k] := a[i,k]/a[k,k];$$
$$a[i,j] := a[i,j] - x[i,k] \cdot a[k,j];$$

3.7.3 Matrix QR factorization

Nested loop program The nested loop program for the computation of the QR factorization of a matrix $[a_{ij}]$ can be written as follows:

```
for k = 1 to N − 1
    for i = k + 1 to N
        for j = k to N
        if j = k
            (θ[i, k], a[k, k]) := F1(a[k, k], a[i, k]);
        if j > k
            (a[k, j], a[i, j]) := F2(a[k, j], a[i, j], θ[i, k]);
```

where $F_1(x, y)$ is a function which computes the argument and length of the vector $[x, y]^t$, and $F_2(x, y, \alpha)$ is a function which rotates the vector $[x, y]^t$ over the angle α.

This program can be equivalently written as follows:

```
for k = 1 to N − 1
    for i = k + 1 to N
        for j = k to N
            (a[i, j], a[k, j], θ[i, k]) := Fk,j(a[i, j], a[k, j], θ[i, k]);
```

where the function $F_{k,j}(a[i,j], a[k,j], \theta[i,k])$ will compute $F_1(a[k,k], a[i,k])$ if $j = k$, and it will compute $F_2(a[k,j], a[i,j], \theta[i,k])$ if $j > k$.

The iteration space $\mathcal{I}$ of this program is

$$\mathcal{I} = \{I = [k, i, j]^t | 1 \leq k \leq N-1, k+1 \leq i \leq N, k \leq j \leq N\}.$$

The variables in the loopbody are:

(1) $a[i,j]$, or $a[f(I)]$, $f(I) = A_f I$;
(2) $a[k,j]$, or $a[g(I)]$, $g(I) = A_g I$;
(3) $\theta[i,k]$, or $\theta[h(I)]$, $h(I) = A_h I$;

where

$$A_f = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; A_g = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{ and } A_h = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

The null spaces of the indexing matrices A_f, A_g and A_h are all of dimension one, and propagation vectors in $Nul(A_f)$, $Nul(A_g)$, and $Nul(A_h)$ are $c_f = [1,0,0]^t$, $c_g = [0,1,0]^t$, and $c_h = [0,0,1]^t$ respectively.

For this program we have

$$\mathcal{U} = \mathcal{W} = \{a[f], a[g], \theta[h]\};$$
$$\mathcal{U}^\theta = \mathcal{W}^\theta = \{\theta[h]\}, \text{ hence } \mathcal{X}^\theta = \{\theta[h]\} \text{ and } \mathcal{Y}^\theta = \mathcal{Z}^\theta = \emptyset; \text{ and}$$
$$\mathcal{U}^a = \mathcal{W}^a = \{a[f], a[g]\}, \text{ hence } \mathcal{X}^a = \{a[f], a[g]\} \text{ and } \mathcal{Y}^a = \mathcal{Z}^a = \emptyset.$$

Single assignment program Let $\mathcal{F} = \{f, g\}$, and let $a[f]$, $a[g]$ and $\theta[h]$ be replaced in the loopbody by three new variables $\phi_f[I]$, $\phi_g[I]$ and $\phi_h[I]$ respectively. Then, according to the procedure given in section 3.5 we will obtain the following single-assignment program:

$$\forall I = [k,i,j]^t \in \mathcal{I}:$$

$$\phi_f[I] := \begin{cases} \phi_f[I'] & \text{if } I' = pred(I,\mathcal{D}),\ f(I) = f(I') \text{ or} \\ \phi_g[I'] & \text{if } I' = pred(I,\mathcal{D}),\ f(I) = g(I') \\ & \text{where } \mathcal{D} = Def(\mathcal{F}, f(I)) \\ a[f(I)] & \text{otherwise} \end{cases} ; \quad \text{(s2.1)}$$

$$\phi_g[I] := \begin{cases} \phi_g[I'] & \text{if } I' = pred(I,\mathcal{D}),\ g(I) = g(I') \text{ or} \\ \phi_f[I'] & \text{if } I' = pred(I,\mathcal{D}),\ g(I) = f(I') \\ & \text{where } \mathcal{D} = Def(\mathcal{F}, g(I)) \\ a[g(I)] & \text{otherwise} \end{cases} ; \quad \text{(s2.2)}$$

$$\phi_h[I] := \begin{cases} \phi_h[I - c_h] & \text{if } I - c_h \in \mathcal{I} \\ \theta[h(I)] & \text{otherwise} \end{cases} ;$$

$$(\phi_f[I], \phi_g[I], \phi_h[I]) := F_{k,j}(\phi_f[I], \phi_g[I], \phi_h[I]);$$

Localization

A: converting s2.1 into propagations.

Below we show that the conditions $I' = pred(I,\mathcal{D})$ and $f(I) = g(I')$ cannot be satisfied in statement s2.1. That is, we show that for all $I' \in \mathcal{I}$, if $f(I) = g(I')$, then $I \prec I'$.

Let $I = [k,i,j]^t$ and $I' = [k',i',j']^t$. Assume $f(I) = g(I') \Leftrightarrow A_f I = A_g I' \Leftrightarrow [i,j]^t = [k',j']^t$. That is, $k' = i$ and $j' = j$. Hence $I' = [k',i',j']^t = [i,i',j]^t$, and $I - I' = [k,i,j]^t - [i,i',j]^t = [k-i, i-i', 0]^t$. But as $I \in \mathcal{I}$, it follows that $i \geq k+1$, or $k - i < 0$, hence, $I - I' \prec 0$ or $I \prec I'$.

Therefore, statement s2.1 is equivalent to the following propagation statement

$$\forall I = [k,i,j]^t \in \mathcal{I}\text{: } \phi_f[I] := \begin{cases} \phi_f[I'] & \text{if } I' = pred(I,\mathcal{D}) \\ a[f(I)] & \text{otherwise} \end{cases};$$

where $\mathcal{D} = Def(f, f(I))$. Or equivalently

$$\forall I = [k,i,j]^t \in \mathcal{I}\text{:} \\ \phi_f[I] := \begin{cases} \phi_f[I - c_f] & \text{if } I - c_f \in \mathcal{I} \Leftrightarrow i > k+1 \\ a[f(I)] & \text{otherwise} \end{cases};$$

B: converting s2.2 into propagations.

In order to convert the statement s2.2 into a propagation statement, we first determine the set

$$D = Def(\mathcal{F}, g(I)) = Def(f, g(I)) \cup Def(g, g(I)), \text{ where}$$
$$Def(f, g(I)) = \{I' | f(I') = g(I), I' \in \mathcal{I}\}, \text{ and}$$
$$Def(g, g(I)) = \{I' | I' = I + pc_g, p \in \mathbf{Z}, I' \in \mathcal{I}\}.$$

To determine the set $Def(f, g(I))$, let $I = [k,i,j]^t$ and $I' = [k',i',j']^t$. Then

$$f(I') = g(I) \Leftrightarrow A_f I' = A_g I \Leftrightarrow [i',j']^t = [k,j]^t \Rightarrow i' = k \text{ and } j' = j.$$

Therefore, $I' = [k',k,j]^t$. Since $I' \in \mathcal{I}$, we have $1 \leq k' \leq N-1$, $k'+1 \leq k \leq N$, and $k' \leq j \leq N$. Hence

$$Def(f, g(I)) = \{I' | I' = [k',k,j]^t, 1 \leq k' \leq N-1, k'+1 \leq k \leq N, k' \leq \\ j \leq N\}.$$

Since $k'+1 \leq k$, we have $I'' = max_{\prec}(Def(f, g(I))) = [k-1,k,j]^t$. Moreover, $I'' \prec I$ since $I'' - I = [-1, k-i, 0]^t \prec 0$.

Now we show that $I'' \prec min_{\prec}(Def(g, g(I)))$. Since $Def(g, g(I)) = \{I' | I' = I + pc_g, p \in \mathbf{Z}, I' \in \mathcal{I}\}$, $I''' = min_{\prec}(Def(g, g(I))) = [k, k+1, j]^t$ since $I''' - c_g \notin \mathcal{I}$. So we have $I'' \prec I'''$ since $I''' - I'' = [1,1,0]^t \succ 0$.

To summarize, $I' = pred(I, \mathcal{D})$ in statement s2.2 is given by

$$I' = \begin{cases} I - c_g & \text{and } g(I) = g(I') \quad \text{if } I - c_g \in \mathcal{I} \\ I - [1,1,0]^t & \text{and } g(I) = f(I') \quad \text{if } I - c_g \notin \mathcal{I} \text{ and } \\ & \qquad\qquad\quad I - [1,1,0]^t \in \mathcal{I} \end{cases}.$$

Therefore, the statement s2.2 is equivalent to the following propagation statement:

$$\phi_g[I] := \begin{cases} \phi_g[I - c_g] & \text{if } I - c_g \in \mathcal{I} \\ \phi_f[I - [1,1,0]^t] & \text{if } I - c_g \in \mathcal{I} \text{ and } I - [1,1,0]^t \in \mathcal{I} \\ a[f_g(I)] & \text{otherwise} \end{cases};$$

Or equivalently

$$\phi_g[I] := \begin{cases} \phi_g[I - c_g] & \text{if } i > k+1 \\ \phi_f[I - [1,1,0]^t] & \text{if } i = k+1 \text{ and } k \geq 2 \\ a[f_g(I)] & \text{otherwise} \end{cases};$$

As result, we obtain the following RA for the QR matrix decomposition:

$$\begin{aligned} &\forall I = [k,i,j]^t \in \mathcal{I}: \\ &\quad \phi_f[I] := \begin{cases} \phi_f[I - c_f] & \text{if } i > k+1 \\ a[f(I)] & \text{otherwise} \end{cases}; \\ &\quad \phi_g[I] := \begin{cases} \phi_g[I - c_g] & \text{if } i > k+1 \\ \phi_f[I - [1,1,0]^t] & \text{if } i = k+1 \text{ and } k \geq 2 \\ a[f_g(I)] & \text{otherwise} \end{cases}; \\ &\quad \phi_h[I] := \begin{cases} \phi_h[I - c_h] & \text{if } j > k \\ \theta[h(I)] & \text{otherwise} \end{cases}; \\ &\quad (\phi_f[I], \phi_g[I], \phi_h[I]) := F_{k,j}(\phi_f[I], \phi_g[I], \phi_h[I]); \end{aligned}$$

Note that the value of k and j can also be propagated to iteration I through a boundary of $\mathcal{I}$ by introducing two additional variables $\phi_1[I]$ and $\phi_2[I]$, and the following propagation statements:

$$\phi_1[I] := \begin{cases} \phi_1[I - [1,0,0]^t] + 1 & \text{if } I - [1,0,0]^t \in \mathcal{I} \\ k & \text{otherwise} \end{cases};$$

$$\phi_2[I] := \begin{cases} \phi_2[I - [0,0,1]^t] + 1 & \text{if } I - [0,0,1]^t \in \mathcal{I} \\ j & \text{otherwise} \end{cases};$$

3.8 Concluding remarks

We have presented in this chapter a formal and systematic method to convert, when possible, a class of nested loop programs into firstly, single-assignment programs and then into RAs for systolic array implementation. Several examples are given.

The method presented provides also a systematic way of analyzing data dependences in nested loop programs. We believe that our approach is complementary to those of parallel/vectorizing compilers (see e.g., [Kuc78], [PKL80], [PW86]): instead of determining which iterations (statements) are independent of each other (i.e., parallelism), we determine which iterations depend strictly on each other. For instance, our method can be used to assist users to analyze and modify their programs in order to have more benefit from a parallel/vectorizing compiler. See e.g., [Los88] for such an example. On the other hand, we can use some of these compiler techniques to simplify the extraction of dependence vectors. For instance, we can use the technique described in [KMC72] to partition a loop program into a few coupled loop programs which have simpler loopbodies (see section 3.3).

Currently, an interactive CAD (computer-aided design) tool is being developed to assist designers to analyze and to convert nested loop programs into single-assignment and RA programs. The generalization of the method presented here in order to handle more complicated loop programs will be considered in future.

3.9 Appendix 3.A

In this appendix we describe a way to obtain an integer solution of the equation $AJ+b=BI+c$ [NW88].

Definitions

We denote the determinant of a matrix C by $det(C)$. An $n \times n$ matrix of integers C is called a unimodular matrix if $det(C) = \pm 1$.

An $n \times n$ nonsingular integral matrix $H = h_{ij}$ is said to be in Hermite normal form [NW88] if (a) H is lower triangular, i.e., $h_{ij} = 0$ for $i < j$; (b) $h_{ii} > 0$ for $i = 1, \ldots, n$; and (c) $h_{ij} \leq 0$ and $|h_{ij}| \leq h_{ii}$ for $i > j$.

Let $A \in \mathbf{Z}^{m \times n}$, $m \leq n$, and $rank(A) = m$. Then there is an $n \times n$ unimodular matrix C such that $AC = (H, 0)$, where H is in Hermite normal form. For instance, let $A = \begin{bmatrix} 2 & 6 & 1 \\ 4 & 7 & 7 \end{bmatrix}$. Then we have $AC = (H, 0)$, where

$$H = \begin{bmatrix} 1 & 0 \\ -3 & 5 \end{bmatrix}, \text{ and } C = \begin{bmatrix} 1 & 3 & -7 \\ 0 & -1 & 2 \\ -1 & 0 & 2 \end{bmatrix}.$$

Moreover, let $C = (C_1, C_2)$, with C_1 an $n \times m$ and C_2 an $n \times (n-m)$ matrix. Then (1) the set $\{x \in \mathbf{Z}^n \mid Ax = b\} \neq \emptyset$ if and only if $H^{-1}b \in \mathbf{Z}^m$, and (2) if $H^{-1}b \in \mathbf{Z}^m$, then all x can be written as $C_1 H^{-1} b + C_2 z$, $z \in \mathbf{Z}^{n-m}$.

The proof of this statement is classical, and lengthy; we refer the reader to [NW88] for details. Instead, we give a procedure for the computation of C and H in polynomial time [NW88](chapter 4).

Proposition 3.15 *Let $A = (a_1, ..., a_n)$, $gcd(a_{is}, a_{it}) = r$, and $pa_{is} + qa_{it} = r$, where a_i is the i-th column of A, $gcd()$ is the* greatest common divisor *operator, and p and q are relatively prime. Then there exists an $n \times n$ unimodular integer matrix C such that $AC = A'$, where*

$$\begin{aligned} a'_l &= a_l \quad \textit{for} \quad l \neq s, t; \\ a'_s &= pa_s + qa_t; \\ a'_t &= -\tfrac{a_{it}}{r} a_s + \tfrac{a_{is}}{r} a_t. \end{aligned}$$

In particular, $a'_{is} = r$ and $a'_{it} = 0$.

Proof: Take C to be an identity matrix in all but columns s and t. In column s, we have $c_{ss} = p$, $c_{ts} = q$, and $c_{is} = 0$ otherwise. In column t, we have $c_{st} = -a_{it}/r$, $c_{tt} = a_{is}/r$, and $c_{it} = 0$ otherwise. It is readily verified that $AC = A'$, and $det(C) = pa_{is}/r + qa_{it}/r = 1$.

□

The Hermite Normal Form Algorithm

initialization: $i = 1$.

Step 1: Work on row i. Set $j \leftarrow i + 1$.

Step 2: Work on row i and columns i and $j > i$.

- If $a_{ij} = 0$, do nothing.
- Otherwise find $r = gcd(a_{ii}, a_{ij})$ and p, q relatively prime such that $pa_{ii} + qa_{ij} = r$.
- Set $A \leftarrow AC$, where C is the unimodular matrix described in proposition 3.15, with $s = i$, $t = j$.
- If $j < n$, set $j \leftarrow j + 1$ and return to Step 2.
- If $j = n$, go to Step 3.

Step 3: Work on row i and column i. If $a_{ii} < 0$, then set $A \leftarrow AC$, where C multiplies column a_i by -1.

Step 4: Work on row i and column $j < i$. Set $j \leftarrow 1$. Set $A \leftarrow AC$, where C replaces column a_j by $a_j - \lceil a_{ij}/a_{ii} \rceil a_i$. If $j = i - 1$, set $i \leftarrow i + 1$. If $i > m$, stop. Otherwise return to Step 1. If $j < i - 1$, set $j \leftarrow j + 1$ and return to Step 4.

□

This algorithm will produce a unimodular matrix C, and a H with $(H, 0) = AC$. Note that this procedure gives us also a way of computing the null space of A as well, obviously, $Nul(A) = \{C_2 z | z \in \mathbf{Z}^{n-m}\}$.

Remark 3.6 It can be shown in a similar way that when $rank(A) = k < min(m, n)$, then there is an $n \times n$ unimodular matrix C, a permutation matrix P, and a $k \times k$ Hermite matrix H such that $PAC = \begin{bmatrix} H & 0 \\ 0 & 0 \end{bmatrix}$. Details are omitted here.

Now let us go back to the equation $AJ + b = BI + c \Leftrightarrow AJ = BI + c'$, with $c' = c - b$. We show now that J can be written as $J = Q'I + p$, where $Q' \in \mathbf{Q}^{n \times n}$ and $p \in \mathbf{Q}^n$. We only consider the case $rank(A) = m$ here. The case $rank(A) < m$ is similar, and is omitted.

Assume that $rank(A) = m$. Then there is an $n \times n$ unimodular matrix C, such that $AC = (H, 0)$, where H is an $m \times m$ matrix in Hermite normal form. Let $C = (C_1, C_2)$, with C_1 an $n \times m$ and C_2 an $n \times (n - m)$ matrix. Then, as we know, the solution

of equation $Ax = d$ can be written as $x = C_1H^{-1}d + C_2z$, $z \in \mathbf{Z}^{n-m}$. Put $J = x$, and $BI + c' = d$. Then we have

$$J = C_1H^{-1}(BI + c') + C_2z = C_1H^{-1}BI + C_1H^{-1}c' + C_2z.$$

Put $Q' = C_1H^{-1}B$, and $p = C_1H^{-1}c' + C_2z$ then we have the desired result.

Chapter 4

A Highly Parallel Architecture for Computer Image Synthesis

In this chapter we present a highly parallel architecture for the fast rendering of computer images. The system is based on an algorithm, usually referred to as the *radiosity method*, proposed by Greenberg *et al.* [GTGB84], [CG85] for the computer generation of realistic images. It consists of two loosely coupled sub-systems, and we will focus our discussion on the first sub-system, which computes form-factors by a new method based on a ray-tracing algorithm. Since rays can be traced independently, a high degree of parallelism can be achieved. An implementation of the ray-tracing algorithm using pipelined Cordic processors [BDdL86], [LHDB88] is described. Also, a feasible architecture for the entire sub-system is presented. (The second sub-system solves the system of radiosity equations by means of Gauss-Seidel iterations. A multiprocessor Gauss-Seidel iterative system solver is described in [BD87a].)

This chapter is organized as follows. In section 4.1 we give a brief discussion of the radiosity method proposed by Greenberg *et al.* In section 4.2 we describe a new method for computing form-factors by means of ray-tracing. In section 4.3 we present an algorithm for computing ray-patch intersections in Cordic arithmetic and its implementation using pipelined Cordic processors. In section 4.4 we present an architecture for the computation of a matrix of form-factors.

4.1 The radiosity method

In [GTGB84], a light model was introduced for 3D computer image synthesis. In this model, the interaction of light between diffusely reflecting surfaces is computed by solving (diffuse reflection) radiation field equilibrium equations. This method is, therefore, frequently referred to as the (light) radiosity method. It gives a good prediction of global illumination effects [GTGB84], [CG85].

The radiosity method calculates object-to-object diffuse reflections of light within a complex environment. The object surfaces are assumed to be "ideal" diffuse ("Lambertian" reflection). That is, the incident light is reflected from a surface in all directions with equal intensity. In this way, each surface is treated as a uniform secondary light source, and the history of rays is lost after reflection from a surface. The total rate of light energy leaving a surface is called the radiosity of the surface (the energy rate has dimension energy per unit time per unit area per unit frequency interval), which is equal to the sum of emitted energy and reflected energy of the surface. The reflected energy of the surface is equal to its reflection coefficient times the total incident energy, and the total incident energy on the surface is the sum of incident energies radiated from all surfaces, including itself.

To generate a computer image of an environment, the surfaces of the objects in the environment are subdivided into small areas, called patches. A patch should be small enough such that its radiosity can be assumed to be a constant.

Assume that there are N patches $P_1, ..., P_N$ in an closed environment. Let b_j, e_j, and ρ_j be the radiosity, the rate of emitted light energy, and the reflectivity of patch P_j, respectively. Then, we can write the following energy balance equation:

$$b_j = e_j + \rho_j \sum_{i=1}^{N} b_i f_{i,j}, \ 1 \leq j \leq N, \tag{4.1a}$$

where $f_{i,j}$ is called the *form-factor* from P_i to P_j. The form-factor $f_{i,j}$ gives the fraction of the energy radiated by P_i which is received by P_j.
Equivalently, we can write Eq. 4.1a in matrix form as:

$$(I_{N\times N} - \Lambda F)b = E, \tag{4.1b}$$

where $I_{N\times N}$ is the $N\times N$ identity matrix, $F=\{f_{i,j}\}$ is a matrix, $b=\{b_j\}$ and $E=\{e_j\}$ are column vectors, and $\Lambda=diag(\rho_j)$ is a diagonal matrix.

Eq. 4.1 describes an equilibrium energy balance within the environment. When the form-factor matrix F is known, the radiosity vector b can be obtained by solving the set of linear equations. It can be then used for rendering of final images of the environment; see e.g., [CG85] for details.

The computation of the form-factor matrix F is fairly time-consuming. Indeed, the exact calculation of (even) a single form-factor requires the evaluation of a double surface integral [RH78], [SC78]. Using the hemisphere method introduced by Nusselt [RH78], the double surface integral can be replaced by a single surface integral, which allows considerable computational savings. Recently, based on the hemisphere method, an approximation technique called "hemicube" method was introduced [CG85] (see section 4.2). Although the form-factor calculation is simplified and speeded up by this approximation technique, it takes a great deal of time in the image generations. In this chapter we present an algorithm-specific hardware system to accelerate the generation of form-factor matrix F. Also, a new parallel algorithm, which is suitable for VLSI implementation, is presented for the computation of form-factors.

Note that solving Eq. 4.1 is also an expensive task since the dimension of the matrix $I_{N\times N}-\Lambda F$ tends to be large. The generation of color images will be even more expensive: it will require to solve the system of linear equations Eq. 4.1 repetitively because the reflectivities ρ_j of the patches are wave-length dependent. A multiprocessor system is described in [BD87a] for the rapid solution of the system of linear equations Eq. 4.1b.

4.2 Computing form-factors by ray-tracing

In this section we describe a ray-tracing algorithm to approximate form-factors. The algorithm is inherently parallel since all the ray-patch intersections can be computed

independently.

Form-factors

Within a closed environment, the form-factor $f_{i,j}$ from a surface A_i to a surface A_j is defined as the average energy radiated by A_i and received by A_j.

Let dA_i be an element of the surface A_i, and dA_j an element of A_j. Let $\bar{n}_{p,i}$ and $\bar{n}_{p,j}$ be the normals of dA_i and dA_j respectively. Moreover, let r be the surface-to-surface distance between dA_i and dA_j, and the angles ϕ_i and ϕ_j, $0 \leq \phi_i, \phi_j \leq \frac{\pi}{2}$, be as shown in Fig. 4.1. Then, the form-factor $f_{i,j}$ is given by [RH78], [CG85]:

$$f_{i,j} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} \Gamma(dA_i, dA_j) dA_j dA_i, \tag{4.2}$$

where $\Gamma(dA_i, dA_j)$ is defined as

$$\Gamma(dA_i, dA_j) = \begin{cases} 1 & \text{if } dA_i \text{ can "see" } dA_j; \\ 0 & \text{if not.} \end{cases}$$

we say that dA_i can see dA_j if there is direct ray from dA_i to dA_j. The form-factor $f_{i,i}$ is zero for a planar patch since it cannot see itself.

Let

$$c_{i,j} = \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} dA_j. \tag{4.3a}$$

($c_{i,j}$ is also referred to as the "configuration factor" [SC78].) If we ignore the influence of Γ, then Eq. 4.2 can be written as

$$f_{i,j} = \frac{1}{A_i} \int_{A_i} c_{i,j}.dA_i \tag{4.3b}$$

Hemisphere and hemicube methods

To simplify the computation of form-factors, Nusselt [RH78] introduces a hemisphere projection method, which is based on a simple interpretation of Eq. 4.3a. As illustrated in Fig. 4.2a, where for the sake of simplicity the surfaces A_i and A_j are taken to be 2-dimensional, the factor $c_{i,j}$ is given by the fraction of the x-axis covered by the projection of A_j on the unit circle according to the hemisphere method. The computational procedure for form-factors is simplified and made more accurate by this method since no numerical integration is required for computing $c_{i,j}$.

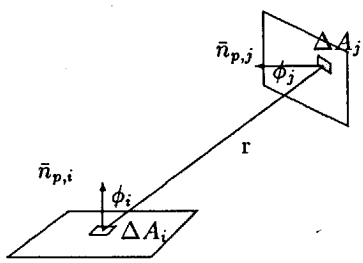


Figure 4.1: Form-factor geometry

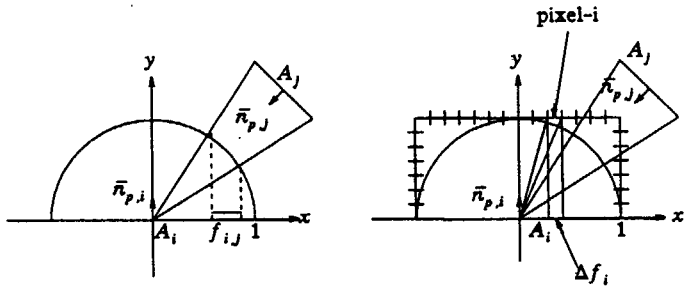
When the distance r is rather large in relation to the area of the two surfaces, the form-factor $f_{i,j}$ will be approximately given by $c_{i,j}$. We assume that this is the case for our patches.

The hemicube projection method [CG85] gives an approximation of the projection of a surface onto a hemisphere. This method is illustrated in Fig. 4.2b for the 2-dimensional case. As shown in Fig. 4.2b, a hemicube is constructed around the hemisphere, the surface of the hemicube is divided into "pixels", and to each pixel p_i "delta-form-factor" Δf_i is assigned. The hemicube method projects A_j onto the hemicube instead of the hemisphere. The form-factor is then approximated by $\sum_j \Delta f_j$, where the Δf_j's are the delta-form-factors of the pixels that are covered by the projection. To take the influence of the function Γ in Eq. 4.2 into account, when a pixel p_i is covered by the projections of more than one surface, then the delta-form-factor Δf_i is assigned to the closest surface.

Since the projection of a surface onto a hemicube is much easier to compute than the projection of a surface onto a hemisphere, the hemicube method simplifies the computation of a form-factor considerably. Further, since all the delta-form-factors can be pre-calculated and stored in a table, a great number of computations can be replaced by fast table-lookups.

Computing hemisphere projection by ray-tracing

The projection of a surface onto a hemicube can also be approximated by means of ray-tracing. Instead of projecting a surface onto the hemicube, rays can be cast



(a) (b)

Figure 4.2: (a) Hemisphere method. (b) Hemicube method

from the center of the hemicube through the center of the pixels. A pixel is then considered to be covered by the projection if the ray corresponding to the pixel intersects the surface. This method is clearly as accurate as the hemicube method, and it is inherently parallel since all the ray-patch intersections can be computed independently. We will follow this ray-tracing approach to compute form-factors.

Since this ray-tracing approach avoids the direct (numerical) computation of the projections, we can assign to the rays the delta-form-factors of the pixels defined directly on the hemi*sphere*. A straightforward way to define pixels on a hemisphere is shown in Fig. 4.3. In Fig. 4.3, the surface of the hemisphere is approximated by small areas $\Delta A = r^2 \Delta\theta \Delta\psi \sin\psi$ in the spherical coordinate system. To compute the delta-form-factors, let K be a given integer constant, and let $\Delta\theta = \Delta\psi = \frac{2\pi}{K}$, $\theta = (\Delta\theta)i$, $i = 0, 1, ..., K-1$, and $\psi = (\Delta\psi)j$, $j = 0, 1, ..., \lfloor \frac{K}{4} \rfloor$, where $\lfloor\ \rfloor$ represents the floor operator. Then, the delta-form-factor $\Delta f_{i,j}$ of a pixel (i, j) when $j \neq 0$ is given by

$$\Delta f_{i,j} = \frac{\cos\phi_i \cos\phi_j}{\pi r^2} \Delta A(i,j) = \frac{\cos\psi}{\pi r^2} r^2 \Delta\theta \Delta\psi \sin\psi = \frac{2\pi}{K^2} \sin 2\psi. \quad (4.4)$$

When $j = 0$, the area of the pixel is given by $\Delta A = \pi r^2 \sin^2 \Delta\psi$, and its delta-form-factor is equal to $\sin^2 \Delta\psi$.

Due to the symmetry, $\Delta f_{i,j}$ in Eq. 4.4 depends only on ψ (or j since $\psi = (\Delta\psi)j$). This can be used to reduce the size of the look-up table of delta-form-factors to $K+1$.

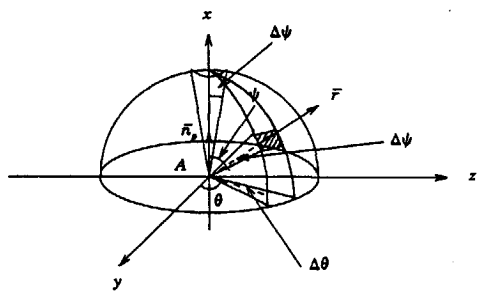


Figure 4.3: Hemisphere subdivision

From now on we will refer to the hemisphere that is placed at the origin $(0,0,0)^t$, above the y-z plane as the *reference hemisphere*, where the superscript t denotes vector transposition. Similarly, we will refer to the rays cast from the origin through the center of the pixels of the reference hemisphere the *reference rays*. A reference ray is specified by a unit vector $\bar{r} = (x_r, y_r, z_r)^t$ which indicates the direction of the ray.

4.3 A Cordic ray-tracing algorithm

In this section we describe an algorithm to compute the intersection of a ray with a planar patch in Cordic arithmetic. Also, an implementation of the algorithm using pipelined Cordic processors is given.

4.3.1 Cordic arithmetic

The Cordic computation technique was introduced by Volder [Vol59], and generalized by Walther [Wal71]. It is a bit-recursive algorithm for fixed-point execution of planar vector rotations

$$\Phi_m(\beta) = \begin{bmatrix} \cos(\sqrt{m}\beta) & -\sqrt{m}\sin(\sqrt{m}\beta) \\ \frac{1}{\sqrt{m}}\sin(\sqrt{m}\beta) & \cos(\sqrt{m}\beta) \end{bmatrix}$$

in circular ($m=1$), linear ($m=0$), or hyperbolic ($m=-1$) modes (otherwise called *coordinate systems*) indicated by the parameter m.

In each mode, Cordic can be used to perform two kinds of operations, called *rotation* and *vectorization* respectively. When its operation is rotation, it will receive a triple $<\bar{x}=(x,y)^t,\beta,m>$ as input, and produce the product $\Phi_m(\beta)\bar{x}$ as output. When its operation is vectorization, it will receive a pair $<\bar{x}=(x,y)^t,m>$ as input, and compute the length and the argument of $\bar{x}$ according to the parameter m. (The length and the argument of $\bar{x}$ in coordinate system m are given by $\sqrt{x^2+my^2}$ and $\frac{1}{\sqrt{m}}\tan^{-1}(\sqrt{m}y/x)$ respectively.)

The basic idea of Cordic is to compute $\Phi_m(\beta)$ in terms of a sequence of "micro rotations", and then to compute each micro rotation using shift (i.e., multiplication of a power of 2) and addition operations only.

Let $\beta=\sum_{j=0}^{p}\sigma_j\alpha_{m,j}$, where $\sigma_j=\pm1$ for all j, and $\alpha_{m,j}$'s are pre-determined positive angles ($\alpha_{m,p}$ is the smallest angle which determines the angle resolution). Then $\Phi_m(\beta)$ can be written as $K_m\prod_{j=0}^{p}\omega_m(\sigma_j,\alpha_{m,j})$, where

$$K_m=\prod_{j=0}^{p}\cos(\sqrt{m}\alpha_{m,j})$$

is a scaling factor, and

$$\omega_m(\sigma_j,\alpha_{m,j})=\begin{bmatrix}1 & -\frac{m}{\sqrt{m}}\tan(\sqrt{m}\sigma_j\alpha_{m,j})\\ \frac{1}{\sqrt{m}}\tan(\sqrt{m}\sigma_j\alpha_{m,j}) & 1\end{bmatrix}$$

is a micro rotation.

In [BDdL86], [LHDB88], a VLSI implementation of Cordic arithmetic is described. In this implementation, the execution of the micro rotations are pipelined. The term $\frac{1}{\sqrt{m}}\tan(\sqrt{m}\alpha_{m,j})$ in $\omega_m(\sigma_j,\alpha_{m,j})$, $0\le j\le p$, is implemented as either $2^{-s_j}+\eta_{m,j}2^{-s'_j}$ or 2^{-s_j}, with $\eta_{m,j}=\pm1$, and s_j and s'_j pre-determined positive integers. The s_j, s'_j, and $\eta_{m,j}$, $j=0,...,p$, are determined in such a way that K_m can be implemented by simple shift operations ($K_1=1/2$ and $K_{-1}=4$). The $\alpha_{m,j}$'s satisfy the inequality $|\beta-\sum_{j=0}^{p}\sigma_j\alpha_{m,j}|\le\alpha_{m,p}$ for any $\beta\in[\pi,-\pi]$. The main features of the VLSI Cordic processor are summarized in Tab. 4.1.

In the following discussion, we will use the symbol shown in Fig. 4.6a to represent such a Cordic processor. Only vector and angle entries are shown in the figure, control signals (for the selection of functions and operation modes) are omitted.

Table 4.1: Cordic chip features

process	size	Number of pins	Number of transistors	Accu-racy	Throughput
2μCMOS	1.2 cm^2	144	70000	16 bits	10^7 rotations/s

4.3.2 A Cordic ray-tracing algorithm

In the following discussion, we assume that a patch P is defined by the inner area of a convex planar polygon with a simple (non-self-intersecting) boundary, such as a triangle, a rectangle, etc., and a patch is represented by the unit normal $\bar{n}_p = (x_p, y_p, z_p)^t$ of the plane N of the patch, the coordinate of the center $\bar{P}_c$, and an ordered list of vertices $\bar{P}_i$ of the patch:

$$P : \{\bar{n}_p, \bar{P}_c, \bar{P}_i \mid \bar{P}_i = (x_i, y_i, z_i)^t,\ i = 1, ..., m,\ m \geq 3\}.$$

A point is inside a patch if the *winding number* of the boundary of the patch with respect to that point is ± 1 (the boundary of the patch is excluded). The winding number of a boundary with respect to a given point is defined as the net number of times that a point on the boundary wraps around the given point while the boundary point makes one complete traversal of the boundary [NS80].

A patch can only radiate light energy into the hemisphere enclosing its outward normal. This hemisphere can be represented by a set of vectors (directions) given by $\{\bar{v} \mid \bar{v}^t \cdot \bar{n}_p > 0\}$, where "$\cdot$" denotes the vector inner product operator. We call this side of the patch its front-side. Similarly, a patch can only receive light energies on its front-side.

From now on we assume that the vertices of a patch are numbered clockwise when viewed from the front-side of the patch.

We assume that a ray $\bar{R}$ is represented by its origin $\bar{r}_0 = (x_0, y_0, z_0)^t$ and a unit vector $\bar{r} = (x_r, y_r, z_r)^t$ indicating the direction of the ray. In this way, a point $\bar{R}(t)$ on the ray is parameterized as:

$$\bar{R}(t) = t\bar{r} + \bar{r}_0,\ \ t > 0,$$

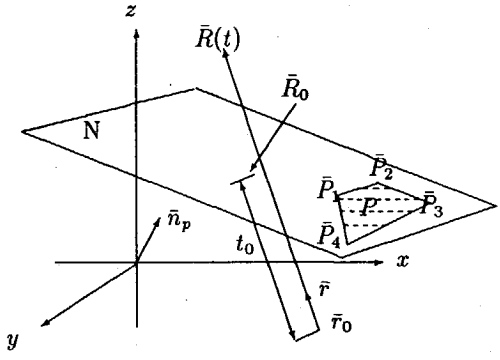


Figure 4.4: Ray-tracing planar convex patch

where t measures the distance from $\bar{r}_0$ to the point along the ray.

We compute the intersection of a ray with a patch in two steps (see Fig. 4.4). First, the intersection $\bar{R}_0 = \bar{R}(t_0)$ of the ray $\bar{R}(t)$ with the plane N of the patch (if any) is computed. Then, we test whether $\bar{R}_0$ is enclosed by the patch P. The ray intersects the patch if $\bar{R}_0$ is inside the patch; there is no intersection otherwise.

I. Computing the intersection of a ray with the plane of a patch

Assume that a ray $\bar{R}(t)$ intersects the plane N of a patch P at $\bar{R}_0$ from the front-side. Let $\bar{P} \neq \bar{R}_0$ be an arbitrary point in the plane of the patch. Then the following conditions must hold:

$$\begin{cases} \bar{n}_p^t \cdot (\bar{R}_0 - \bar{P}) = 0 \\ \bar{r}^t \cdot \bar{n}_p < 0 \\ t_0 > 0 \end{cases} . \tag{4.5}$$

We will assume from now on that $\bar{P} = \bar{P}_1$, the first vertex of the patch.

Let Σ be a square orthogonal matrix, i.e., $\Sigma\Sigma^t = I$. Then Eq. 4.5 is invariant under a transformation induced by Σ. That is:

$$\begin{aligned} &(\Sigma\bar{n}_p)^t \cdot \Sigma(\bar{R}_0 - \bar{P}_1) = 0 \iff \\ &\quad (\Sigma\bar{n}_p)^t \cdot (t_0\Sigma\bar{r} + \Sigma\bar{r}_0 - \Sigma\bar{P}_1) = 0; \end{aligned} \tag{4.6a}$$

$$(\Sigma\bar{r})^t \cdot \Sigma\bar{n}_p < 0; \tag{4.6b}$$

$$t_0 > 0. \tag{4.6c}$$

To compute $\bar{R}_0$, we first choose Σ in such a way that $\Sigma\bar{n}_p = (1,0,0)^t$, and then use the Σ to transform the vectors $\bar{r}, \bar{r}_0, \bar{P}_1, \bar{P}_2, ..., \bar{P}_m$ into $\bar{r}', \bar{r}_0', \bar{P}_1', \bar{P}_2', ..., \bar{P}_m'$. That is,

$$(1,0,0)^t = \Sigma\bar{n}_p, \tag{4.7a}$$

$$(\bar{r}', \bar{r}_0', \bar{P}_1', \bar{P}_2', ..., \bar{P}_m') = \Sigma(\bar{r}, \bar{r}_0, \bar{P}_1, \bar{P}_2, ..., \bar{P}_m). \tag{4.7b}$$

The required Σ can be computed in Cordic arithmetic by vectoring the normal $\bar{n}_p$ of the patch P in the circular mode, which solves the following equation:

$$(1,0,0)^t = \Sigma\bar{n}_p = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix}, \tag{4.8}$$

where $\alpha = \tan^{-1}(y_p/x_p)$, and $\beta = \tan^{-1}(z_p/x_p^*)$, with $x_p^* = x_p\cos\alpha + y_p\sin\alpha$. Since Σ depends on two arguments α and β, we will write $\Sigma(\alpha,\beta)$ instead of Σ when necessary.

Let P' and N' denote the patch P and the plane of the patch N after the transformation respectively, see Fig. 4.5a (recall that the normal of N' is $(1,0,0)^t$). Then N' is given by $x = d$, where $|d|$ is the distance between N' and the $y-z$ plane, and all the vertices of P' have the same x-component d.

We determine next whether the transformed ray $t\bar{r}'+\bar{r}_0'$ intersects the transformed patch P'. Let $\bar{r}' = (x_r', y_r', z_r')^t$, $\bar{r}_0' = (x_0', y_0', z_0')^t$, and $\bar{P}_1' = (x_1', y_1', z_1')^t$. Then from Eq. 4.6a, t_0 must satisfy

$$t_0 x_r' + x_0' - x_1' = 0. \tag{4.9a}$$

As shown in Fig. 4.5b, the conditions (4.6b) and (4.6c) will only be satisfied when

$$\begin{cases} x_r' < 0 \\ x_0' > x_1' \end{cases}. \tag{4.9b}$$

In this case, the ray intersects the plane of the patch from the front-side. When t_0 is known, the intersection point $\bar{R}_0'$ of the transformed ray with the plane N' will be given by

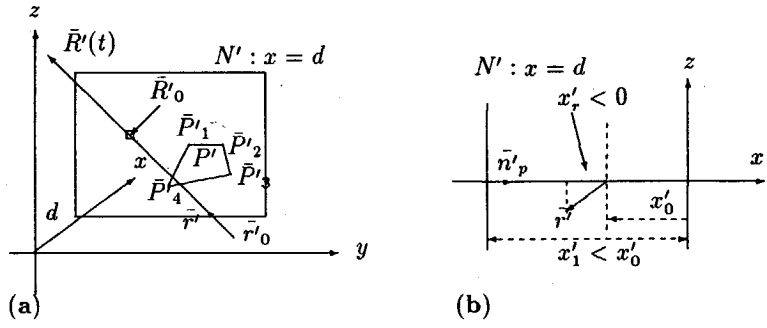


Figure 4.5: (**a**) Patch transformation. (**b**) Intersection condition

$$\bar{r'}_0 + t_0\bar{r'} = (x'_0, y'_0, z'_0)^t + t_0(x'_r, y'_r, z'_r)^t. \tag{4.10}$$

Eqs. 4.9a and 4.10 can both be solved in Cordic arithmetic. First we compute t_0 by "vectoring" the vector $(x'_r, x'_0 - x'_1)$ in the linear mode, i.e., by determining the transformation

$$\begin{pmatrix} * \\ 0 \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ t_0 & 1 \end{bmatrix} \begin{pmatrix} x'_r \\ x'_0 - x'_1 \end{pmatrix}, \tag{4.11}$$

where "*" means *don't care*. This will produce t_0 as result.

Then we compute $t_0 y'_r$ and $t_0 z'_r$ in Eq. 4.10 by rotating the vectors $(y'_r, 0)$ and $(z'_r, 0)$ over t_0 in the linear mode of Cordic:

$$\begin{pmatrix} * \\ t_0 y'_r \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ t_0 & 1 \end{bmatrix} \begin{pmatrix} y'_r \\ 0 \end{pmatrix}, \tag{4.12a}$$

$$\begin{pmatrix} * \\ t_0 z'_r \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ t_0 & 1 \end{bmatrix} \begin{pmatrix} z'_r \\ 0 \end{pmatrix}. \tag{4.12b}$$

(These operations are of course equivalent to division and multiplication respectively).

Since the vertices $\bar{P}'_1, ..., \bar{P}'_m$ of P' and $\bar{R}'_0$ have all the same x-component, we treat them from now on as 2-dimensional vectors defined in the $y-z$ plane.

II. Testing whether $\bar{R}'_0$ is enclosed by P'

Once the intersection point $\bar{R}'_0$ is computed, we have to determine next whether this point is inside the transformed P' (obviously, $\bar{R}(t_0)$ is inside P if and only if $\bar{R}'_0$ is inside P').

To determine whether $\bar{R}'_0$ is inside P', we introduce vectors

$$\bar{V}_i = \bar{P}'_i - \bar{R}'_0, \; i = 1, ..., m. \tag{4.13a}$$

If one of them is equal to zero, then $\bar{R}'_0$ coincides with a vertex of P', and there is no intersection. Assume that all the $\bar{V}_i$'s are not equal to zero. Let $\phi_i \in [\pi, -\pi]$ be the argument of $\bar{V}_i$, $i = 1, ..., m$, and let $\Phi(\alpha)$ denote a circular rotation over the angle α. Then we compute the vectors $\bar{V}'_i$ as follows:

$$\bar{V}'_{(i) mod\, m} = \Phi(-\phi_{i-1}) \bar{V}_{(i) mod\, m}, \; i = 2, ..., m, m+1, \tag{4.13b}$$

where *mod* denote a modulo operator. The order in which the vectors are computed must be kept as indicated.

Proposition 4.1 *Let P' be a patch which is represented by a convex planar polygon, and let $\bar{V}_i$, ϕ_i, $\Phi(-\phi_i)$, and $\bar{V}'_i$, $i = 1, ..., m$ be defined as above. Then $\bar{R}'_0$ is inside P' if and only if the z-components of all the vectors $\bar{V}'_i$ are non-zero and have equal signs.*

Proof: Assume that $\bar{u}_i$ is the unit vector in the direction of $\bar{V}_i$, $i = 1, 2, ..., m$. If we rotate a vector $\bar{u}$ from $\bar{u}_{i-1}$ to $\bar{u}_{(i) mod\, m}$ for $i = 2, ..., m, m+1$, then the origin will be enclosed by the convex patch P' if the sum of angles over which $\bar{u}$ is rotated is equal to $\pm 2\pi$. Since P' is assumed to be convex, all these angles are less than π, and the direction of the rotations are all the same. According to Eq. 4.13b, the argument of $\bar{V}'_{(i) mod\, m}$ is equal to the angle over which $\bar{u}$ will rotate from $\bar{u}_{i-1}$ to $\bar{u}_{(i) mod\, m}$ for all i. Since they are all less than π, the direction of the rotations of $\bar{u}$ is indicated by the sign of the z-component of the vectors $\bar{V}'_i$.

□

The computation of ϕ_i and $\bar{V}'_{i+1}$ in Eq. 4.13b can both be carried out in Cordic arithmetic. The angle ϕ_i can be computed by vectoring $\bar{V}_i$ in the circular mode of

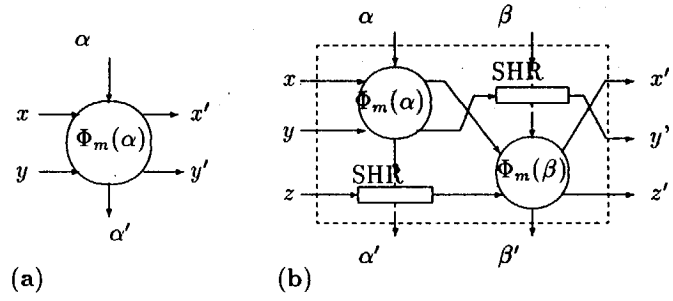


Figure 4.6: (**a**) Cordic processor. (**b**) Processor unit PE-a

Cordic, and $\bar{V}'_{i+1}$ can be computed by rotating the vector $\bar{V}_{i+1}$ over the angle $-\phi_i$, also in the circular mode of Cordic.

Remark 4.1 Let B be the bounding box of P' defined by

$$B = \{(y_{min}, y_{max}),\ (z_{min}, z_{max})\},$$

where y_{min} and y_{max} are the minimum and the maximum of the y-components of the vertices of P' respectively, similarly for z_{min} and z_{max}. Then $\bar{R}'_0$ is outside P' if it is outside B. Clearly, the above test is not necessary if $\bar{R}'_0$ is outside B.

4.3.3 A Cordic intersection processor

In hardware, we use a pipelined unit $PE-a$ as shown in Fig. 4.6b to compute the $\Sigma(\alpha, \beta)$ of Eq. 4.7a and the vector transformations of Eq. 4.7b. It consists of two pipelined Cordic processors and two shift registers (SHRs) for data synchronization. Both SHRs have the same throughput and latency as a Cordic processor.

Let L denote the latency of a pipelined Cordic processor, then the latency of $PE-a$ will be equal to $2L$; it has the same throughput as a Cordic processor.

When we pass the vector sequence $\bar{n}_p, \bar{r}_0, \bar{P}_1, \bar{r}, \bar{P}_2, ..., \bar{P}_m$ through the $PE-a$ by vectoring the first vector $\bar{n}_p$ in the circular mode of Cordic and rotating the rest of the vectors in the same mode, we will obtain as output the transformed vectors $(1,0,0)^t, \bar{r}'_0, \bar{P}'_1, \bar{r}', \bar{P}'_2, ..., \bar{P}'_m$ as required in Eq. 4.7.

Given the sequence $(1,0,0)^t, \bar{r'}_0, \bar{P'}_1, \bar{r'}, \bar{P'}_2, ..., \bar{P'}_m$, or equivalently,

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} x'_0 \\ y'_0 \\ z'_0 \end{pmatrix}, \begin{pmatrix} x'_1 \\ y'_1 \\ z'_1 \end{pmatrix}, \begin{pmatrix} x'_r \\ y'_r \\ z'_r \end{pmatrix}, \begin{pmatrix} x'_2 \\ y'_2 \\ z'_2 \end{pmatrix}, ..., \begin{pmatrix} x'_m \\ y'_m \\ z'_m \end{pmatrix},$$

we fed it then into a pre-alignment network, called $PreAN$, to produce the following sequences A and B:

$$A: \begin{pmatrix} x'_r \\ x'_0 - x'_1 \end{pmatrix}, \begin{pmatrix} y'_r \\ 0 \end{pmatrix}, \begin{pmatrix} z'_r \\ 0 \end{pmatrix}, \begin{pmatrix} * \\ * \end{pmatrix} \quad ..., \quad \begin{pmatrix} * \\ * \end{pmatrix}$$

$$B: \quad \begin{pmatrix} * \\ * \end{pmatrix}, \quad \begin{pmatrix} * \\ * \end{pmatrix}, \begin{pmatrix} y'_0 \\ z'_0 \end{pmatrix}, \begin{pmatrix} y'_1 \\ z'_1 \end{pmatrix}, ..., \begin{pmatrix} y'_m \\ z'_m \end{pmatrix}$$

("*" means *don't care*). The network $PreAN$ can be implemented to have maximum latency 2 and the same throughput as that of a Cordic processor; details are omitted.

The output sequences of $PreAN$ are directly used by another pipelined unit PE–b shown in Fig. 4.6c to evaluate Eqs. 4.11 and 4.12. Similarly to PE–a, PE–b uses two pipelined Cordic processors and two SHRs. The upper Cordic processor performs vectoring operations, and the lower Cordic processor performs rotation operations, both in linear mode. $PE-b$ has obviously the same throughput and latency as a Cordic processor. The output sequences produced by $PE-b$ are as follows:

$$\begin{pmatrix} t_0 y'_r \\ * \end{pmatrix}, \begin{pmatrix} t_0 z'_r \\ * \end{pmatrix}, \begin{pmatrix} * \\ * \end{pmatrix} \quad ..., \quad \begin{pmatrix} * \\ * \end{pmatrix}$$

$$\begin{pmatrix} * \\ * \end{pmatrix}, \quad \begin{pmatrix} y'_0 \\ z'_0 \end{pmatrix}, \begin{pmatrix} y'_1 \\ z'_1 \end{pmatrix} \quad ..., \quad \begin{pmatrix} y'_m \\ z'_m \end{pmatrix}$$

They are fed into a post-alignment network called $ProAN$, which has the same throughput and latency as $PreAN$, to produce the vector sequence $\bar{V}_1, \ldots, \bar{V}_m$ as defined in Eq. 4.13a.

Finally, we use PE–c shown in Fig. 4.6d to compute ϕ_i and $\bar{V'}_{i+1}$ of Eq. 4.13b, with as input the sequence $\bar{V}_1, \ldots, \bar{V}_m, \bar{V}_1$. As shown in Fig. 4.6d, $PE-c$ uses two Cordic processors which are configured similarly to PE–b, however, its upper Cordic processor performs vectoring operations in the circular mode, and its lower Cordic

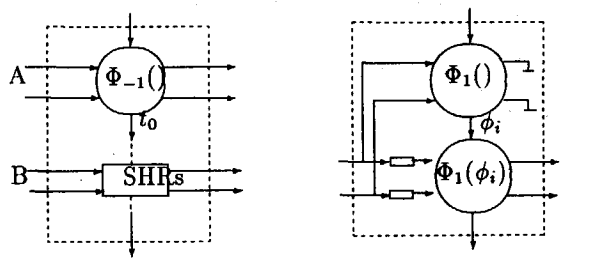


Figure 4.6: (**c**) Processor unit PE-b. (**d**) Processor unit PE-c

processor performs rotation operations, also in the circular mode. The latency of this processor unit is $L+1$.

The lower Cordic processor will output the sequence $\bar{V}_2, \ldots, \bar{V}_m, \bar{V}_1$, from which we can determine whether or not the sign of the z-components of these vectors are the same.

Putting $PE-a$, $PreAN$, $PE-b$, $ProAN$, and $PE-c$ in cascade (see Fig. 4.6e), we obtain a pipelined intersection processor (IP), which is able to compute 10^7 ray-path intersections per second. The latency of IP is $4L+5$. (When the support $\bar{r}_0$ of each ray is equal to zero, then $PreAN$ and $ProAN$ can be both implemented with latency 1.)

Remark 4.2 In this implementation, the condition given in Eq. 4.9b and the bounding box information of a transformed patch (see remark 4.1) are not used since all the computations are pipelined. The bounding box information is however used in an alternative implementation described in [BD89b].

4.4 Architecture for form-factor computations

In this section we describe an array architecture for the computation of a matrix F of form-factors as required in Eq. 4.1. Assuming that there are N patches $P_1, ..., P_N$,

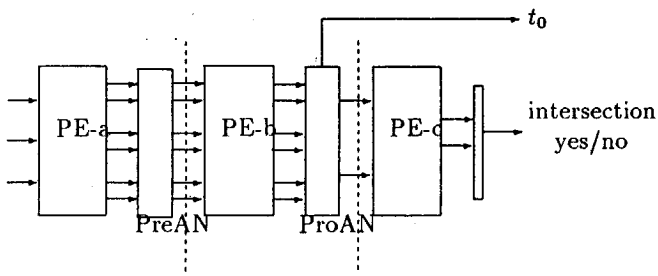


Figure 4.6: (e) Intersection processor IP

then N^2-N form-factors $f_{i,j}$ have to be computed, one for each patch pair (P_i, P_j), $i \neq j$ ($f_{i,i}=0$ for planar patches).

The required matrix F of form-factors can be generated with a high degree of parallelism: all the form-factors can be computed in parallel, and all the rays can be processed in parallel as well within the computation of each form-factor. There is, therefore, a large number of choices of system architectures.

The computation of form-factors have however to be organized in an optimal way, since the number of available processing elements in a realistic hardware system will be much fewer than the number of patches and rays to be processed. In our design, we have chosen as a solution to pipeline the operations to reduce the number of processing elements. The pipelined operations are, moreover, divided into stages separated by fast memories (FIFOs, or first-in-first-out buffers). Such division allows us to adjust parameters involved in the architecture, such as the size of memories, the speed of data transmissions, etc., at the end of the design so that the form-factors can be computed in an efficient way: a great number of unnecessary computations can be skipped in the subsequent stages while keeping the number of interruptions of pipelined operations to a minimum.

Another issue which is worthy of mentioning is the trade-off between the pre-processing and the on-line generation of data. For instance, many computations can be saved if we sort the patches according to their normals, and number the patches with equal normal consecutively: blocks of zero entries will then be created

around the diagonal of the form-factor matrix F since the form-factor defined by two patches with equal normal is zero. Further, since the normal of a patch has to be vectorized several times in form-factor computations, we can also compute it in advance and store the results for later use. However, such pre-processing of patches will take extra time, processing power and storage. Creating and storing intermediate data is especially critical and is unacceptable due to the already large amount of data. Taking into account that fast pipelined processors (Cordic) will be used in our system, we will not perform any such pre-processing of patches. (It is, however, natural to assume that the patches obtained by the subdivisions of a larger one are numbered in succession. The form-factor matrix tends therefore to have block of zeros around its diagonal).

Architecture for form-factor computations

The form-factor matrix F is computed row by row. Below we restrict our discussion to the computation of one, say, the ith row $F_{i,1},...,F_{i,N}$ of the matrix F. (More rows can be computed in parallel by duplicating the hardware). All these form-factors are computed in the coordinate system defined by the patch P_i, i.e., the origin of the coordinate system is the center $\bar{P}_{c,i}$ of P_i, and the positive x-axis is in the direction of the normal $\bar{n}_{p,i}$ of the patch. (Note that the form-factors are invariant under the rotation and translations of the coordinate system).

Assume that the patches $P_1, ..., P_N$ are already defined in the coordinate system of P_i, then the rays cast from the hemisphere of P_i are simply given by the reference rays (see section 4.2), which will be denoted by $\bar{R}_1, ..., \bar{R}_M$, with M the total number of reference rays.

The main part of the computations for generating the form-factors $F_{i,1},...,F_{i,N}$ is given by the following loop program.

Program-1:

for $j = 1$ to M

    $T[j] := \infty$;

    $I[j] := \perp$;

for $i = 1$ to N

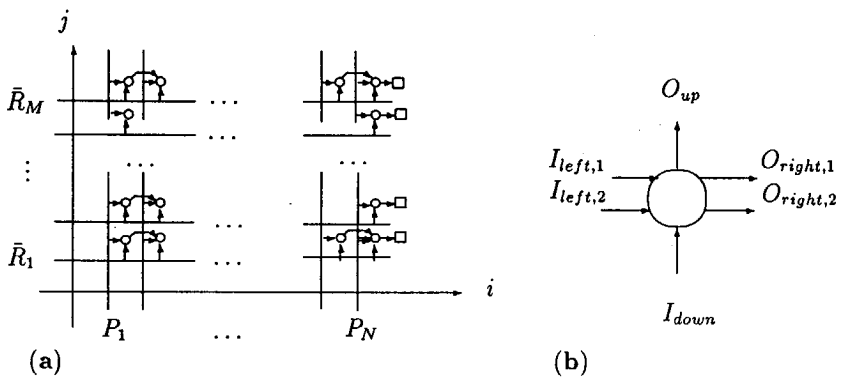


Figure 4.7: (a) Dependence graph of form-factor computations. (b) Virtual Processor

```
for j = 1 to M
begin
    t := intersection(R̄_j, P_i);                    (S.1)
    if t < T[j] then T[j] := t; I[j] := i;          (S.2)
end
```

where $\perp$ means *undefined*, and $intersection(\bar{R}, P)$ is a function which returns the distance measured from the support of the ray $\bar{R}$ to the intersection point of $\bar{R}$ with the patch P, if any; it will return the value ∞, otherwise.

The output of this program is a vector $I[j]$, $j = 1, ..., M$; if $I[j] = k$, then ray $\bar{R}_j$ intersects patch P_k. To compute the form-factors $F_{i,1}, ..., F_{i,N}$, we first initialize them to zeroes, and then we add the delta-form-factorΔf_j of the ray $\bar{R}_j$ to $F_{i,k}$ if $I[j] = k$, $1 \leq k \leq N$. Mapping the operations of this last step to hardware is straightforward; details are omitted.

An equivalent dependence-graph representation of this program is shown in Fig. 4.7a, where each node represents a virtual processor with three inputs and three outputs as shown in Fig. 4.7b (not to be confused with Cordic processor shown in Fig. 4.6a). If in Fig. 4.7a we identify each processor by its $i - j$ coordinates, then the functionality of the processor (i, j) can be specified as follows:

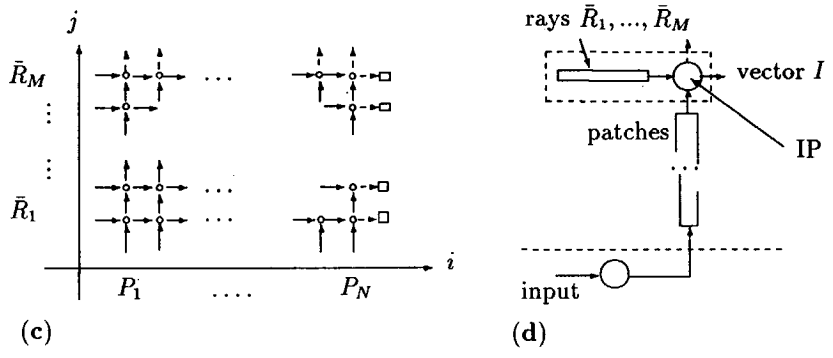


Figure 4.7: (c) Localized Dependence graph. (d) Computing the ith row of F using one IP

I_{down}: get P_i;
$I_{left,1}$: get $\bar{R}_j$;
$I_{left,2}$: get $\{t', i'\}$;
$t := intersection(\bar{R}_j, P_i)$;
$O_{right,2}$: put $\{t, i\}$ if $t < t'$, or $\{t', i'\}$ otherwise;
O_{up}: put P_i;
$O_{right,1}$: put $\bar{R}_j$;

where $\{t, i\}$ means that ray $\bar{R}_j$ intersects patch P_i with $0 < t \leq \infty$.

From Fig. 4.7a we can obtain a *localized* dependence graph as shown in Fig. 4.7c (where the inputs $I_{left,1}$, $I_{left,2}$ and outputs $O_{right,1}$, $O_{right,2}$ of each processor are each represented by a single edge), from which various (systolic) array implementations can be derived.

Although there are many possibilities to implement the dependence graph shown in Fig. 4.7c in hardware, we consider a single intersection processor implementation shown in Fig. 4.7d to be practical. In this implementation, the I/O bandwidth for loading patches is much lower than that of the internal data communication because one patch is tested against many rays for possible intersections. When there are more intersection processors available, we will use them to compute more

rows of the form-factor matrix F in parallel. (Besides, the number of available IPs in a realistic system can hardly match the number of rows of F).

As shown in Fig. 4.7d, the main computations will take place in the dashed box, which consists of a buffer containing all the rays $\bar{R}_1, ..., \bar{R}_M$, and an intersection processor IP. It computes the inner loop of program-1, and produces the vector $I[j]$, $j = 1, ..., M$ as output.

Up to now we have assumed that all the patches $P_1, ..., P_N$ are defined in the coordinate system of P_i. Since this will not always be the case, we have to use some additional hardware, shown in Fig. 4.7d (below the dashed line) to transform the patches to the coordinate system of P_i. Such coordinate transformation for a patch P is defined by $\Sigma P - \Sigma P_{c,i}$, where Σ is a rotation matrix such that $\Sigma \bar{n}_{p,i} = (1,0,0)^t$, $\bar{n}_{p,i}$ is the normal of P_i, and $P_{c,i}$ is the center of the patch P_i. The processor configuration $PE-a$ with some additional adders can be used for both computing Σ and performing the transformation.

Note that after this transformation, we can skip a patch if (a) it is found below the $y-z$ plane (i.e., the maximum of the x-coordinates of its vertices is smaller than or equal to zero), or (b) it has the same normal as P_i. It will produce a zero form-factor in these cases since it cannot be seen by P_i.

4.5 Concluding remarks

We have presented in this chapter a highly parallel architecture for the rapid generation of computer images using the light radiosity model.

The system presented is also suitable for implementing the iterative radiosity method described in [CGIB86]. However, it does not include the new results described in e.g., [ICG86], [WCG87], [SPL88], for the rendering of an environments containing non-diffuse and specular surfaces. An implementation, which takes these effects into account, is recently described in [YHDD89]. The issues of how to store patches in a database and how to retrieval patches from the database in an efficient way are not addressed here.

The system is not yet implemented. Even the architecture study presented here

is far from complete. However, this work shows that it is feasible to map the radiosity method which can be used to produce high-quality computer-synthesized images on hardware. We have just provided a beginning of parallel implementation of the light radiosity method.

This work has demonstrated that to design a complex VLSI system we need a powerful computer-aided design system which allows us to specify and verify a design in an interactive and hierarchical way. The CAD system must moreover support us to make design decisions, to perform design evaluations (simulations), and to explore and document design alternatives. Currently, a CAD system, called HIFI, with these objectives is in the development, see e.g., [AD88], [vdHdLDD89], [DdLvdHD89].

The material used in this chapter has been partially published in [BD87b], [BD87c], [BD89b].

Chapter 5

Concluding Remarks

In this thesis we have studied the problem of the systematic design of regular VLSI processor arrays, in particular, systolic arrays. A systolic array has the advantage of modularity and regularity, local interconnection, and highly pipelined processing, which are crucial to efficient implementation in VLSI.

In chapter 2 we presented a methodology to design fixed-size systolic arrays that are suitable for solving problems of arbitrary sizes. The method can also be used in the design of wavefront arrays. As described in [Kun88], a simple way to compare a systolic array and its wavefront array counterpart is

$$\text{Wavefront array} = \text{systolic array} + \text{data flow computing.}$$

The method and theory presented in chapter 2 provide us with not only a means to adjust size, dimension, I/O bandwidth, etc., of a systolic array, but also a means to adjust and manipulate its interconnection pattern (see section 2.3.2). The latter can be used to map several algorithms to one fixed-size array with (weakly) programmable processing elements, or to map a given systolic algorithm to an existing systolic computing network.

Further investigation is required in order to extend the method to answer, among others, the following questions: Given a class of systolic algorithms, what is the optimal systolic array that can be used to execute all the algorithms in the class? how to design it? and how does one program the processing elements? Conversely, given a systolic array consisting of (weakly) programmable processing elements (e.g.,

transputers), what is the class of algorithms that can be executed optimally on this array? If a problem can not be solved using one systolic array, when and how can it be solved using several systolic arrays?

In chapter 2 we have assumed that algorithms are given in the form of regular algorithms (RAs). However, the class of algorithms which are available in RA form is rather limited, and the conversion of a given algorithm into an RA is generally not an easy task. In chapter 3 we studied the problem of when and how a nested loop program can be converted into an RA. Our discussion has shown that even a simple nested loop program may possess complicated data dependences, and the class of nested loop programs that can directly be mapped onto systolic arrays is rather limited.

However, the following questions remain to be answered: How to handle broadcasts that can not be localized? If some part of the dependence graph of a nested loop program is irregular, how to eliminate such irregular subgraphs? Note that it is sometimes possible to project the dependence graph to a lower dimension so that a regular dependence graph can be obtained. For instance, the CORDIC processor described in [BDdL86], [LHDB88] is in fact a linear systolic array with nearest-neighbor interconnections only. It can be obtained by the projection of a 2-dimensional dependence graph into a regular dependence graph of dimension one; the interconnections within each node (processor) are, however, less regular. What if a nested loop program cannot be converted into an RA? It is sometimes possible in such a situation to rearrange the operations (iterations) of the program to obtain a new program which can successfully be converted into RAs. A typical example of this is the conversion of the transitive closure and the shortest path problem in graph theory into RAs studied by Kung *et al.*, see [KLL87], [Kun88] for details. Can such a process be formalized?

Although the class of algorithms that are implementable on systolic arrays is restricted, the theory and implementation of systolic arrays provides us with a general guideline for reformulating an algorithm to make it suitable for VLSI massive parallel processing. In addition to the techniques mentioned above (i.e., re-order operations, localize data dependences, etc.), an algorithm has frequently to be reformulated in

terms of new type of operations such that (a) the resulting algorithm requires only a few types of simple operations (i.e., operation unification); (b) branch and control are kept to a minimum; (c) the algorithm must be intrinsically stable numerically; and (d) the algorithm must possess massive parallelism and its data dependences can be characterized as much as possible by a regular data flow structure. See also [Dew88]. A good example of this is the systolic array implementation of a class of feed-forward matrix inversion algorithms described in [Jai89], [JD89].

Using this guideline, in chapter 4 we examined an algorithm proposed by Greenberg *et al.* [GTGB84], [CG85], called the radiosity method, for the generation of realistic computer images. We have reformulated one of the most time-consuming part of the algorithm, i.e., the form-factor computations, and obtained a highly parallel architecture, in which form-factors are computed using one type of operation: ray-tracing (i.e. computing the intersection of a ray and a patch). Since rays can be traced independently, the algorithm is highly parallel, with a simple and regular data flow (see section 4.6). Moreover, an algorithm and its implementation using pipelined Cordic processors for ray-tracing planar polygon patches are described. (Therefore, all the ray-patch intersections can be computed using Cordic operations.) This work shows that it is feasible to map the radiosity method which can be used to produce high-quality computer-synthesized images on VLSI parallel architecture. However, it is still just the beginning of efficient parallel implementation of the radiosity method. The question still remains: How to design a computer graphics engine to produce high-quality computer-synthesized realistic images in real-time ?

We hope that we have been able to convince the reader that a VLSI processor array is an important and practical means in real-time applications, and that it can be designed in a formal and systematic way.

Bibliography

[AD88] J. Annevelink and P. Dewilde. Hifi: a functional design system for vlsi processing arrays. In *Proc. Int. Conf. on Systolic Arrays*, pages 413–452, San Diego, 1988.

[AHU74] A.V. Aho, J.E. Hopcraft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Reading, MA: Addison-Wesley, 1974.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[BCKT79] U. Banerjee, S.C. Chen, D.J. Kuck, and R.A. Towle. Time and parallel processor bounds for fortran-like loops. *IEEE Trans. Computers*, C-28(9), Sept. 1979.

[BD87a] Jichun Bu and Ed F. Deprettere. A parallel vlsi algorithm for fast sparse matrix solution by gauss-seidel iteration. In *Proc. ISCAS*, 1987.

[BD87b] Jichun Bu and Ed.F. Deprettere. A vlsi algorithm for computing form-factors in radiative transfer computer image synthesis. In *Computer Graphics 1987: proceedings of CG international'87*, pages 180–193, Springer-Verlag, May 1987.

[BD87c] Jichun Bu and Ed.F. Deprettere. A vlsi system architecture for high-speed radiative transfer 3d image synthesis. In *Proc. EUROGRAPHICS*, August 1987.

[BD88a] Jichun Bu and Ed F. Deprettere. Analysis and modeling of sequential iterative algorithms for parallel and pipeline implementations. In *Proc. IEEE Symp. on Circuit and Systems*, pages 1961–1965, 1988.

[BD88b] Jichun Bu and Ed F. Deprettere. Converting sequential iterative algorithms to recurrent equations for automatic design of systolic arrays. In *Proc. ICASSP*, pages 2025–2028, 1988.

[BD89a] Jichun Bu and Ed.F. Deprettere. On the localization problem in systolic algorithm design. In *Proc. IEEE Symp. on Circuits and Systems*, Portland, 1989.

[BD89b] Jichun Bu and Ed.F. Deprettere. A vlsi system architecture for high-speed radiative transfer 3d image synthesis. *Visual Computer*, (5):121–133, 1989.

[BDdL86] Jichun Bu, Ed F.A. Deprettere, and A. de Lange. On the optimization of pipelined silicon cordic algorithms. In *Proc. EUSIPCO*, 1986.

[CG85] Michael F. Cohen and Donald P. Greenberg. The hemi-cube, a radiosity solution for complex environments. In *ACM Proc. of Siggraph'85 19(3)*, pages 31–40, July 1985.

[CGIB86] Michael F. Cohen, Donald P. Greenberg, David S. Immel, and Philip J. Brock. An efficient radiosity approach for realistic image synthesis. *IEEE Computer Grahphics and Applications*, 26-35, March 1986.

[Che86] M.C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *J. Parallel and Distributed Computing*, 3:461–491, 1986.

[CS84] P. Cappello and K. Steiglitz. Unifying vlsi array design with linear transformations of space-time. In *Advances in Computing Research*, pages 23–65, JAI Press Inc., 1984.

[DdLvdHD89] P.M. Dewilde, A.A.J de Lange, A.J van der Hoeven, and F. Deprettere. Hifi: an object oriented system for the structural synthesis of signal processing algorithms and the vlsi compilation of signal flow graphs. In *Proc. IFIP Int. Workshop on Applied Formal Methods for Correct VLSI Design*, Nov. 1989.

[Dew88] P. Dewilde. New algebraic methods for modelling large-scale integrated circuits. *Int. Journal of Circuit Theory and Applications*, 16:473–503, 1988.

[DI85] J.M. Delosme and Ilse C.F. Ipsen. An illustration of a methodology for the construction of efficient systolic architectures in vlsi. In *Proc. 2nd Int. Symp. on VLSI Technology*, 1985.

[DI86] J.M. Delosme and Ilse C.F. Ipsen. Design methodology for systolic arrays. In *Proc. SPIE*, 1986.

[FFW85] J.A.B Fortes, K.S. Fu, and B.W. Wah. Systematic approaches to the design of algorithmically specified systolic arrays. In *Proc. ICASSP*, pages 26–29, March 1985.

[FGQ86] P. Frison, P. Gachet, and P. Quinton. Designing systolic arrays with diastol. In *VLSI SIGNAL PROCESSING II*, pages 93–105, IEEE Press, New York, 1986.

[FM84] J.A.B Fortes and D.I. Moldovan. Data broadcasting in linearly scheduled array processors. In *Proc. 11th Symp. on Comp. Architecture*, 1984.

[GKT79] L. Guibas, H.T. Kung, and C.D. Thompson. Direct vlsi implementation of combinatorial algorithms. In *Proc. Conf. on VLSI: Architecture, Design and Fabrication*, pages 509–525, January 1979.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Battaile Bennett. Modeling the interaction of light between diffuse surfaces. *Computer Graphics*, 18(3):213–222, July 1984.

[HB85] K. Hwang and F.A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill Book Company, 1985.

[HC80] K. Hwang and Y.H. Cheng. Vlsi computing structure for solving large scale linear systems of equations. In *Proc. Parallel Processing Conf.*, pages 217–227, 1980.

[HC81] K. Hwang and Y.H. Chung. Partitioned algorithms and structures for large-scale matrix computations. In *Proc. 5th Symp. Comput. Arithmetic*, May 1981.

[Hel85] D. Heller. Partitioning big matrices for small systolic arrays. In S.Y. Kung, H.J. Whitehouse, and T. Kailath, editors, *VLSI and Modern Signal Processing*, pages 185–199, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[HNS87] S. Horiike, S. Nishida, and T. Sakaguchi. A design method of systolic arrays under the constraint of the number of the processors. In *Proc. ICASSP*, 1987.

[ICG86] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environment. *Computer Graphics*, 20(4):133–142, August 1986.

[Jai86] K. Jainandunsing. Optimal partitioning scheme for wavefront/systolic array processors. In *Proc. IEEE Symp. on Circuits and Systems*, 1986.

[Jai89] K. Jainandunsing. *Parallel Algorithms for Solving Systems of Linear Equations and Their Mapping on Systolic Arrays*. PhD thesis, Delft University of Technology, Delft, The Netherlands, January 1989.

[JD89] K. Jainandunsing and Ed.F. Deprettere. A new class of parallel algorithms for solving systems of linear equations. *SIAM J. Sct. Stat. Comput.*, 10(5):880–912, September 1989.

[Joh82] L. Johnson. A computational array for the qr method. In *Proc. M.I.T. Conf. Advanced Research VLSI*, Jan. 1982.

[KD89] P.F.C. Krekel and Ed.F. Deprettere. A systolic algorithm and architecture for solving sets of linear equations with multi-band coefficient matrix. *Journal of VLSI Signal Processing*, 1:143–152, 1989.

[KL80] H.T. Kung and C.E. Leiserson. Systolic arrays for vlsi. In *Sparse Matrix Proceedings*, pages 245–282, Philadelphia:Society of Industrial and Applied Mathematicians, 1980.

[KLL87] S.Y. Kung, S.C. Lo, and P.S. Lewis. Optimal systolic design for the transitive closure and the shortest path problems. *IEEE Trans. on Computers*, 603–614, May 1987.

[KMC72] D.J. Kuck, Y. Muraoka, and S.C. Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Trans. Computers*, C-21(12), Dec. 1972.

[KMW67] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[Kre89] P.F.C. Krekel. *Partitioning and Clustering in Regular Index Spaces.* Technical Report, Delft University of Technology, Delft, The Netherlands, 1989.

[Kuc78] D.J. Kuck. *The Structure of Computers and Computations.* John Wiley & Sons, Inc., 1978.

[Kuh80] R.H. Kuhn. Transforming algorithms for single-stage and vlsi architectures. In *Proc. Workshop Interconnection Networks for Parallel and Distributed Processing*, 1980.

[Kun79] H.T. Kung. Let's design algorithms for vlsi systems. In *Proc. Caltech Conf. on VLSI*, pages 65–90, Jan. 1979.

[Kun80a] H.T. Kung. The structure of parallel algorithms. *Adv. Comput.*, 19:65–111, 1980.

[Kun80b] S.Y. Kung. Vlsi array processor for signal processing. In *Proc. Conf. on Adv. Res. in Integrated Circuits*, Jan. 28-30 1980.

[Kun82] H.T. Kung. Why systolic architectures? *Computer*, 37–45, Jan. 1982.

[Kun84] S.Y. Kung. On supercomputing with systolic/wavefront array processors. *Proceedings of the IEEE*, 39–46, July 1984.

[Kun88] S.Y. Kung. *VLSI Array Processors.* Prentice-Hall International Editions, 1988.

[Lam74] L. Lamport. The parallel execution of do loops. *Comm. ACM*, 12(2), February 1974.

[LHDB88] A.D. Lange, A.V.D. Hoeven, Ed.F. Deprettere, and J. Bu. An optimal floating-point pipeline cmos cordic processor. In *Proc. ISCAS*, 1988.

[LK88] P. Lee and Z.M. Kedem. Synthesizing linear array algorithms from nested for-loop algorithms. *IEEE Trans. on Computers*, 37(12), December 1988.

[Los88] M.L.F. Lossie. *The Generalized Schur Algorithm: Round-off Analysis, Vectorization, and an Application to VLSI Modelling.* Technical

Report, Delft University of Technology, Delft, The Netherlands, May 1988.

[LRS83] C.E. Leiserson, F.M. Rose, and J.B. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. 3rd Caltech Conf. on VLSI*, pages 87–116, Computer Science Press, 1983.

[LS83] C.E. Leiserson and J.B. Saxe. Optimizing synchronous systems. *J. VLSI and Computer Systems*, 1(1), 1983.

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

[Mol82] D.I. Moldovan. On the analysis and synthesis of vlsi algorithms. *IEEE Trans. Computers*, C-31(11):1121–1126, Nov. 1982.

[Mol83] D.I. Moldovan. On the design of algorithms for vlsi systolic arrays. *Proceedings of the IEEE*, 71(1), January 1983.

[MW84] L.W. Miranker and A. Winkler. Space-time representation of computational structures. *Computing*, 32:93–114, 1984.

[MWF84] D.I. Moldovan, C.I. Wu, and J.A.B. Fortes. Mapping an arbitrary large qr algorithm into a fixed-size vlsi array. In *Proc. of the 1984 Int. Conf. on Parallel proc.*, pages 21–24, Aug. 1984.

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

[NLV87] J.J. Navarro, J.M. Llaberia, and M. Valero. Partitioning: an essential step in mapping algorithms into systolic array processors. *IEEE Computer*, July 1987.

[NS80] Martin E. Newell and Carlo H. Sequin. The inside story on self-intersecting polygons. *LAMBDA*, Second Quarter:20–24, 1980.

[NS82] D. Nassimi and S. Sahni. Data broadcasting in simd computers. *IEEE Trans. Computers*, C-30(2), February 1982.

[NW88] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, Inc., 1988.

[PKL80] D.A. Padua, D.J. Kuck, and D.H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Trans. Computers*, C-29(9), Sept. 1980.

[PS82] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization, Algorithms and Complexity.* Prentice-Hall, Inc., 1982.

[PW86] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations and supercomputers. *Comm. ACM*, 29(12), Dec. 1986.

[Qui84] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. 11th Annual Int. Symp. on Comput. Arch.*, pages 208–214, June 1984.

[Raj86] S.V. Rajopadhye. *Synthesis, Verification and Optimization of Systolic Arrays.* PhD thesis, Department of Computer Science, University Utah, December 1986.

[Rao85] S.K. Rao. *Regular Iterative Algorithms and Their Implementations on Processor Arrays.* PhD thesis, Stanford University, October 1985.

[RF87] S.V. Rajopadhye and R.M. Fujimoto. Systolic array synthesis by static analysis of program dependencies. In *Proc. of PARLE*, June. 1987.

[RH78] Siegel Robert and John R. Howell. *Thermal Radiation Heat Transfer.* Hemisphere Publishing Corp., 1978.

[RK86] S.K. Rao and T. Kailath. What is a systolic algorithm. In *Proc. SPIE Vol.614 Highly Parallel Signal Processing Architectures*, 1986.

[RK88a] S.K. Rao and T. Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proceedings of the IEEE*, 76(3), March 1988.

[RK88b] V.P. Roychowdhury and T. Kailath. Regular processor arrays for matrix algorithms with pivoting. In *Proc. Int. Conf. on Systolic Arrays*, 1988.

[RK88c] V.P. Roychowdhury and T. Kailath. Regular processor arrays for matrix algorithms with pivoting. *Submitted to Comm. ACM*, Feb. 1988.

[RPF86] S.V. Rajopadhye, S. Purushothaman, and R.M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proc. Sixth Conf. on Foundations of Software Technology and Theoretical Computer Science*, Feb. 1986.

[RTRK88] V. Roychowdhury, L. Thiele, S.K. Rao, and T. Kailath. On the localization of algorithms for vlsi processor arrays. In *VLSI and Signal Processing*, pages 459–470, IEEE Press, New York, 1988.

[SC78] E.M. Sparrow and R.D. Cess. *Radiation Heat Transfer.* Hemisphere Publishing Corp., 1978.

[SC90] C.J. Scheiman and P. Cappello. A processor-time minimal systolic array for transitive closure. In *To appear in Conf. Int. Conf. on Application Specific Array Processors*, 1990.

[SK85] R. Schreiber and P.J. Kuekes. Systolic linear algebra machines in digital signal processing. In S.Y. Kung, H.J. Whitehouse, and T. Kailath, editors, *VLSI and Modern Signal Processing*, pages 389–405, Prentice-Hall, Englewood Cliffs, NJ, 1985.

[SPL88] Min-Zhi Shao, Qun-Sheng Peng, and You-Dong Liang. A new radiosity approach by procedural refinements for realistic image synthesis. *Computer Graphics*, 22(4):93–101, August 1988.

[Thi88] L. Thiele. On the hierarchical design of vlsi processor arrays. In *Proc. IEEE Symp. on Circuits and Systems*, pages 2517–2520, Helsinki, 1988.

[Thi89a] L. Thiele. Design of local concurrent algorithms. In *Proc. MTNS*, Amsterdam, 1989.

[Thi89b] L. Thiele. On the design of piecewise regular processor arrays. In *Proc. IEEE Symp. on Circuits and Systems*, Portland, 1989.

[Ull84] J.D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.

[US89] L. Thiele U. Schwiegelshohn. Linear systolic arrays for matrix computations. *Journal of Parallel and Distributed Computing*, 28–39, (7) 1989.

[vdHdLDD89] A.J van der Hoeven, A.A.J. de Lange, Ed.F. Deprettere, and P. Dewilde. A new model for the high level description and simulation of vlsi networks. In *Proc. 26th DAC conf.*, June 1989.

[vDQ88] V. van Dongen and P. Quinton. Uniformization of linear recurrence equations: a step towards the automatic synthesis of systolic arrays. In *Proc. Int. Conf. Systolic Arrays*, pages 473–482, May 25-27 1988.

[Vol59] J.E. Volder. The cordic trigonometric computing technique. *IRE Trans. Electronic Computers*, EC-8(3):330–340, 1959.

[Wal71] J.S. Walther. An unified algorithm for elementary functions. In *Proc. Spring Joint Comput. Conf. AFIPS press*, page 397, 1971.

[WCG87] J.R. Wallace, M.F. Cohen, and D.P. Greenberg. A two-pass solution to the rendering equation-a synthesis of ray-tracing and radiosity methods. In *ACM Proc. of Siggraph'87 21(4)*, pages 311–320, July 1987.

[WD88] Y. Wong and J.M. Delosme. Broadcast removal in systolic algorithms. In *Proc. Int. Conf. Systolic Arrays*, pages 403–412, May 25-27 1988.

[YHDD89] A.C. Yilmaz, S. Hagestein, Ed.F. Deprettere, and P. Dewilde. A hardware solution to the generalized two-pass approach for rendering of artificial scenes. *To be published*, 1989.

Samenvatting

In dit proefschrift wordt een systematische manier voor het ontwerpen van reguliere VLSI-processor arrays, in het bijzonder systolische arrays, voorgesteld en theoretisch uitgewerkt. Meer specifiek wordt met de conceptuele en constructieve uitwerking van de volgende twee cruciale stappen een bijdrage geleverd tot het systematiseren van het ontwerptraject: (1) het afbeelden van reguliere algoritmen op systolische arrays waarvan de afmetingen constant zijn, (2) het omzetten van imperatieve geneste-lus-algoritmen (in de vorm van standaard computerprogrammas) in equivalente *reguliere algoritmen* die als formele specificatie voor een architectuursynthese veel geschikter zijn dan de gegeven specificatie. Een andere bijdrage van dit proefschrift is een nieuw VLSI-algoritme voor het displayen van artificiele, computergegenereerde foto-realistische beelden. Dit algoritme is een goed voorbeeld van voor het succesvol oplossen van complexe maar realistische problemen wat in het algemeen aan een architectuur synthese vooraf zal gaan.

Hoofdstuk 2 gaat over het architectuur ontwerp. Het ontwerpen van systolische arrays is conceptueel wel bekend. Uitgangspunt is een volledig explicit geometrisch model van het algoritme. Dit model is formeel en niet suggestief ten aanzien van implementaties. Het systolisch array dat in de ontwerp sessie uit dit model verkregen wordt heeft eveneens een omvang die direct samenhangt met de uitgestrektheid van de graaf. Erg realistisch is dit niet. Inderdaad, een array kan niet willekeurig groot zijn en algoritmen die alleen in termen van *maat* verschillen kunnen niet afgebeeld worden op eenzelfde array omdat arrays niet zo maar ingekrompen of geexpandeerd kunnen worden. Wij hebben een methode ontwikkeld om systolische arrays te ontwerpen die een constante complexiteit hebben in termen van aantallen processor-

cellen, onafhankelijk van de afmetingen van het algoritme dat erop afgebeeld moet worden; dit wil zeggen voor het implementeren van een klasse van algoritmen die zijn geparametriseerd met betrekking tot hun afmetingen. De methode voor het ontwerpen van systolische arrays met vaste afmetingen die wij voorstellen en uitwerken in dit hoofdstuk bevat onder andere alle in de literatuur voorgestelde technieken voor het reduceren van de afmetingen van een array. Ze bevat ook modificaties en uitbreidingen van deze methoden en nieuwe methoden. Het verband tussen al deze reductie mogelijkheden wordt verduidelijkt en aanbevelingen voor het doen van keuzes worden gedaan. De procedure die wordt voorgesteld komt in het kort hierop neer: (a) ontwerpen van een (conceptueel) array met onbepaalde afmetingen, (b) partitioneren van het array zodat de afmeting van een enkele partitie, ongeveer de gewenste array afmeting is. Zo'n partitie wordt het gereduceerd array genoemd. (c) Het gereduceerd array wordt (eventueel) verder gereduceerd door het toepassen van processor- (cel-, element-) klustering. Twee klustertechnieken worden beschreven die toegepast kunnen worden om bepaale aanpassingen te realizeren, zoals herleiden van aantal processorcellen tot een minimum, verder reduceren van de dimensie en/of de afmetingen van het gereduceerde array, in balans brengen van locale geheugencapaciteiten en externe communicatiebandbreedtes.

Hoofdstuk 3 gaat over algoritme specificaties. Algoritmen waarvoor een systolische implementatie bestaat worden systolische algoritmen genoemd. Deze vormen een sub-klasse van de groep van zogenoemde reguliere algoritmen (RA's). In dit hoofdstuk geven wij een mathematische analyse van klassieke lus-programmas gericht op het systematisch omzetten van zulke programmas in RA specificatie vorm. De transformatie gaat dan in 2 stappen: (a) het omzetten van het gegeven programma in "single-assignment" vorm waarin alle data-afhankelijkheden in de vorm van algebraische expressies explicit zijn gemaakt, (b) het localiseren, indien mogelijk, van de data-afhankelijkheden. In onze methode worden iteraties (recursie stappen) opgespoord die in stricte zin van elkaar afhankelijk zijn. Dit in tegenstelling tot wat sommige paralleliserende en/of vectoriserende compilers doen, namelijk bepalen welke iteraties of statements onafhankelijk van elkaar zijn. Voor het ontwerpen van systolische algoritmen is het belangrijk te weten welke iteraties van elkaar afhankelijk

zijn en hoe die afhankelijkheid is. De communicatie-mogelijkheden tussen (macro) operaties zijn immers beperkt bij zulke realizaties.

Hoofdstuk 4 gaat over het ontwerpen van algoritmen. Er wordt een algoritme en een bijbehorende parallel-VLSI-architectuur gepresenteerd voor de computer-synthese van hoge-resolutie, foto-realistische weergave van objecten en interieurs. Daarbij wordt gesteund op de zogenoemde "Radiosity-methode" die de lichtintensiteit van de omgeving berekent volgens een fysich model. In dit hoofdstuk hebben we willen aantonen, aan de hand van een voorbeeld, dat het niet voldoende is te beschikken over "een" algoritme (programma) en de krachtige middelen uit hoofdstukken 3 en 4 waarmee RA specificaties en gereduceerde arrays kunnen worden geconstrueerd. Het ontwerptraject van de architectuur kan niet meer uit het gegeven algoritme halen dan er in zit, en de meeste algoritmen zijn imperatieve lus-programmas die zijn geoptimaliseerd naar Von-Neuman architecturen toe. Deze zijn serieel, en wat de rekencapaciteit betreft gebaseerd op vermenigvuldigen en accumulatie als basis operaties. Zulke algoritmen zullen dus opnieuw ontworpen moeten worden, rekening houdend met de randvoorwaarden van de VLSI-implementatie. Daarom wordt in dit hoofdstuk aandacht besteed aan dat herformuleringsprocess, dat beoogt de geschiktheid van het algoritme (programma) voor reguliere VLSI-implementatie te verhogen, indien mogelijk. In de radiosity-methode worden zogenoemde vormfactoren uitgerekend die geometrische zichtbaarheden meten van de "elementen" in de ruimte die het object of de omgeving modelleren. We geven een ray-tracing-algoritme voor het berekenen van deze factoren. Deze stralen waarmee transmissies en reflecties van licht worden gesimuleerd kunnen onafhankelijk van elkaar worden "getrokken". Daarom is een hoge graad van inherent parallelisme aanwezig, en dit aspect wordt uitgebuit door operaties op verschillende niveaus te pipelinen. Voor de vormfactorberekening wordt een VLSI-algoritme en een bijbehorende architectuur gegeven.

About the Author

Jichun Bu was born in Changchun, China, on March 22, 1960. In 1978 he graduated from the Beijing No.5 High School in Beijing, China. After a short stay of 3 months at the Qinghua University in Beijing, China, in February 1979 he came to the Netherlands for a technical study course.

From March to September 1979 he followed an introductory Dutch language course in Delft, the Netherlands. Then he started his study of electrical engineering at the Delft University of Technology where he received in February 1985 the Ingenieurs'degree (the equivalent of an M.Sc.) with distinction.

From March to June 1985 he worked in the Automatic System Design Group of the Department of Electrical Engineering, Eindhoven University of Technology, in Eindhoven, the Netherlands.

In July 1985 he joined the Laboratory of Network Theory of the Department of Electrical Engineering at the Delft University of Technology, where he embarked on his work towards a Ph.D. From February 1989 to June of that year he was a visiting researcher at the Institute of Microelectronics, University of Saarlands, in Saarbrucken, West Germany.