Automatic Derivation of Systolic Arrays for LU Decomposition

Sanjay V. Rajopadhye

CIS-TR-87-12
November 6, 1987

Department of Computer and Information Science
University of Oregon

Automatic Derivation of Systolic Arrays for LU Decomposition

Sanjay V. Rajopadhye
Computer Science Deptartment
University of Oregon
Eugene, Or 97403

Abstract

We present a number of systolic arrays for decomposing a matrix into its lower and upper triangular factors (LU-decomposition). These architectures have been *formally derived* using techniques for synthesizing systolic arrays from affine recurrence equations, and the entire design process can be automated. The derivation highlights two important aspects of our synthesis methodology. Firstly, the initial specification is a high level one similar to a nested loop program. We illustrate the use a technique called *explicit pipelining* to automatically localize the data dependencies. Secondly, the architectures that we present have interesting features such as control signals, and specialized behavior of certain processors (such as boundary processors). We describe how these characteristics (and also processor initialization signals) can be derived automatically.

1 Introduction

Over the last decade there have been tremendous advances in computational power due to the advent of VLSI technology. However, the price for this power is the increased complexity of the devices. In order to combat the complexity it is essential to use a disciplined approach during the design process. One such discipline is the systolic array, a parallel architecture that consists of *regular* interconnections of a *large* number of very *simple* processors. The term *systolic array* was first proposed by Kung and Leiserson [8,9,10], and they also addressed the issues of efficient layout of such circuits [11]. Initially, a large number of papers described architectures for specific problems such as matrix multiplication, L-U decomposition of matrices, solving a set of equations, convolution, dynamic programming, etc. (see [7,8,9] for an extensive bibliography). Subsequently, a number of researchers addressed the problem of *automatically* deriving systolic arrays, notably Fortes, Moldovan et. al. [4,5,6,12], Quinton [13], Delosme and Ipsen [1,2,3], (and many others, see [14] for an detailed survey).

In this paper we present a number of architectures for LU-decomposition of a matrix. Each of these has been rigorously derived by using techniques that we have developed elsewhere [14,15,16]. While similar synthesis methods have been proposed by other researchers, our method has two main advantages which are illustrated by the designs presented here. Firstly, the initial specifications for our design are more general than those used in earlier approaches because they have *affine*, rather than *uniform* dependencies. We then use a special technique called *explicit pipelining* that enables us to automatically localize the dependencies. Secondly and more importantly, the architectures derived here have a number of interesting features *viz* specialized computation by certain processors (such as boundary processors), non-uniform data-flow governed by control signals, specialized computation governed by control signals, etc. We are able to formally derive each of these computational aspects, as well as processor initialization.

The rest of this paper is organized as follows. In the next section (Section 2 we briefly describe the synthesis methodology. Then, in Section 3 we define the initial specification for the LU-decomposition problem (as an Affine Recurrence Equation). Section 4 deals first with pipelining the dependencies of the recurrence, and then determining a timing function (the optimal one) for it. In each of the subsequent sections we derive different systolic implementations for the recurrence by choosing appropriate allocation functions. Each choice of allocation function also affects the different "control planes" and hence generates the special features of the final array. Finally we conclude by comparing these architectures in terms of processor utilization, I/O complexity, etc. and indicate a few directions for further work.

2 Overview of the Synthesis Methodology

In a typical scenario for synthesizing systolic arrays one starts with an initial algorithm defined by the computation of a function at all points in an *index-space* (*viz* the integer lattice points in a subset of Euclidean space). An example of such an algorithm is a simple (loop) program. The range of values over which the loop indices are permitted to vary define exactly the index-space (which is n-dimensional if there are n nested loops); the body of the loop represents the function which is computed at all the points in this domain, and thus serves to define the granularity of the processor. The synthesis problem then reduces to mapping the original index-space to a *space-time* domain (i.e., assigning to each point in the original domain a *place* and a *time*). The two parts of the mapping function are referred to as the *allocation function* (for the space component) and *timing function* (time component) respectively. Such a mapping must satisfy certain constraints as described below.

- The mapping must be bijective i.e., two distinct points in the index-space should be mapped to two distinct points in the space-time domain. If it were not so, the computations from two distinct points in the problem domain would be scheduled on the same processor at the same time.
- The data dependencies of the original algorithm must be rendered spatially and temporally local since systolic arrays have nearest neighbor interconnections (spatial locality) and a finite memory in each processor (which mandates temporal locality since the value used by any processor must have been produced by its neighbor only a finite number of "clock-ticks" ago).
- These transformed dependencies must be *uniform* over the whole space-time domain, since the processors in a systolic array are *identical* and have similar interconnections independent of their physical location in the array. The importance of this requirement has been demonstrated elsewhere [14].

Under this basic framework it is mathematically advantageous to view the initial algorithm as a *recurrence equation* i.e., an equation defined as

$$f(p) = g(f(q_1), f(q_2), \ldots, f(q_k))$$

where g represents the loop body and p and $q_1 \ldots q_k$ are n-dimensional points. The domain is defined separately and is usually a convex hull. In much of the earlier work, the recurrence was restrited to be a *Uniform Recurrence Equation* (URE). In UREs the difference $p - q_i$ between a point p and any

other point q_i on which it depends, was required to be a *constant vector* independent of p.[1] It is easy to show that if *affine transformations* are applied to such an index-space, the transformation yields another index-space that also has uniform dependencies. This means that the third constraint mentioned above can be easily satisfied by appropriate *affine* timing and allocation functions. Hence, by choosing appropriate affine timing and allocation functions, it can be guaranteed that the processors will have identical interconnections. A number of earlier researchers have developed necessary and sufficient conditions for the existence of such timing and allocation functions . They also showed that the problem of determining such functions reduces to a constraint optimization problem with a *constant* number of constraints.

We have argued elsewhere [16] that requiring the dependencies to be uniform at the *initial specification* level is unduly restrictive and proposed a more generalized class of recurrences called *Affine Recurrence Equations* (AREs). In AREs each of the q_i's are *affine* functions of p, i.e., $q_i = \mathrm{A}_i p + \mathrm{b}_i$ where A_i is an $n \times n$ matrix and b_i is an $n \times 1$ vector.[2] In synthesizing systolic arrays from AREs [14,16] we focussed our attention on affine timing and allocation functions. We have shown that under such transformations the dependencies themselves remained affine (and hence did not represent a systolic array). It was therefore necessary to *explicitly pipeline* the dependencies to render them uniform. A similar idea was discussed by Fortes and Moldovan [6].

The basic idea behind pipelining is as follows. In an ARE, the computation of f at any point p requires the value of $f(q)$ as one of its arguments (where $q = \mathrm{A}p + \mathrm{b}$). Clearly the difference $p - q$ is not a constant vector, but depends on p. To render the dependency uniform let us first try to find another point p' that also requires the value of $f(q)$. We want p' to be "close to" p in the sense that $p' - p$ is a constant vector independent of p (say ρ). If such a point exists, and somehow it can be correctly scheduled (i.e., all its arguments can be made available at p') then we may also schedule p as follows. Instead of getting its argument $f(q)$ from the point q it gets it from p' (which is just a constant vector away and which has just used the value in its own computation). To do this we introduce an additional *pipelining function* $f'(p)$ at all points in the domain. This function merely passes its arguments on unchanged. Then the value of $f(q)$ is available at point p from p', i.e., $p + \rho$ (which in turn gets it from $p + 2\rho$ and so on). Carrying this argument to its inductive conclusion, the pipelining function has the following form.

$$f'(p) = \begin{cases} f(p) & \text{if } \pi p = \theta \\ f'(p + \rho) & \text{otherwise} \end{cases}$$

[1] This means that if one has say, a three dimensional index-space $[i, j, k]$, then all assignment statements in the loop body *must* have the form `u[i,j,k] := g(.. u[i+a,j+b,k+c] ..)` where a, b and c are integer constants.

[2] This means that our loop body can contain statements such as `u[i,j,k] := g(.. u[i',j',k'] ..)` where *each* of i' j' and k' are *affine* functions of i, j and k.

The expression $\pi p = \theta$ is a "termination condition" to test if the point p is on a domain boundary (the basis case for the inductive argument above), and the new (uniform) dependency that has been introduced is ρ. It has been shown that ρ corresponds to the null space of the matrix A. The pipelining function f' does not impose any additional "computational" requirements on the processor since at the domain boundary its value is simply the same as f, and at internal points it is the same as f' at a neighboring point. Once the dependencies have been made uniform we may use the earlier techniques for determining appropriate timing and allocation functions. As we shall see later, the control signals and the specialized processor behavior are determined precisely by the linear conditions like $\pi p = \theta$ that are introduced during the pipelining (and to any conditional expressions present in the original specification).

3 Initial Specification of LU Decomposition

LU-Decomposition of a square matrix A is the problem of determining two matrices L and U such that A = L · U and L is lower-triangular (with diagonal elements equal to 1) and U is upper-triangular. This corresponds to solving n^2 equations (one for each element in the A matrix) for n^2 unknowns, and by simple algebraic manipulation (omited here for brevity) the computation can be defined by the following recurrence.

$$\begin{aligned} l_{i,j} &= f(i,j,j) \text{ for } i > j \\ u_{i,j} &= f(i,j,i-1) \text{ for } i \leq j \end{aligned}$$

where $f(i,j,0)$ is $a_{i,j}$ and

$$f(i,j,k) = \begin{cases} f(i,j,k-1) & \text{if } i = k \\ f(i,j,k-1)/f(k,j,k-1) & \text{if } j = k \\ f(i,j,k-1) - f(i,k,k) * f(k,j,k-1) & \text{otherwise} \end{cases} \quad (1)$$

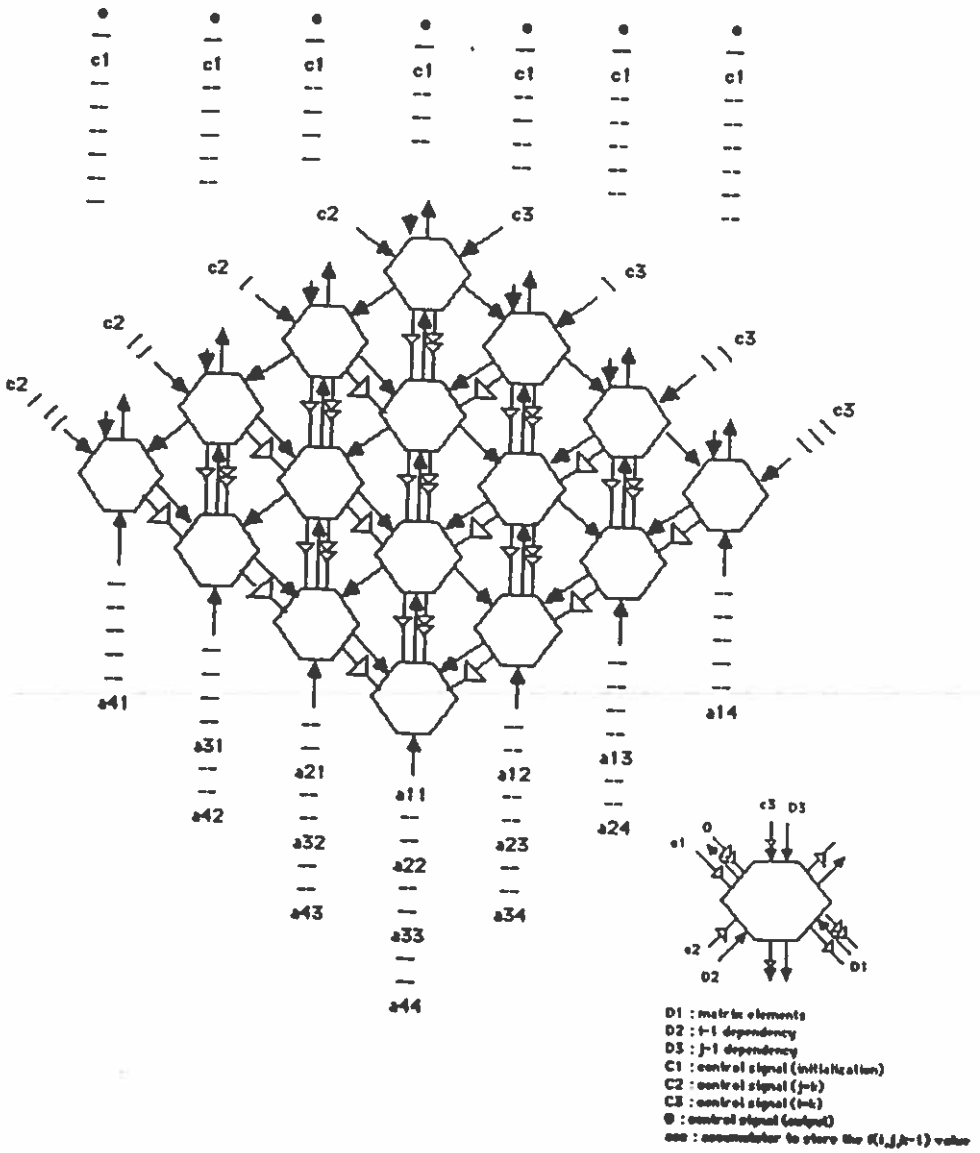
The domain for this recurrence (see Figure 1) is a pyramid with [1,1,1], [1,n,1], [n,1,1] and [n,n,1] as its base (ABCD) and [n,n,n] as its vertex, E. The input values $a_{i,j}$ enter the computation at the base ABCD and the outputs are available at the two diagonal faces ABE (corresponding to the plane $i = k$ where the $u_{i,j}$ values are computed) and ADE (except for points on the line AE) corresponding to the $j = k$ plane where $l_{i,j}$ are computed. Note that $l_{i,j}$ (*viz.* the value of $f(i,j,k)$ when $j = k$) is computed by using a division operation (the second line in Eqn 1) while the computation at all other points in the domain involve a simple multiply-subtract operation.

[13] Quinton, P. *The Systematic Design of Systolic Arrays.* Tech. Rep. 216, Institut National de Recherche en Informatique et en Automatique INRIA, July 1983.

[14] Rajopadhye, S.V. *Synthesis, Otimization and Verification of Systolic Architectures.* PhD thesis, University of Utah, Salt Lake City, Utah 84112, December 1986.

[15] Rajopadhye, S.V. and Fujimoto, R.M. Systolic array synthesis by static analysis of program dependencies. In *Proceedings, Parallel Architectures and Languages, Europe*, Springer Verlag LNCS No 258, Eindhoven, the Netherlands, June 1987.

[16] Rajopadhye, S.V., Purushothaman, S., and Fujimoto, R.M. On synthesizing systolic arrays from recurrence equations with linear dependencies. In *Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer Verlag, LNCS No 241, New Delhi, India, December 1986.



Left Half Processors

```
if C2=1 then
   begin
        acc := acc / D-3
        D2-out := acc
   end
   else begin
        acc := acc - D2 * D3
        D2-out := D2
   end
D3-out := D3
C2-out := C2
O-out := O
```

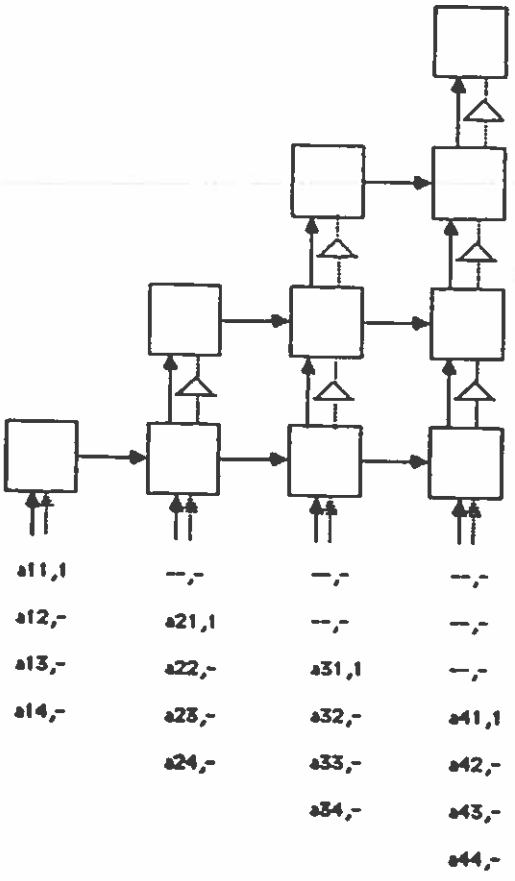
Central Processors

```
if C2=1 then
   begin
        D3-out := acc
        D2-out := 1
   end
   else begin
        acc := acc - D2 * D3
        D3-out := D3
        D2-out := D2
   end
O-out := O
```

Right Half Processors

```
if C3=1 then
        D3-out := acc
   else begin
        acc := acc - D2 * D3
        D3-out := D3
   end
        D2-out := D2
C3-out := C3
O-out := O
```

Figure 3: Architecture 2: $a(i,j,k) = [i,j]$



Diagonal Processors

```
horiz-out := vert-in
vert-out := vert-in
```

Interior Processors

```
if c=1 then
begin
   acc := vert-in / horiz-in
   horiz-out := horiz-in
   vert-out := acc
end
else begin
   vert-out := vert-in -
                horiz-in * acc
   horiz-out := horiz-in
end
```

Figure 4: Architecture 3: $a(i,j,k) = [i,k]$