The Algebraic Path Problem Revisited

Sanjay Rajopadhye[1], Claude Tadonki[2], and Tanguy Risset[1]

[1] IRISA, Rennes, France, http://www.irisa.fr/cosi/Rajopadhye
[2] University of Yaounde, Cameroon, cmtado@uycdc.uninet.cm

Abstract. We derive an efficient linear SIMD architecture for the algebraic path problem (APP). For a graph with n nodes, our array has n processors, each with $3n$ memory cells, and computes the result in $3n^2 - 2n$ steps. Our array is ideally suited for VLSI, since the controls is simple and the memory can be implemented as fifos. I/O is straightforward, since the array is linear. It can be trivially adapted to run in multiple passes, and moreover, this version *improves* the work efficiency. For any constant α, the running time on $\frac{n}{\alpha}$ processors is no more than $(\alpha+2)n^2$. The work is no more than $(1+\frac{2}{\alpha})n^3$ and can be made as close to n^3 as desired by increasing α.
Keywords: transitive closure, shortest path, matrix inversion, Warshall-Floyd & Gauss-Jordan elimination, systolic synthesis, scheduling, space-time mapping, recurrence equations.

1 Introduction

The algebraic path problem (APP) unifies a number of well-known problems into a single algorithm schema. It may be stated as follows. We are given a weighted graph $G = \langle V, E, w\rangle$ with vertices $V = \{1, 2, \ldots n\}$, edges $E \subseteq V \times V$ and a weight function $w : E \to S$, where S is a closed semiring $\langle S, \oplus, \otimes, *, \mathbf{0}, \mathbf{1}\rangle$ (closed in the sense that $*$ is a unary "closure" operator, defined as the infinite sum $x* = x \oplus (x \otimes x) \oplus (x \otimes x \otimes x) \oplus \ldots$. A path in G is a (possibly infinite) sequence of nodes $p = v_1 \ldots v_k$ and the weight of a path is defined as the product $w(p) = w(v_1, v_2) \otimes w(v_2, v_3) \otimes \ldots \otimes w(v_k, v_{k-2})$. Let $P(i, j)$ denote the (possibly infinite) set of all paths from i to j. The APP is the problem of computing, for all pairs i, j, such that $0 < i, j \leq n$, the value $d(i, j)$ defined as follows ($\bigoplus$ is a "summation" operator for $\mathcal{S}$)

$$d(i,j) = \bigoplus_{p \in P(i,j)} w(p)$$

Warshall's transitive closure (TC) algorithm, Floyd's shortest path (SP) algorithm and the Gauss-Jordan matrix inversion (MI) algorithm are but instances of a single, generic algorithm for the APP (henceforth called the WFGJ algorithm), the only difference being the semiring involved. It's sequential complexity (and hence the work) is n^3 semiring operations.

There has been considerable research on implementing the APP (or particular instances thereof) on systolic arrays (see Table 1). Most of the early work was

Authors	Application	Area	Time
Guibas et al. [5]	TC	n^2	$6n$
Nash-Hansen [9]	MI	$3n^2/2$	$5n$
Robert-Tchuenté [13]	MI	n^2	$5n$
Kung-Lo [6]	TC	n^2	$7n$
Rote [15]	APP	n^2	$7n$
Robert-Trystam [14]	APP	n^2	$5n$
Kung-Lo-Lewis [7]	TC & SP	n^2	$5n$
Benaini et al. [1]	APP	$n^2/2$	$5n$
Benaini-Robert [2]	APP	$n^2/3$	$5n$
Scheiman-Cappello [16]	APP	$n^2/3$	$5n$
Clauss-Mongenet-Perrin [4]	APP	$n^2/3$	$5n$
Takaoka-Umehara [17]	SP	n^2	$4n$
Rajopadhye [11]	APP	n^2	$4n$
Risset-Robert [12]	APP	n^2	$4n$
Chang-Tsay [3]	APP	n^2	$4n$
Djamegni et al. [18]	APP	$n^2/3$	$4n$
Linear arrays (systolic and otherwise)			
Kumar-Tsay [10]	APP	n^2	$7n^2$
Kumar-Tsay [10]	APP	n^2	$7n^2$
Myoupo-Fabret [8]	APP	n^2/α	$(3+\frac{4}{\alpha})n^2$

Table 1. Systolic implementations for the WFGJ algorithm and its instances (SP, TC and MI respectively denote shortest path, transitive closure and matrix inversion). The table is not exhaustive but in rough chronological order.

ad hoc, while later work used systematic design methods. All the implementations take $\Theta(n)$ time on $\Theta(n^2)$ processors. In the early 90's a new localization that reduced the running time from $5n$ to $4n$ was proposed independently by a number of authors [3, 11, 12, 17]. The techniques of Scheiman-Cappello and Benaini-Robert [2, 16] can be used to reduce the number of processors in these fast arrays to $n^2/3$, as shown by Djamegni et al. [18]. However, all the arrays to date sacrifice work efficiency by a constant factor.

In practice this is an important limitation. A direct implementation of any of these architectures by partitioning to run on a dedicated VLSI array will not improve the work complexity.

In this paper, we seek an SIMD VLSI architecture that does not suffer from this problem. We formally derive an architecture with only $\Theta(n)$ processors and $\Theta(n^2)$ time and also improve the work efficiency. Specifically, the array has n processors, and computes the result in $3n^2 - 2n$ time steps. The array can be easily adapted to run in multiple passes on $p < n$ processors, without sacrificing but rather, with a *gain* in work efficiency. With α passes (on $\frac{n}{\alpha}$ processors) the running time is no more than $n^2(\alpha + 2)$, and hence, the work is at most $(1+\frac{2}{\alpha})n^3$. Thus by increasing the number of passes we can approach as close to a work optimal implementation as desired. Finally, on a *fixed* number of processors (i.e., where $\alpha = n/p$ is not *constant* but proportional to n) our implementation is work optimal.

Our architecture is not "purely" systolic because there are three shift registers of length n in each processor. Note however, that since a shift register is simply a linear array of registers, one can argue that our architecture is really a two dimensional $n \times n$ systolic array where only the boundary processors do the actual computation—internal "processors" are just registers. Pragmatically, the array is very modular and ideally suited for VLSI implementation.

The remainder of this paper is organized as follows. The following section describes the notation and necessary background. Section 3 presents and intuitive description of our linear SIMD array. Next, Sect. 4 presents its formal derivation and analysis of its performance. We conclude in Sect. 5.

2 Notation and Background

We now recall the Warshall-Floyd-Gauss-Jordan algorithm for the APP and describe some of its important properties. The algorithm is based on the following SRE, where $D_0 = \{i, j, k \mid 0 < i, j \leq n; 0 \leq k \leq n\}$ is the domain of F.

$$d(i,j) = \{i,j \mid 0 < i,j \leq n\} : F(i,j,n) \tag{1}$$

$$F(i,j,k) = \begin{cases} D_0 \cap \{i,j,k \mid k = 0\} & : a_{i,j} \\ D_0 \cap \{i,j,k \mid i = j = k\} & : F(i,j,k-1)* \\ D_0 \cap \{i,j,k \mid i = k \neq j\} & : F(k,k,k) \otimes F(i,j,k-1) \\ D_0 \cap \{i,j,k \mid j = k \neq i\} & : F(i,j,k-1) \otimes F(k.k.k) \\ D_0 \cap \{i,j,k \mid i \neq k; j \neq k\} & : F(i,j,k-1) \oplus \\ & \quad (F(i,k,k) \otimes F(k,j,k-1)) \end{cases} \tag{2}$$

Apart from the initialization plane, $k = 0$ the domain of F is an $n \times n \times n$ cube. For a given value of k, we will call the point $[k, k, k]$ on the line $i = j = k$ as the *pivot*, and points on the respective planes $i = k \neq j$ and $j = k \neq i$ (i.e., points of the form $[k, j, k]$ and $[i, k, k]$, respectively) as the *pivot row* and *pivot column*. The remainder of the points are called the *interior* points. Observe that the four main (apart from the initialization) clauses of (2) correspond to the pivot, the pivot row, the pivot column and the interior points, respectively. We also call the point $[k+1, k+1, k]$ for $j \neq k$, as the *first interior point* of the k-th plane, and the points $[k+1, j, k]$ as the *first interior row*.

Observe that for any plane $k =$ const, we have the following computation order (assuming an unbounded number of processors). First, the pivot $[k, k, k]$ is computed, since it depends on a value from the "preceding" plane. Next, the rest of the pivot column, namely the points $[i, k, k]$ for $i \neq k$ (respectively, the pivot row—points $[k, j, k]$ for $j \neq k$) are computed, since they depend directly on the pivot. Finally, the interior points are computed since they all depend directly on the pivot column. Also note that the pivot of plane $k+1$ depends directly on the first interior point, and hence no computation in the next plane can start before it. In other words, there is a critical path of length three between two successive pivots, namely $[k,k,k] \to [k+1,k,k] \to [k+1,k+1,k] \to [k+1,k+1,k+1]$. Hence the fastest running time of the algorithm is $3n$. Indeed, it is well known [11] that the optimal parallel schedule for the SRE is as follows.

$$t_f(i,j,k) = \begin{cases} D_0 \cap \{i,j,k \mid k=0\} & : 0 \\ D_0 \cap \{i,j,k \mid i=j=k\} & : 3k-2 \\ D_0 \cap \{i,j,k \mid i=k\neq j\} & : 3k-1 \\ D_0 \cap \{i,j,k \mid j=k\neq i\} & : 3k-1 \\ D_0 \cap \{i,j,k \mid i\neq k; j\neq k\} & : 3k \end{cases} \tag{3}$$

This schedule assumes an unbounded number of processors, $\Theta(n^2)$, to be precise, with global communication: in particular, broadcasts of the pivot (and the pivot column) are necessary. Table 1 illustrates that the locality of communication required for a systolic implementation imposes a slowdown from $3n$ to $4n$ or more.

Before we proceed further, let us derive a modified version of the SRE (1-2), which we shall use in the remainder of this paper. For notational convenience, we define the operator $\dot{+}$ by $i\dot{+}j = (i+j) \bmod n$. Now we apply the transformation $(i,j,k \to (i-k) \bmod n, (j-k) \bmod n, k)$ to all points in the (sub) domain $\{i,j,k \mid 0 < i,j,k \leq n\}$ of D_0. By applying the standard transformation rules for recurrence equations, we obtain the following SRE, where the new domain of F is $D_1 = \{i,j,k \mid 0 \leq i,j < n; 0 < k \leq n\} \cup \{i,j,k \mid k = 0 < i,j \leq n\}$.

$$d(i,j) = \begin{cases} \{i,j \mid 0 < i,j < n\} & : F(i,j,n) \\ \{i,j \mid 0 < i < n; j = n\} & : F(i,0,n) \\ \{i,j \mid i = n; 0 < j < n\} & : F(0,j,n) \\ \{i,j \mid i = j = n\} & : F(0,0,n) \end{cases} \tag{4}$$

$$F(i,j,k) = \begin{cases} D_1 \cap \{i,j,k \mid k=0\} & : a_{i,j} \\ D_1 \cap \{i,j,k \mid i=j=0\} & : F(i\dot{+}1,j\dot{+}1,k-1)* \\ D_1 \cap \{i,j,k \mid i=0<j\} & : F(i\dot{+}1,j\dot{+}1,k-1) \otimes F(0,0,k) \\ D_1 \cap \{i,j,k \mid i>0=j\} & : F(0,0,k) \otimes F(i\dot{+}1,j\dot{+}1,k-1) \\ D_1 \cap \{i,j,k \mid i,j>0\} & : F(i\dot{+}1,j\dot{+}1,k-1) \oplus \\ & \quad (F(i,0,k) \otimes F(1,j\dot{+}1,k-1)) \end{cases} \tag{5}$$

What we have just applied is the well known reindexation transformation of Kung Lo and Lewis [7]. It effectively brings the pivot to the vertical line $i=j=0$ and the pivot rows and columns to the sides ($i=0$ and $j=0$) of the domain D_1. It may be visualized as a toroidal shift of each plane relative to the previous (this also explains why the old $F(i,j,k-1)$ argument is systematically replaced by $F(i\dot{+}1,j\dot{+}1,k-1)$).

3 A Linear Systolic Array: Intuitive Description

We now present the main intuition behind our array. Specifically, we describe the order in which computations are performed by each processor (the local schedule) without entering into details of whether and why it is valid.

In our architecture, the k-th plane is executed by processor k. Thus each processor computes a $n \times n$ matrix as defined by the SRE (4-5). However, the order in which the processor computes its values is rather special. The elements are computed "principal submatrix" by "principal submatrix" as follows.

- The pivot element $F(0,0,k)$ is computed first. This is the base case.
- After the $(i-1)$-th principal submatrix has been computed, the i-th row and column of the i-th principal submatrix are calculated in an alternating manner: $F(i,0,k)$, $F(0,i,k)$, $F(i,1,k)$, $F(1,i,k)$, $\ldots F(i,i-1,k)$, $F(i-1,i,k)$. Finally the i-th diagonal element, $F(i,i,k)$ is computed. This completes the computation of the i-th principal submatrix.

We emphasize that this specifies the *relative* order of computations performed on a given processor, and that there may be intervening idle cycles. Figure 1-a illustrates this schedule for the first processor (for which there are no idle cycles).

Also note that this is merely the order in which the elements of the k-th slice are computed. They are not ***sent*** to the next processor in this order (this is what causes idle cycles). Rather, the processor first performs a toroidal shift and then sends the elements (of this shifted matrix) in the same "principal submatrix" order described above. The reason for this is the $(i\dot{+}1, j\dot{+}1, k-1)$ dependency—every computation depends (at least) on the element that would be "just below it" if toroidally shifted. Thus the first interior point is the first value sent, and the pivot is the *last*. In the penultimate $2n-2$ steps, the pivot row and column are sent.

Let us now look at the implications of this on the schedule of processor 2 (Fig. 1.b). Since the first element that it can compute is its own pivot, and this depends on $F(1,1,1)$ which is itself computed at $t=4$, processor 2 starts only at $t=5$. However, for the next two steps, it is idle, because (i) the values computed by processor 1 are not sent, and moreover (ii) the next computation that processor 2 can perform is $F(1,0,2)$, which depends on $F(2,1,1)$, and that value is computed only at $t=7$. From here on, the 2×2 principal submatrix is finished as per the above schedule. At the start of the third principal submatrix, there are again two idle cycles. This repeats at the start of each row (as indicated by the 2 ■ in Fig. 1.b). However, there are no idle cycles at the start of the *last* row because this corresponds to the previous processor's pivot row and column, which were computed well in advance.

A close look at the schedule of processor 3 (Fig. 1.c) will now make the pattern clear. We see that there are now two additional idle cycles at the start of each row (processor 2 imposes and additional latency of two cycles on each row). However, we also see that in the penultimate row, the processor can now catch up (there are only 2 idle cycles instead of 4). This is because there were no idle cycles in the last row of the *previous* processor. As always, the schedule in the order of the "principal submatrix" and every point $F(i,j,k)$ is computed immediately after the point $F(i\dot{+}1, j\dot{+}1, k-1)$ is sent by the previous processor.

We leave blank the table for processor 4 (Fig. 1.d), as an exercise for the reader. Please fill it up before proceeding further, study it carefully, and in particular, try to determine a closed form formula that gives the value of the

1	3	6	11	18	27	38	51
2	4	8	13	20	29	40	53
5	7	9	15	22	31	42	55
10	12	14	16	24	33	44	57
17	19	21	23	25	35	46	59
26	28	30	32	34	36	48	61
37	39	41	43	45	47	49	63
50	52	54	56	58	60	62	64

(a) $k = 1$

	5	9	14	21	30	41	54	67
2 ■	8	10	16	23	32	43	56	69
2 ■	13	15	17	25	34	45	58	71
2 ■	20	22	24	26	36	47	60	73
2 ■	29	31	33	35	37	49	62	75
2 ■	40	42	44	46	48	50	64	77
2 ■	53	55	57	59	61	63	65	79
	66	68	70	72	74	76	78	80

(b) $k = 2$

	11	17	24	33	44	57	70	83
4 ■	16	18	26	35	46	59	72	85
4 ■	23	25	27	37	48	61	74	87
4 ■	32	34	36	38	50	63	76	89
4 ■	43	45	47	49	51	65	78	91
4 ■	56	58	60	62	64	66	80	93
2 ■	69	71	73	75	77	79	81	95
	82	84	86	88	90	92	94	96

(c) $k = 3$

(d) $k = 4$

	71	85	98	111	124	137	150	163
12 ■	84	86	100	113	126	139	152	165
10 ■	97	99	101	115	128	141	154	167
8 ■	110	112	114	116	130	143	156	169
6 ■	123	125	127	129	131	145	158	171
4 ■	136	138	140	142	144	146	160	173
2 ■	149	151	153	155	157	159	161	175
	162	164	166	168	170	172	174	176

(d) $k = 8$

Fig. 1. Illustration of the schedule for $n = 8$. Each table shows the time instants at which the elements of $F(i, j, k)$ are computed by processor k (for $k = 1 \ldots 4$, and $k = 8$ the final step). The x ■'s denote x idle ticks during which the processor does nothing. The table for $k = 4$ is left blank as an exercise for the reader.

time instant as a function of i, j and k (hint: first concentrate only on points on the diagonals of each plane, and remember that i and j range from 0 to $n - 1$).

The following remark can now be easily (after the little exercise) verified.

Remark 1. The closed form of the function described in Fig. 1 is given as follows.

$$t(i, j, k) = \begin{cases} \text{if } k = 0 & \text{then} \quad 0 \\ \text{if } i \geq j & \text{then} \quad t_d(i, k) - 2(i - j) \\ \text{if } i < j & \text{then} \quad t_d(j, k) - 2(j - i) + 1 \end{cases} \tag{6}$$

where

$$t_d(i, k) = \begin{cases} \text{if } i + k \geq n & \text{then} \quad n^2 + 2n(i + k - n) + n - i - 1 \\ \text{if } i + k \leq n & \text{then} \quad (i + k)^2 + k - 1 \end{cases} \tag{7}$$

Observe that $t_d(i, k)$ is the value of $t(i, j, k)$ on the diagonal (i.e., where $i = j$).

4 Formal Derivation of the Array

We have so far seen the processor allocation and an intuitive explanation of the schedule of our architecture. However, we do not have a proof of correctness of its validity. We now resolve this question.

Theorem 1. *The function $t(i,j,k)$ defined by (6-7) is a valid schedule for the variable F in the* SRE *(4-5).*

Outline of Proof: We need to show that whenever a point $(i,j,k) \in D_1$ depends on another $(i',j',k') \in D_1$, then $t(i,j,k) > t(i',j',k')$. Since the dependency function (and even the number of dependencies) is different in different subdomains of D_1, the proof must follow the structure of SRE (4-5). In fact, the schedule is valid iff we show that the (boolean) recurrence (8) can be reduced to a tautology. By substituting from (6) this reduces to showing a number of implications, each of the form that a set of affine constraints (bounds of a subdomain) imply that a polynomial is strictly positive. This is tedious but simple. Indeed, it can even be done automatically with a mechanical theorem prover. ∎

$$X(i,j,k) = \begin{cases} D_1 \cap \{i,j,k \mid i=j=0\} : t(i,j,k) > t(i\dot{+}1, j\dot{+}1, k-1) \\ D_1 \cap \{i,j,k \mid i=0<j\} : t(i,j,k) > \max(\\ \qquad t(i\dot{+}1, j\dot{+}1, k-1), t(0,0,k)) \\ D_1 \cap \{i,j,k \mid i>0=j\} : t(i,j,k) > \max(\\ \qquad t(0,0,k), t(i\dot{+}1, j\dot{+}1, k-1)) \\ D_1 \cap \{i,j,k \mid i,j>0\} \quad : t(i,j,k) > \max(t(i\dot{+}1, j\dot{+}1, k-1), \\ \qquad t(i,0,k), t(1,j,k-1)) \end{cases} \tag{8}$$

Recall that the processor allocation function is defined as follows.

$$\forall (i,j,k) \in D_1, \quad p(i,j,k) = k \tag{9}$$

Now, we must show that the schedule and allocation function are not in conflict, i.e., no two distinct points in D_1 are mapped the the same processor at the same time. The proof follows directly from Remark 2 below, which itself can be easily verified.

Remark 2. $t(i,j,k) = t(i',j',k) \Rightarrow i = i'$ and $j = j'$.

Remark 3. The running time of the array is given by $t(n-1,n-1,n) = 3n^2 - 2n$ and hence its work efficiency is $\frac{1}{3}$.

The algorithm can be implemented on only $p < n$ processors by using multiple passes. However, the analysis of the running time is a little involved. In the single pass array, the k-th processor starts the first pass at $t_s(k) = t(0,0,k) = k^2 + k - 1$ and is active until $t_f(k) = t(n-1, n-1, k) = n^2 + 2n(k-1)$. It is thus active for precisely $t_a(k) = t_f(k) - t_s(k) + 1 = n^2 + 2n(k-1) - (k^2+k) + 2$, and since $n \geq k$ this *increases* with k. Indeed, this should be obvious since each processor has exactly n^2 computations to perform, but as k increases, the processor has more and more idle cycles. Hence if we start the next pass as soon as the *first* processor is free (i.e., at $t = n^2 + 1$), there will be conflicts.

Ideally, we desire that the last processor, p, start its next pass exactly at $t_f(p) + 1$ so that there will be no wasted time on this processor. Since there are exactly $t_s(p)$ cycles from the start of a pass (on the first processor) to the time at which the last processor starts the pass, we can achieve our goal if the *first* processor starts its second pass at $t_f(p)+1-t_s(p)$, i.e., at $t_a(p)+1$. Since the first processor is active on the first pass for the first n^2 of these cycles, the number of idle cycles for the first processor between the first two passes is $2n(p-1)+2$, which is of the same order as the total memory (registers) in the array.

The argument holds between any two passes, and since there are $\frac{n}{p}$ passes, we have the following.

Theorem 2. *The total running time on p processors is given by*

$$T_p = \frac{n^3}{p} + 2n^2\frac{p-1}{p} - (n-p)(p+1) + \frac{2n}{p} - 2 \tag{10}$$

Corollary 1. *For any (positive integer) constant α, the total running time on $p = \frac{n}{\alpha}$ processors is as given below, assuming that $n \approx n + \alpha \approx n - \alpha$, and ignoring constant terms.*

$$T_p \approx n^2\left(\alpha + 2 - \frac{\alpha-1}{\alpha^2}\right) \leq n^2(\alpha+2) \tag{11}$$

The work of the multiple pass array on $\frac{n}{\alpha}$ processors is $n^3(1+\frac{2}{\alpha}-\frac{\alpha-1}{\alpha^3})$. Hence, we can improve the work efficiency simply by increasing α, i.e., by sacrificing running time by a constant factor with a corresponding decrease in the number of processors. It is important to note that the savings arise because the first processor is never idle *during* any pass (see Fig. 2), though it emulates a processor which would have idle cycles in the single pass array. A formal proof of correcness of the multiple pass implementation is omited due to space constraints.

The above analysis assumes that α is a constant, i.e., we always use $\Theta(n)$ processors, scaling the architecture with the problem size. On the other hand, it is also clear from (10) that with only a fixed number of processors, we have a work optimal implementation, with a running time dominated by the $\frac{n^3}{p}$ term.

5 Conclusion

We have presented an SIMD architecture for the APP, which unifies unifies many well-known problems into a single algorithm schema. Unlike most previous work which proposed arrays with $\Theta(n^2)$ processors and $\Theta(n)$ time, all entailing a loss of work optimality, our architecture is a linear array of n processors and a running time of $3n^2$ (thus performing three times the work of the sequential algorithm). However, a simple multipass version of the array on $\frac{n}{p}$ processors achieves a

Fig. 2. Illustration of the "space-time" behavior of two arrays (for $n = 32$), one for a single pass which finishes at $t = 3008$, and one with 8 processors performing 4 identical passes finishing at $T = 5678$. A processor is active during the solid line and idle during the gaps. Observe how the first processor has no idle time *during* any pass.

running time that is as close to a work optimal as desired. The architecture is derived using the well known systolic synthesis methods but with extensions involving polynomial schedules. Each processor in the array has 3 fifos of n-registers, and the architecture is well suited for direct VLSI implementation. We also note that a simple extension is a "block" version of the architecture, which would be more suitable for implementation on a general purpose parallel machine. For such an implementation, and interesting problem is the choice of the block size that optimizes the total running time, a problem we treat elsewhere.

Acknowledgments C. Tadonki was partially supported by the *Microprocessors and Informatics* Program of the United Nations University, Tokyo, Japan, and the French agency *Aire Développement* through the project *Calcul Parallèle*.

References

1. A. Benaini, P. Quinton, Y. Robert, Y. Saouter, and B. Tourancheau. Synthesis of a new systolic architecture for the algebraic path problem. *Science of Computer Programming*, 15:135–158, 1990.
2. A. Benaini and Y. Robert. Space-time minimal systolic arrays for gaussian elimination and the algebraic path problem. In *ASAP 90: International Conference on Application Specific Array Processors*, pages 746–757, Princeton, NJ, September 1990. IEEE Press.
3. P. Y. Chang and J. C. Tsay. A family of efficient regular arrays for the algebraic path problem. *IEEE Transactions on Computers*, 43(7):769–777, July 1994.

4. P. Clauss, C. Mongenet, and G-R. Perrin. Synthesis of size-optimal toroidal arrays for the algebraic path problem: A new contribution. *Parallel Computing*, 18:185–194, 1992.
5. L. Guibas, H. T. Kung, and Clark D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Proc. Conference on Very Large Scale Integration: Architecture, Design and Fabrication*, pages 509–525, January 1979.
6. S. Y. Kung and S. C. Lo. A spiral systolic algorithm/architecture for transitive closure problems. In *ICCD 85: International Conference on Circuit Design*, pages 622–626, Rye Town, NY, 1985. IEEE.
7. S. Y. Kung, S. C. Lo, and P. S. Lewis. An optimal systolic design for the transitive closure and the shortest path problems. *IEEE Transactions on Computers*, C-36(5):603–614, May 1987.
8. J. F. Myoupo and C. Fabret. A modular systolic linearization of the warshall-floyd algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):449–455, may 1996.
9. J. G. Nash and S. Hansen. Modified faddeew algorithm for matrix multiplication. In *Proc., SPIE Workshop on Real-Time Signal Processing*, pages 39–46. SPIE, 1984.
10. V. K. Prasanna Kumar and Y-C Tsai. Designing linear systolic arrays. *Journal of Parrallel and Distributed Computing*, (7):441–463, may 1989.
11. S. V. Rajopadhye. An improved systolic algorithm for the algebraic path problem. *INTEGRATION: The VLSI Journal*, 14(3):279–296, Feb 1993.
12. T. Risset and Y. Robert. Synthesis of processors arrays for the algebraic path problem: Unifying old results and deriving new architectures. *Parallel Processing Letters*, 1:19–28, 1991.
13. Y. Robert and M. Tchuenté. Résolution systolique de systèmes linéaires denses. *RAIRO Modélisation et Analyse Numérique, Technique et Sciences Informatiques*, 19(2):315–326, 1985.
14. Y. Robert and D. Trystam. Systolic solution of the algebraic path problem. In W. Moore, A. McCabe, and R. Urquhart, editors, *Systolic Arrays*, 1, pages 171–180, Oxford, UK, 1987. Adam Hilger.
15. Günter Rote. A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion). *Computing*, 34(3):191–219, 1985.
16. Chris J. Schieman and Peter R. Cappello. A processor-time minimal systolic array for transitive closure. In *International Conference on Application Specific Array Processors*, pages 19–30, Princeton, NJ, September 1990. IEEE Computer Society, IEEE Computer Society Press.
17. T. Takaoka and K. Umehara. An efficient VLSI circuit for the all pairs shortest path problem. *Journal of Parallel and Distributed Computing*, 16:265–270, 1992.
18. C. Tayou Djamegni, P. Quinton, S. Rajopadhye, and T. Risset. Derivation of systolic algorithms for the algebraic path problem by recurrence transformations. *Parallel Computing*, To appear, 1999.