SYSTOLIC COMPUTATION AND VLSI

Mark R. Kramer & Jan van Leeuwen

RUU-CS-82-9
June 1982

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–531454
The Netherlands

SYSTOLIC COMPUTATION AND VLSI

Mark R. Kramer & Jan van Leeuwen

Technical Report RUU-CS-82-9
June 1982

Department of Computer Science
University of Utrecht
P.O. Box 80.002, 3508TA Utrecht
the Netherlands

SYSTOLIC COMPUTATION AND VLSI *

Mark R. Kramer & Jan van Leeuwen

Department of Computer Science, University of Utrecht
P.O. Box 80.002, 3508TA Utrecht, the Netherlands

Abstract. The notion of systolic computation is due to H.T. Kung and C.E. Leiserson. It describes the computation by means of a network of simple processing elements that rhythmically act on regular streams of data passing through the system. We explain the paradigms of systolic algorithm design through a discussion of systolic queues, stacks and trees. An integral approach is given to matching problems and matrix multiplication performed by (2-dimensional) systolic arrays. As a novel contribution we present a systolic algorithm for inverting a nonsingular $n \times n$ matrix in $O(n)$ time.

1. Introduction.

With the advent of VLSI-technology (cf. Mead and Conway [17]) it has become feasible to design circuits with tens of thousands of components and integrate them on a single chip of silicon. The large degree of parallelism that can be incorporated in circuits of this size gives VLSI-chips at least the potential of providing extremely fast and efficient computing devices. The development of systolic algorithms as initiated by Kung and Leiserson [13] can be viewed as an approach aimed at exploiting this potential through the use of modular design principles and fully parallel and pipelined data processing (streaming).

A VLSI-chip as understood here is described by its constituent components (processing elements) and their interconnections (the wiring pattern). We will simply assume that processing elements consist of a single "cell",

* Notes for a contribution to the 4th Advanced Course on Foundations of Computer Science, held June 14-25 (1982) in Amsterdam, the Netherlands.

even though they may be built of several transistors and like components To facilitate the design of the chip, we insist that the actual variety of different processing elements used be kept as small as possible. The wiring pattern needs to be simple and regular for the same reason, with only local connections of processing elements and no long wires that need more area and more energy to drive and that are invariably slower. The algorithm performed on the chip should exploit the parallel processing power of the many available cells, which includes both a "division of labor" and the use of pipelining through the circuit.

The VLSI-chips that conform to these rules are characterised by simple geometries (arrays) of cells, with the cells rhythmically acting on one or more streams of data that smoothly move across the chip. The algorithms underlying the operation of the chip have been termed "systolic" (Kung and Leiserson [13]) for the analogy with the rhythmic pulsing of blood through the arteries. Systolic algorithms and architectures have been studied extensively by H.T. Kung (see e.g [10], [11]) and several co-workers. Independently similar goals have been pursued by the research group of T. Legendi (see e.g. [14]) in the study of regular "fields" of cellular processors, a hardware oriented outgrowth of the theory of cellular automata as known from e.g. [5].

In these notes we shall illustrate the paradigms of systolic algorithm design through a number of examples that, aside from being of interest in their own right, do provide some useful special circuits for inclusion in more involved designs. In Section 2 we discuss a design for systolic priority queues due to Leiserson [15] and show how it can be used to sort n keys in $O(n)$ time and to implement ordinary queues and stacks with $O(1)$ response times. In Section 3 we discuss the "tree machine" as proposed by Bentley and Kung [3] and the systolic trees of Leiserson [15], aimed again at the implementation of a fast priority queue. We analyse some shortcomings of both designs and the solutions for it proposed by Song [21] and Ottmann, Rosenberg and Stockmeyer [18], respectively. In Section 4 we provide a uniform treatment

of pattern matching (as in Foster and Kung [8]), comparison-problems (as in Kung and Lehmann [12]) and matrix multiplication (as in Kung and Leiserson [13] and Katona [9]), which all use a similar idea of pipelining on a 1- or 2-dimensional array of cells. In Section 5 a (novel) systolic algorithm is presented to invert an $n \times n$ nonsingular matrix in $O(n)$ time. The algorithm is based on Gaussian elimination and assumes that no pivoting is required. The algorithm was proposed for implementation on parallel architectures before (Pease [19]), but the systolic version presented here shows the intricacy of pipelining to achieve greater speed. Each of the Sections demonstrates a particular way of describing the actions of a systolic circuit: operational in terms of actions of cells on their contents (Section 2), programmed for a small repertoire of instructions (Section 3), operational in terms of actions on the data streams (Section 4) and functional (Section 5, as in the framework of e.g. Katona [9]).

From an abstract point of view, systolic algorithms are not very different from multi-processor algorithms with the processors acting in fully synchronised order. Two levels of timing can be distinguished in the design of a systolic algorithm: (i) the global timing required to synchronize the cells as a network, and (ii) the local (internal) timing required for the operation within a cell. The latter is not trivial, for we shall design processing elements to perform non-elementary operations (like multiplication) on b-bit numbers. Nevertheless we shall view these operations as atomic and requiring only one "tick" of the global clock.

2. (Linear) systolic queues and stacks.

We shall primarily discuss the design of a systolic priority queue, i.e., a structure that supports INSERT/DELETE/XMIN ("extract smallest key") commands on a set that never contains more than N keys. The design consists of a linear array of cells (see figure 1) with its I/O connection to the environment left of the first cell. Every cell has two registers, A and B, each of which can contain a key of the set. If

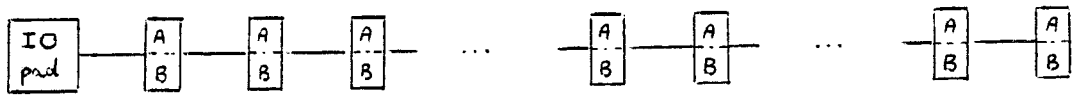


figure 1

a register carries no key, we assume it contains a default value ∞ larger than any conceivable key. Our aim is to maintain the set of $n \leq N$ keys in increasing order, with all keys in the A-registers of a contiguous initial segment of the array. The B-registers are used for transporting newly inserted keys to their proper position in the ordering. If a key got to its right place, it moves the current key out of the A-register and replaces it, while the latter takes over in 'streaming' right. It should be clear that an insertion thus has an effect that ripples on through the array, moving all subsequent keys one place up. As soon as one insertion has moved one step one can initiate a next one, for the first cell would now be idle.

A deletion would require a special signal to be sent up the array, to search for and delete a particular key. All keys to the right should subsequently move down one place, which they will appear to do one after the other and just fast enough to give instructions coming in from the left an undistorted impression of contignity of the data set. The extraction of the smallest key is a special case, with an additional instruction to send the key (always in the A-register of the first cell) left to output. For uniformity we only program the array to do the latter kind of deletion.

The cells will be timed such that the odd and even numbered cells "beat" alternately. When it acts, a cell will compare its register contents with the register contents of its neighbour to the left. The timing of the array is such that this neighbour is momentarily in-active and (thus) it is safe to inspect it. The "IO pad" (see figure 1) should act as a left neighbour when it has to. At other times it can route keys

in and out. Let A_ℓ and B_ℓ denote the register contents of the A and B registers of a left neighbour, respectively. The cells will cycle through the following program:

do
  cycle 1 : copy B_ℓ into B ; rearrange the keys in A_ℓ, A and B such that $A_\ell \le A \le B$;
  cycle 2 : rest
od.

Observe that values in a B-register indeed move right and are swapped in place when the proper place in the ordering ("before the current A-value") is reached. If A_ℓ contains ∞ (ultimately the result of a deletion or XMIN), then keys right of it will move a step down (in ripple fashion). The copying of a B_ℓ-register (in cycle 1) could, but need not destroy the value of this register. The IO-pad has an A and a B port that is set as follows, to interact properly with the first cell of the array and allow for the processing of commands:

(i) to insert a key k, A is set to $-\infty$ and B to k (which thus prepares it for moving up the B-registers),

(ii) to extract a smallest key, both A and B are set to ∞. In the next cycle (or the one after that, depending on the timing of the array) the smallest key will have moved into the A-register and can be extracted. The first cell now has a value ∞ in its A-register, which will be filled from the right immediately in the following step,

(iii) when idling, A is kept at $-\infty$ and B at ∞.

The design has no provision for signaling when the queue is "full". Overflow will result in the loss of elements at the end. The following conclusion should be evident ([15]):

Theorem A linear systolic array of size N can process INSERT/XMIN (and DELETE) commands with $O(1)$ response times, as long as the

number of keys in the set remains $\leq N$ at any moment.

The design immediately leads to a fast on-line sorter which operates in $O(n)$ time on a set of n keys. Just feed the keys into the queue and extract the smallest from it in n consecutive steps. (The method of Armstrong and Rem [2] is a special case of this for bitwise sorting.)

The principle of the systolic priory queue can be used to implement ordinary queues and stacks. A straightforward idea is to stamp elements with a natural number, which gives their ordering in the queue or stack, and to subsequently use it for a key to move the elements into the array. We do not propose that this be used but merely note that it is a convenient way of conceptualising the required data movements through the A and B registers. The <u>systolic queue</u> is like a systolic priority queue in which the elements are kept sorted in inverse order of arrival. Thus, new keys always move up through the B-registers until the first open A-register at the end of the line is reached. Deletions from the front of the queue are executed exactly like the XMIN command. We no longer need ∞ for reasons of ordering and simply use it to mean that a register is "empty". The cells of the systolic queue can abide by the following simple program (recall that the odd and even numbered cells still alternate in action):

```
do
  cycle 1 : if
              A_ℓ & B_ℓ & A → copy B_ℓ into B ;
            □ A_ℓ & B_ℓ & ¬A → copy B_ℓ into A ; set B to ∞ ;
            □ ¬A_ℓ & ¬B_ℓ & A → copy A into A_ℓ ; set A and B to ∞
            fi ;
  cycle 2 : rest
od.
```

(The if .. □ .. □ .. fi is a guarded command, which effectively acts like a <u>skip</u> if no guard happens to be satisfied. The & - notation is

a simple shorthand for "... is not empty and".) Note that, as deletions occur at the front of the queue, there will never be more than one empty A-register in between two occupied ones.

Theorem. A linear systolic array of size N can process ENQUEUE/DEQUEUE commands with $O(1)$ response times, as long as the number of elements in the set remains $\leq N$.

A systolic stack is like a systolic priority queue in which elements are kept sorted in exact order of arrival. Thus, a newly inserted key always forces the element in the A-register of the first cell to move up, which in turn forces all elements in the queue to step one position to the right. A deletion is again very similar to the XMIN command and causes all elements to do one step left. The cells of a systolic stack, again, can do with a simplified program:

```
do
  cycle 1 : if
              A_f & B_f & A → copy A into B ; copy B_f into A ;
            [] A_f & B_f & ¬A → copy B_f into A ; set B to ∞ ;
            [] ¬A_f & ¬B_f & A → copy A into A_f ; set A and B to ∞
            fi ;
  cycle 2 : rest
od.
```

Theorem. A linear systolic array of size N can process PUSH/POP commands with $\Theta(1)$ response times, as long as the number of elements in the set remains $\leq N$.

3. Systolic trees.

With the many uses of trees in search structures (cf. [1]) it seems

advantageous to connect N processing elements by a tree-based circuit rather than in a linear array. Again the cells can have several

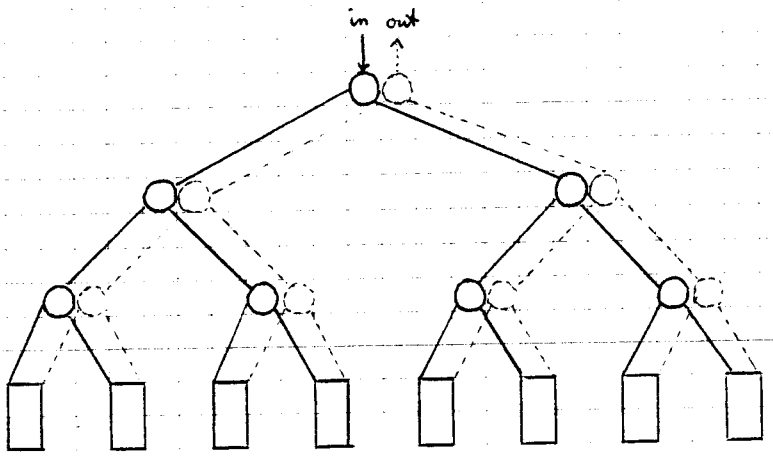


figure 2

registers, but for the moment we assume that each cell can hold at most one key. The tree-circuit (see figure 2) is best thought of as consisting of two strata: one to route information down to the cells, and another to merge information and route it upward. The ○-nodes simply split or duplicate messages in one step, the □-nodes are more complicated and can perform some function on the two incoming messages (data) from below to determine what message to pass upward. Packaging a tree-machine on a small chip is a non-trivial matter discussed at some length in Song [21] and Bhatt and Leiserson [4].

The important characteristic of the tree machine is that it allows for extremely efficient broadcasting of information to all processors simultaneously. Assuming the processors all act in one step, the result information can be merged and sent back up the tree equally fast. (One might call it "inverse broadcasting".) In the applications that follow we shall assume that the machine is pipelined. This means that there is a steady movement of wavefronts down the ○-tree and up the □-

tree, with every wavefront carrying its own information (corresponding to one command) and spanning an entire level of the tree. A distinction should be made between the compute time (i.e., the time between submitting a request and receiving the corresponding output) and the period of a command (i.e., the time between its wavefront and the next, or: the interval of time that must pass before a next command can be entered).

Metatheorem. A tree machine of size N can process any admissible command with a compute time of $O(\log N)$ and a period of $O(1)$, as long as the number of elements in the set remains $\leq N$.

The metatheorem does not always hold, but it serves as a criterion to test the suitability of the tree machine for a particular set of commands. We shall try an implementation of a simple dictionary, i.e., a structure that supports MEMBER/INSERT/DELETE commands on a set that never contains more than N keys. We assume throughout that keys are unique and (thus) that there never occur two identical keys in the tree. It means there never is an insertion of a key that is already present. (It only points to the first of several problems inherent to the systolic approach...)

A MEMBER(k) command is easy. Broadcast it to all processors, let them answer yes or no depending on the key they contain and merge the answers up the tree to check that there was at least one "yes". It takes a compute of $O(\log N)$ and a period of $O(1)$. An INSERT(k) is harder, because there is no point in broadcasting the instruction and store k in every available (free) A-register as a result! To guide the search for a free register, one could add to each O-node a counter that keeps track of the number of free registers in the cells it covers in its subtree. The INSERT command is moved in a direction with count > 0, while the counters that it passes are (of course) immediately decreased by 1, until it eventually reaches one free register where k gets stored. It complicates the tree machine tremendously, but does give an $O(\log N)$ compute time and

an $O(1)$ period. To process a DELETE(k) one would need to update the counters along the search path towards k only. A DELETE thus causes severe problems, because the search path can only be traced during the wave back up the tree, i.e., after having located k. This forces the period for DELETE commands to remain at $O(\log N)$, apparently.

Theorem. A tree machine of size N can process MEMBER/INSERT/DELETE commands with a compute time of $O(\log N)$ and a period of $O(1)$, as long as the number of elements in the set remains $\leq N$. (It is assumed that INSERTs always add new keys and that DELETEs always remove existing keys.)

Proof

The argument is due to Song [21]. Suppose the B-register of every processor is made available to hold any integer $\in 1..N$ or ∞. If a processor holds a key, then its B-register contains ∞ ("empty"). If it holds no key, then the B-register contains a value that essentially is its position in a free space list. A (global) counter F is maintained at the root of the machine to keep track of the total number of free A-registers. If F has value f, the "tickets" 1 to f are somehow distributed over (i.e., contained in the B-registers of) the now available processors. An INSERT(k) command is tagged with the current value of F and broadcasted to all processors. The value of F is rightaway decreased by 1 and the actual insertion of k only takes place in the processor whose B-register holds ticket f. (The B-register is subsequently erased.) A DELETE(k) command is tagged with the current value of F plus 1 and broadcasted down. The value of F is incremented and the processor that contains k stores the tag in its B-register. (This time the A-register is, of course, erased.) □

The problem of handling extraneous insertions/deletions (called "redundant" insertions/deletions in [18]) remains, at least for the time

being.

The counting of free processors became necessary, because we apparently lost the possibility of shifting keys left and right in the array to maintain the set in a contiguous initial segment of the processors. Define an L-machine to be a tree machine of some size N with the processors connected into a linear array (see figure 3). We

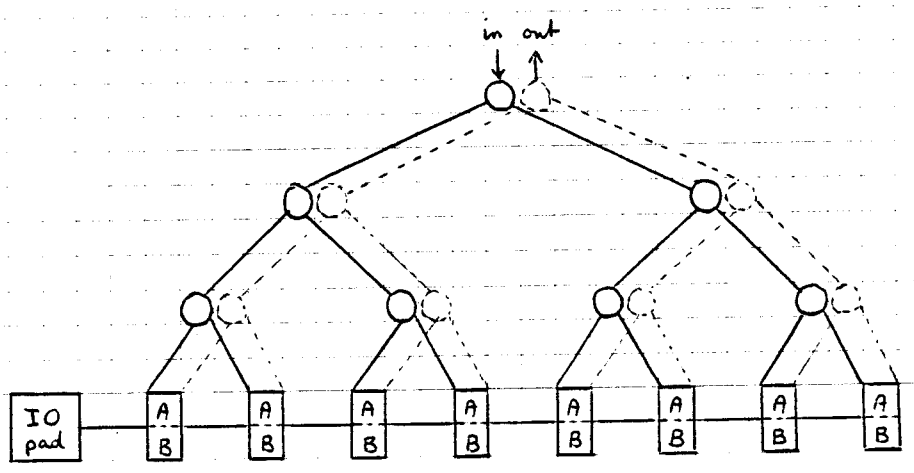


figure 3

assume as in Section 1 that an IO-pad marks the left end of the array. The L-machine retains the facility of the tree machine to broadcast instructions to all processors but combines it with the attractive feature of linear arrays to move keys among neighboring cells. We shall assume that the wavefronts activate the processors simultaneously and do not succeed one another faster than the processors need to complete their cycle. The packaging of L-machines on one or more chips is a bit harder but can be done along the same lines as for tree machines (cf. Song [21]). We shall exploit the L-machine to implement an extended dictionary structure that supports MEMBER / INSERT / DELETE / XMIN commands. We shall ignore MEMBER commands, as they are treated exactly as for tree machines.

Commands are broadcasted from the root to all processors. Depending on the instruction received the processors shift their keys right (if they recognise that an insertion took place to their left) or left (if they recognise that a deletion took place to their left or the command was an XMIN) or not at all (if they recognise there is no need for it). The processors can decide the required action by comparing k to their A-register and the A-register of their left neighbor. When timed right, the machine maintains the set of keys in increasing order in an initial segment of the array. In particular, an XMIN will automatically shift the smallest key onto the IO-pad.

Theorem. An L-machine of size N can process MEMBER/INSERT/DELETE/XMIN commands with a compute time of $O(\log N)$ and a period of $O(1)$ as long as the number of elements in the set remains $\leq N$. (It is assumed that INSERTs always add new keys and that DELETEs always remove existing keys.)

Extraneous insertions (i.e., INSERT(k) commands with k already in the set) lead to severe problems, for they make processors shift their keys right while they shouldn't. Likewise extraneous deletions (i.e., DELETE(k) commands with k not in the set) make processors erroneously shift keys left, crushing the smallest key $> k$ currently in the set. A possible way out suggested in [18] is to allow "holes" in the array, i.e., to accept that some processors in the initial segment of the array hold no key. In this way, an INSERT command could proceed as indicated (because it would merely create an additional hole) and the procedure for DELETEs could simply consist of the erasure of the key to be deleted. We only need to make sure that the number of holes doesn't grow out of hand, to endanger e.g. the rapid answering of XMIN commands. We shall distinguish between empty locations at the right end of the array (marked with ∞) and embedded empty locations at the front

(marked by *)

Theorem. An L-machine of size N can process MEMBER/INSERT/DELETE/XMIN commands with a compute time of $O(\log N)$ and a period of $O(1)$, as long as the number of elements in the set remains $\leq N/2$. (Extraneous insertions/deletions are allowed.)

Proof

The technique is due to Ottmann, Rosenberg and Stockmeyer [18] but we render it in a considerably simplified form. The set will be maintained such that the following invariants hold: (I_1) the first processor is not starred, and (I_2) every starred processor has a non-starred right neighbour.

Extraneous insertions can lead to two consecutive starred processors and (ordinary!) deletions to even three starred processors in a row, assuming a deletion simply "stars" the deleted key. Define an auxiliary command COMPRESS, which makes a processor shift its key left if the left neighbour is starred. To reinstate the invariant, it clearly is sufficient to let every processor right of the presumed location of the update do one or two COMPRESSes after having performed the regular steps for an INSERT or a DELETE command, respectively. An XMIN will always move the smallest key correctly out onto the IO-pad due to (I_1), but the left shift generated for the entire array could bring (or rather: leave) a star in the first cell. Fortunately (I_2) is preserved in a left shift and guarantees that the second processor is not starred. To maintain the invariant (I_1) it thus suffices to do one COMPRESS after every XMIN.

(I_1) and (I_2) imply that at most $\lfloor N/2 \rfloor$ processors may get (and remain) starred, thus reducing the effective capacity of the machine to $\lceil N/2 \rceil$ keys. □

Ottmann, Rosenberg and Stockmeyer [18] argue that the compute time for the given set of commands can be reduced to $O(\log n)$ by making the

sorted chain of keys through the (upper) levels of the tree and creating a proper barrier to "bounce" signals back to the root.

4. Systolic arrays.

Until now we only saw designs in which data and instructions can enter through a single IO-port. One- and two-dimensional systolic arrays are designed so vectors of data can enter into the computation simultaneously, by letting all processors on the boundary have IO-ports to the environment (see e.g. figure 4). In this Section we shall explore the potential of the parallel pipelining of data in a number of applications. We shall primarily discuss the problem of comparing tuples $(a_1, \dots, a_N)$ and $(b_1, \dots, b_N)$, but the underlying principles will extend to familiar problems like the systolic multiplication of two $N \times N$ matrices.

Let us suppose first that $(a_1, \dots, a_N)$ is fixed and that we want to compare it to many tuples $(b_1, \dots, b_N)$. A first design that comes to mind is shown in figure 4. It is an array of N processors, in which

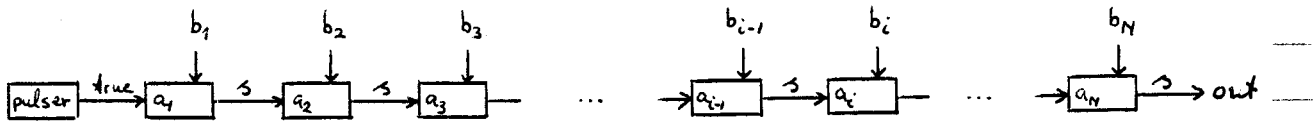
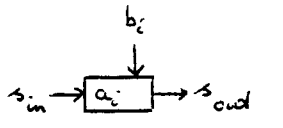


figure 4

the i^{th} processor is fixed to contain the i^{th} component of the tuple a. The tuple b is entered such that its i^{th} component is input to the same processor, so a comparison of a_i and b_i can take place. A signal s (starting out with value true) is chased from left to right to collect the results. The tuples match if and only if s still has value true

figure 5



$$s_{out} = s_{in} \wedge (a_i = b_i)$$

when it reaches the right end of the array. Figure 5 shows the simple action of every processor.

The design of figure 4 appears to have a period of $O(N)$. yet there is no reason to let the 1^{st}, 2^{nd}, ... processor idle as soon as the s-signal has passed. To take advantage of it, we shall modify the design and use a skewed input format (figure 6): for all $1 \le i \le N-1$ the datum b_{i+1} is input into the $(i+1)^{st}$ processor exactly one clock pulse after b_i is input to the i^{th}. The net effect is that the $(i+1)^{st}$ component is input

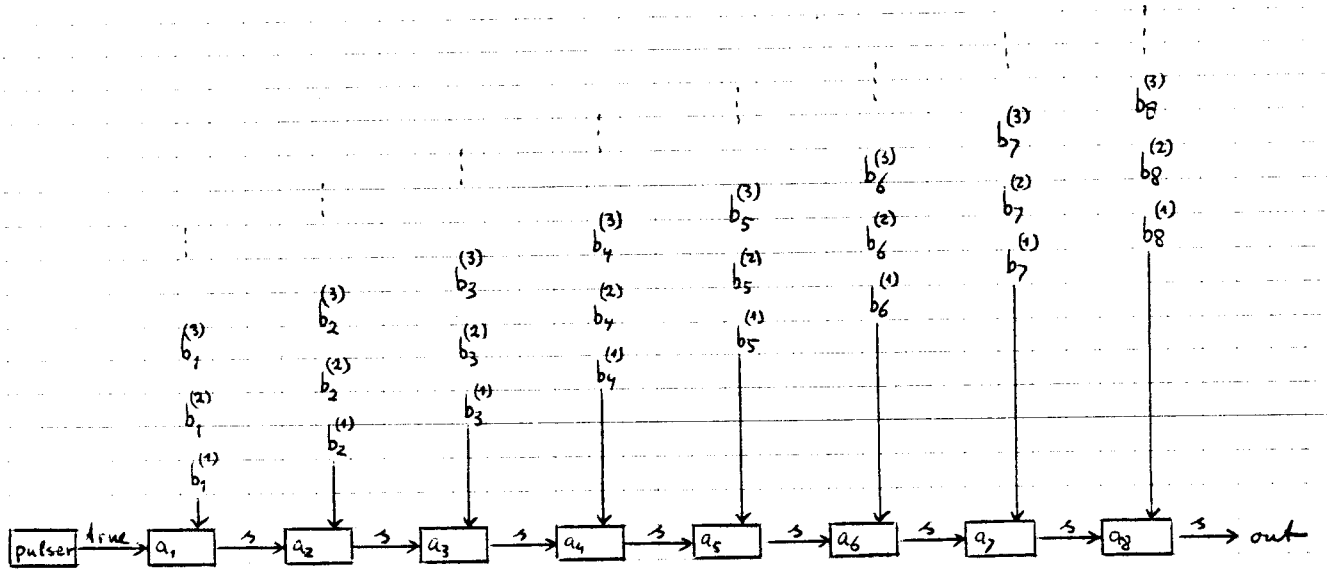


figure 6

just when the result signal s of the comparison between $(a_1, \dots, a_i)$ and $(b_1, \dots, b_i)$ comes in from the left. It should be clear that this design can be pipelined with a period of $O(1)$. The result of a comparison is available one clock pulse after entering the last component of a b-tuple.

Theorem. A linear systolic array of size N can do tuple comparisons with a fixed vector of length N with compute time $O(N)$ and period $O(1)$, using a skewed input format.

It is interesting to note that there is nothing special about using the operations $\wedge$ and $=$ in the processors (cf. figure 5). If we use operations $+$ and $\cdot$ instead s essentially accumulates the inner product of the tuples $(a_1, \ldots, a_N)$ and $(b_1, \ldots, b_N)$ as vectors. We thus have a systolic algorithm for a variety of problems that all conform to the same algebraic laws.

Now imagine that we have a "long" string $b = b_1 b_2 b_3 \cdots b_n$ $(n \geq N)$ and enter the tuples $(b_1, \ldots, b_N)$, $(b_2, \ldots, b_{N+1})$, $(b_3, \ldots, b_{N+2})$ and so on. The result is that $(a_1, \ldots, a_N)$ gets compared to every substring of b of length N, and the linear systolic array effectively becomes a <u>pattern matcher</u>. Closer inspection shows that it isn't really necessary to

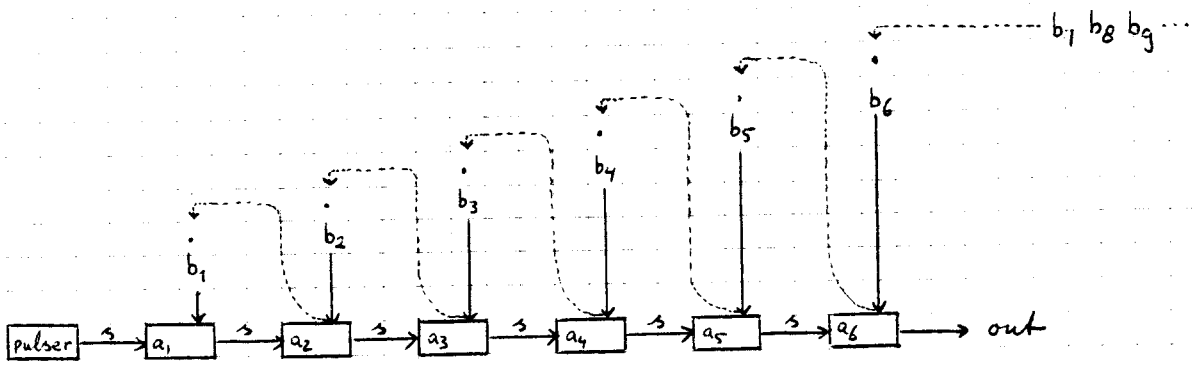


figure 7

enter all tuples separately. If we let each tuple lag by one additional clock pulse in time, then the next symbol required at the port of a processor can be sent over from its right neighbour and be received just when it is needed (see figure 7). Immediately after inputting the i^{th} symbol the array outputs a signal (s) that indicates whether the last N symbols match with $(a_1, \ldots, a_N)$ or not. Figure 8 shows the design we now

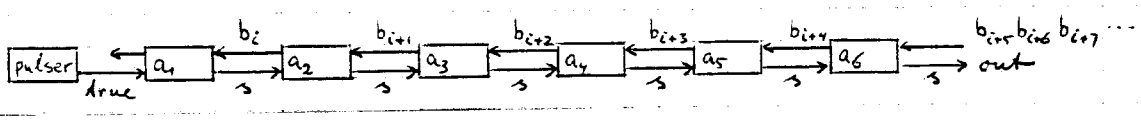


figure 8

have in its essential form. The string b is steadily moving through the array and the output indicates whether a match occurs or not. It is exactly the design of a <u>systolic pattern matcher</u> as given by Foster and Kung [8].

Now suppose we have a series of K tuples $a^{(k)} = (a_1^{(k)}, \cdots, a_N^{(k)})$ and $b^{(k)} = (b_1^{(k)}, \cdots, b_N^{(k)})$ and we wish to compare all couples $a^{(k)}$ and $b^{(k)}$, for $1 \le k \le K$. Returning to the design of figure 6 it is clear that a similar algorithm will do, provided we make sure that every processor is loaded with the proper $a_i^{(k)}$ at the right time to match a corresponding $b_i^{(k)}$. The solution is, of course, not to fix the a-symbol in a processor

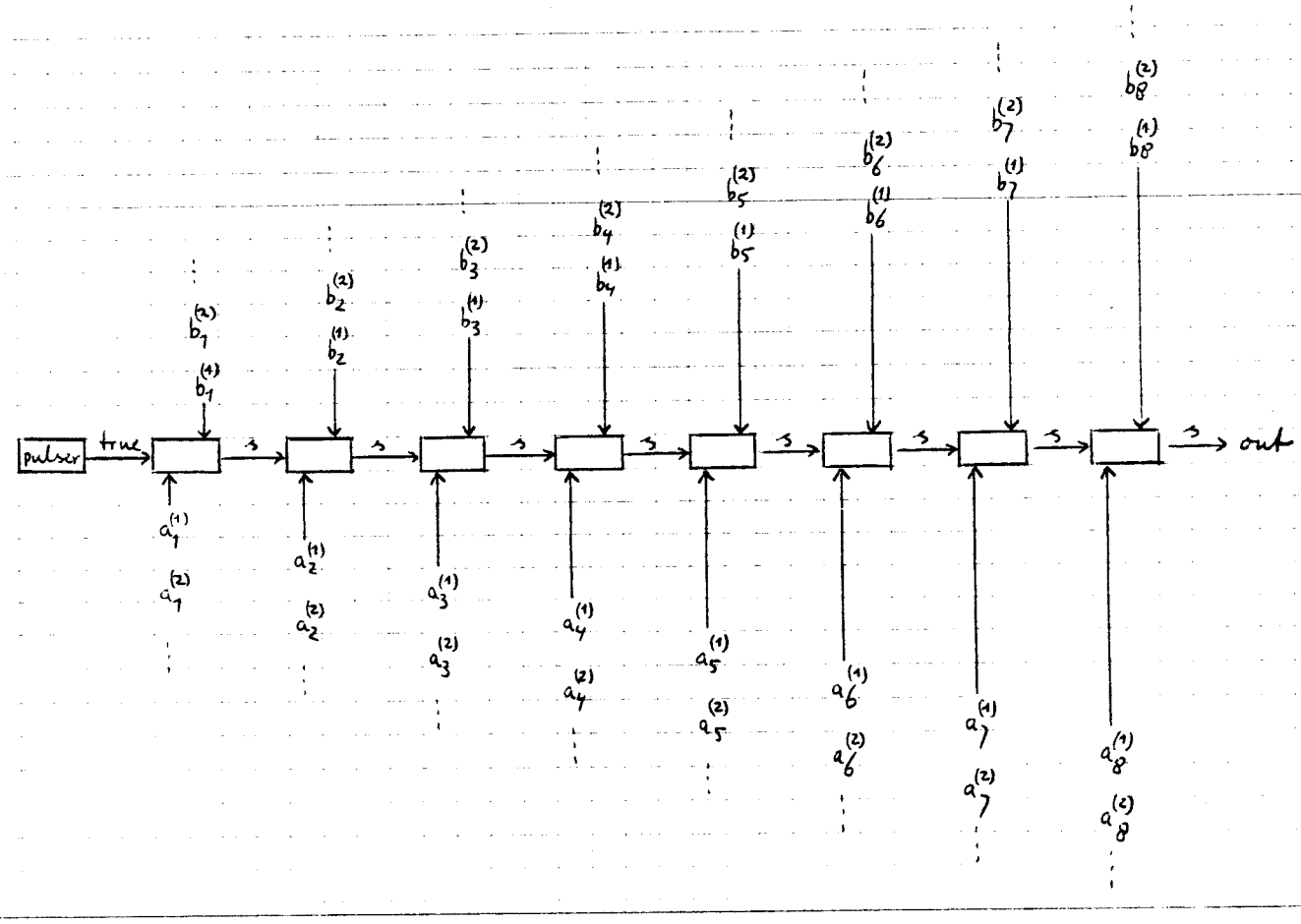


figure 9

but to feed in the a-tuples one after the other, in the same (but sym-

metrical) skewed manner as the b's. After reading in another pair of symbols $a_N^{(k)}$ and $b_N^{(k)}$ into the N^{th} processor, the array will output the result of comparing the entire tuples $a^{(k)}$ and $b^{(k)}$ in the cycle. The design is more appealing perhaps if we use the operations + and · instead of $\neq$ and $=$ in the processors.

Theorem. A linear systolic array of size N can compute inner products of pairs of vectors of length N with a compute time of O(N) and a period of O(1), using a skewed input format.

The result is a classical one in pipelined computation, which thus holds in the current framework as well.

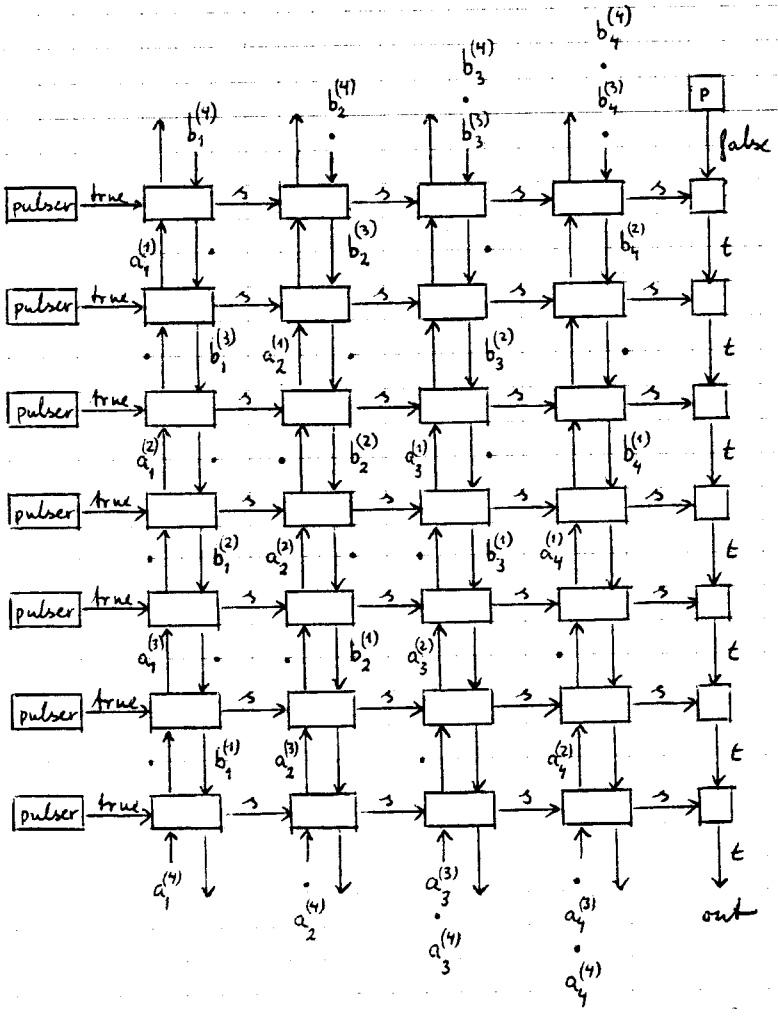


figure 10

Finally suppose that we wish to compare every a-tuple to every b-tuple and output signals which indicate with every (say) b-tuple whether it was matched at least once. In this way we would be able to select the tuples in the intersection of the two sets. The idea is, of course, to use a K-fold repetition of the linear design of figure 9 so each of the a-tuples comes across each of the b-tuples. A slight problem arises in the stepping pattern if we do so. For example, after $a_1^{(1)}$ and $b_1^{(1)}$ have met in a processor they both move and $a_1^{(2)}$ and $b_1^{(2)}$ immediately take their position. It means that $a_1^{(1)}$ and $b_1^{(2)}$ and $a_1^{(2)}$ and $b_1^{(1)}$ pass without meeting in a processor, i.e., without an opportunity to compare them! As shown in figure 10 the problem is easily solved by separating both the a-tuples and the b-tuples by a "dummy" tuple (dummies are drawn as dots). The number of processor rows must be increased to $2K-1$ to make it work. To collect the results of the comparisons an additional column of processing elements is added to the right of the array. The t-signal sent down is "$\vee$"-ed with the s-signal coming in from every row. Observe that the t-signal steps along with the b_N-symbol of every $b^{(k)}$. By the time $b_N^{(k)}$ leaves the bottom right processor, the value of the t-signal that appears on the out-wire during the next clock period will indicate whether $b^{(k)}$ matched any of the a-tuples! The design has been suggested by Kung and Lehmann [12] for use in a relational database multiprocessor environment.

Theorem. A two-dimensional systolic array of size $O(K)$ by $O(N)$ can process two sets of K tuples of length N and determine their intersection in $O(K+N)$ time.

An interesting result is obtained if we again replace the operations $\wedge$ and $=$ in a processor by $+$ and $\cdot$. In this case the design of figure 10 essentially computes all inner products $c_{kl} = a^{(k)} * b^{(l)}$ $(1 \le k, l \le N)$. If we omit the rightmost column and let each "s"-wire carry its value to

output, then we get the inner products in the skewed order as

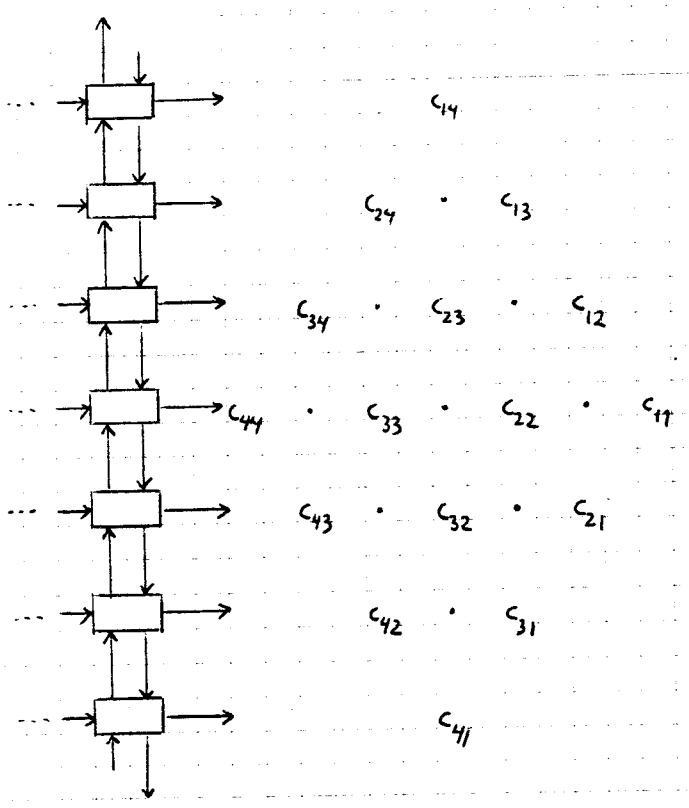


figure 11

shown in figure 11. Now let $K = N$ and interpret $a^{(1)}$ to $a^{(N)}$ as the rows of an $N \times N$ matrix A and $b^{(1)}$ to $b^{(N)}$ as the columns of an $N \times N$ matrix B. The systolic array we designed exactly produces the coefficients of the product matrix $C = A * B$, with a compute time of $O(N)$!

Theorem A two-dimensional array of size $O(N)$ by $O(N)$ can process two $N \times N$ matrices and output their product (as a matrix) in a compute time of $O(N)$. No (substantially) smaller array will do to get the same compute time.

Proof

It only remains to show that the $O(N^2)$ size bound of the systolic array cannot be substantially improved. This is a simple consequence of a theorem of Savage [20] that states that a matrix multiplier with

area A and compute time T must satisfy $AT^2 = \Omega(N^4)$. □

In general the design can be applied to multiply $K \times N$ and $N \times K$ matrices. (See Katona [9] for some other uses of the design.)

5. A systolic matrix inverter.

In this Section we shall develop a systolic algorithm to invert an $N \times N$ matrix $A = (a_{ij})$ in time $O(N)$, using $O(N^2)$ processors. The algorithm is based on Gaussian elimination (see e.g. [6]) and assumes that no pivoting is needed during the process. It serves here to demonstrate the intricacies of pipelined computation. We shall first discuss the basic algorithm and modify the data movement in it to suit our purposes.

The Gaussian algorithm to compute A^{-1} (without pivoting) operates as follows. Extend A to a $N \times 2N$ matrix $(A\ I)$ by juxtaposing an $N \times N$ identity matrix and apply elementary transformations to it until a matrix $(I\ B)$ is obtained. Then $B = A^{-1}$. Permissible transformations are rowmul's (multiply a row by a scalar) and rowsub's (subtract a scalar multiple of a row from another row). The precise algorithm we use is due to Pease [19]:

```
for i := 1 to N do
  begin
    rowmul: multiply the i^th row by a_ii^-1;
        for j := 1 to N do
          begin
            if j ≠ i then rowsub: subtract a_ji times the i^th row
                from the j^th row
          end
  end;
```

(In the algorithm a_{ii} and a_{ji} are used as variables and thus denote the

values in the corresponding locations of A as they are at the time of reference.) Figure 12 shows how the algorithm works on a simple example.

$$\left(\begin{array}{ccc|ccc} 2 & 0 & 2 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{array}\right) \quad (i=1) \Longrightarrow \quad \left(\begin{array}{ccc|ccc} 1 & 0 & 1 & \frac{1}{2} & 0 & 0 \\ 0 & -1 & 0 & -\frac{1}{2} & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{array}\right) \quad (i=2) \Longrightarrow$$

$$\left(\begin{array}{ccc|ccc} 1 & 0 & 1 & \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 & \frac{1}{2} & -1 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 1 & 1 \end{array}\right) \quad (i=3) \Longrightarrow \quad \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{3}{2} & -2 & -2 \\ 0 & 1 & 0 & \frac{1}{2} & -1 & 0 \\ 0 & 0 & 1 & -1 & 2 & 2 \end{array}\right)$$

Figure 12

To obtain an algorithm within our present framework, we shall aim at having a processor in every "cell" (square) of the $N \times 2N$ matrix. The key to systolicism is to observe the necessary data movement, to make it regular, and pipeline it. Consider Gauss's algorithm for $i = 1$. To do the rownul one could pass on the value of a_{11}^{-1} to the subsequent processors in the first row. Passing it on means: multiply, and send a_{11}^{-1} to the right neighbour. To do the rowsub for j ($j \neq 1$) one must spread the value of a_{j1} through the row, and next get the values from the first row (that should sift down through every column) and do the required multiply (by a_{j1}) and subtract (from the resident row element). It suggests that we better start by passing on the entire first column and temporarily store the values in an auxiliary field of the processors in every column. Only in the first row can we immediately do the necessary Gauss step right-away. Next we pass all (now modified) values of the first row down to the rows below and do the rowsub's, one after the other. This gives the algorithm the flavor of a cyclic (i.e., repeting) 2-phase process:

send the first column right, and (next) send the first row down. For the sake of systolism it would be desirable if this cycle could be repeted for $i=2$, $i=3$, ... Observe that the first row is turned into a unit vector (cf. figure 12) and never again is the sight of much activity after the first step. Thus, while sending the first column right, we might as well move the column and let it "exchange" its way over to the end, thus effectively moving the 2nd, 3rd, ... column left one position. If we do a similar exchange with the first row (which puts it at the bottom of the matrix and moves up all other rows by one), then we have created a starting position as before (although effectively with $i=2$ and e.g. a_{22} in the upper left corner). With this modifi-

$$\begin{pmatrix} 2 & 0 & 2 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} -1 & 0 & -\frac{1}{2} & 1 & 0 & 0 \\ 1 & \frac{1}{2} & 0 & 0 & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{pmatrix} \Rightarrow$$

$$\begin{pmatrix} \frac{1}{2} & -\frac{1}{2} & 1 & 1 & 0 & 0 \\ 1 & \frac{1}{2} & 0 & 0 & 1 & 0 \\ 0 & \frac{1}{2} & -1 & 0 & 0 & 1 \end{pmatrix} \Rightarrow \begin{pmatrix} \frac{3}{2} & -2 & -2 & 1 & 0 & 0 \\ \frac{1}{2} & -1 & 0 & 0 & 1 & 0 \\ -1 & 2 & 2 & 0 & 0 & 1 \end{pmatrix}$$

$\Rightarrow$: "send" the first column right, next "send" the first row down.

Figure 13

cation the algorithm has become completely regular and, as we shall see, amenable to pipelining. After N cycles of the 2-phase process all rows are back in their original order, but the columns are still halfway a full shift over 2N places. It means that the inverse now appears in the first rather than the second $N \times N$ block of the array and has effectively overwritten A, which is just as well. Figure 13 shows the modi-

fied algorithm applied to the example matrix of before.

As it stands each cycle of the algorithm takes $O(N)$ parallel moves and (thus) the entire routine takes $O(N^2)$ time, assuming the processors can be timed right to run it. Observe that each cycle of the algorithm consists of two "waves": one moving from left to right (doing a rowmul in the first row and spreading the a_{ij}'s in the lower rows) and a next

the wave from left to right:

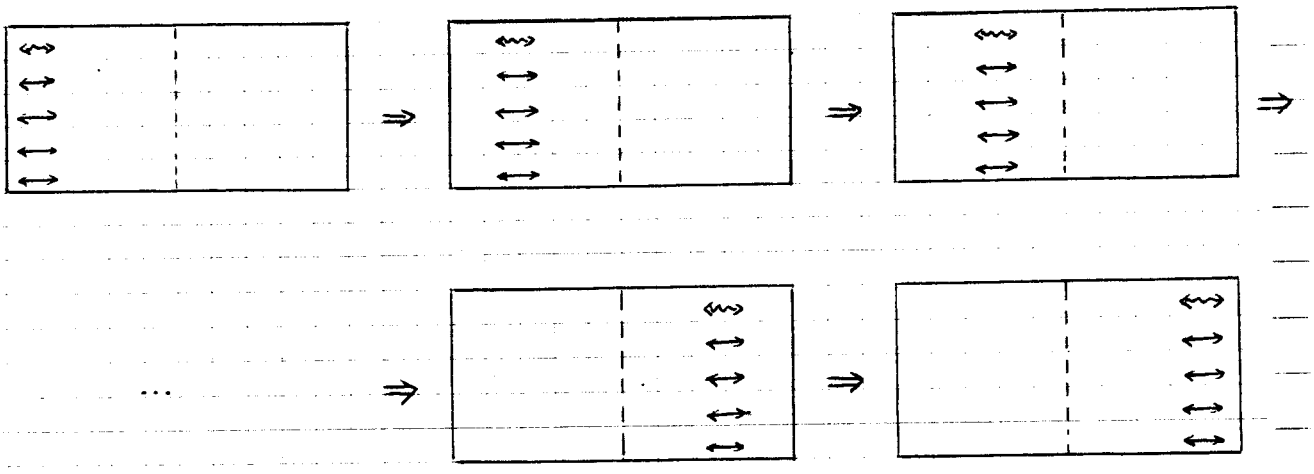


Figure 14

the subsequent wave from up to down:

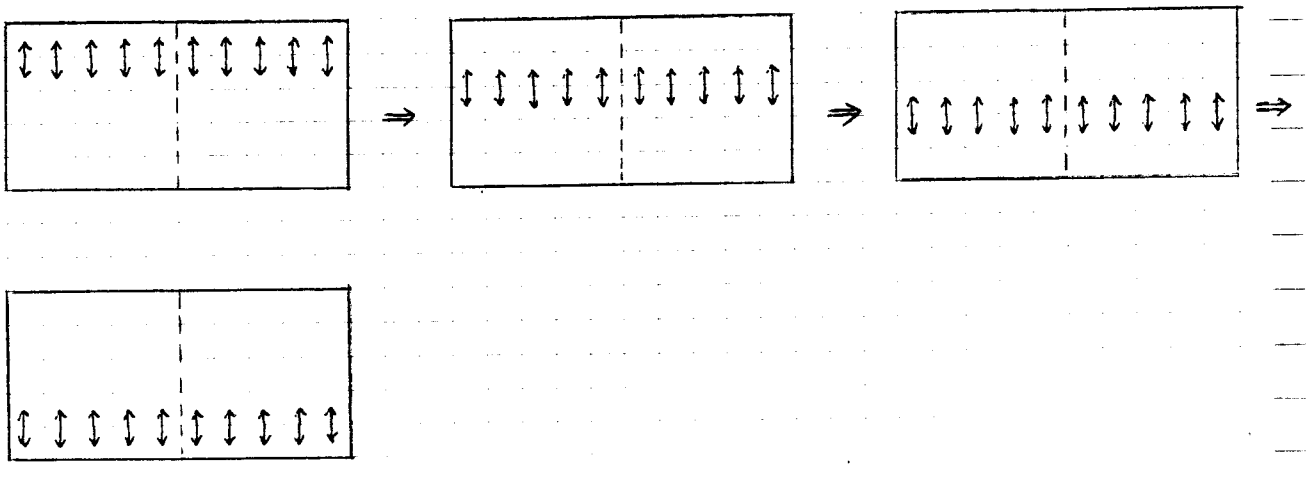


Figure 15

one from "up" to "down" (exchanging the values of the first row downwards and executing the necessary rowsub's, using the a_{ij} values just deposited in the preceding wave). Figures 14 and 15 show how the waves progress, with "↔" and "↕" indicating where the "action" occurs. The wiggly "⇜⇝" in the first wave indicates that a rowinul is carried out along the way.

Note that the down wave could begin at a processor as soon as the right wave has passed. And after a down wave has passed, the right wave of the next cycle can be started up. It follows that the regular wave pattern of the algorithm allows us to pipeline the computation and let the waves follow another at a short distance (only one or two clock periods). To achieve it we have to "skew" the wave fronts, so they indeed move completely in parallel and in pipelined fashion. Figures 14bis and 15bis show how this looks like for separate waves, and figure 16 shows the

a skewed right wave:

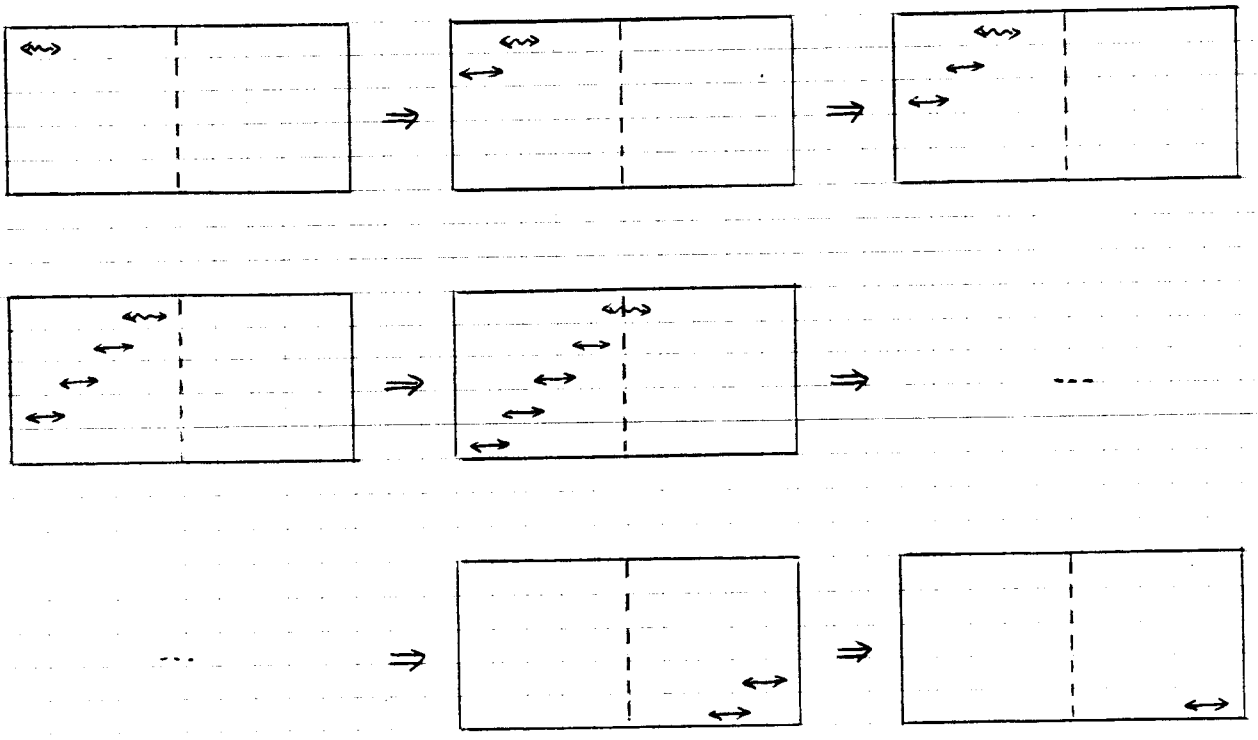


Figure 14bis

a skewed down wave :

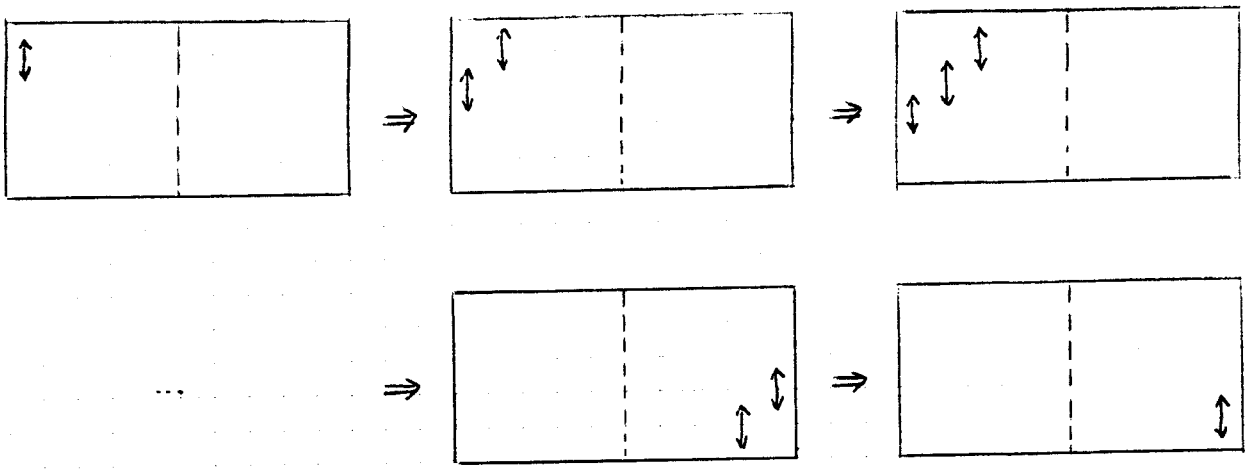


figure 15 bis

pipelining :

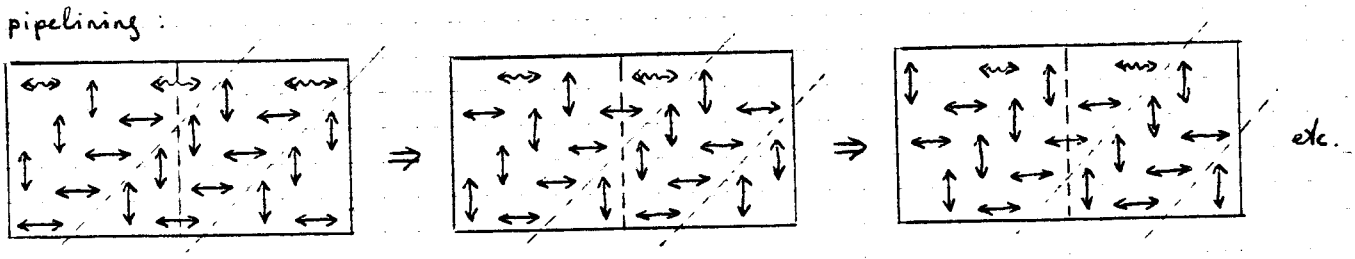


figure 16

waves of a cycle trailing (between / /) across from the upper left to the lower right corner.

Theorem. A two-dimensional systolic array of size N by 2N can compute the inverse of a (stored) N×N matrix in O(N) time using Gauss's algorithm (assuming no pivoting is needed).

Proof.

Put a processor in every cell of the N × 2N matrix. Each processor will have two data registers and a periodic clock (or state indicator). The first (a-)register contains the value of the matrix element that is stored here, the second (b-)register contains the datum that is being passed on. The clock essentially cycles through four states, corresponding

to the neighbour with whom information is exchanged (watch e.g. the continuous activity at a single cell in figure 16) : l ("get value from the b-register of the left neigbor"), r ("exchange"), u ("exchange") and d ("compute the rowsub"). We shall actually let the downward exchanges move through the a-registers. The processors along the boundaries will be considered separately as they miss some of the "neighbours" referred to. The activity of the array is started by sending a control signal immediately preceding the front of the very first cycle, which turns everybody's clock to l.

For a concrete description of the actions of a processor we shall adopt Kadona's method of specifying transitions through "configuration terms" ([9]). Processors will be placed in categories according to their location in the array (see figure 17) : category I are all cells in the

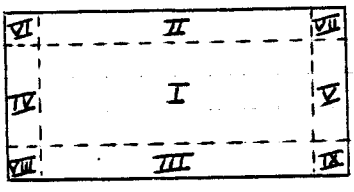


figure 17

interior, and categories II to IX are spread along the boundary. The transitions can be grouped as follows :

(i) send and exchange the multiplier through the first row (this implements the wiggly ones)

category	processor		next state	
VI	a l	⇒	a' r	
II	b | l	⇒	b r	
VI II	b r | a	⇒	a' u	with $a' = a \cdot b$
VII	l	⇒	1 r	

(ii) send and exchange the a_{j1} right (through the b-registers) as in the first wave of a cycle

category	processor		nextstate
IV VIII	a l	⇒	a r
I III	b — l	⇒	b r
IV VIII I III	r — a	⇒	a u
V IX	b — l	⇒	b b r

For the processors with no right neighbour we have, in addition:

category	processor		nextstate
VII V IX	r	⇒	u

(iii) send and exchange the elements of the first row down (through the a-registers) according to the second wave of a cycle

category	processor		nextstate	
	a			
IV I V VIII III IX	u	⇒	a d	
VI II VII IX I V	a_1 d	⇒	a' l	with $a' = a_2 - b \cdot a_1$
	a_2 b			

For the processors with no neighbour above or below we have, in addition:

category	processor		nextstate
VI II VII	u	⇒	d
VIII III IX	d	⇒	l

The configuration terms only display the register contents that are of use in a transition. Unspecified ("empty") fields remain unaltered. The last wave of the final (N^{th}) cycle should carry a control signal that turns processors off (as they return to the l state again).

Observe that N cycles of the algorithm thus trail over the array. The compute time is easily seen to be $O(N)$. □

A simple observation enables us to reduce the size of the processor array from N by 2N to N by N. (A related observation occurs in [19].) Returning to the non-pipelined version of the algorithm, consider the right

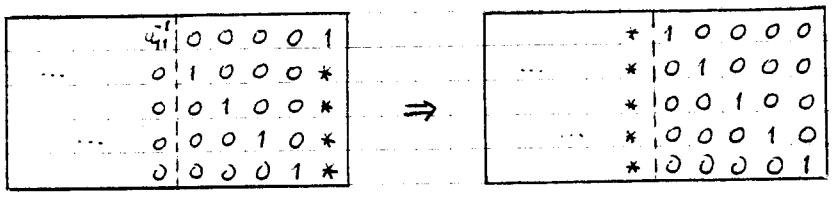
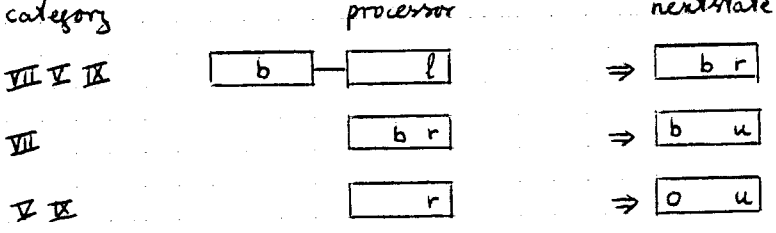


figure 18

N by N block after the first (left to right) wave of a cycle has ended and exchanged the first column towards the last position in the processor array (see figure 18). The next (downward) wave will turn this vector into a unit and exchange it towards the bottom position, thus effectively turning the entire right block back into the identity matrix! It follows that we may as well eliminate the right block from the processor array, provided we let the processors along the (new) right boundary act as if the block was still there:



All other transitions remain as they were. The entire procedure of pipelining, of course, holds true for the curtailed array as well. There is no hope that the size of the array can be reduced to anything less than $O(N^2)$ if the linear processing time is to be maintained, in view of the results of Savage [20]. By noting (cf. [1], thm 6.8) that

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}^{-1} = \begin{pmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{pmatrix}$$

it is not hard to see that the lowerbounds for matrix multiplication (cf Section 4) apply to inversion as well. Hence, the systolic matrix inverter is essentially optimal with respect to the "AT^2" measure.

The design is interesting, for it proves that Gauss's algorithm contains a tremendous degree of parallelism. The most interesting part perhaps is the possibility to pipeline, which makes crucial use of the assumption that no pivoting is needed. The assumption is valid for e.g. symmetric matrices that are known to be positive definite (cf. [7]). If pivoting is required (for whatever reason, including numeric stability), then a third wave is added to every cycle which moves "backward" and thus prevents the expedience of pipelining. Cycles still need $O(N)$ time, but the execution can no longer be overlapped. The $O(N^2)$ algorithm for matrix inversion that results still improves upon the fastest algorithms known in the sequential case. (This is Pease's result, cf. [19].)

It should take little effort to modify the systolic matrix inverter to a systolic "LU-decomposer". The underlying algorithm (again ignoring the need to pivot) is quite similar, but only does its rowsub's for $j > i$. After N cycles this produces the matrices U (upper triangular) and L^{-1} (lower triangular) in distinct portions of the processor array (see figure 19). Applying the inversion algorithm in the right block, the desired L

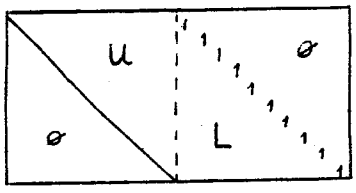


figure 19

matrix is obtained. In Kung and Leiserson [13] the LU-decomposition was produced slightly differently.

6 References

(Reference [16] is not cited in the text.)

[1] Aho, A.V., J.E. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley Publ. Comp., Reading, Mass., 1974.

[2] Armstrong, P.N. and M. Rem, A serial sorting machine, preprint, Dept of Computer Science, California Institute of Technology, Pasadena, Cal., 1978.

[3] Bentley, J.L. and H.T. Kung, Two papers on a tree-structured parallel computer, Techn. Rep. CMU-CS-79-142, Dept of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1979.

[4] Bhatt, S.N. and C.E. Leiserson, How to assemble tree machines, Proc. 14th Annual ACM Symp. Theory of Computing, San Francisco, 1982, pp. 77-84.

[5] Codd, E.F., Cellular automata, Acad. Press, New York, NY, 1968.

[6] Faddeev, D.K. and V.N. Faddeeva, Computational methods of linear algebra, Freeman, San Francisco, Cal., 1963.

[7] Forsythe, G.E. and C.B. Moler, Computer solutions of linear algebraic systems, Prentice-Hall Inc., Englewood Cliffs, NJ, 1967.

[8] Foster, M.J. and H.T. Kung, The design of special purpose VLSI chips, Computer 13 (1980) 26-40.

[9] Katona, E., Cellular algorithms for binary matrix operations, in: W. Händler (ed.), CONPAR 81, LN-CS vol 111, Springer Verlag, Heidelberg, 1981, pp. 203-216.

[10] Kung, H.T., Let's design algorithms for VLSI systems, Techn. Rep. CMU-

CS-79-151, Dept of Computer Science, Carnegie-Mellon Univ., Pittsburgh Pa, 1979

[11] Kung, H.T., Why systolic architecture, Computer 15 (1982) 37-45.

[12] Kung, H.T and P.L Lehmann, Systolic (VLSI) arrays for relational database operations, Techn. Rep. CMU-CS-80-114, Dept of Computer Science Carnegie Mellon Univ, Pittsburgh, Pa, 1980.

[13] Kung, H.T. and C.E. Leiserson, Systolic arrays for (VLSI), Techn Rep. CMU-CS-79-103, Dept of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1979. (See also: [17], chap 8.)

[14] Legendi, T., Cellular algorithms and their verification, in: W. Händler (ed.), CONPAR 81, LN-CS vol 111, Springer Verlag, Heidelberg, 1981, pp. 164-188.

[15] Leiserson, C.E., Systolic priority queues, Techn. Rep. CMU-CS-79-115, Dept of Computer Science, Carnegie-Mellon Univ, Pittsburgh, Pa., 1979.

[16] Leiserson, C.E., Area-efficient VLSI computation, Ph.D. Theris, Techn. Rep. CMU-CS-82-108, Dept of Computer Science, Carnegie-Mellon Univ, Pittsburgh, Pa., 1982

[17] Mead, C.A. and L.A. Conway, Introduction to VLSI systems, Addison-Wesley Publ Comp., Reading, Mass., 1980.

[18] Ottmann, Th., A.L. Rosenberg and L.J. Stockmeyer, A dictionary machine (for VLSI), Rep. RC 9060 (#39615), IBM TJ Watson Research Cntr., Yorktown Heights, NY, 1981.

[19] Pease, M.C., Matrix inversion using parallel processing, J. ACM 14 (1967) 757-764.

[20] Savage, J.E., Area-time tradeoffs for matrix multiplication and related problems in VLSI models, J. Comp. Syst. Sci. 22 (1981) 230-242.

[21] Song, S.W., On a high-performance VLSI solution to database problems, Ph.D. Thesis, Techn. Rep. CMU-CS-81-142, Dept of Computer Science, Carnegie-Mellon Univ, Pittsburgh, Pa., 1981.