SHEAR SORT: A TRUE TWO-DIMENSIONAL SORTING TECHNIQUE FOR VLSI NETWORKS

Isaac D. Scherson† , Sandeep Sen† and Adi Shamir‡

† Department of Electrical and Computer Engineering
University of California
Santa Barbara, CA 93106

‡ Department of Applied Mathematics
Weizman Institute of Science
Rehovot, Israel 76100

Abstract

Given a sequence of numbers which can be mapped into an $m \times n$ array, sorting rows and columns is shown to yield an overall sorted sequence. This unusually simple procedure is proven to require $O(\log_2 m)$ iterations by analysing the data movement in the array under successive row and column sorts. An efficient bubble sort network suitable for VLSI implementation, with near-optimal $area-time^2$ performance is a direct application of the row-column sorting technique. The ease in implementation for practical VLSI chips is also demonstrated.

I. Introduction

The problem of sorting numbers on a two-dimensional array has been studied by various researchers ([1], [2], [10]) and more recently by Leighton[3], Lang et al.[8] and Tseng et al.[12]. This problem involves routing of each data item to a distinct position of the array predetermined by some indexing scheme. Three different schemes have been considered by Thompson & Kung[1] : row major, shuffled row major, and snake-like row major (see Figure 1). The corresponding column-major forms can be considered equivalent to the schemes above. Most of the above referenced algorithms are based on two dimensional adaptations of very efficient sorting algorithms for linear sequences like bitonic sort[9] in [1] and [2], and odd-even merge sort[9] in [10] and these are essentially recursive in nature. Even though they perform optimally within a constant factor of the lower bound of O(n) for a $n \times n$ array, these algorithms spend most of the time in routing data to appropriate processors. Consequently, the complexity of the execution time is dominated by the number of unit routing steps in a SIMD model. Thompson and Kung[1] derived a 4(n-1) lower bound from an initial configuration where elements on the opposite corners of an $n \times n$ array have to be exchanged.

However, mere figures of time complexity appear to be superficial, when we consider the complexity of the control structure for the complicated data routing during the successive stages of recursion. One hardly needs to overemphasize the cost of the communication overheads in a VLSI implementation. In this context one can be more enthusiastic about the algorithm in [12], which in spite of being recursive in nature gets away with minimal control and achieves a bound within $O(\log n)^{(a)}$ of the optimal. In the realm of sorting a two-dimensional array of numbers, a seemingly "nice" way would be to sort rows and columns (since it involves sorting on smaller problems of approximately $\sqrt{n}$ size) and "hope" that somehow a combination of these two operations will terminate in a sorted sequence. Unfortunately, such a procedure doesn't seem to work when implemented in a straight-forward manner and indeed Leighton[3] observes that ".. if the matrix were square, we would essentially be sorting rows and columns which is well known to leave entries arbitrarily away from the correct sorted position". This was in obvious reference to the row-major indexing scheme. Paradoxically things fall into place when one sorts the rows in a snake-like row-major form without increasing the complexity of the procedure. This simple algorithm, which we call **shear-sort** , will be formally introduced in the next section. Section III will provide the analysis of the algorithm. In section IV we discuss very efficient and simple VLSI implementation of the algorithm using only bubble-sort network and in section V we discuss a method to optimize the algorithm by simple manipulations.

Row-major

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Snake-like row-major

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

Shuffled row-major

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

Figure 1. Indexing schemes.

II. Row-column sort

Let Q = [$q_{i,j}$] be an $m \times n$ matrix onto which we have mapped a linear integer sequence S. Sorting the sequence S is then sorting the elements of Q in some predetermined indexing scheme. We suggest an iterative algorithm in which every iteration consist of the two basic operations:

(1) Row-sort - Sort independently all the row vectors of Q such that adjacent rows are sorted in opposite directions (alternate rows in the same direction). In a normal snake-like row-major indexing scheme, sort the first row from left to right (increasing). At the end of this step, $q_{i,j} \leq Q_{i,j+1}$ for all i = 1, 3, 5,..2p+1 and $q_{i,j} \leq q_{i,j+1}$ for all i = 2, 4, 6,..2p.

(2) Column-sort - Sort independently, in an ascending order from top to bottom all column vectors of Q. After this step $q_{i,j} \leq q_{i+1,j}$ for all j = 1, 2, ..n.

The shear-sort algorithm is defined as a repetitive application of steps 1 and 2 until one of the following conditions is satisfied :

(a) all the columns are sorted, i.e. no element has moved in the present column-sort **after a row-sort**, or

(b) no element has moved in the present row-sort **after a column sort**.

A step by step application of the algorithm is shown in Figure 2.

(a) Throughout this paper log will be assumed to be to the base 2 unless otherwise mentioned.

1	4	9	13
2	6	10	14
3	7	11	15
5	8	12	16

Fig 2 a Rows and columns are sorted in row-major form but the array as a whole is unsorted.

1	4	9	13
14	10	6	2
3	7	11	15
16	12	8	5

Fig 2 b After the first rowsort in snake-like row-major indexing scheme.

1	4	6	2
3	7	8	5
14	10	9	13
16	12	11	15

Fig 2 c End of iteration 1

1	2	4	6
8	7	5	3
9	10	13	14
16	15	12	11

Fig 2 d Rowsort of iteration 2

1	2	4	3
8	7	5	6
9	10	12	11
16	15	13	14

Fig 2 e At the end of iteration 2 all elements are in their final sorted rows.

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

Fig 2 f A snake-like row-major sorted sequence.

The reader may note that the algorithm will terminate into a sorted snake-like row-major sequence from the definition of such an indexing scheme. It is trivial to observe that, if the rows and columns are sorted in the directions corresponding to the algorithm, the array is sorted. Thus conditions (a) and (b) are equivalent and are necessary and sufficient conditions.

To give the reader an insight into the 'mechanism' of this sorting technique and throw some light on how such a simple procedure results in a sorted sequence we will obtain a very loose upper bound using an informal analysis. In the next section we will derive a tight upper bound using an indirect technique.

Consider the n smallest elements in Q and assume they are randomly distributed over the rows and columns of the array. The first column-sort will move these elements to the first row if initially they all happen to be in different columns. However, if all n smallest elements are in the same column (only possible for $m \geq n$), by virtue of alternating sorting direction on rows, a row-sort will have the effect of moving the elements on odd rows to the leftmost column of the array and the remaining half to the rightmost column. *This phenomenon is analogous to normal shear of a column and hence the name of this algorithm* . Clearly, at the beginning of the second iteration the n smallest elements have moved up into the upper half of Q. The second row-sort will then pair the elements on the two leftmost columns for the odd rows and on the two rightmost columns for the even rows. Following the same reasoning, it is not difficult to see that the column sort of the p iteration will move the n smallest elements of Q to the band defined from rows 1.. $\frac{n}{2^p}$. Without loss of generality we can assume n to be a power of 2 and conclude that the n smallest elements of Q will move to their sorted positions in at most log n iterations.

It follows from the above discussion that for m < n, the n smallest elements move into their sorted position in at most log m iterations. It is evident that once the first row is in place, it will continue to do so throughout the remaining iterations. Therefore we now face a problem of sorting the reduced array Q_{-1} of dimension $m-1 \times n$. Each time a row is in its place, we reduce the problem to a smaller array on which the smallest elements are brought to the 'first' row in $\lceil log(m-k) \rceil$ iterations where k is the number of previously discarded 'first' rows. The total number of iterations to sort the array is thus

$$\sum_{k=0}^{k=m-1} \lceil \log(m-k) \rceil$$

which is bounded from above by mlog m.

In this simple analysis we assumed that after the 'first' row of array Q_{-k} is in place all the remaining elements are randomly distributed in the reduced array Q_{-k-1}. This is not the case as we will show in the next section which gives a much better bound of O(log m) iterations.

III. Analysis of the algorithm

We shall prove that shear-sort converges in O(log m) iterations, for an $m \times n$ array, by showing that the elements go within a specified distance of its final sorted row in every successive iteration. It will be seen that an arbitrary element (and hence all elements) goes within $\frac{m}{2^p}$ of its final sorted row after p iterations, from which the O(log m) bound follows immediately. We will use here an application of the [0-1]principle (Knuth[6]) (for a direct combinatorial proof see Scherson & Sen[13]). For the purpose of applying the [0,1] principle, we will visualize our sorting algorithm as a sorting network of logm + 1 stages, where in each stage we sort all the rows in a row-major snake-like form followed by sorting all the columns (Figure 3).

Consider the simple case of a $2 \times n$ array containing arbitrary number of 0's and 1's.

(a) After the rows are sorted, the 0's in the first row will be packed to the left, while 0's in the second row will be pushed to the right. Let us denote the number of 0's in the first and second rows by n_1 and n_2, respectively (Figure 4).

(b) Depending on the value of $n_1 + n_2$, sorting the columns (in this case a simple compare exchange), will result in one of the following

Case 1 ($n_1 + n_2 < n$) : the bottom row will contain only 1's and the top row will be a mixture of 0's and 1's.

Case 2 ($n_1 + n_2 = n$) : the top row will contain only 0's and the bottom row will have only 1's.

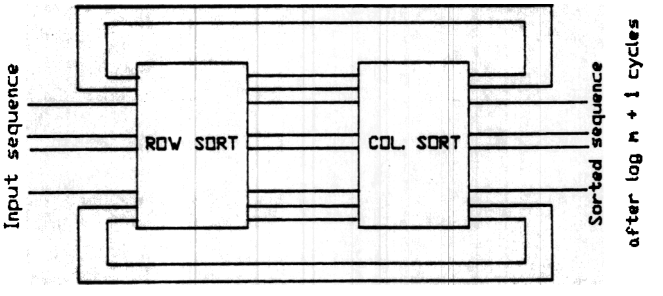

Figure 3. The algorithm as a sorting network.

0	0	0	0	0	0	1	1	1	1
1	1	1	1	1	1	1	0	0	0

Figure 4a. The top row is sorted from left to right and the bottom row from right to left(non-decreasing)

0	0	0	0	0	0	1	0	0	0
1	1	1	1	1	1	1	1	1	1

Figure 4b. Afterthe column sort, one of the two rows(second in this case) contains only 0's or 1's. In a more favorable case, both rows are clean.

Case 3 : ($n_1 + n_2 > n$) : the top row will contain only 0's and the bottom row will contain an unordered sequence of 0's and 1's.
Out of these cases, Case 2 is the most favorable, since the rows are already sorted and in the other cases we have a sorted row (first in case 1 and second in case 2). Also, the rows are mutually ordered i.e., all the elements of the top row are less than or equal to the elements of the bottom row. In the next row-sort the remaining row is ordered and sorting is complete. Henceforth we shall refer to a row as being 'clean', if it contains identical elements (only 0's or only 1's). Analogously, a row consisting of both 0's and 1's will be called 'dirty'.

The above discussion makes it clear that in a pair of adjacent rows, after the first iteration at least one of the rows is 'cleaned', i.e. it consists of only 0's or 1's. We will now extend the above phenomenon to an $m \times n$ array. Without loss of generality assume m to be a power of 2. After the row-sort of the first iteration we have 0's squeezed to the left on the odd- numbered rows and packed to the right on the even-numbered rows. Consider that a column sort consists of the following two stages
(1) For an element $q_{i,j}$, do a compare exchange with the element $q_{i+1,j}$, for i = 1,3 ..,m-1. For i = 2,4 ..,m, compare exchange $q_{i,j}$ with $q_{i-1,j}$. (This is equivalent to sorting m/2 pairs of adjacent rows independently.)
(2) Sort the columns normally i.e. across the entire columns.

It should be clear that decomposition of a column sort in the above manner is not going to affect the ordering resulting from a normal column sort. The extra (hypothetical) step makes the analysis much simpler. From the previous discussion, we know that following step 1 there is at least one clean row among every pair of odd-even rows. We may now consider an entire 'clean' row as a single entity during the column sort since a 'clean' row continues to be clean throughout the subsequent iterations. This follows from the observation that, when a 'clean' row of 0's is compared with a 'dirty' row across the columns, the clean row bubbles to the top row remaining 'clean'. Similarly a 'clean' row of 1's sinks to the bottom. Clearly, two clean rows, when compared across the columns cannot 'dirty' themselves. It may be easier to visualize this process of column sort being a bubble sort, in which all elements of a row vector are simultaneously compared to the corresponding elements in their adjacent row. The reader should note that the actual algorithm used for sorting columns does not necessarily have to be a bubble-sort since the ordering obtained is independent of the algorithm used. Row-sorts do not affect the clean rows.

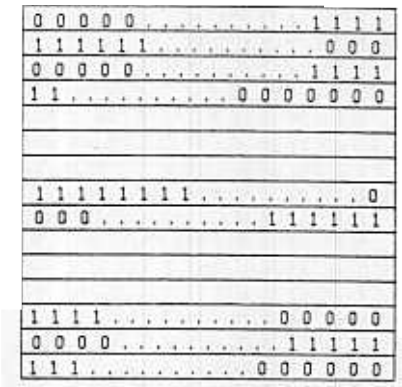

Fig 5 a We can visualize the column sorting as
1. sorting on pairs of rows across all columns which cleans atleast half the number of dirty rows followed by
2. sorting on columns across the entire array. Note that cleaned rows in previous step continues to remain 'clean' thereafter.

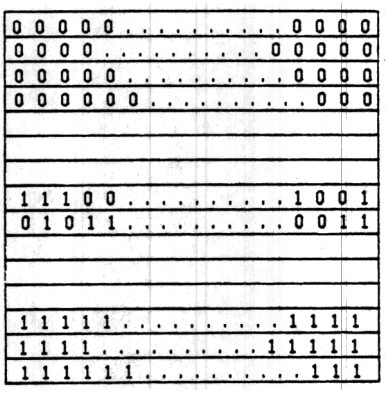

Fig 5 b The 'clean' rows of 0's bubble up at the top of the array and the 'clean' rows of 1's settle down. Note that the 'dirty' rows are contiguous somewhere between the bands of 'clean' rows.

It follows, that after the column-sort step of iteration 1, we have at least half (m/2) rows 'clean', with the clean rows of 0's at the top and 'clean' rows of 1's at the bottom. The 'dirty' rows will occupy a band of at most m/2 rows in the middle of the array (Figure 5). Actually we may find less than m/2 'dirty' rows since two 'dirty' rows may mutually 'clean' each other during the column sort. In the course of the second iteration, half of the m/2 'dirty' rows will become clean by applying the same argument to the contiguous band of dirty rows while the clean rows continue to be clean. By a repetitive application of this phenomenon of cleaning half the number of dirty row with every successive iteration, we may

conclude that there can be at most one dirty row after log m iterations. An additional row-sort orders this dirty row which actually separates the clean rows of 0's from the clean rows of 1's in the sorted array. Thus, for an initial array consisting of an arbitrary number of 0's and 1's **the shear-sort algorithm terminates successfully in logm +1 iterations.**

The classification of rows as 'clean' and 'dirty' in constructing the complexity analysis was extremely helpful since it reduced the burden of keeping track of every element in the array to only m rows. At this point the implications in a generalized array consisting of elements which may not be only 0's or 1's, is not clear. Nevertheless, we can state the following important result:

Theorem 1 : An $m \times n$ array of elements can be sorted using shear-sort in time proportional to $\lceil \log m \rceil + 1$ [m*(time for row-sort) + n*(time for column sort)].

The proof follows immediately from the previous discussion and the [0,1] principle. The tightness of this bound should be obvious from the the informal discussion at the end of the previous section. In the next section we will present an efficient implementation of this algorithm on a mesh-connected processor array.

The **average-case** performance analysis can also be simplified using the [0-1] principle. An important corollary following from the discussion of clean and dirty rows can be summarized as follows:

Corollary 1 : A 0/1 array consisting of 'd' dirty rows initially, can be sorted using $\lceil \log d \rceil$ iterations of shear sort.

Theorem 1 can be stated as a special case, since all m rows can be dirty initially. Consider O(m) 0's uniformly distributed in the $m \times n$ array, the other elements being 1's. Because of the uniform distribution, we may conclude, that on the average, m/2 rows will be dirty. From Corollary 1, the average number of iterations is logm - 2 or O(log m). This shows that the algorithm is optimal within minor variations of the basic shear-sort paradigm.

IV. VLSI Implementation of shear-sort

In spite of the terrible performance of a normal single processor bubble sort, efforts have been directed towards obtaining efficient VLSI implementations ([5], [6]) because of the inherent simplicity of the algorithm. For this purpose a parallel version of bubble sort viz. odd-even transposition sort([6]) has been adopted. By using crossing sequence techniques, several researchers have shown that the optimal AT^2 bound for sorting n elements is O(n^2) in word model and O($n^2 \log^2 n$) in bit model ([3],[5],[6]). The normal N/2 processor bubble sort where each processor performs one compare-exchange operation during each of the N iterations behaves horribly (O(n^3)) with respect to the AT^2 measure. This remains unchanged even by using completely pipelined bit-parallel comparison exchange modules to sort more than one problem instance. The pipelined scheme consists of O (n^2) comparators which reduces the effective area by a concurrency factor(n) - the time remaining unchanged ([5]).

Theorem 2 : The AT^2 performance of shear-sort implemented with a bubble-sort network is $O(n^4 \log^3 n)$ for n^2 elements.

Figure 6c shows the implementation of the shear-sort using a pipelined scheme where each of the n rows(columns) are pipelined through this sorting network. The 'Transpose/ Detranspose ' network aligns the array properly for the next column (row) sort. Following Leighton's[3] argument, the Transpose/Detranspose network needs n non-unit length wires and hence occupies $O(n^2)$ area where the transposition is performed in n parallel stages by hardwiring the rows to the corresponding columns (and vice versa- see Figure 6b). The bubble-sort network consists of n^2 comparators and thus the total area of the network is O($n^2 \log n$). We will need 2n word steps to sort all n rows (columns). Each comparator is capable of performing a compare-exchange operation of two O(log n) bit numbers

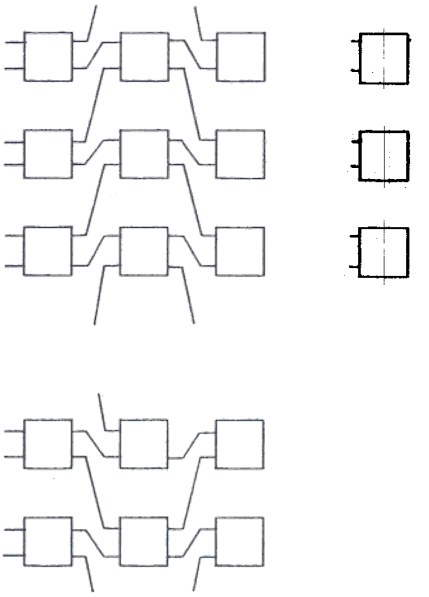

Figure 6a. An N/2 by N comparators Bubble-sorter
Each box represents a compare-exchange module

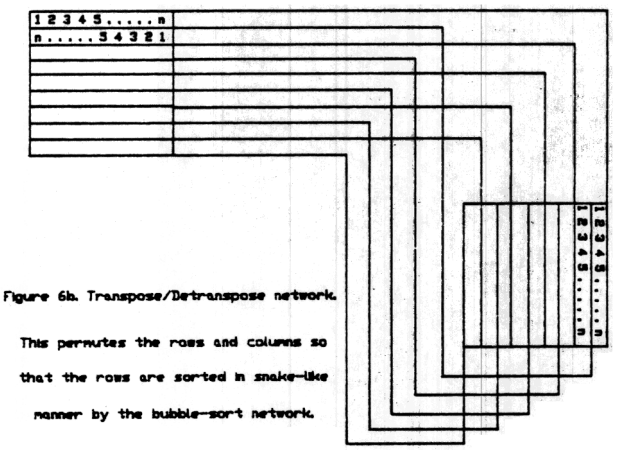

Figure 6b. Transpose/Detranspose network.
This permutes the rows and columns so that the rows are sorted in snake-like manner by the bubble-sort network.

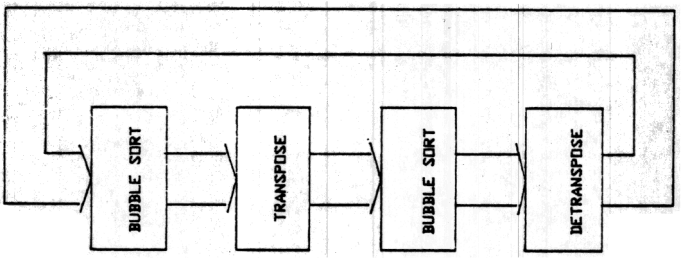

Figure 6c. Block diagram of Shear-sort using Bubble-sort network.
log n + 1 passes through this network will sort any sequence

in O(1) time. As observed previously the Transpose/Detranspose network also needs O(n) time for each iteration. Since we need log n iterations the AT^2 performance for this scheme will be

$$O(n^2 \log n) \times O((n \log n)^2) = O(n^4 \log^3 n)$$

This is only O($\log^3 n$) away from the lower bound. A similar result can be obtained for the bit model by using bit-serial compare exchange modules. Each compare-exchange module can perform a compare exchange operation every O(log n) time units and can fit into an O(1) by O(log n) unit rectangle. Thus the bubble-sort circuit occupies an area of O($n^2 \log n$) units. The Transpose/Detranspose circuit consists of n non-unit length wires which occupy an area of O (n^2) units. Each of these wires routs O(n logn) bits of data and thus takes O(n logn) units of time to complete the operation. The total time is O($n \log^2 n$) and so the AT^2 measure for this scheme is O($n^4 \log^5 n$). This is only a factor of $\log^3 n$ away from the optimal which is O($n^4 \log^2 n$) for O(n^2), O(log n) bit numbers ([3]).

As noted by Thompson[5], this network needs very little in the way of control as no complicated operations are involved and may be more attractive than its AT^2 performance indicates (being a couple of log n factors away from the optimal). There is hardly any need to overemphasize that this scheme of sorting which exploits the powerful property of the algorithm has made bubble-sort comparable to some more sophisticated VLSI sorting networks as far as $area-time^2$ trade-off is concerned.

Figure 7 shows a more regular network for accomplishing the required compare exchanges, though it needs slightly more area (a factor of logn more than the previous approach). We have managed to incorporate the comparators for carrying out row and column sorts within the mesh- connected network, which is very similar to an ILLIAC IV like machine. The horizontal and the vertical compare-exchange modules are used during row-sort and column sort stages respectively. A ROW/COLUMN control line decides which set of the comparators is being used. A ROWDIR control line which controls the direction of compare exchange for the horizontal comparators, achieves alternating sorting directions in rows. A single row/column of elements 'oscillates' between two adjacent row(column) n/2 times, each of which carries out the required compare-exchanges needed for two iterations of odd-even transposition sort. The ROWDIR is changed during every clock cycle of the row-sort stage, so that adjacent rows are sorted in opposite directions. The overlapped comparators in the figure actually represent single comparators multiplexed between the row and column sorts. Each cell is drawn with two inputs, but only one of them is selected according to the row or column sort stages. After O(logm +1) such iterations the array is sorted in a snake-like ordering.

A few comments may be expedient at this point to emphasize the ease in implementation and the expandability of this scheme which are of major concern to any VLSI chip designer. The simplicity in the control structure, which is the result of purely iterative nature of the algorithm, and the regularity of the layout due to use of only nearest- neighbor type operations, make it ideally suitable for systolic implementation. Each chip containing a subset $k \times k$ of the $n \times n$ components will be connected to four similar chips in the four (North, East, South, West) directions, thus presenting no additional complications for chip interconnections. To account for the pin limitations in a chip, and packaging a maximum number of components in a single chip, we can use bit serial communications between the nearest neighbors. This would not change the asymptotic performance since a compare-exchange operation is of the same complexity as a bit-serial communication time of a 'b' bit words (O(b) time). For example, in a 80 pin package, we need at most 10 pins for the global control signals leaving us with 70 pins for the inter-chip communication. Rounding this to the nearest power of two, i.e. 64, we can integrate a 16×16 mesh-connected compare-exchange modules in a single chip (for a $k \times k$ array, we need 4k data pins).

While this scheme is theoretically inferior to Lang et al.[8] by a factor of logn, it may prove to be more feasible for reasonable sized arrays by sacrificing a little time-performance in the bargain.

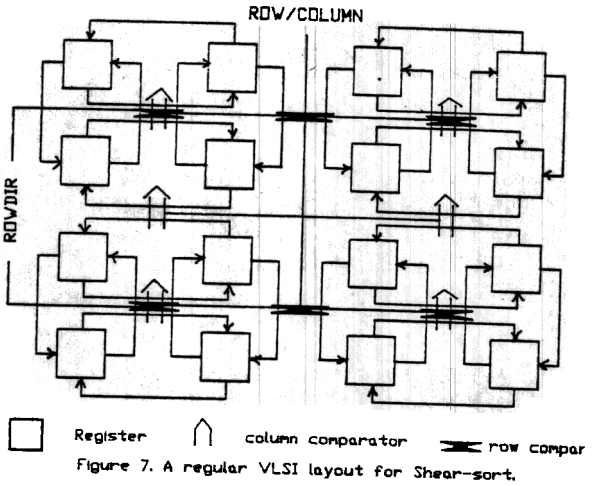

Figure 7. A regular VLSI layout for Shear-sort.

V. Optimization of the shear sort

A further reduction in execution time for shear-sort is possible for a 'brick-wall' type sorting using a simple manipulation in the dimension of the array of n^2 elements. Recall from section III, that the band of 'dirty' elements keeps on shrinking by a factor of 2 in every iteration. This is the basis for an O(log m) (for an $m \times n$ array) iteration convergence. Thus, during the successive stages of the algorithm, it is enough to sort the elements in a column that fall within the 'dirty' band, instead of sorting an entire column during the column-sort stage. Another way of looking at it is that, since all the elements move within $\frac{m}{2^p}$ rows after p iterations (discussed more in [13]), we need to perform only $\frac{m}{2^p}$ steps of the brick-wall sort after p iterations.

For an $m \times n$ array, this means we perform m, m/2, m/4 .. iterations of brick-wall sort in successive column-sort stages. However, no such optimization seems possible for the the row-sort stage since the algorithm is non-adaptive so that we keep on performing all the 'n' compare-exchange steps throughout the course of execution. This yields a complexity of n(logm +1) + 2(m - 1) compare exchange steps. For n^2 elements, this can be expressed as

$$\frac{n^2}{m}(logm + 1) + 2(m - 1)$$

By choosing $m = \frac{n\sqrt{logn}}{\sqrt{2}}$, we get a complexity of

$$O(\ 2\sqrt{2}n\sqrt{logn} + \frac{nloglogn}{\sqrt{2logn}}\ )$$

The second term can be approximated to o(n) since $\frac{loglogn}{\sqrt{logn}} = o(1)$, thereby yielding the following result:

Theorem 3: A rectangular array of n^2 elements can be sorted using shear-sort in time proportional to O($n\sqrt{logn}$).

The behavior of this function is very close to linear since $\sqrt{logn}$ is less than 5 for well over tens of millions i.e. more than the practical size of any sorting chip. In fact it outperforms most of the well-known algorithms for number of elements below a couple of tens of thousands which is not rare in practical situations. The cautious reader may have noticed that an extra compare exchange step is needed during the column sort stage depending on the boundary of the dirty band. If the dirty band starts from an even row during all the iterations of column sort the complexity function will be affected by an additional log m steps, which is negligible.

This improvement can be very easily incorporated into the VLSI implementation shown in Figure 7. The number of iterations of odd-even transposition sort during row and column sort stages can be easily controlled by the number of 'oscillations' between adjacent rows(columns) of comparators.

VI. Conclusions

The feasibility of sorting a rectangular array of elements by sorting rows and columns was demonstrated. The algorithm was shown to execute in at most O(log m) iterations and in section III we traced the data movement during each iteration. Recall that we showed that all elements moved within O($\frac{m}{2^p}$) rows of their final destination row in O(p) iterations. Also taking advantage of this phenomenon it is possible to optimize a simple algorithm like bubble sort. Sorting a row(column) is actually sorting $\sqrt{n}$ elements in a n element sequence which may be expensive. However this basic operation may be optimized by using a sorting network as was demonstrated in section IV. The complexity of this algorithm is more appropriately expressed as O ($\sqrt{n} \times k \times \log n$) for an 'n' element sequence organized as a square array, where k is the time for sorting $\sqrt{n}$ elements and this is O($\sqrt{n} \log n$) for a single processor sort which was used as a basis for the multiprocessor implementation. For the sorting network allowing pipelining this turns out to be O($(\sqrt{n} + k)\log n$) which at the best will give us a time of O($\sqrt{n} \log n$).

This algorithm can be also executed effectively in any MIMD type machine which allows independent access to rows and columns in a shared memory model. A multiprocessor architecture which can implement the algorithm elegantly because of its orthogonal access to the memory banks and originally conceived as a high performance graphics system which can draw very fast vectors of any orientation ([11]) is discussed in Scherson & Sen[13]. In a similar architecture, Tseng et al.[12] propose a combination of bitonic sort and a single processor O(nlogn) sort, which can achieve identical complexity figures when implemented on mesh-connected processors. That process also sorts rows and columns in successive stages though it requires a more complicated control structure being recursive in nature. The direction of sorting in rows change dynamically in successive levels of recursion, as also does the number of independent sequences in the columns.

In an effort to optimize the algorithm further, by trying to reduce the cost of sorting entire rows and columns by partial sorting an interesting variation is mapping a two dimensional odd-even transposition sort on this row-major snake-like indexing scheme. Each iteration will consist of performing compare-exchange on elements ($x_{2i-1,j}$, $x_{2i,j}$) on all rows independently followed by the same procedure on all the columns ($x_{i,2j-1}$, $x_{i,2j}$) and then repeating the same with the elements ($x_{2i,j}, x_{2i+1,j}$) in all rows and elements ($x_{i,2j}, x_{i,2j+1}$) in all columns. It is not difficult to see that such an algorithm will result in a sorted sequence. It was speculated in [13] that the upper bound for the number of iterations may be $\sqrt{n}$ giving an optimal and an even simpler algorithm. However it turns out to be totally inefficient running into O(n) iterations like a normal bubble-sort which makes us hopeful about the optimality of shear sort within similar variations.

References

[1] C.D. Thompson & H.T. Kung , "Sorting on a Mesh-Connected Parallel Computer," Communications of the ACM, vol 20, no. 4, April 1977.

[2] D. Nassimi & S. Sahni, "Bitonic Sort on a Mesh-Connected Parallel Computer," IEEE Transactions on Computers, vol C-27, no 1, January 1979.

[3] T. Leighton , "Tight Bounds on the Complexity of Parallel Sorting," IEEE Transactions on Computers, vol C-34, no. 4, April 1985.

[4] D. Gale & R.M. Karp ," A Phenomenon in the Theory of Sorting," Journal of Computer and System Sciences, no. 6 , 1972.

[5] C.D. Thompson, "The VLSI Complexity of Sorting ," IEEE Transaction on Computers, vol C-32, no. 12, December 1983.

[6] D.E. Knuth, "The Art of Computer Programming," vol. 3, Addison-Wesley, 1973.

[7] J.D. Ullman, "The Computational Aspects of VLSI ," Computer Science Press, 1984.

[8] Lang Hans-Werner et al., "Systolic sorting on a Mesh Connected Network," IEEE Transactions on Computers, vol C-34, no. 7, July 1985.

[9] K. Batcher, "Sorting Networks and their applications," in *Proc.* AFIPS Spring Joint Comput. Conf, vol 32, 1968.

[10] M. Kumar & D.S. Hirschberg, "An efficient implementation of Batcher's odd-even merge algorithm and its application in parallel sorting schemes," IEEE Transaction on Computers, vol C-32, March 1983.

[11] I.D. Scherson, " A Parallel Processing Architecture for Image Generation and Processing, " Preliminary Report, E.C.E. Report No. 84-20, U.C.S.B., August 1984.

[12] P.S. Tseng, K. Hwang and V.K. Prasanna Kumar, " A VLSI based Multiprocessor Architecture for implementing parallel algorithms, " International Conference on Parallel Processing, Aug. 1985.

[13] I.D. Scherson and Sandeep Sen, " A Characterization of a Parallel Row-column sorting technique for rectangular arrays, " ECE Technical Report No. 85-14, U.C.S.B., August 1985.