UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

SELECTION ON MATRICES CLASSIFYING ROWS AND COLUMNS

A. A. Bertossi, S. Olariu, M.C. Pinotti and S.-Q. Zheng

September 2002

Technical Report # DIT-02-0073

Selection on Matrices Classifying Rows and Columns

A.A. Bertossi* S. Olariu† M.C. Pinotti‡ S.-Q. Zheng§

Abstract

The median problem transforms a set of N numbers in such a way that none of the first $\frac{N}{2}$ numbers exceeds any of the last $\frac{N}{2}$ numbers. A comparator network that solves the median problem on a set of r numbers is commonly called an *r-classifier*. This paper shows how the well-known Leighton's Columnsort algorithm can be modified to solve the median problem of $N = rs$ numbers, with $1 \leq s \leq r$, using an r-classifier instead of an r-sorting network. Overall the r-classifier is used $O(s)$ times, namely the same number of times that Columnsort applies an r-sorter. A hardware implementation is proposed that runs in optimal $O(s + \log r)$ time and uses an $O(r \log r(s + \log r))$ work. The implementation shows that when $N = r \log r$ there is a classifier network solving the median problem on N numbers in the same $O(\log r)$ time and using the same $O(r \log r)$ comparators as an r-classifier, thus saving a $\log r$ factor in the number of comparators over an $(r \log r)$-classifier.

Keywords: comparator network, selection network, classifier, median problem, hardware algorithm.

1 Introduction

Advances in VLSI technology have made possible to implement algorithm-structured chips as building blocks for high-performance computing systems. For example, it is widely accepted to endow general-purpose computer systems with a special-purpose parallel sorting device, invoked whenever its services are needed. The design of such a sorting chip is based on *sorting networks*, namely, networks of comparators that sort their input numbers into order.

A relevant problem closely related to sorting is the *median* problem, where one asks for classifying a set of numbers into halves, in such a way that each number in one class is at least as

*Department of Computer Science, Via Mura Anteo Zamboni, 7, University of Bologna, 40127 Bologna, ITALY, bertossi@cs.unibo.it

†Department of Computer Science, Old Dominion University, Norfolk, VA 23529-0162, USA, olariu@cs.odu.edu

‡Department of Computer Science and Telecommunications, University of Trento, 38050 Povo, Trento, ITALY, pinotti@science.unitn.it

§Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083-0688, USA, sizheng@utdallas.edu

large as all of those in the other class [5]. Solving such a problem is a frequent computation, that occurs in database monitoring to compute order statistics and approximated sorting [10], in parallel scheduling to schedule the tasks with the minimum or maximum priorities [15], and in breadth-first searching algorithms, like the M algorithm and the bidirectional algorithm, used in the decoding of convolutional codes [4]. These applications, along many others, motivate the study of *classifier networks*, that is, networks of comparators that solve the median problem, which could also be implemented as VLSI chips.

There is a wide literature on the design and analysis of sorting networks [1, 3, 7, 8, 12, 13]. Clearly, any *r-sorter* (i.e. a sorting network that sorts r input numbers) is also an *r-classifier* (i.e. a classifier network that solves the median on the same r input numbers). However, classifiers have not to do as much as sorters. Therefore, there is a rich literature also on the design and analysis of classifiers [4, 6, 9, 11, 14], which yields to simpler and more efficient networks than sorters. In fact, it is well-known that effective r-sorters are still based on Batcher's networks [3] and require $O(\log^2 r)$ time, while the existence of r-sorters taking $O(\log r)$ time is only of theoretical interest, due to the enormous constant hidden in the big-oh notation of the AKS network [1]. In contrast, there exist r-classifiers taking effectively $O(\log r)$ time, where the constant hidden in the big-oh notation is very small [6].

Leighton [8] devised a simple and effective sorting algorithm, called *Columnsort*, which repeatedly applies an r-sorter. The Columnsort algorithm sorts in column-major order a matrix $A[0 \ldots r-1, 0 \ldots s-1]$, with $r \geq 2(s-1)^2$ and $r \equiv 0 \bmod s$. It consists of 8 passes: the odd passes are sorting passes, while the even passes are data movement passes. During passes 1, 3, 5, and 7, each column of A is locally sorted by means of an r-sorter. During pass 2 the numbers of A are taken in column-major order and put back in A in row-major order, while during pass 4 the numbers of A are taken in row-major order and put back in column-major order. In passes 6 and 8, the numbers are shifted forward or backward, respectively, by $\lfloor \frac{r}{2} \rfloor$ positions. Overall, the r-sorter is applied $O(s)$ times. Note that data movement passes are used merely for the purpose of sorting only the columns, but one could properly group consecutive rows with r numbers per group and then apply the r-sorter to sort each group of rows.

The original motivation for Leighton's Columnsort algorithm was indeed that the AKS network by itself provides a means to sort r items in $O(\log r)$ time using $O(r \log r)$ comparators, but this implies that a total of $O(r \log^2 r)$ work (i.e., time $\times$ comparators) is used, which is inefficient by a factor of $\log r$. Observed that the AKS network can be pipelined, Columnsort shows how to optimally sort $N = r \log r$ numbers, arranged into an $r \times \log r$ matrix, in $O(\log r)$ time and $O(r \log^2 r)$ work.

Leighton's solution obtains an optimal work using a sorter of smaller size than that of the input. Applying a similar reasoning to the median problem, one may use an r-classifier to classify

$N \geq r$ numbers. In this case, $\Omega(N/r + \log r)$ time is needed, since no more than r numbers can be processed simultaneously and $\Omega(\log r)$ is a lower bound on the network depth [16], and $\Omega(r \log r)$ comparators are required, as proved by Alekseyev [2].

Based on the considerations above, the following natural questions arise: "Does Columnsort solve the median problem if classifiers replace sorters?", and if the answer is negative: "How should one modify Columnsort in order to efficiently solve the median problem using classifiers instead of sorters?". This would imply that the median problem can be solved using a simpler and smaller (in its constant factors) network than the AKS sorting network, and thus raising a new question: "Given the ability to use an r-classifier for solving the median problem of r numbers in $O(\log r)$ depth using $O(r \log r)$ comparators, is it possible to derive a circuit that can find the median of $N = r \log r$ numbers using the same asymptotic depth and number of comparators as an r-classifier?" In the affirmative case, there would be a circuitry for N numbers which has size smaller by a $\log r$ factor than that of an N-classifier network.

In this paper answers to the above questions are given. As a preliminary, let the behavior of Columnsort be briefly analyzed when used to solve the median problem still applying an r-sorter. After pass 4 of Columnsort, every number is within $(s-1)^2$ of its correct sorted position [8]. Since $r \geq 2(s-1)^2$, the numbers of A are already separated, except perhaps either those in the central column, if s is odd, or those in the lowest half and in the highest half of the two central columns, if s is even. More formally, in the third sorting pass one only needs either to sort column $A[*, \frac{s-1}{2}]$, if s is odd, or to sort $A[\frac{r}{2} \ldots r-1, \frac{s}{2}-1] \cup A[0 \ldots \frac{r}{2}-1, \frac{s}{2}]$, if s is even (hereafter, $A[*, j]$ and $A[i, *]$ denote column j and row i of A, respectively, while $A[i \ldots h, j \ldots k]$ denotes the submatrix of A given by the specified rows and columns). Thus, overall, 3 sorting passes, instead of 4, are enough for Columnsort to solve the median problem using an r-sorter.

In order to answer to the first question, consider now what happens if one tries to solve the median problem still using Columnsort, but substituting the r-sorter with an r-classifier. In the following, a counterexample, built on a particular matrix A, is exhibited where Columnsort fails because the median number remains in its original position in the wrong half. Such a counterexample can be generalized to a matrix A with arbitrary size $r \times s$ in such a way that the median number can be hidden virtually in any position of the wrong half of A. Let $r = 54$, $s = 6$ and let the input numbers be all the integers between 1 and $rs = 324$. Consider the sequence of the above numbers sorted from 1 to 324. Remove from the sequence the median number 162 and its successor 163, and place 163 between 53 and 54 and 162 between 217 and 218. Then, store the modified sequence in A in column-major order, and apply the Columnsort algorithm using the 54-classifier, instead of the 54-sorter. The 54-classifier merely separates the 27 smallest numbers from the 27 largest numbers, but no particular order within each half can be assumed. In particular, applying the 54-classifier in each odd pass, each column of A may remain unchanged. If this is the case, it is

easy to see that the set of numbers is transformed in such a way that 162 and 163 remain in their original positions. Therefore Columnsort does not solve the median problem since the numbers 162 and 163 remain, respectively, in the second half and in the first half of A.

Thus, the answer to the first question, "Does Columnsort work if classifiers replace sorters?", is negative. In order to give answers to the other two questions, the rest of this paper is structured in two parts.

The first part consists of Sections 2 and 3 which provide a high level description of two algorithms that show how Columnsort can be modified to solve the median problem of $N = rs$ numbers, with $1 \leq s \leq r$, using as a basic operation the r-classifier invocation. In details, Section 2 describes the *Row-Column-Selection* algorithm which takes a logarithmic number of passes to solve the median problem on a matrix A of size $r \times s$, with $r = 2^k$ and any $s = 2^h$, where $1 \leq h \leq k$, using an r-classifier (when $h = k$ such an algorithm works also on a square matrix). Although the number of passes is $O(\log s)$ for Row-Column-Selection and $O(1)$ for Columnsort, both such algorithms apply their comparator network, namely an r-classifier or an r-sorter, respectively, the same number of times, that is $O(s)$. Section 3 describes another algorithm, called *Three-Pass-Selection*, which solves in 3 passes the median problem on a matrix A of size $r \times s$, with $1 \leq s \leq \sqrt{\frac{r}{2}}$. This algorithm shows that, replacing the sorter with the classifier in the Columnsort algorithm, the median problem can be solved still maintaining a constant number of passes as long as s is bounded by $O(\sqrt{r})$. Therefore, both Three-Pass-Selection and Row-Column-Selection provide an answer to the second question, "How should one efficiently modify Columnsort using classifiers instead of sorters?" Moreover, they are based on the simpler r-classifier by Jimbo and Maruoka [6], which takes effectively $O(\log r)$ time, whereas, to obtain the same time performance, Columnsort has to employ the ineffective r-sorter by Ajtai, Komlos, and Szemeredi [1].

The second part of this paper consists of Section 4, which presents a hardware algorithm, based on both the Three-Pass-Selection and Row-Column-Selection algorithms, which solves the median problem on $N = rs$ numbers, with $1 \leq s \leq r$. Such an algorithm achieves an optimal $O(s + \log r)$ time and uses $O(r \log r)$ comparators. Overall, $O(rs \log r + r \log^2 r)$ work is done which, when $s = \Omega(\log r)$, is better by a factor of $\log r$ than the $O(N \log^2 N) = O(rs \log^2 r)$ work used by an N-classifier. In particular, when the number s of columns of A is $O(\log r)$, the hardware algorithm gives an affirmative answer to our third question, showing that it is possible to build a classifier network that can find the median of $O(r \log r)$ numbers in the same $O(\log r)$ time and using the same $O(r \log r)$ number of comparators as an r-classifier. The hardware algorithm strictly enforces conflict-free memory accesses and is based on a simple architecture, feasible for VLSI implementation.

2 Row-Column-Selection Algorithm

The goal of this section is to solve the median problem on a set of $N = rs$ numbers, with $1 \leq s \leq r$, stored in a matrix $A[0 \ldots r-1, 0 \ldots s-1]$, using an optimal number of applications of an r-classifier.

For the sake of simplicity, in this section, both r and s are assumed to be powers of two, that is, $r = 2^k$ and $s = 2^h$, for any $1 \leq h \leq k$.

The building block of the algorithm is a 4-*Partition* procedure which receives in input $2r$ numbers, grouped as C_0, C_1, C_2, C_3, each of size $r/2 = 2^{k-1}$, and partitions the numbers such that all the numbers in C_j are not larger than those in C_{j+1}, with $0 \leq j \leq 3$.

The *Row-Column-Selection* algorithm consists of $h = \log s$ recursions, that reduce the number of columns from s to 2.

At each recursion, Row-Column-Selection halves the number c of columns of the previous recursion and perceives A as composed by 4 submatrices A_1, A_2, A_3, A_4, each of size $r/2 \times c/2$, as depicted in Figure 1. In particular, the i-th recursion works on a matrix A of size $r \times c$, where $c = s/2^i$, and $0 \leq i \leq \log s - 1$ (clearly, when $i = 0$, $c = s$).

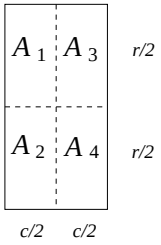

Figure 1: The submatrices A_1, A_2, A_3, A_4 in which A is decomposed.

Each recursion executes two passes: a *row-pass*, followed by a *column-pass*. The row-pass examines A row-by-row, applying $c/2$ times the 4-Partition procedure to groups of $2r/c$ rows. The column-pass examines A column-by-column, applying $c/2$ times the 4-Partition procedure to pairs of columns. At the end of a recursion, the numbers belonging to A_1 are followed by all the numbers in $A_3 \cup A_4$, while those belonging to A_4 are preceded by all the numbers in $A_1 \cup A_2$. Therefore, the numbers in A_1 belong to the $rc/2$ smallest numbers of A, while those in A_4 belong to the $rc/2$ largest numbers of A. The algorithm is, then, recursively applied to the halved sub-matrix of A consisting of A_2 and A_3.

In order to formally describe the above algorithm, the details of how to perform the 4-Partition procedure using the r-classifier are given first.

The 4-Partition procedure, shown in Figure 2, invokes 6 times the r-classifier. In each invocation, r-classifier (C, C') receives r numbers, grouped as C and C', each of size $r/2$, and rearranges them

procedure 4-Partition (C_0, C_1, C_2, C_3)
begin
0. r-classifier (C_0, C_1);
1. r-classifier (C_2, C_3);
2. r-classifier (C_1, C_2);
3. r-classifier (C_0, C_3);
4. r-classifier (C_0, C_1);
5. r-classifier (C_2, C_3);

end

Figure 2: The 4-Partition procedure.

so that every number in C is followed by all the numbers in C'. The 4-Partition procedure executes a computation similar to an *odd-even sort* on 4 items, where each comparison is replaced with a call to the r-classifier. However, the advantage of this procedure versus odd-even sort is that calls $2i$ and $2i+1$, with $0 \leq i \leq 2$, could be performed simultaneously, for a total of 3 parallel phases, thus saving 1 parallel phase over odd-even sort.

Lemma 1 *The 4-Partition procedure partitions the $2r$ input numbers into 4 groups C_0, C_1, C_2, C_3, each of size $r/2 = 2^{k-1}$, such that all the numbers in C_j are not larger than those in C_{j+1}, with $0 \leq j \leq 3$.*

Proof. Let a and b be the medians of $C_0 \cup C_1$ and $C_2 \cup C_3$, respectively. After invoking the r-classifier on $C_0 \cup C_1$ and on $C_2 \cup C_3$, a and b belong to C_0 and C_2, respectively. If $a \leq b$, C_0 is followed by all the numbers in $C_1 \cup C_3$ and C_3 is preceded by all the numbers in $C_0 \cup C_2$. Therefore, C_1 and C_2 need to be separated. Instead, if $b < a$, C_2 is followed by all the numbers in $C_1 \cup C_3$, while C_1 is preceded by all the numbers in $C_0 \cup C_2$, and thus C_0 and C_3 need to be separated. After invoking r-classifier (C_1, C_2) and r-classifier (C_0, C_3), the median problem on the input set has been solved, that is all the numbers in $C_0 \cup C_1$ precede those in $C_2 \cup C_3$. Therefore, to obtain the 4 partition, it is enough to invoke again the r-classifier on $C_0 \cup C_1$ and on $C_2 \cup C_3$. □

At each recursion, the 4-Partition procedure is repeatedly called by the *Row-Column-Pass* procedure, as illustrated in Figure 3.

The row-pass consists of $c/2$ iterations: during the i-th iteration, the numbers belonging to r/c consecutive rows of $A_1, \ldots, A_4$ are rearranged into 4 groups in such a way that the numbers in

procedure Row-Column-Pass (A, r, c)
begin
/* Row-Pass */
  for $i = 0$ **to** $\frac{c}{2} - 1$ **do**
    4-Partition($A[i\frac{r}{c} \dots (i+1)\frac{r}{c} - 1, 0 \dots \frac{c}{2} - 1]$, $A[\frac{r}{2} + i\frac{r}{c} \dots (i+1)\frac{r}{c} - 1, 0 \dots \frac{c}{2} - 1]$,
      $A[i\frac{r}{c} \dots (i+1)\frac{r}{c} - 1, \frac{c}{2} \dots c - 1]$, $A[\frac{r}{2} + i\frac{r}{c} \dots (i+1)\frac{r}{c} - 1, \frac{c}{2} \dots c - 1]$);
  endfor
/* Column-Pass */
  for $i = 0$ **to** $\frac{c}{2} - 1$ **do**
    4-Partition($A[0 \dots \frac{r}{2} - 1, i]$, $A[\frac{r}{2} \dots r - 1, i]$, $A[0 \dots \frac{r}{2} - 1, i + \frac{c}{2}]$, $A[\frac{r}{2} \dots r - 1, i + \frac{c}{2}]$);
  endfor
end

Figure 3: The Row-Column-Pass procedure.

$A[i\frac{r}{c} \dots (i+1)\frac{r}{c} - 1, 0 \dots \frac{c}{2} - 1]$ are not larger than those in $A[\frac{r}{2} + i\frac{r}{c} \dots (i+1)\frac{r}{c} - 1, 0 \dots \frac{c}{2} - 1]$, which are followed by $A[i\frac{r}{c} \dots (i+1)\frac{r}{c} - 1, \frac{c}{2} \dots c - 1]$, which in their turn are not larger than $A[\frac{r}{2} + i\frac{r}{c} \dots (i+1)\frac{r}{c} - 1, \frac{c}{2} \dots c - 1]$).

Similarly, the column-pass consists of $c/2$ iterations: during the i-th iteration, with $0 \leq i \leq c/2 - 1$, the $2r$ numbers stored in the two columns $A[*, i]$ and $A[*, i + c/2]$ are rearranged in such a way that all the numbers of $A[i\frac{r}{c} \dots (i+1)\frac{r}{c} - 1, 0 \dots \frac{c}{2} - 1]$ are not larger than those in $A[\frac{r}{2} + i\frac{r}{c} \dots (i+1)\frac{r}{c} - 1, 0 \dots \frac{c}{2} - 1]$, which are followed by those in $A[0 \dots \frac{r}{2} - 1, i + \frac{c}{2}]$, which in their turn are not larger than $A[\frac{r}{2} \dots r - 1, i + \frac{c}{2}]$.

The Row-Column-Pass procedure requires $O(c)$ applications of the r-classifier network, since there are c iterations and each iteration invokes the 4-Partition procedure. It is worthy to note that different iterations work on groups of disjoint rows or columns, and therefore, as it will be discussed in Section 4, all the $c/2$ iterations of the row-pass or column-pass could be performed in pipeline.

Figure 4 summarizes the *Row-Column-Selection* algorithm, which works for any $s = 2^h$, with $1 \leq h \leq \log r$, and hence even when A is a square $r \times r$ matrix. Its correctness comes from the following lemma.

Lemma 2 *At the end of the Row-Column-Pass procedure,*

- *no number in A_1 is larger than any number in $A_3 \cup A_4$, and*
- *no number in A_4 is smaller than any number in $A_1 \cup A_2$.*

algorithm Row-Column-Selection $(A,\ r,\ s)$
begin
/* let A be perceived as the 4 matrices A_1, A_2, A_3, A_4 shown in Figure 1 */
  if $s = 2$ **then**
    4-Partition (A_1, A_2, A_3, A_4)
  else
    Row-Column-Pass $(A,\ r,\ s)$;
    Row-Column-Selection $(A_2 \cup A_3,\ r,\ \frac{s}{2})$
  endif
end

Figure 4: The Row-Column-Selection algorithm.

Proof. Only the first claim is proved, as the proof of the second one follows from a mirror argument.

By contradiction, let $u \in A_1$ be strictly larger than one number $v \in A_3 \cup A_4$. Assume u and v belong, respectively, to columns $A[*, i]$ and $A[*, j]$, where $0 \leq i \leq c/2 - 1$ and $c/2 \leq j \leq c - 1$, and $j \neq i + c/2$.

Since the columns were separated in 4 parts, there are at least $\frac{3}{2}r + 1$ numbers not smaller than u in columns $A[*, i] \cup A[*, i + c/2]$, and at least $r + 1$ numbers not larger than v in columns $A[*, j - c/2] \cup A[*, j]$. Let U and V denote these sets of numbers, respectively. Observe that all the numbers that at the end of the row-pass belonged to the pair of columns i and $i + c/2$ remain in the same pair of columns at the end of the column-pass. Moreover, at the end of the row-pass, for a fixed $q \in [0, c/2 - 1]$, the number stored in $A_1[p, i]$ is not larger than all the $\frac{3}{2}r$ numbers in $A_2[\frac{r}{2} + p, *]$, $A_3[p, *]$, and $A_4[\frac{r}{2} + p, *]$, where $q\frac{r}{c} \leq p \leq (q + 1)\frac{r}{c} - 1$. Generalizing to the other submatrices, at the end of the row-pass, the number stored in $A_t[p, i]$ is not larger than the $\frac{(4-t)}{2}r$ numbers in $A_{t+1}[((t - 1) \bmod 2)\ \frac{r}{2}+p, *], \ldots, A_4[\frac{r}{2}+p, *]$, where $1 \leq t \leq 4$ and $q\frac{r}{c} \leq p \leq (q+1)\frac{r}{c}-1$.

Consider now the elements U in the pair of columns i and $i + c/2$ of A. By the previous observations, each element in U implies that there are some elements not smaller than u on columns $j - c/2$ and j, and therefore forbids some positions for the elements V in such columns. The number of forbidden positions for V is minimized when U contains all the elements in $A_4[*, i+c/2]$, $A_3[*, i + c/2]$, and $A_2[*, i]$. Hence, altogether U forbids at least r positions for the elements V in columns $j - c/2$ and j. Since there are $2r$ positions available, out of which r are forbidden, only r

positions are available for the elements in V.

This shows that $u \geq v$ is impossible. In conclusion, every element in A_1 is followed by all the elements in A_3 and A_4. □

Consider now how many applications $C(s)$ of the r-classifier are required. Since the Row-Column-Pass procedure requires as many r-classifier applications as the number of columns in A, which halves at each recursion, the relation holds

$$C(s) = C\left(\frac{s}{2}\right) + O(s),$$

whose solution is $C(s) = O(s)$.

Although the number of passes is $O(\log s)$ for Row-Column-Selection and $O(1)$ for Columnsort, both such algorithms apply their comparator network, namely an r-classifier or an r-sorter, respectively, the same number of times, that is $O(s)$.

3 Three-Pass-Selection Algorithm

The goal of this section is to show how the Row-Column-Selection algorithm can be modified to solve the median problem in 3 passes when the number of columns is $O(\sqrt{r})$.

Again, consider the median problem on a set of $N = rs$ numbers, stored in a rectangular matrix $A[0 \ldots r-1, 0 \ldots s-1]$, with r rows and s columns, where it is assumed that $1 \leq s \leq \sqrt{\frac{r}{2}}$ and $\frac{r}{2s}$ is an integer. The algorithm presented in this section, called *Three-Pass-Selection algorithm* , perceives A composed by 4 submatrices A_1, A_2, A_3, A_4. With respect to the Row-Column-Selection algorithm, however, such matrices are arranged in a different way, as depicted in Figure 5. The submatrix A_1 consists of the uppermost $\frac{r}{2} - s$ rows of A, $A[0, *], A[1, *], \ldots, A[\frac{r}{2} - s - 1, *]$, A_2 of the s rows $A[\frac{r}{2} - s, *], \ldots, A[\frac{r}{2} - 1, *]$, A_3 of the s rows $A[\frac{r}{2}, *], \ldots, A[\frac{r}{2} + s - 1, *]$, and finally A_4 by the lowermost $\frac{r}{2} - s$ rows $A[\frac{r}{2} + s, *], \ldots, A[r - 1, *]$.

The Three-Pass-Selection algorithm works in three passes: a *row-pass*, which examines A row-by-row, a *column-pass*, which explores A by columns, and a final pass on $A_2 \cup A_3$.

The row-pass consists of s iterations: during the i-th iteration, with $0 \leq i \leq s - 1$, the r numbers stored in the submatrix $G_i = A[i\frac{r}{s} \ldots (i+1)\frac{r}{s} - 1, *]$ are rearranged into 4 submatrices $G_{i,0}, \ldots, G_{i,3}$ of $\frac{r}{2s} - 1, 1, 1$, and $\frac{r}{2s} - 1$ rows, respectively, in such a way that all the numbers in $G_{i,j}$ are no larger than all the numbers in $G_{i,j+1}$, with $0 \leq j \leq 2$. Specifically, as depicted in Figure 6, $G_{i,0}$ consists of the first $\frac{r}{2s} - 1$ rows of G_i, that is $G_{i,0} = A[i\frac{r}{s} \ldots i\frac{r}{s} + \frac{r}{2s} - 2, *]$, $G_{i,1} = A[i\frac{r}{s} + \frac{r}{2s} - 1, *]$, $G_{i,2} = A[i\frac{r}{s} + \frac{r}{2s}, *]$, and finally $G_{i,3}$ consists of the last $\frac{r}{2s} - 1$ rows of G_i, namely $G_{i,3} = A[i\frac{r}{s} + \frac{r}{2s} + 1 \ldots (i+1)\frac{r}{s} - 1, *]$. This partitioning operation is performed invoking the 4-*Skewed-Partition* procedure on G_i.

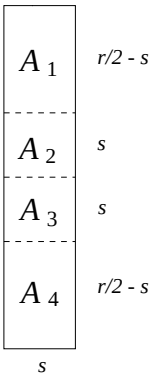

Figure 5: The submatrices A_1, A_2, A_3, A_4 in which A is decomposed.

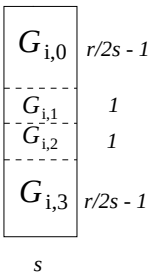

Figure 6: The submatrices $G_{i,0}, G_{i,1}, G_{i,2}, G_{i,3}$ in which G_i is partitioned.

The column-pass consists also of s iterations: during the i-th iteration, the r numbers of column $A[*, i]$ are rearranged, again by invoking the procedure 4-Skewed-Partition, into four groups of size $\frac{r}{2} - s, s, s$, and $\frac{r}{2} - s$, respectively, in such a way that all the numbers in column i of the submatrix A_j, that is $A_j[*, i]$, are followed by those in column i of A_{j+1}, namely $A_{j+1}[*, i]$, for $1 \leq j \leq 3$.

The final pass considers the $2s^2$ numbers stored in A_2 and A_3, which are rearranged, by the 2-*Partition* procedure, in such a way that all the s^2 numbers in A_2 are followed by the s^2 numbers in A_3.

In conclusion, Figure 7 summarizes the Three-Pass-Selection algorithm (the implementations of the procedures 4-Skewed-Partition and 2-Partition will be discussed later).

The correctness results from the following lemma.

Lemma 3 *At the end of the second pass of the Three-Pass-Selection algorithm ,*

- *no number in A_1 is larger than any number in $A_3 \cup A_4$, and*
- *no number in A_4 is smaller than any number in $A_1 \cup A_2$.*

algorithm Three-Pass-Selection (A, r, s)
begin
/* let A be perceived as the 4 matrices A_1, A_2, A_3, A_4 shown in Figure 5 */
/* Row-Pass */
  for $i = 0$ **to** $s - 1$ **do**
    4-Skewed-Partition$(G_{i,0}, G_{i,1}, G_{i,2}, G_{i,3})$;
  endfor
/* Column-Pass */
  for $i = 0$ **to** $s - 1$ **do**
    4-Skewed-Partition$(A_1[*, i], A_2[*, i], A_3[*, i], A_4[*, i])$;
  endfor
/* Final-Pass */
  2-Partition(A_2, A_3);
end

Figure 7: The Three-Pass-Selection algorithm .

Proof. The proof is similar to that of Lemma 2. As before, only the first claim is proved.

By contradiction, let $u \in A_1$ be strictly larger than one number $v \in A_3 \cup A_4$. Assume u and v belong, respectively, to columns $A[*, i]$ and $A[*, j]$, where $0 \leq i \neq j \leq s - 1$. Since the columns were separated in 4 parts, there are at least $\frac{r}{2} + s + 1$ numbers not smaller than u in column $A[*, i]$, and at least $\frac{r}{2} + 1$ numbers not larger than v in column $A[*, j]$. Let U and V denote these sets of numbers, respectively. Observed that all the elements that at the end of the row-pass belonged to column i remain in the same column at the end of the column-pass, let denote with $F_{k,t}$, where $0 \leq k \leq s - 1$ and $0 \leq t \leq 3$, the numbers at the end of the row-pass were stored in column i of $G_{k,t}$.

Hence, every number e that belongs to $F_{k,2}$ is followed by $\frac{r}{2s} - 1$ rows of G_k whose numbers are not smaller than e at the end of the row-pass, as well as at the end of the column-pass. Similarly, every number e that belongs to $F_{k,1}$ is followed by $\frac{r}{2s}$ rows of G_k, and therefore by one more row of G_k with respect to $F_{k,2}$. Finally, every e belonging to $F_{k,0}$ is followed by $\frac{r}{2s} + 1$ rows of G_k, and thus by one more row of G_k with respect to $F_{k,1}$.

Consider now the set U of numbers in column i. By the previous observation, each number in U forces on the other columns of A, and especially on column j, some numbers not smaller than u, and therefore it forbids some positions for the numbers V in column j. The amount of forbidden

positions for V is minimized when U contains all the numbers in

$$\bigcup_{\substack{0 \le k \le s-1 \\ 1 \le t \le 3}} F_{k,t}$$

Altogether U forbids at least $\frac{r}{2}$ positions for the numbers V in column j. Since there are r positions available, out of which $\frac{r}{2}$ are forbidden, only $\frac{r}{2}$ positions are available for the numbers in V.

This shows that $u \ge v$ is impossible. In conclusion, every number in A_1 is followed by all the numbers in A_3 and A_4, and therefore A_1 cannot contain the median of A. □

Note that the Three-Pass-Selection algorithm requires only 3 passes because after executing the row-pass and the column-pass, only $2s^2 \le r$ numbers remain in A_2 and A_3 and they can be separated by a single r-classifier application. Note that, if $2s^2 < r$, it is sufficient to fill the r-classifier with any additional $\frac{r}{2} - s^2$ numbers taken from A_1 and any additional $\frac{r}{2} - s^2$ numbers taken from A_4.

It is worth noting that both the Three-Pass-Selection and Columnsort algorithms uses a constant number of passes and apply their comparator network (namely, a classifier or a sorter, respectively) the same number of times, that is $O(s)$.

In order to implement the 4-Skewed-Partition procedure, observe that an r-classifier alone can only partition r numbers into two halves, each of size $\frac{r}{2}$, such that all the numbers of the first half are not larger than those of the second half. Therefore, to accomplish the final goal of the procedure, one needs to extract the s largest numbers of the first half, and the s smallest numbers of the second half. This can be achieved using a classifier device a bit more complex than a simple r-classifier, as it will be shown in the next section, which however has the same asymptotic depth and size as the simple r-classifier.

As regard to the time complexity, note that different iterations of the row-pass and column-pass work on groups of disjoint rows and columns, respectively. Therefore all the s iterations of the same pass can be performed in pipeline. In the next section, a hardware algorithm, for $1 \le s \le r$, will be devised that combines the advantages of both the Three-Pass-Selection and Row-Column-Selection algorithms and achieves an optimal $O(\log r + s)$ time using an $O\left(r \log r(\log r + s)\right)$ work.

4 Hardware Implementations

In this section, a hardware algorithm is presented for solving the median problem on $N = rs$ numbers, where $1 \le s \le r$ (for the sake of simplicity, it is assumed that r, $\log r$ and s are powers of two). Such an algorithm combines the paradigms of both the Three-Pass-Selection and Row-Column-Selection algorithms. Specifically, the hardware algorithm, called *Combine-Selection*,

algorithm Combine-Selection $(A,\ r,\ s)$
begin
  if $s = \log r$ **then**
    Three-Pass-Selection $(A,\ r,\ s)$
  else
    Row-Column-Pass $(A,\ r,\ s)$;
    Combine-Selection $(A_2 \cup A_3,\ r,\ \frac{s}{2})$
  endif
end

Figure 8: The high level description of the hardware algorithm.

behaves recursively as Row-Column-Selection while the number of columns remains larger than $\log r$, but it acts as Three-Pass-Selection as soon as the number of columns becomes $\log r$, as illustrated in Figure 8.

First, an architectural framework is exhibited which consists of r memory modules and a slightly modified r-classifier network, which includes also some simple networks for performing maximum and minimum computations. Then, pipeline schemes are presented for all the proposed algorithms, which read/write a row or a column of A from/to the memory modules in constant time and achieve optimal time performance.

4.1 Architecture

Figure 9 depicts the architecture for $r = 8$. The basic architectural features of the design include:

(i) A *data memory* organized into r independent memory modules $M_0, M_1, \ldots, M_{r-1}$. Each memory module M_i is randomly addressed by an address register AR_i, associated with an adder. All the registers AR_i's can be loaded simultaneously by addresses broadcast from the control unit. When all such addresses are the same, the r locations addressed simultaneously are referred to as a *memory line*.

(ii) A set of data registers, R_i, $0 \leq i \leq r-1$, each capable of containing a number, whose purpose is to interface the extended r-classifier device with the memory modules. In constant time, the r elements in the data registers can be loaded in parallel into the addressed registers, or can be stored in parallel into the r modules addressed by address registers.

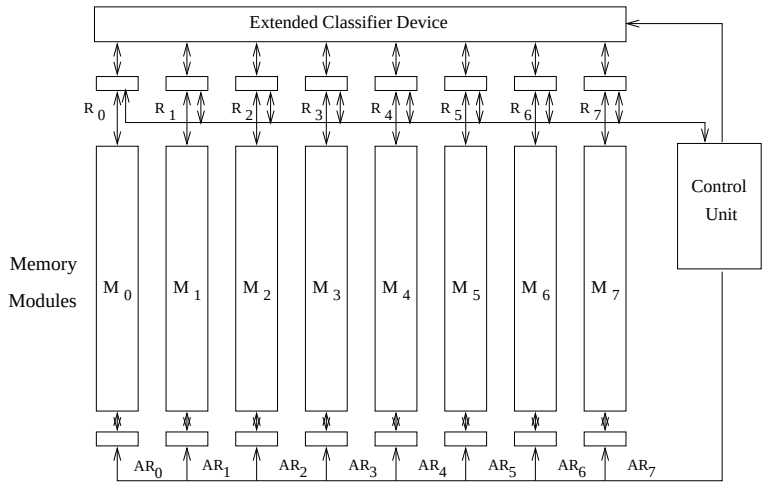

Figure 9: The proposed architecture with $r = 8$.

(iii) An extended classifier device consisting of:

- an r-*classifier* network of I/O size r and depth $O(\log r)$;
- $\log r$ *maximum* networks $max_0, \ldots, max_{\log r - 1}$; for $0 \leq i \leq \log r - 1$, each max_i has $O(r)$ comparators and $O(\log r)$ depth, is capable of performing a maximum computation, and is equipped with a register MR_i, which can store $\log r$ numbers;
- $\log r$ *minimum* networks $min_0, \ldots, min_{\log r - 1}$; for $0 \leq i \leq \log r - 1$, each min_i has $O(r)$ comparators and $O(\log r)$ depth, is capable of performing a minimum computation, and is equipped with a register mR_i, which can store $\log r$ numbers;
- a *demultiplexer* to route the outcome of the r-classifier either to a suitable max_i/min_i or to the memory;
- an $(r - 2\log r)$-*classifier* network of I/O size $r - 2\log r$ and depth $O(\log r)$.

The structure of the extended classifier device is illustrated in Figure 10 for $r = 16$ and $\log r = 4$.

(iv) A *control unit* (CU, for short), consisting of a *control processor* capable of performing simple arithmetic and logic operations and of a *control memory* used to store the control program as well as the control data. The CU generates control signals for the demultiplexer, for the r- and $(r - 2\log r)$-classifiers, for the registers and for memory accesses. The CU can broadcast an address to all memory modules and to the data registers, and can read an element from any data registers. These operations are assumed to take constant time.

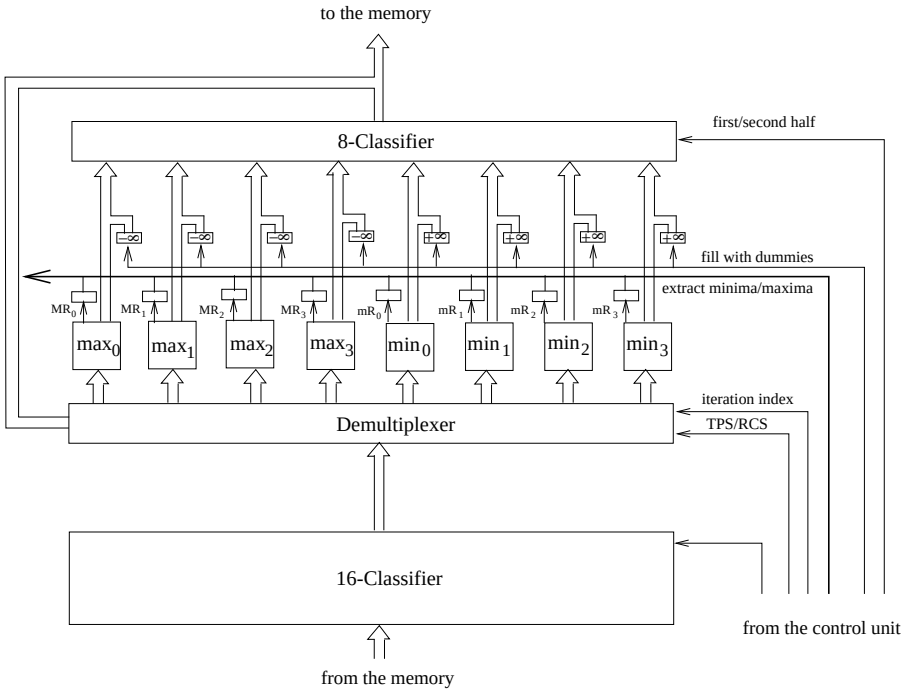

Figure 10: The structure of the extended classifier for $r = 16$ and $\log r = 4$.

To achieve high performance for the hardware implementation, the r-classifier must be filled at each instant with a new set of r numbers. This can be accomplished only if *conflict-free* access is guaranteed to the memory modules storing the rows and the columns of A, namely only when all the r elements of the same row or column can be simultaneously read from or written to the r memory modules in constant time. Hereafter it is assumed that A is stored in such a way that each column of A forms a memory line, namely, it is kept in r memory locations having the same address in all the modules. Precisely, the generic element $A[i, j]$ of column j is stored in position j of module M_i. However, in this way, each row is stored in the same memory module. Therefore the elements of the same row cannot be retrieved conflict-free, but must be accessed one by one, requiring a time linear in the row length. To overcome this drawback, the hardware implementations of the proposed algorithms replace access to rows with access to diagonals. This does not hurt Lemmas 2 and 3 whose proofs are based on a counting argument consisting of how many numbers of a row intersect a column. Since such a quantity remains the same when replacing rows with diagonals, the correctness of the hardware implementation of the row-passes is guaranteed.

4.2 Implementation of Three-Pass-Selection

In this subsection, a detailed description of the hardware implementation of the Three-Pass-Selection algorithm is given, only in the case of interest for Combine-Selection, that is when the number of columns is exactly $\log r$.

The building block of the Three-Pass-Selection algorithm is the 4-Skewed-Partition procedure, which works on r numbers, corresponding either to a subset of $\frac{r}{\log r}$ consecutive rows or to a single column of A, depending whether a row-pass or a column-pass is performed.

During the i-th iteration of the row-pass, the r-classifier is filled up with the $\frac{r}{\log r}$ rows of $G_i = A[i\frac{r}{\log r} \ldots (i+1)\frac{r}{\log r} - 1, *]$ in order to separate them into the 4 submatrices $G_{i,0}, \ldots, G_{i,3}$ of $\frac{r}{2\log r} - 1, 1, 1$, and $\frac{r}{2\log r} - 1$ rows, respectively. Since as said before the hardware implementation accesses conflict-free diagonals instead of rows, the generic element $A[i\frac{r}{\log r} + h, k]$, belonging to the submatrix G_i, is retrieved from and stored back by the CU in position k of the memory module $M_{h\log r + (k+i) \bmod \log r}$, where $0 \leq h \leq \frac{r}{\log r} - 1$ and $0 \leq k \leq \log r - 1$. Then, in this way, during the i-th iteration of the row-pass, each classifier call can access r locations conflict-free, one for each memory module.

During the i-th iteration of the column-pass, the four groups accessed by the 4-Skewed-Partition procedure correspond to a single column, which is stored in memory line i, whose elements can be retrieved and stored back by the CU without memory conflicts.

The $\log r$ iterations of the row-pass or column-pass are performed in pipeline, starting the i-th iteration at time instant i. Since the r-classifier network works in $C\log r$ time, where $C \approx 2$ [6], the r-classifier ends to handle the i-th iteration at time $i + C\log r$, with $0 \leq i \leq \log r - 1$.

During the generic i-th iteration of the row-pass or column-pass, the smallest $r/2$ numbers, output by the r-classifier, are given in input to the max_i network, while the largest $r/2$ numbers are given to the min_i network. In conclusion, the pair of networks max_i and min_i is devoted to the single i-th iteration. Then, for $\log r$ times, max_i (resp., min_i) extracts in pipeline the maximum (resp., minimum), stores it in its associated MR_i (resp., mR_i) register, and replaces the extracted value with a dummy $-\infty$ (resp., $+\infty$) value. In particular, max_i (resp., min_i) extracts the first maximum (resp., minimum) at time $i + C\log r + \log r$, and it extracts the $\log r$-th maximum (resp., minimum) of the same iteration at time $i + C\log r + 2\log r - 1$. Subsequently, the CU fills the $(r - 2\log r)$-classifier twice with the content of the minimum/maximum network, in order to clean the significant $\frac{r}{2} - \log r$ values from the $\log r$ dummy values. At instant $i + C\log r + 2\log r$, the CU fills the $(r - 2\log r)$-classifier network with the $\frac{r}{2}$ numbers still stored in max_i along with additional $\frac{r}{2} - 2\log r$ dummy $-\infty$ values. At the next instant, the CU fills the classifier with the other numbers stored in min_i along with additional $\frac{r}{2} - 2\log r$ dummy $+\infty$ values. Hence, every 2 instant of time the $(r - 2\log r)$-classifier network is filled with the content of a different pair of maximum and minimum comparator networks, which correspond to different iterations of

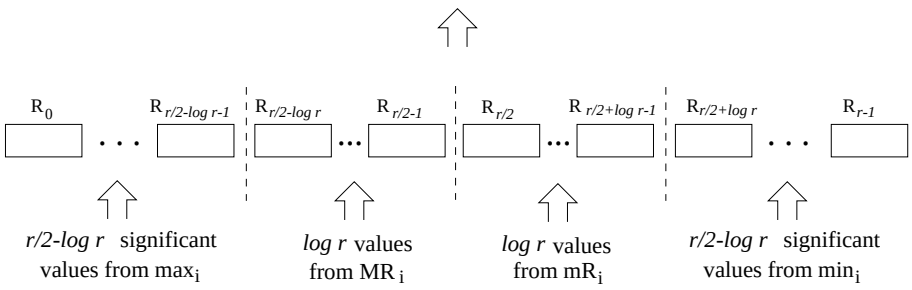

Figure 11: The loading of the data registers at the end of the i-th iteration of a row-pass or column-pass.

the same row-pass or column-pass. Once the $(r - 2\log r)$-classifier has separated the content of max_i, the CU moves the $\frac{r}{2} - \log r$ largest numbers (i.e., the significant values of max_i) to the data registers $R_0, \ldots, R_{\frac{r}{2}-\log r-1}$. At the next instant the content of min_i has been separated, and the CU moves the $\frac{r}{2} - \log r$ smallest numbers (i.e., the significant values of min_i) to $R_{\frac{r}{2}+\log r}, \ldots, R_{r-1}$. Moreover, the CU also moves the $\log r$ numbers already stored in MR_i to $R_{\frac{r}{2}-\log r}, \ldots, R_{\frac{r}{2}-1}$, and those stored in mR_i to $R_{\frac{r}{2}}, \ldots, R_{\frac{r}{2}+\log r-1}$. The loading of the data registers is shown in Figure 11. Hence, the i-th iteration is concluded storing back conflict-free the content of the data registers into the memory.

Observe that the $(r - 2\log r)$-classifier works in pipeline on all the $\log r$ iterations of the same row-pass or column-pass producing the output of the same iteration in two subsequent instants. Therefore, since the $(r - 2\log r)$-classifier works in $O(\log r)$ time, overall $O(\log r)$ time is taken to accomplish an entire row-pass or column-pass.

The final pass of the Three-Pass-Selection algorithm is implemented filling in $O(\log r)$ time the r-classifier network with the $2\log^2 r$ numbers of $A_2 \cup A_3$ along with any additional $\frac{r}{2} - \log^2 r$ numbers taken from A_1 and any additional $\frac{r}{2} - \log^2 r$ numbers taken from A_4. A single application of the r-classifier accomplishes the separation required by the final pass. Thus, the final pass takes time $O(\log r)$.

Note that to compute the actual median number of the entire matrix A, it is enough to extract the maximum from the smallest $\frac{r}{2}$ elements output by the final pass. This can be accomplished in $O(\log r)$ time by using any maximum comparator network max_i.

Overall, the hardware implementation of the Three-Pass-Selection algorithm takes $O(\log r)$ time, to solve the problem on $N = r\log r$ numbers, which is optimal due to the lower bound given in [16]. In order to evaluate the work, observe that each of the $2\log r$ maximum and minimum

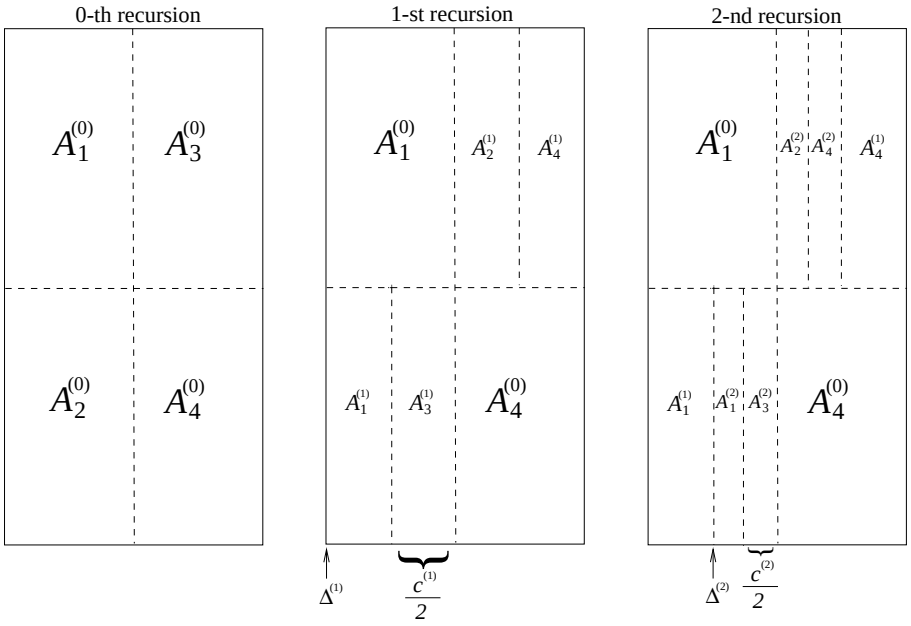

Figure 12: In place partitioning of A at the j-th recursion, with $j = 0, 1, 2$.

networks is implemented by a tree of $O(r)$ comparators, for a total of $O(r \log r)$ comparators. Since both the r-classifier and the $(r - 2\log r)$-classifier employ $O(r \log r)$ comparators, a total work of $O(r \log^2 r)$ is used. Note that the above hardware implementation gives an affirmative answer to our third question, showing that it is possible to build a classifier network that can find the median of $r \log r$ numbers in the same $O(\log r)$ time and using the same $O(r \log r)$ comparators as an r-classifier.

4.3 Implementation of Combine-Selection

This subsection describes the hardware implementation of the Combine-Selection algorithm which behaves as Row-Column-Selection to reduce the number of columns from s to $\log r$, and then invokes Three-Pass-Selection.

At first, it is shown how the recursion on the submatrix $A_2 \cup A_3$ in Figure 8 is realized. The algorithm proceeds partitioning the numbers of A as depicted in Figure 12.

The first call within the main program to Combine-Selection is an anomalous recursion, denoted hereafter as 0-th recursion. The 0-th recursion partitions A into $A_1^{(0)}, A_2^{(0)}, A_3^{(0)}$ and $A_4^{(0)}$ as described in Section 2. In the subsequent recursions, the hardware implementation works in place on the submatrices A_2 and A_3 obtained from the previous recursion, partitioning them in a convenient way. Let j be the recursion, with $j \geq 0$, and let $c^{(j)}$ be the number of columns of each matrix

group	element	module	position
C_0	$A[i\frac{r}{s}+h,k]$	$h\frac{s}{2}+(k+i) \bmod \frac{s}{2}$	k
C_1	$A[\frac{r}{2}+i\frac{r}{s}+h,k]$	$\frac{r}{2}+h\frac{s}{2}+(k+i) \bmod \frac{s}{2}$	k
C_2	$A[i\frac{r}{s}+h,\frac{s}{2}+k]$	$h\frac{s}{2}+(k+i) \bmod \frac{s}{2}$	$\frac{s}{2}+k$
C_3	$A[\frac{r}{2}+i\frac{r}{s}+h,\frac{s}{2}+k]$	$\frac{r}{2}+h\frac{s}{2}+(k+i) \bmod \frac{s}{2}$	$\frac{s}{2}+k$

Table 1: Memory access to a generic element of a group during the i-th iteration of the 0-th recursion, where $0 \leq i \leq \frac{s}{2}$, $0 \leq k \leq \frac{s}{2}-1$ and $0 \leq h \leq \frac{r}{s}-1$.

$A_1^{(j)}, A_2^{(j)}, A_3^{(j)}, A_4^{(j)}$. Clearly, $c^{(j)} = \frac{s}{2^j}$. When $j \geq 1$, the j-th recursion separates $A_2^{(j-1)}$ and $A_3^{(j-1)}$, respectively, in $A_1^{(j)}, A_3^{(j)}$ and $A_2^{(j)}, A_4^{(j)}$, and invokes the $(j+1)$-th recursion on $A_2^{(j)} \cup A_3^{(j)}$, as shown in Figure 12 for $j = 1, 2$. Moreover, defined

$$\Delta^{(j)} = \begin{cases} 0 & j = 1 \\ \Delta^{(j-1)} + \frac{1}{2}c^{(j-1)} & j \geq 2, \end{cases}$$

the j-th recursion fills the $\frac{c^{(j)}}{2}$ columns $\Delta^{(j)}, \ldots, \Delta^{(j+1)}-1$ of $A_2^{(0)}$ with numbers that belong to the set of the smallest $\frac{rs}{2}$ numbers. Similarly, the j-th recursion fills the $\frac{c^{(j)}}{2}$ columns $\frac{c}{2}+\frac{c^{(j)}}{2}, \ldots, \frac{c}{2}+c^{(j)}-1$ of $A_3^{(0)}$ with numbers that belong to the set of the largest $\frac{rs}{2}$ numbers.

In the following, the details of the conflict-free access memorization of the Row-Column-Pass procedure are given. The building block of such a procedure is the 4-Partition procedure of Figure 2, which works on 4 groups C_0, C_1, C_2, C_3. Such groups correspond to subsets of rows or columns, depending whether a row-pass or a column-pass is performed. During the 0-th recursion, in the i-th iteration of the row-pass, the Row-Column-Pass procedure accesses $4\frac{r}{s}$ rows. Since the hardware implementation accesses diagonals instead of rows, the generic element $A[i\frac{r}{s}+h,k]$, belonging to group C_0, is retrieved from and stored back in position k of the memory module $M_{h\frac{s}{2}+(k+i) \bmod \frac{s}{2}}$, where $0 \leq h \leq \frac{r}{s}-1$ and $0 \leq k \leq \frac{s}{2}-1$. Similarly, the element $A[\frac{r}{2}+i\frac{r}{s}+h,k]$, belonging to group C_1, is accessed in position k of the module $M_{\frac{r}{2}+h\frac{s}{2}+(k+i) \bmod \frac{s}{2}}$, where $0 \leq h \leq \frac{r}{s}-1$ and $0 \leq k \leq \frac{s}{2}-1$. The memory access for a generic element of each group is summarized in Table 1. Observing the table above, one realizes that each classifier call, which works on a pair of groups as specified in Figure 2, accesses r locations, one for each memory module.

During the 0-th recursion, in the i-th iteration of the column-pass, the four groups accessed by the Row-Column-Pass procedure correspond to the two memory lines i and $i+\frac{s}{2}$, and therefore can be retrieved and stored back without memory conflicts.

group	element	module	position
C_0	$A[\frac{r}{2} + i\frac{r}{c^{(j)}} + h, \Delta^{(j)} + k]$	$\frac{r}{2} + h\frac{c^{(j)}}{2} + (k+i) \bmod \frac{c^{(j)}}{2}$	$\Delta^{(j)} + k$
C_1	$A[i\frac{r}{c^{(j)}} + h, \frac{c}{2} + k]$	$h\frac{c^{(j)}}{2} + (k+i) \bmod \frac{c^{(j)}}{2}$	$\frac{c}{2} + k$
C_2	$A[\frac{r}{2} + i\frac{r}{c^{(j)}} + h, \Delta^{(j+1)} + k]$	$\frac{r}{2} + h\frac{c^{(j)}}{2} + (k+i) \bmod \frac{c^{(j)}}{2}$	$\Delta^{(j+1)} + k$
C_3	$A[i\frac{r}{c^{(j)}} + h, \frac{c}{2} + \frac{c^{(j)}}{2} + k]$	$h\frac{c^{(j)}}{2} + (k+i) \bmod \frac{c^{(j)}}{2}$	$\frac{c}{2} + \frac{c^{(j)}}{2} + k$

Table 2: Memory access to a generic element of a group during the i-th iteration of the j-th recursion, where $0 \leq i \leq \frac{c^{(j)}}{2}$, $1 \leq j \leq \log s$, $0 \leq k \leq \frac{c^{(j)}}{2} - 1$ and $0 \leq h \leq \frac{r}{c^{(j)}} - 1$.

In the next recursions, the computation proceeds in place on submatrices $A_2^{(0)}$ and $A_3^{(0)}$. During the j-th recursion, with $j \geq 1$, in the i-th iteration of the row-pass, the Row-Column-Pass procedure accesses $4\frac{r}{c^{(j)}}$ rows. According to the diagonal implementation, the generic element $A[\frac{r}{2}+i\frac{r}{c^{(j)}}+h, k]$, belonging to group C_0, is retrieved from and stored back in position $\Delta^{(j)}+k$ of the memory module $M_{\frac{r}{2}+h\frac{c^{(j)}}{2}+(k+i) \bmod \frac{c^{(j)}}{2}}$, where $0 \leq h \leq \frac{r}{c^{(j)}} - 1$ and $0 \leq k \leq \frac{c^{(j)}}{2} - 1$. The access to the generic elements of the groups is illustrated in Table 2. Again, from the above table, it is easy to see that each classifier call accesses r locations, one for each memory module.

Finally, during the j-th recursion, with $j \geq 1$, in the i-th iteration of the column-pass, the Row-Column-Pass procedure accesses the columns $\Delta^{(j)} + i$ and $\Delta^{(j)} + \frac{c^{(j)}}{2} + i$ of $A_2^{(0)}$, and the columns $\frac{s}{2} + i$ and $\frac{s}{2} + \frac{c^{(j)}}{2} + i$ of $A_3^{(0)}$. Note that each column is stored as a memory line, and therefore can be retrieved and stored back without memory conflicts.

Although a careful conflict-free access for the matrix A has been adopted, a simple pipeline implementation of the 4-Partition procedure allows to start performing at two consecutive instants just classifier calls $2i$ and $2i+1$, with $0 \leq i \leq 2$, since the input for classifier calls 2 and 4 is supplied only after the output of the previous calls 1 and 3 is obtained. With this local perspective, the 4-Partition procedure requires $O(\log r)$ time and the j-th recursion of the Combine-Selection algorithm, which calls Row-Column-Pass, requires $c^{(j)}$ iterations of 4-Partition for an overall time of $O(c^{(j)} \log r)$.

For the classifier network to operate at full capacity, and therefore to have the j-th recursion of Combine-Selection taking $O(c^{(j)})$ time, a global overview of the computation must be taken into account. In fact, an efficient implementation of the Row-Column-Pass procedure can be provided which exploits, by means of an *interleaved pipelining*, the parallelism inherent in its $c^{(j)}$ iterations.

In order to describe the behaviour of the interleaved pipelining, consider again the generic j-th recursion of Combine-Selection, with $j \geq 0$, and focus on the row-pass. The $\frac{c^{(j)}}{2}$ calls to the 4-Partition procedure are performed as follows. The computation starts with classifier call 0 performed in simple pipeline fashion on all the data given by Row-Column-Pass for the iterations $0, 1, \cdots, c^{(j)}/2$. Clearly, this is possible because the classifier call 0 is applied every time to a different

set of data. Then, in a perfectly similar fashion, simple pipeline is used to carry out every classifier call 1 of 4-Partition on all the data given by Row-Column-Pass for the iterations $0, 1, \cdots, c^{(j)}/2$. The same approach is followed for the remaining classifier calls of 4-Partition. Moreover, the same interleaved pipelining strategy is used with the 6 classifier calls of 4-Partition within the column-pass. Note that two classifier calls on the same input data are at least $c^{(j)}/2$ iterations apart. Therefore, as long as $c^{(j)}/2$ is not smaller than the depth of the classifier network, the interleaved pipelining can proceed without interruptions.

The $(j+1)$-th recursion of the Combine-Selection algorithm operates on a number of columns $c^{(j+1)} = c^{(j)}/2$ and hence also the number of iterations of Row-Column-Pass halves. The r-classifier works in $C \log r$ time, where $C \approx 2$ [6]. Therefore, as soon as the number of iterations becomes $\log r$, the interleaved pipelining cannot be applied anymore without slowing the computation. The computation then proceeds as in the Three-Pass-Selection implementation described in the previous subsection where the number of columns is $\log r$.

To evaluate the time complexity of the above pipeline implementation of the Combine-Selection algorithm, observe that in the j-th recursion of Combine-Selection, Row-Column-Pass invokes $c^{(j)} = s/2^j$ times 4-Partition, which in turn calls 6 times the classifier. During all the recursions of Combine-Selection performed according to the interleaved pipelining, including the 0-th recursion, a new classifier call starts executing at each subsequent instant. Therefore, the overall time required by the interleaved pipeline is

$$\sum_{j=0}^{\ell} \frac{6s}{2^j} + C \log r = O(s),$$

where $\ell = \log \frac{s}{\log r}$.

In addition, the final computation performed according to the Three-Pass-Selection implementation requires $O(\log r)$ time. Therefore, Combine-Selection takes an optimal $O(s + \log r)$ time, to solve the problem on $N = rs$ numbers, with $s \leq r$. Since an extended classifier network of depth $O(\log r)$ and $O(r \log r)$ comparators is employed, a total work of $O(rs \log r + r \log^2 r)$ is used.

4.4 Implementation of Row-Column-Selection

For the sake of completeness, this subsection sketches how the Row-Column-Selection algorithm of Figure 4 can be implemented in hardware. Such an algorithm does not use the 4-Skewed-Partition as a subroutine, and therefore it can be implemented just using an r-classifier instead of the extended classifier device employed by the Combine-Selection algorithm. In this way, the hardware requirement is kept as simple as possible, but the time performance is optimal only when the number of columns is $\Omega(\log r \log \log r)$.

The hardware implementation of Row-Column-Selection differs from that of Combine-Selection only when the number of iterations becomes less than or equal to $\log r$. At that point, Row-Column-

Selection continues to recursively half the number of columns invoking the Row-Column-Pass procedure, but the interleaved pipelining cannot be applied anymore to implement Row-Column-Pass. The computation, then, proceeds by means of a semi-interleaved pipeline, which is similar to the interleaved pipeline described above except that the executions of the classifier calls $2i$ of all iterations start only after the output of the previous classifier calls are obtained, with $0 \leq i \leq 2$. In particular, when $\log \frac{s}{\log r} + 1 \leq j \leq \log \frac{s}{4}$, the j-th recursion of Row-Column-Selection performed according to the semi-interleaved pipelining requires time $O(c^{(j)} + \log r) = O(\log r)$. Since there are $O(\log \log r)$ recursions performed in the semi-interleaved pipeline fashion, $O(\log r \log \log r)$ time is required to complete the Row-Column-Selection computation. In conclusion the hardware implementation of Row-Column-Selection takes $O(s + \log r \log \log r)$ time, which is optimal when $s = \Omega(\log r \log \log r)$, and uses $O(r \log r(s + \log r \log \log r))$ work.

5 Conclusions

This paper has shown how the well-known Leighton's Columnsort algorithm can be modified so as to solve the median problem using classifier networks instead of sorting networks. In particular, two median algorithms have been presented. Both algorithms apply the classifier no more time than Columnsort applies the sorter, but use a simpler and more effective network, and can be efficiently implemented in hardware.

The first algorithm takes a logarithmic number of passes and can be implemented using just an r-classifier. The second algorithm takes 3 passes and uses a slightly modified r-classifier, which however has the same depth and the same number of comparators (in order of magnitude) as the simple r-classifier. In particular, such an algorithm shows that it is possible to build a classifier network that can solve the median problem of $O(r \log r)$ numbers using optimal $O(\log r)$ time and $O(r \log r)$ comparators as an r-classifier. Finally, such two algorithms can be combined together leading to a hardware algorithm which solves the median problem of $N = rs$ numbers, for $1 \leq s \leq r$, in optimal $O(\log r + s)$ time and using $O(r \log r(\log r + s))$ work.

It is worthy to note that the extended classifier network has replaced a simple r-classifier only to implement the 4-Skewed-Partition procedure. However, given an (n, m)-classifier which classifies its n input numbers into the m smallest numbers and the $n - m$ largest ones, such a procedure could be easily implemented by connecting in cascade the output of an r-classifier with an $(r/2, r/2 - \log r)$-classifier and an $(r/2, \log r)$-classifier. The extended classifier has been introduced in the architecture because $\log r$ and $r/2 - \log r$ are not $\Omega(r)$. Indeed, according to [6], an (n, m)-classifier can be obtained maintaining the same time and work performances as an n-classifier only when $m = \Omega(n)$,

However, several questions still remain open. The $\Omega(N \log N)$ lower bound on the number of comparators given in [2] holds only for networks with I/O size N. On the other side, $\Omega(N)$ is a lower bound on the work for any algorithm using comparisons [5]. Hence, any hardware algorithm that uses an r-classifier has a trivial $\Omega(N + r \log r)$ lower bound on the work. Therefore, a challenge for the future is either to design a hardware algorithm that matches such a trivial lower bound or to prove a higher lower bound on the work. Moreover, one could generalize the methods presented here for solving the K-Selection problem for an arbitrary K, where it is asked to classify a set of N numbers so as to separate the K smallest numbers and the $N - K$ largest ones.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, Sorting in $c \log n$ Parallel Steps, *Combinatorica*, 3, (1983), 1–19.

[2] V.E. Alekseyev, Sorting Algorithms with Minimum Memory, *Kibernetica*, 5, (1969), 99–103.

[3] K.E. Batcher, Sorting Networks and Their Applications, *Proc. of AFIPS Conference*, (1968), 307–314.

[4] J. Belzile, Y. Savaria, D. Haccoun, and M. Chalifoux, Bounds on the Performance of Partial Selection Networks, *IEEE Trans. on Communications*, 43, (1995), 1800–1809.

[5] M. Blum, R.W. Floyd, V. Pratt, R.L. Rivest, and R.E. Tarjan, Time Bounds for Selection, *Journal of Computer and System Sciences*, 7, (1973), 448–461.

[6] S. Jimbo and A. Maruoka, A Method of Constructing Selection Networks with $O(\log n)$ Depth, *SIAM Journal of Computing*, 25, (1996), 709–739.

[7] M. Kutylowski, K. Lorys, B. Oesterdiekhoff, and R. Wanka, Periodification Scheme: Constructing Sorting Networks with Constant Period, *Journal of ACM*, 47, (2000), 944–967.

[8] F.T. Leighton, Tight Bounds on the Complexity of Parallel Sorting, *IEEE Transactions on Computers*, C-34, (1985), 344–354.

[9] F.T. Leighton, Y. Ma, and T. Suel, On Probabilistic Networks for Selection, Merging and Sorting, *Theory of Computing Systems*, (1997), 559–582.

[10] G.S. Manku, S. Rajagopalan, and B.G. Lindsay, Approximate Medians and Other Quantiles in One Pass and with Limited Memory, *ACM Sigmod Int'l Conf. on Data Management*, (1998).

[11] S. Olariu, M.C. Pinotti, and S.Q. Zheng, An Optimal Hardware-Algorithm for Selection Using a Fixed-Size Parallel Classifier Device, *6th Int'l Conference on High Performance Computing*, Calcutta, India, (1999), 284–288.

[12] S. Olariu, M.C. Pinotti, and S.Q. Zheng, How to Sort N Items Using a Network of Fixed I/O, *IEEE Trans. on Parallel and Distributed Systems*, 10, (1999), 487–499.

[13] S. Olariu, M.C. Pinotti, and S.Q. Zheng, An Optimal Hardware-Algorithm for Sorting Using a Fixed-Size Parallel Sorting Device, *IEEE Transactions on Computers*, 49, (2000), 1310–1324.

[14] N. Pippenger, Selection Networks, *SIAM Journal of Computing*, 20, (1991), 878–887.

[15] B.W. Wah and K.L. Chen, A Partitioning Approach to the Design of Selection Networks, *IEEE Transactions on Computers*, 33, (1984), 261–268.

[16] A.C. Yao, Bounds on Selection Networks, *SIAM Journal on Computing*, 9, (1980), 566–582.