EFFICIENT ARCHITECTURES FOR HIDDEN SURFACE REMOVAL

Chaitali Chakrabarti

Department of Electrical Engineering
Arizona State University
Tempe, AZ 85287-5706
chaitali@asu.edu

Lori Lucke

Department of Electrical Engineering
University of Minnesota
Minneapolis, MN 55455
lucke@ee.umn.edu

ABSTRACT

We present several new efficient architectures to solve the hidden surface problem in the feature domain. All the architectures operate on segments (instead of pixels) and create a list of visible segments for each scan line. We present two new semi-systolic architectures consisting of an array of M processors, where M is the maximum number of overlapping segments. Both architectures require presorting of the segment endpoints and have a latency of $O(N)$, where N is the number of input segments. We present two new sorting network architectures which do not require any presorting of the endpoints. These architectures consist of $O(\log N)$ stages of segment merge units and are based on odd-even merge sort and running merge sort.

1. INTRODUCTION

Hidden surface removal is a problem encountered during the projection of a set of 3-dimensional objects onto a 2-dimensional viewing surface. The objects may be projected from a specific perspective or, as in our case, in parallel along a particular direction. As the objects are projected, it is necessary to determine which lines or surfaces are visible and must be displayed. Numerous algorithms have been proposed in the past for the hidden surface removal problem [1, 2]. When one scanline of the image is processed at a time, the hidden surface removal problem reduces to the hidden segment removal problem.

Most of the existing graphics processors implement the z-buffer algorithm for hidden segment removal. During line scan, the z-buffer algorithm compares each new polygon point to a point stored in the z-buffer. If the new point is closer to the viewer, then its' color and depth values replace the old values in the frame and z-buffers. Thus objects closer to the viewer overwrite objects that are further away. While this algorithm is easy to map onto multiprocessor systems, it requires a large amount of processing power, since the frame buffer is read and written multiple times at each pixel location. As the demand for portable and real-time display of virtual images grows, it is necessary to develop parallel algorithms and architectures which can meet the dual demands of high throughput and low power. Hidden surface algorithms that determine the visible segments in the feature domain prior to writing the segments to the frame buffer can meet these demands. Recently several systolic architectures have been proposed which operate on segments instead of pixels [3, 4]. The architecture proposed in [3] requires $2N$ processors and $3N$ time steps while the architecture in [4] requires $> 2N$ time steps and M processors, where N is the total number of segments, and M is the maximum number of overlapping segments. Both architectures require presorting of the segments by their left endpoints.

In this paper we present four new architectures to solve the hidden segment problem. All the architectures create a list of visible segments for each scan line. The first two architectures require presorting of the segment endpoints and are mapped onto semi-systolic arrays of size M. Architecture 1 requires a maximum of $2N$ time steps while Architecture 2 requires only N time steps but uses a more complex processor architecture. Architectures 3 and 4 do not require any presorting and are mapped onto sorting network-based architectures. Architecture 3 utilizes a parallel merge-sort network and is very fast. Architecture 4 utilizes a running order sorter and is slower, but has smaller area. All these architectures are suitable for high speed or low power applications.

2. NEW SYSTOLIC ARCHITECTURES

Definition: The hidden segment problem identifies the visible segment endpoints among a set of N non-intersecting segments. Let segment i be represented as $\{(l_i, zl_i), (r_i, zr_i)\}$ where (l_i, zl_i) is the left endpoint and (r_i, zr_i) is the right endpoint. In this paper we assume that segments never cross. In this case, a single z value is sufficient to describe the segment and we

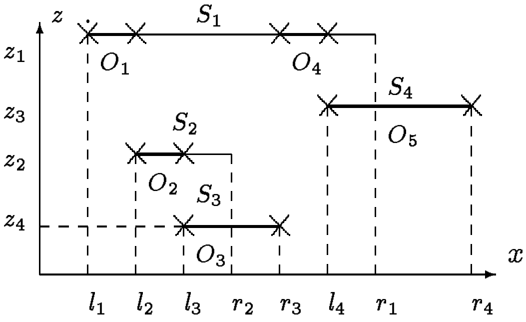

Figure 1: Visible Segment Example

Table 1: Architecture 1 Example

T	Input	Sorted List	Output
0	(l_1,z_1)	S1	$ol_1 = (l_1, z_1)$
1	(l_2,z_2)	S2,S1	$or_1 = (l_2, z_1)$,$ol_2 = (l_2, z_2)$
2	(l_3,z_3)	S3,S2,S1	$or_2 = (l_3, z_2)$,$ol_3 = (l_3, z_3)$
3	(r_2,z_2)	S3,S1	
4	(r_3,z_3)	S1	$or_3 = (r_3, z_3)$,$ol_4 = (r_3, z_1)$
5	(l_4,z_4)	S4,S1	$or_4 = (l_4, z_1)$,$ol_5 = (l_4, z_4)$
6	(r_1,z_1)	S4	
7	(r_4,z_4)		$or_5 = (r_4, z_4)$

also refer to the segment as $\{l_i, r_i, z_i\}$. A segment, k, is visible if there are no other segments with $z_i < z_k$ for all segments i. Consider, for example, the segments shown in Fig. 1. Here there are 4 input segments, S1 - S4, and 5 visible output segments, O1 - O5.

We propose two new systolic architectures which are based on updating a sorted list of potentially visible segments. These architectures are similar to the ones proposed in [5] for order statistic filters.

Architecture 1:

The input to Architecture 1 is a sorted list of left and right endpoints. Each input endpoint is used to update a list of segments sorted by their z values and stored in sorted order. If the input is a left endpoint, then the corresponding segment is added to the sorted list, and if the input is a right endpoint, then the corresponding segment is removed from this list. The list is of size M, where M is the maximum number of overlapping segments. Table 1 shows the operation of this algorithm on the example of Fig. 1. The input left and right endpoints are denoted by (l, z) and (r, z), while the output left and right endpoints are denoted by ol_i and or_i.

The architecture consists of a linear array of M processors, where each processor consists of a comparator, registers, and control logic as shown in Fig. 2. Processor i, $P(i)$, stores a left segment endpoint corresponding to the segment with the ith smallest z-value. The

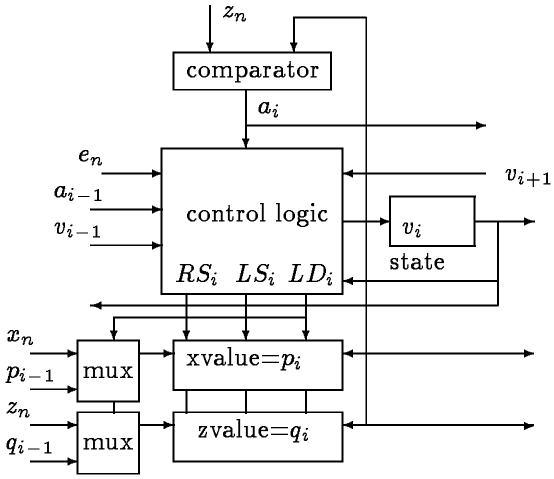

Figure 2: Architecture 1: Processor Logic

segment endpoint consists of p_i, the x-value, and q_i, the z-value. The input to the processor array is a segment endpoint described by $\{e_n, x_n, z_n\}$, where e_n is a bit denoting whether the endpoint is a left or right endpoint ($e_n = 0$ for left), and x_n and z_n are the x and z values of the endpoint. In each cycle, at most one visible segment is output, and it takes $2N$ cycles to generate all the outputs.

Processor $P(i)$ operates as follows. The z-values of the current and new segment are compared, and $a_i = 1$ if $q_i \geq z_n$. The control logic generates the left shift, LS_i, the right shift, RS_i, and the load, LD_i, signals for the segment endpoint registers, depending on the value of a_i and the comparison and shift values from $P(i-1)$. The control logic also uses a validity bit, v_i to determine whether the segment in the processor is valid or not. A right shift occurs when a new left endpoint represents a segment in front of the currently stored segment. A load occurs when the new left endpoint represents a segment in front of the current segment and behind the segment in $P(i-1)$. A left shift occurs when a new right endpoint represents a segment in front of or equal to the current segment.

The output segment endpoints are generated in $P(0)$ in parallel with the sorted list update procedure. Recall that $P(0)$ contains the segment endpoint with the smallest z value. Let $\hat{z}$ equal the z value of the output of the previous left endpoint. Then the output is generated as follows. If $a_0 = 0$ then there is no output since the endpoint is not visible. If $e_n = 0$ and $a_0 = 1$, then $or_{k-1} = (x_n, \hat{z})$ and $ol_k = (x_n, z_n)$. If $e_n = 1$ and $a_0 = 1$, then $or_{k-1} = (x_n, \hat{z})$ and $ol_k = (x_n, q_1)$.

Architecture 2:

The input to Architecture 2 is a list of segments that are sorted by their left endpoints. The array holds a

Table 2: Architecture 2 Example

Time	Input	Sorted List	Output
0	S1	S1	
1	S2	S2,S1	O1=$\{l_1, l_2, z_1\}$
2	S3	S3,S1	O2=$\{l_2, l_3, z_2\}$
3	S4	S4	O3=$\{l_3, r_3, z_3\}$
			O4=$\{r_3, l_4, z_1\}$
			O5=$\{l_4, r_4, z_4\}$

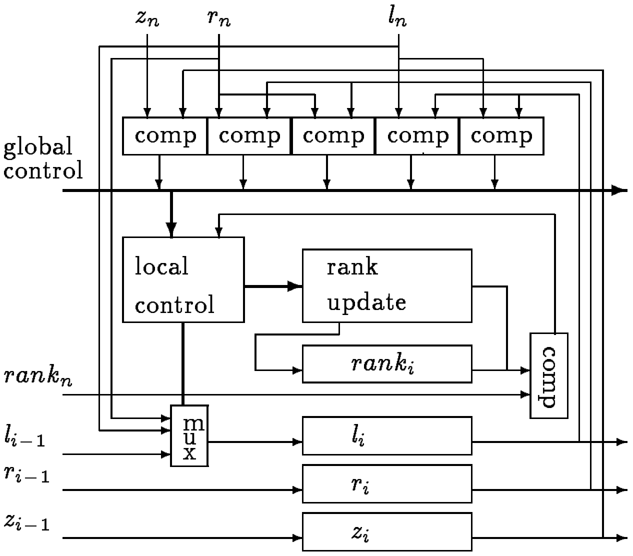

Figure 3: Architecture 2: Processor Logic

list of potentially visible segments implicitly sorted by their z values. Each new segment is compared to all the segments in the array, and segments that are not overlapped by the new segment are output. The visible segment list is then updated depending on whether the new segment is visible or not. Table 2 shows the operation of this algorithm on the example of Fig. 1.

The architecture consists of an array of M processors, where each processor consists of comparators, shift registers and control logic. In each cycle, more than one visible segment is output, and all the output segments are generated in $N+1$ cycles. While the structure of the array is similar to that of Architecture 1, the processors are more complex. Fig. 3 describes one such processor.

Each processor stores a potentially visible segment denoted by $\{l_i, r_i, z_i\}$ where l_i is the left x value, r_i is the right x value, and z_i is the depth of the segment. The segments are stored within the array in the order they are received. Therefore, the storage registers only shift right. Each processor stores a rank, $rank_i$, indicating the relative depth of the corresponding segment. The use of a rank allows us to maintain the visible segments in an implicit sorted list. A new segment, $\{l_n, r_n, z_n\}$, enters the array and is compared to all the stored segments. The results of these comparisons are used to calculate the rank of the new segment, $rank_n$, and to update the rank and endpoints of the remaining segments. For example, if the new segment partially occludes the old segment, then one of the new segment endpoints becomes the endpoint of the old segment. If any stored segments are to the left of the new segment, then they are sent to the output. For example, when Segment 3 enters the array in Table 2, it is to the right of a portion of Segment 2. Since the segments are presorted by their left endpoints, we know that no other segment will occlude the portion of Segment 2 and so this becomes the output segment. We have implemented and simulated Architecture 2 for $M = 8$ using the Mentor Graphics CAD tools and CMOS standard cells.

3. NEW SORTING NETWORK-BASED ARCHITECTURES

We propose two sorting network-based architectures which do not require any presorting of the segment endpoints. The two architectures differ in their input/output behavior (parallel vs serial) but both generate a sorted list of visible segments. However the visible segment problem is different from most sorting problems since the number of output segments differs from the number of input segments. If we begin with a set of N random input segments, then the number of output (visible) segments is at most $2N-1$. Thus existing sorting networks must be modified to generate the sorted visible segment list.

Architecture 3:

Architecture 3 operates on segments in parallel and generates the sorted list of visible segments in parallel. This architecture is based on Batcher's odd/even merge sorting network [6]. While the interconnection of the 2 networks is the same, the compare-swap units of the odd-even merge network are replaced with *Merge A* units to handle overlapping segments. The architecture consists of $\log N$ stages, where N is the number of input segments. Each stage consists of a segment-merge, $SM(k,k)$, which merges two sorted lists of visible segments of size k. If N is a power of 2, then Stage 1 consists of $N/2$ $SM(1,1)$ units, Stage 2 consists of $N/4$ $SM(3,3)$ units, and in general, Stage i consists of $N/2^i$ $SM(2^i-1, 2^i-1)$ units. The number of output segments of $SM(k,k)$ is $2k+1$, which is 1 more than the number of input segments. Fig. 4 describes Architecture 3 for the case when the number of input segments is 4. Here each $SM(1,1)$ unit in Stage 1 generates at most 3 sorted visible segments, which are merged us-

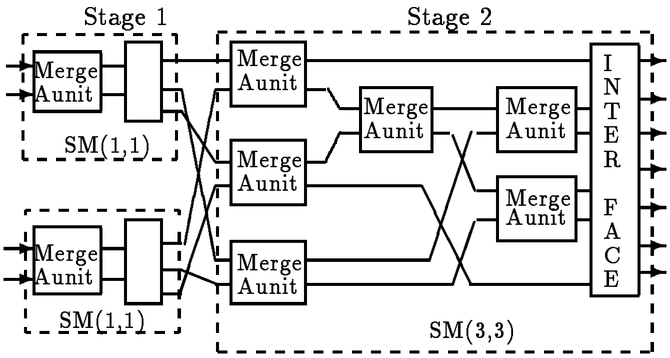

Figure 4: Architecture 3 for $N = 4$

Table 3: Architecture 3 Example

In	Stage 1 Output	Stage 2 Output
S2	T1=$\{l_1, l_2, z_1\}$	O1=$\{l_1, l_2, z_1\}$
S1	T2=$\{l_2, r_2, z_2\}$	O2=$\{l_2, l_3, z_2\}$
	T3=$\{r_2, r_1, z_1\}$	O3=$\{l_3, r_3, z_3\}$
S4	R1=$\{l_3, r_3, z_3\}$	O4=$\{r_3, l_4, z_1\}$
S3	R2=$\{l_4, r_4, z_4\}$	O5=$\{l_4, r_4, z_4\}$

ing a $SM(3,3)$ in Stage 2 to generate 7 visible sorted segments.

The Merge A unit compares two inputs, eliminates the occluded portions of segments, and orders the outputs. Each input consists of at most two segments which have been paired together and are treated as one entity. Fig. 5 describes Merge A unit. Let $I_{1,1}$ and $I_{1,0}$ be the segment pair from input list 1, $I_{2,1}$ and $I_{2,0}$ be the segment pair from input list 2. The outputs are generated in sorted order and consist of two segment pairs: $O_{1,1}$, $O_{1,0}$ and $O_{2,1}$, $O_{2,0}$.

The segments in a pair are tagged with a 1 or a 0 as indicated by the second subscript in the pair. All visible segments are tagged with a 1 *except* if the 2 parent segments $\{l_1, r_1, z_1\}$ and $\{l_2, r_2, z_2\}$ have the following relationship: $z_1 < z_2$, $l_1 > l_2$ and $r_1 < r_2$. Then the output becomes, $O_{1,1} = \{l_2, l_1, z_2\}$, $O_{2,1} = \{r_1, r_2, z_2\}$, and $O_{1,0} = \{l_1, r_1, z_1\}$. Thus the tag 0 segment is generated when a smaller segment is completely in front of a larger segment. Once a tag 0 segment is generated, it does not get modified in the rest of the segment-merge unit. This is because a tag 0 segment can never overlap with any other tag 1 or tag 0 segment. By pairing a tag 0 segment with a tag 1 segment, and propagating it through the network, a tag 0 segment is correctly placed in the sorted list of output segments.

Table 3 describes the operation of Architecture 3 on the example in Fig. 1. In the first stage, the $SM(1,1)$ units merge the input segments $S2$ and $S1$ to generate $T1$, $T2$, and $T3$, and segments $S4$ and $S3$ to generate $R1$ and $R2$. In the second stage, the $SM(3,3)$ unit

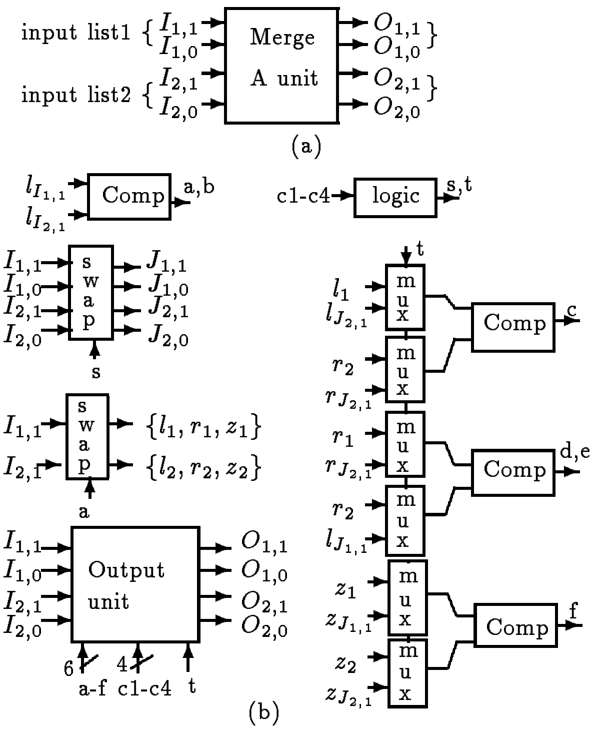

Figure 5: Architecture 3: Merge A unit (a) Block diagram (b) Logic design

merges sorted segment lists $(T1, T2, T3)$ with $(R1, R2)$ and generates the output segments $O1$ through $O5$.

The logic design of Merge A is shown in Fig. 5(b). As segments are merged, portions or whole segments may become occluded or NIL. The computations in the Merge A unit are a function of the number and position of NIL input segments. For instance, if $I_{1,1}$ and $I_{1,0}$ are both NIL, then $O_{1,1} = I_{2,1}$, $O_{1,0} = I_{2,0}$, and $O_{2,1} = O_{2,0}$ =NIL. Control signals $c_1 - c_4$ are used to identify which of the four inputs may be NIL. The endpoints of the input and output segments may be different in only the following 3 cases : (i) $I_{1,0}$ is NIL (ii) only $I_{2,0}$ is NIL and (iii) both $I_{1,0}$ and $I_{2,0}$ are NIL. Control signals s and t are used to distinguish these 3 cases. In case (iii), additional computations are done to determine the segment with the larger left endpoint (among $I_{1,1}$ and $I_{2,1}$) $\{l_1, r_1, z_1\}$, and the segment with the smaller left endpoint $\{l_2, r_2, z_2\}$. The results of these comparisons (control signals c, d, e, f) and $c_1 - c_4$ are fed to the output unit which generates the endpoints of the output segments.

Since the outputs of merge A unit can consist of NIL segments, each segment-merge unit consists of an interface unit to remove the NIL segments from the sorted list. For instance, the interface unit in $SM(3,3)$ (see Fig. 4) removes at least 12-7=5 NIL segments. In general, the interface unit in $SM(2^i - 1, 2^i - 1)$ removes

$(2^{i+2}-4)-(2^{i+1}-1)=2^{i+1}-3$ NIL segments. The interface unit also removes the tags, so that the input to any segment-merge unit consists of single (and not paired) segments.

Architecture 4:

Architecture 4 is based on a running order merge sort [7] where the segment lists are input sequentially and the sorted list of visible segments is generated sequentially. This architecture also consists of $\log N$ stages. Each stage consists of (i) a Merge B unit to merge two sorted segment lists sequentially and (ii) a storage unit to store the segments that have yet to be merged (see Fig. 6a).

Fig. 6b describes the Merge B unit. Let S_1 and S_2 be the two segment lists that are to be merged. In each cycle, a segment I_n from S_1 or S_2 is merged with a potentially visible segment I_s (generated in the previous cycle), and two segments are generated: a visible segment that is output and a potentially visible segment that is stored for the next computation. The computations are summarized below. First, the left endpoints of I_n and I_s are compared and the segment with the larger left endpoint, $J_1=\{l_1,r_1,z_1,q_1\}$, and the one with the smaller left endpoint, $J_2=\{l_2,r_2,z_2,q_2\}$, are identified. Here q_i is used to denote whether the segment is from S_1 or S_2. Next (l_1,r_2), (r_1,r_2) and (z_1,z_2) are compared, and the results of the comparisons are used to generate the output segment and a potentially visible segment. The output unit also generates a NEXT signal which indicates whether the next segment is from S_1 or S_2.

When one of the segment lists is empty, the contents of the storage register is output followed by the contents of the non-empty segment list. Control signals g, h, t and NEXT are used to achieve this. If consecutive segments are of the form $\{l_a,r_a,z_a\}$ and $\{r_a,r_b,z_a\}$, then they are merged in an additional unit (not shown in the figure) before being stored in the next stage.

In each stage, the number of output segments is 1 more than the number of input segments. To support this, additional storage and a controlled input stream are required. The storage unit in Stage i stores 3 segment lists (generated by Stage $i-1$) each of size 2^i-1. Stage i operates on two of the segment lists while the third is computed by Stage $i-1$. A smooth data flow is ensured by adding delays into the input data stream of Stage 1. For example, for $N=8$, 11 clock cycles are required to input the 8 input segments. The input data is fed at times 0,1,-,3,4,-,-,7,8,-,10,11 and delays are inserted for the other cycles.

The latency of this implementation is $2N-3$. This large latency can be reduced by increasing the number of parallel merge units in each stage. For instance, the latency can be reduced by a factor of 2 by including 2 merge units in each stage, and by modifying the last stage such that the odd segments and the even segments are merged in parallel using Merge A unit.

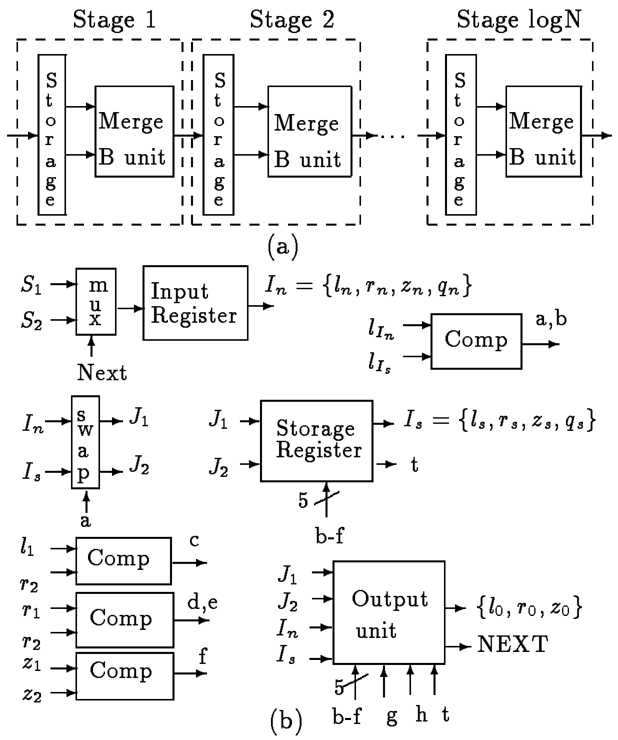

Figure 6: Architecture 4: (a) Block diagram (b) Logic design of Merge B unit

4. REFERENCES

[1] Foley, Van Dam, et. al. *Introduction to Computer Graphics*. Addison-Wesley, 1994.

[2] I. E. Sutherland, R.F. Sproull, and R.A. Schumacker. A characterization of ten hidden surface algorithms. *Computing Surveys*, 6(1):1–55, 1974.

[3] N. H. Vaidya, S. R. Das, P.C. Mathias, and L.M. Patnaik. A systolic algorithm for scanline-based hidden surface removal. In *Proc of 3rd Int'l Conf. on Supercomputing*, pages 239–246, 1988.

[4] T. Risset and S. W. Song. *in Application Specific Array Processors*, chapter A real-time systolic algorithm for on-the-fly hidden surface removal, pages 238–249. IEEE Press, 1993.

[5] C. Chakrabarti and L. Lucke. *in VLSI Signal Processing, VI*, chapter Highly concurrent architectures for recursive median filters. IEEE Computer Society Press, 1993.

[6] K. E. Batcher. Sorting networks and their applicatios. In *AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968.

[7] I. Pitas. Fast algorithms for running ordering and max/min calculation. *IEEE Trans. on Circuits and Sys.*, 36(6), June 1989.