IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 16, 2020, accepted December 14, 2020, date of publication December 25, 2020,
date of current version January 6, 2021.

Digital Object Identifier 10.1109/ACCESS.2020.3047594

Design, Implementation, and Analysis of High-Speed Single-Stage N-Sorters and N-Filters

ROBERT B. KENT, (Life Member, IEEE), AND MARIOS S. PATTICHIS, (Senior Member, IEEE)
Department of Electrical and Computer Engineering, The University of New Mexico, Albuquerque, NM 87131, USA
Corresponding author: Robert B. Kent (rkent@unm.edu)

ABSTRACT There is strong interest in developing high-performance hardware sorting systems which can sort a set of elements as quickly as possible. The fastest of the current FPGA systems are sorting networks, in which sets of 2-sorters operate in parallel in each series stage of a multi-stage sorting process. A 2-sorter is a single-stage hardware block which sorts two values, so any list with more than 2 values must be sorted with a series network of 2-sorters. A primary contribution of this work is to provide a general methodology for the design of stable single-stage hardware sorters which sort more than 2 values simultaneously. This general methodology for N-sorter design, with $N>2$, is then adapted for use in modern FPGAs, where it is shown that single-stage 3-sorters up to 9-sorters have speedup ratios from 2.0 to 3.5 versus the comparable state-of-the-art 2-sorter networks. A design system modification is shown to produce even faster single-stage N-max and N-min filters. When used for max pooling 32-bit data in the fastest analyzed FPGA, a single 9-max filter will process 500 million 9-pixel groups per second (4K:3840x2160 at 500 frames/second). The single-stage 9-median filter using this design methodology, useful in image processing, is shown to have speedup ratios of 3.0 to 4.1 versus state-of-the-art FPGA network implementations, even though its resource usage is comparable to, often better than, the network implementations. Ten 8-bit 9-median filters operating in parallel in the fastest FPGA will process over 5.4 billion pixels/sec (4K at over 600 frames/second).

INDEX TERMS Field programmable gate arrays, FPGA, image filtering, merging, sorting, sorting networks, max pooling, median filters.

I. INTRODUCTION

Some of the most successful efforts to use hardware acceleration for computing tasks have been in the full sort and rank order filtering of unsorted lists of values. The main reason for this is the inherent use of parallel processing in hardware sorting systems such as sorting networks. Two sorting network algorithms introduced by Kenneth Batcher, Odd-Even Merge Sort (O-EMS) and Bitonic Merge Sort, are today's state-of-the-art algorithms for sorting operation speed in FPGAs [1]–[4].

These state-of-the-art sorting networks use 2-sorters as the hardware blocks which operate in parallel during the series of stages in the sorting process. Fig. 1 shows a schematic for a 2-sorter.

When a sorting network only uses 2-sorters, even small lists with more than 2 values must be sorted with a sorting network. Networks which sort 3 or 4 values require 3 stages in series, so these small sorting networks take approximately 3x longer to complete their sort operations, versus one of the single-stage 2-sorters that are used in the network. Fig. 2 shows a schematic for a 2-sorter 4-network, showing its series of 3 stages.

The 2-sorter 4-network shown in Fig. 2 is one of a set of small 2-sorter networks which have historically been considered optimal. They have been considered optimal as they use both a minimum number of series stages and a minimum of hardware resources, i.e. 2-sorters. Until now, 2-sorters have been the only small single-stage sorting blocks that have been available, so this definition of optimal made sense.

The number of series stages for the best 2-sorter networks which sort from 3 up to 9 values are shown in the first row of Table 1. The data in the header row and this first data row, both highlighted in tan, have been known for decades, and remain unchanged today. This data can be found in Donald Knuth's 1998 text [5], but the same data was in his 1973 version as

The associate editor coordinating the review of this manuscript and approving it for publication was Jenny Mahoney.

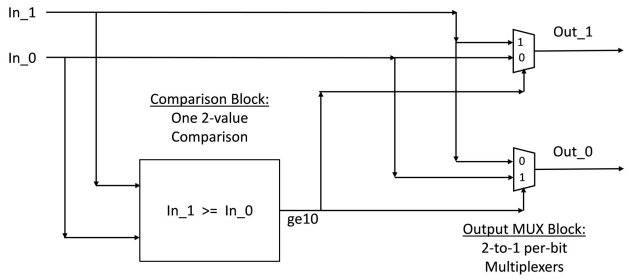


FIGURE 1. Commonly-used stable single-stage hardware 2-sorter.

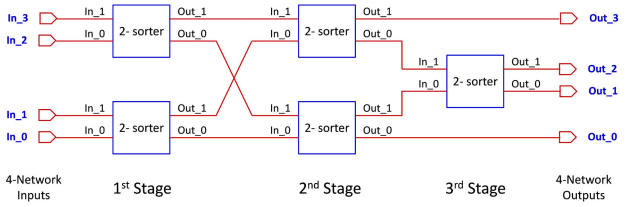


FIGURE 2. Schematic of 3-stage 2-sorter 4-network.

TABLE 1. Estimated N-sorter speedup vs the best 2-sorter networks.

N: Number of Values to Sort	3	4	5	6	7	8	9
N-Network Series Stages [5][6]	3	3	5	5	6	6	7
N-Sorter Equivalent Stages	1	1.5	1.5	2	2	2	2
Estimated N-Sorter Speedup	3	2	3.3	2.5	3	3	3.5

Equivalent Stages: Estimated N-Sorter Prop Delay in 2-Sorter Stage Units
Speedup = N-Network Series Stages / N-Sorter Equivalent Stages

well. A more recent reference [6] still shows the same data. The data in this N-Network Series Stages row is effectively propagation delay data, normalized by the propagation delay of a single 2-sorter.

For N values of 3, 4, 7, and 8, the data for the N-Network Series Stages row in Table 1 come from Odd-Even Merge Sort networks. The data in this row for N values of 5, 6, and 9 are derived from manually designed 2-sorter networks.

The initial motivation for the current paper is to create small single-stage N-sorters, for $N \geq 3$, which will have significantly lower propagation delay values than those found in the first data row of Table 1. Fig. 3 shows a block diagram for these proposed N-sorters.

A comparison of Figs. 1 and 3 shows that a 2-sorter is one instance of an N-sorter design. Novel N-sorters are created when a Fig. 3 design is created for $N \geq 3$.

When the novel N-sorters are implemented in modern Xilinx FPGAs, for N from 3 to 9, it is estimated that their normalized propagation delay values will match those in the pink second row of Table 1. The estimated speedup values of the N-sorters are then listed in the green last row of Table 1. These FPGA estimates are generated and discussed in detail in Section IV-B.

Our first major contribution has been to create a general design system for Fig. 3 N-sorters, for $N \geq 3$. The next important contribution was then to modify the general design

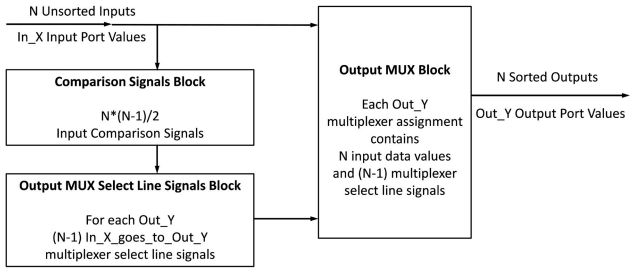


FIGURE 3. General single-stage N-sorter design and data flow.

system as needed in order to implement the novel N-sorters in modern FPGAs. In FPGAs, the target was to produce high speed designs which would meet the estimates listed in the second and third rows in Table 1.

After both N-sorter and comparable 2-sorter N-network designs are synthesized using FPGA software tools, the true propagation delays and speedup ratios are determined, using the synthesis results. As will be seen later, the synthesis N-sorter speedup ratios are in line with the estimated speedup values listed in Table 1.

Although several researchers have discussed the probable advantages of sorting more than 2 input values in a single-stage N-sorter, efforts to produce such N-sorters have been minimal, and not very successful. These previous efforts are discussed in Section II-B.

None of these prior research efforts has provided a successful general system for N-sorter design, with $N \geq 3$, in which the sorters are defined using logic equations. In order to define an N-sorter, the design system introduced here does use logic equations, which are specified in Hardware Description Language (HDL) code. Once N-sorters are defined using synthesizable HDL, often referred to as RTL, they can readily be implemented in hardware such as FPGAs.

In addition to providing full single-stage N-sorters, another major contribution of the new design system is the creation of single-stage rank order N-filters, in which only one or a few of the values from the sorted list are transferred to an output port. Historically [7], the only well-defined single-stage N-filters were 2-max and 2-min filters, each of which is created from a single-stage 2-sorter simply by removing the unused output port.

In the novel system introduced here, several types of single-stage N-filters with $N \geq 3$ are described. It will be shown that the single-stage 9-median filter produced by this system is clearly the fastest available 9-median FPGA filter, and is usually the most resource efficient. The 9-median filter is used to reduce noise while preserving image structure in image processing applications [8].

A modification of the design system is used to produce fast single-stage N-max and N-min filters. All of our FPGA single-stage 3-max to 9-max filters will be shown to be significantly faster than equivalent networks of traditional 2-max filters. If used in max pooling operations over 2×2 or 3×3 image windows, our fast and efficient N-max filters will

benefit the implementation of modern Convolutional Neural Network (CNN) architectures. The speed of our 9-max filters for max pooling will particularly be emphasized in this work.

Another primary motivation for creating these novel single-stage sorters and filters is to use them in fast multiway-merge sorting networks. In the FPGA designs we cover in this article, the sorters and filters introduced here sort a maximum of 9 input values. Fast sort and filter operations for more than 9 values requires that the novel single-stage sorters and filters described here are used in equally novel multiway-merge sorting networks. When sorting larger lists of values, the sorting networks that use these novel single-stage hardware blocks promise to be significantly faster than the current state-of-the-art Odd-Even Merge Sort and Bitonic Merge Sort algorithms. However, a general discussion concerning these promising multiway-merge sorting networks is beyond the scope of this work.

Definitions and a background review are found in Section II. Section III starts off with an example and discussion of how Comparison Counting [5] operates. Our general design system is based on a novel implementation of Comparison Counting, and most of the rest of Section III details how an N-sorter is created in RTL using our system. Design modifications for N-max and N-min filters are also covered.

Section IV describes how the novel N-sorters and N-filters are implemented in FPGAs, and Section V compares the Synthesis results of our single-stage FPGA designs to those of state-of-the-art sorting networks. The three appendices contain speed and resource usage tables for the N-sorters and N-filters, as well as 2-sorter networks, and show how the speed data is used to calculate throughput values and, when appropriate, video frame rates.

II. DEFINITIONS AND BACKGROUND

A. DEFINITIONS

N-Sorter (Nsrtr): An N-sorter is a single-stage hardware block which fully sorts N input values. It consists of N input value ports, logic to perform all $(N*(N-1)/2)$ 2-value comparisons of the input signals, possibly additional internal logic, and the N output value ports which contain the sorted list of values. The commonly used 2-sorter shown in Fig. 1 is the simplest example of an N-sorter, and Fig. 3 generally represents all N-sorter designs.

N-Filter: An N-filter is a single-stage hardware rank order filter. An N-filter is similar to an N-sorter, except that the output port values are only a subset of the sorted list of N input values. Examples of N-filters in which there is only one output value include N-max, N-median, and N-min filters.

Sorting Network (Ntwrk): A sorting network consists of a network of small single-stage hardware sorters and filters, connected in such a way as to sort lists larger than what can be sorted by a single-stage sorter or filter. The small N-sorters and N-filters used in traditional sorting networks are 2-sorters and 2-max and 2-min filters.

An example of such a sorting network is the 4-network shown earlier in Fig. 2. The square blocks shown in Fig. 2 are single-stage hardware 2-sorters, as shown in Fig. 1.

Propagation Delay: A shortened version of worst case propagation delay, propagation delay is the time required for an input signal to propagate to an output along the slowest path in a single-stage or network sorting block.

Max Frequency (MaxFreq): Max frequency is the inverse, 1/propagation_delay, of the worst case propagation delay. This is the maximum clock frequency in which the hardware block is able to complete its sort operation within a single clock cycle.

Speedup: The speedup of an N-sorter versus a comparable 2-sorter N-network is the ratio of the network's propagation delay to the N-sorter's propagation delay.

Resource Increase Ratio: The resource increase ratio of an N-sorter, versus a comparable 2-sorter N-network, is equal to the number of N-sorter hardware resources divided by the number of N-network resources.

Stable Sort [5]: In a stable sort, equal values in the output sorted list are presented in the same order that these values are found in the input list. This is important if these values are keys in key/value pairs.

Mux: Multiplexer

B. N-SORTER AND N-FILTER BACKGROUND

The state-of-the-art algorithms for sorting speed-of-operation in FPGAs are Odd-Even Merge Sort and Bitonic Merge Sort, both of which were introduced in Kenneth Batcher's classic paper [1]. Recent technical papers and texts confirm that these two algorithms are still the state-of-the-art [2]–[4]. These network algorithms use sets of 2-sorters operating in parallel, so their speed is limited by the speed of a 2-sorter.

Various single-stage hardware 2-sorter designs have been described in the technical literature. The 2-sorter shown in Fig. 1 seems to be the most commonly-used 2-sorter currently found in FPGAs or similar hardware types. This type of bit-parallel 2-sorter was described in the literature as early as 2001 [9], and has quite possibly been described earlier. This 2-sorter consists of 2 input ports, a 2-value comparison block to compare the values, and 2 sets of 2-to-1 output multiplexers, in which the comparison result signal is used as the select line for each 2-to-1 multiplexer.

As mentioned in the introduction, the only single-stage filters previously defined were 2-max and 2-min filters, each of which is created by removing the unneeded output port. In [7], a full 2-sorter is called a ''compare-swap element'', and a 2-max or 2-min filter is called a ''select-value element''.

A number of researchers have noted the probable advantages for single-stage N-sorters with $N \geq 3$, but made no attempt to define how such N-sorters could be built. One such paper described how use of single-stage ''k-sorters'' would improve the speed of multiway-merge sorting networks [10]. A more recent paper using similar multiway-merge concepts suggested building ''n-sorters'' using threshold logic [11]. No details were given as to how to build such n-sorters, and

threshold logic does not exist in the FPGAs which are targeted in this work.

In another relatively recent technical paper concerning sorting in FPGAs [7], the authors note that it would be advantageous to sort more than 2 values in a single-stage hardware sorter. The main advantage that these authors see is that fewer hardware resources would presumably be needed for a single-stage N-sorter, versus a multi-stage network of 2-sorters. Once again, there is no discussion in this article concerning how such a larger N-sorter would be built.

Several researchers have attempted to design single-stage hardware N-sorters for $N \geq 3$, with mixed results. One patent that discusses single-stage hardware N-sorters, with N>2, was granted in late 1986 [12]. The patent's Figure 5 appears to setup output multiplexers to correctly sort a list of 3 unsorted values. This Figure 5 constructs the output multiplexers from a 2-level set of AND/OR gates.

However, the inventor does not develop a system of equations for his 3-sorter that can be used to design larger sorters, or even explain the 3-sorter he has created. Since no equations are presented, there is also no information on how the 3-sorter design could be ported to a different hardware type.

A single-stage hardware 3-sorter, called the "Sort-3a Unit", was discussed in 2 technical papers. In one [13], only an equation for the max of 3 values was given. In the second [14], equations for all 3 max, median, and min outputs were presented. The problem with this sorter definition is that, when all 3 values are equal, the max and min values are set to 0, no matter what the common input value is.

One paper shows a block diagram of a hardware 3-sorter in its Figure 2 [15]. However, no equations were given to show how the max, median, and min values are produced.

One patent application does attempt to define a system for building hardware N-sorters for any number of N input values [16]. Although no prior art was referenced in this application, the initial portion of his algorithm seems to be a reasonably straightforward version of **Comparison Counting**, which Donald Knuth describes as Algorithm C, in Section 5.2 of the 1998 2nd Edition of his "Sorting and Searching" textbook [5]. Knuth also seems to call this algorithm **Enumeration Sort**, and notes that this algorithm was first published by Friend [17].

This patent application first attempts to use comparison result counting to determine the output ranks of the N input values, using a hardware NxN comparison matrix. In the matrix, comparison signals of an input with itself are allowed to be hardwired to 1. However, for 2 separate input signals, InA and InB, both InA≥InB and InB≥InA comparison signals are constructed, even though only one of these is needed. This means that his comparison array has $N*(N-1)$ comparison signals, twice as many as the $N*(N-1)/2$ comparisons that are needed.

After the comparison matrix is constructed, the patent application defines an excessive 13 more steps to produce multiplexer select lines for the output port multiplexers. Included in these steps are shift register operations, which transforms a combinatorial algorithm into a more time-consuming one that requires clocking.

III. GENERAL N-SORTER/N-FILTER DESIGN SYSTEM

In Section III, we present our novel methodology for creating stable single-stage N-sorters, with $N \geq 3$. The N-sorter design system described here will work for any hardware type that uses an HDL to specify its designs. The particular HDL used in this work is SystemVerilog (SV) [18], but the design principles defined here are easily implemented using other HDLs as well. Since SystemVerilog uses C syntax for logical equations, the SV RTL code displayed in the following sections should be fairly easy for all to understand.

Section IV will describe how this general design system is optimized in order to implement N-Sorters and N-filters in FPGAs, and FPGA results will then be presented in Section V and in the appendices. However, the principles discussed in this section are not hardware-specific, and can possibly be adapted for software algorithms as well.

Our innovative algorithm uses Comparison Counting to produce fast and efficient N-sorters, and Section III-A shows how Comparison Counting works, using easily understood matrix figures. Sections III-B, III-C, and III-D then detail exactly how our N-sorters are designed, and how Comparison Counting is incorporated in the design methodology.

Section III-E shows how our stable, **non-increasing** N-sorter designs are able to produce stable, **non-decreasing** N-sorters, without modification to the base design. Finally, Section III-F discusses design of single-stage N-filters in general and how design of max and min filters enable their especially fast operation.

A. COMPARISON COUNTING MATRIX EXAMPLE

A comparison result matrix is used here to help clarify exactly how the comparison result counting algorithm works. The comparison matrix configuration for a 7-Sorter is shown in Table 2.

All N-sorters require $N*(N-1)/2$ 2-value comparison result values, which is the number of combinations of 2 values in a list of the N input values. For the 7-sorter, there are then $21 = 7*6/2$ comparison result signals. The unhighlighted lower left half of Table 2 contains the 21 comparison result signals for the 7-Sorter.

The name of each comparison signal is a concatenation of "ge", for ≥, the input port number on the left side of ≥, and then the input port number on the right side of the ≥ operator. For example, the SV initialization statement for ge65 is

```
wire ge65 = ( In_6 >= In_5 );
```

In each column of unhighlighted cells, the comparison results signals are listed which have the column input on the left side of the ≥ operator. In each row of the unhighlighted section, the comparison results signals are listed which have the row input on the right side of ≥ operator.

The lower left half of the matrix is separated from the upper right half by a diagonal of blank cells, in which an input would otherwise be compared to itself. The highlighted

TABLE 2. N-sorter comparison result matrix: 7-sorter configuration.

Right Side		Left Side In_6	Left Side In_5	Left Side In_4	Left Side In_3	Left Side In_2	Left Side In_1	Left Side In_0
	Signals	ge6–	ge5–	ge4–	ge3–	ge2–	ge1–	ge0–
In_6	ge–6		! ge65	! ge64	! ge63	! ge62	! ge61	! ge60
In_5	ge–5	ge65		! ge54	! ge53	! ge52	! ge51	! ge50
In_4	ge–4	ge64	ge54		! ge43	! ge42	! ge41	! ge40
In_3	ge–3	ge63	ge53	ge43		! ge32	! ge31	! ge30
In_2	ge–2	ge62	ge52	ge42	ge32		! ge21	! ge20
In_1	ge–1	ge61	ge51	ge41	ge31	ge21		! ge10
In_0	ge–0	ge60	ge50	ge40	ge30	ge20	ge10	
Column	Total	6	5	4	3	2	1	
Running	Total	21	15	10	6	3	1	

upper right half of the comparison matrix does not contain new comparison result signals. Rather, this half of the matrix is populated by the complemented states of the 21 comparison results which populate the bottom left matrix half. The cell containing the complemented state of a comparison result is mirrored from the cell containing the uncomplemented state, with the "mirror" being the diagonal of blank cells.

The 6 signal states in a given column are the **winners** for the input in that column. That is, each listed signal state indicates that the column input is $\geq$ than the row input in that cell.

To use the comparison matrix for a given input list of values, the input values are placed into the appropriate column and row headers, as shown in the top table in the Table 3 results matrix set. A cell in the lower left half of the matrix is filled in according to whether the column input is $\geq$ the row input. The cells in upper right half of the matrix are filled in with the complemented state of the associated cells in the lower left half of the matrix.

TABLE 3. N-sorter comparison result matrix: 7-sorter input list example.

Right Side		Left Side In_6	Left Side In_5	Left Side In_4	Left Side In_3	Left Side In_2	Left Side In_1	Left Side In_0
	Values	21	18	21	9	24	18	21
In_6	21		.	.	.	1	.	.
In_5	18	1		1	.	1	.	1
In_4	21	1	.		.	1	.	.
In_3	9	1	1	1		1	1	1
In_2	24	.	.	.	.		.	.
In_1	18	1	1	1	.	1		1
In_0	21	1	.	1	.	1	.	
Column	Total	5	2	4	0	6	1	3
Out_Y		Out_5	Out_2	Out_4	Out_0	Out_6	Out_1	Out_3

In the results matrix, a '1' represents true. A '.' (period) represents false

Signals that are true:	Output	Value	Input
In_2_goes_to_Out_6	Out_6	24	In_2
In_6_goes_to_Out_5	Out_5	21	In_6
In_4_goes_to_Out_4	Out_4	21	In_4
	Out_3	21	In_0
In_5_goes_to_Out_2	Out_2	18	In_5
In_1_goes_to_Out_1	Out_1	18	In_1
In_3_goes_to_Out_0	Out_0	9	In_3

For each column, the 1's are summed up and the total is entered into the Column Total row of the Table 3 top table. The column total for a given column indicates which output that column input goes to. The column totals are distinct, even though there are duplicate values in the input list.

The smaller bottom table in the Table 3 set rearranges some of the data from the top table, and is sorted according to the output port number. The highlighted rows in this table indicate that there are the two values which are duplicated in the input list. The value 21 occurs 3 times in the input list, and the value 18 occurs twice.

As shown in the smaller table, for a set of duplicate values, the output order of those duplicated values matches the order of those values in the input list. This indicates that this 7-Sorter (and all N-Sorters created using this system) produces a stable sort.

The comparison result matrix can aid in understanding how Comparison Counting operates, but this type of matrix is not actually implemented in an N-sorter definition, and no column adders are used. Sections III-B to III-D will present the details of how an N-sorter in our system implements the Comparison Counting algorithm presented in this section.

B. PORT AND COMPARISON SIGNAL DEFINITIONS

Specific notation will be used to define the general N-sorter design system, beginning with the naming of the input and output port signals. Fig. 4 shows the naming of the port signals for a 7-sorter defined in SV code. In Fig. 4, the input and output values are all 32-bit unsigned integers, but the methodology can be implemented for any type of numeric value, and for any bit width.

```
module sort_7_values
   (
    input  [ 31 : 0 ]  In_6   ,
    input  [ 31 : 0 ]  In_5   ,
    input  [ 31 : 0 ]  In_4   ,
    input  [ 31 : 0 ]  In_3   ,
    input  [ 31 : 0 ]  In_2   ,
    input  [ 31 : 0 ]  In_1   ,
    input  [ 31 : 0 ]  In_0   ,

    // the max value
    output [ 31 : 0 ]  Out_6  ,
    output [ 31 : 0 ]  Out_5  ,
    output [ 31 : 0 ]  Out_4  ,
    output [ 31 : 0 ]  Out_3  ,
    output [ 31 : 0 ]  Out_2  ,
    output [ 31 : 0 ]  Out_1  ,
    output [ 31 : 0 ]  Out_0
    // the min value
    ) ;
```

FIGURE 4. N-sorter SystemVerilog port signal definitions: 7-sorter example.

The maximum output value is found at the output port with the maximum numeric suffix. For this 7-sorter, the maximum value is at Out_6. The minimum value is always found at Out_0, no matter what the size of the sorter. The other output values are in sorted order, between the maximum and minimum values.

The input values have the same numeric suffixes as the output values. The values at the input ports are unordered, of course, but the input ports are used in a very systematic manner inside the N-sorter. In particular, the $(N*(N-1)/2)$ comparison signals for the N input ports are strictly defined, as is shown in the 2 columns of signal declarations in Fig. 5. The comparison operator is always "$\geq$", and the input with the higher numeric suffix is always on the left side of the $\geq$ operator. The comparison signals are all generated in

```
// 21 comparisons for  7-sorter      // 15 comparisons for  6-sorter

wire   ge65  = ( In_6 >= In_5 ) ;
wire   ge64  = ( In_6 >= In_4 ) ;    wire   ge54  = ( In_5 >= In_4 ) ;
wire   ge63  = ( In_6 >= In_3 ) ;    wire   ge53  = ( In_5 >= In_3 ) ;
wire   ge62  = ( In_6 >= In_2 ) ;    wire   ge52  = ( In_5 >= In_2 ) ;
wire   ge61  = ( In_6 >= In_1 ) ;    wire   ge51  = ( In_5 >= In_1 ) ;
wire   ge60  = ( In_6 >= In_0 ) ;    wire   ge50  = ( In_5 >= In_0 ) ;

// 10 comparisons for  5-sorter      //  6 comparisons for  4-sorter

wire   ge43  = ( In_4 >= In_3 ) ;
wire   ge42  = ( In_4 >= In_2 ) ;    wire   ge32  = ( In_3 >= In_2 ) ;
wire   ge41  = ( In_4 >= In_1 ) ;    wire   ge31  = ( In_3 >= In_1 ) ;
wire   ge40  = ( In_4 >= In_0 ) ;    wire   ge30  = ( In_3 >= In_0 ) ;

//  3 comparisons for  3-sorter      //  1 comparison  for  2-sorter

wire   ge21  = ( In_2 >= In_1 ) ;
wire   ge20  = ( In_2 >= In_0 ) ;    wire   ge10  = ( In_1 >= In_0 ) ;
```

FIGURE 5. The $(N*(N-1)/2)$ comparison result signals: 7-sorter example.

parallel in the Comparison Signals Block, the top left block shown in Fig. 3.

Hardware synthesis tools typically provide a simple and efficient hardware block which produces a 2-value comparison signal, and it is assumed that this default block will be used. If desired, the default 2-value comparison block can be replaced with a user-designed block, but this will not impact the general N-sorter design system described here. The 2-value comparison block size scales with bit width, so a 32-bit block will utilize 4 times the number of hardware resources used by an 8-bit block.

C. OUTPUT MULTIPLEXERS

The output multiplexers define the output ports for the sorted list of inputs, and operate in parallel in the Output MUX Block shown at the right in Fig. 3. In the N-sorter general design system, the code in the output mux block is quite simple. Fig. 6 shows the 7-sorter Out_2 RTL output port assignment. The other 6 output ports for the 7-sorter are assigned in the same manner. These output port assignments use C-style ternary or conditional syntax.

```
assign  Out_2
    =
       ( In_6_goes_to_Out_2  ?  In_6 :
       ( In_5_goes_to_Out_2  ?  In_5 :
       ( In_4_goes_to_Out_2  ?  In_4 :
       ( In_3_goes_to_Out_2  ?  In_3 :
       ( In_2_goes_to_Out_2  ?  In_2 :
       ( In_1_goes_to_Out_2  ?  In_1 :  In_0  ) ) ) ) ) ) ;
```

FIGURE 6. N-to-1 output multiplexer using ternary notation: 7-sorter example.

Although perhaps not obvious, the simple output port assignment shown in Fig. 6 specifies that one output multiplexer will be constructed for each bit of the output port. Therefore, the output bit multiplexers scale with bit width. A 32-bit port will use 4 times the number of hardware resources that an 8-bit port uses.

Like any general N-sorter output assignment, the Fig. 6 assignment contains all N input values, as well as $N-1$ multiplexer select lines. The select line signals, which have In_X_goes_to_Out_Y names, determine which input port value goes to a particular output port. A maximum of 1 of the select line signals can be true for a given output port Out_Y.

Note that no In_0_goes_to_Out_Y signal is needed. If none of the other In_X_goes_to_Out_Y signals is true, then In_0, by default, must be the input that goes to Out_Y.

In the 7-sorter results matrix shown in Table 3, the In_0 column is highlighted in green. Although the winners total in this column indicates that In_0 will go to Out_3, no In_0_goes_to_Out_3 signal is used in our system. The In_0 data is transferred to Out_3 because no other In_X_goes_to_Out_3 signal is true.

D. OUTPUT MUX SELECT LINE SIGNALS

The definition of the In_X_goes_to_Out_Y multiplexer select line signals is at the heart of the N-sorter design system, as these signals implement the novel Comparison Counting algorithm in our design methodology. These signals are generated in parallel in the Output MUX Select Line Signals Block at the bottom left in Fig. 3.

For each input In_X, the comparison signals that determine which output that In_X goes to are the $N-1$ comparison signals in which In_X is compared to the other inputs. In the 7-sorter matrix Table 2, the 6 signals that determine which output an input goes to are the 6 signals listed in the column for that input.

There are 2^{N-1} permutations of the 1 and 0 states of these $N-1$ comparison signals, and in our design system each permutation becomes a product term in one of the N In_X_goes_to_Out_Y SOP equations for that input. In order to determine which In_X_goes_to_Out_Y equation a product term belongs in, the number of winners in the product term is totaled. Once again, a winner for an input is a comparison result that indicates that input is $\geq$ the input it is compared to. The number of winners in an input's product term, when that product term is true, indicates the output port that the input goes to. Once the number of winners is determined, the product term is OR'd into the In_X_goes_to_Out_{Total_of_winners} SOP equation.

Fig. 7 shows the full SOP equation for the 7-sorter's select line signal In_6_goes_to_Out_5. For In_6, all of the winners are uncomplemented signal states. Each of the product terms in Fig. 7 has 5 uncomplemented winners, and one complemented signal state that is not a winner. If any product term in this SOP equation is true, then 7-sorter input In_6 will go to output Out_5.

```
wire    In_6_goes_to_Out_5  =

(   ge65  &&   ge64  &&   ge63  &&   ge62  &&   ge61  && ! ge60  ) ||
(   ge65  &&   ge64  &&   ge63  &&   ge62  && ! ge61  &&   ge60  ) ||
(   ge65  &&   ge64  &&   ge63  && ! ge62  &&   ge61  &&   ge60  ) ||
(   ge65  &&   ge64  && ! ge63  &&   ge62  &&   ge61  &&   ge60  ) ||
(   ge65  && ! ge64  &&   ge63  &&   ge62  &&   ge61  &&   ge60  ) ||
( ! ge65  &&   ge64  &&   ge63  &&   ge62  &&   ge61  &&   ge60  )   ;
```

FIGURE 7. One of $N*(N-1)$ N-sorter MUX select lines: 7-sorter example.

It should be clear that each product term in the Fig. 7 SOP equation has exactly 5 winners. After some thought, it should

also be clear that these are the only product terms for In_6 that have 5 winners. This set of product terms is the necessary and sufficient set that will transfer the In_6 value to Out_5.

There is one particular product term, which is highlighted in Fig. 7, and the signal states in this product term match those listed in the In_6 column in Table 3. Since only one of the 2^6 product terms can be true for a given set of inputs, only one of the In_6_goes_to_Out_Y select line SOP signals can be true as well. For the example data used in Table 3 then, our 7-sorter design will ensure that In_6 goes to Out_5, and no other In_6_goes_to_Out_Y select line signal, is true.

E. REVERSING THE OUTPUT SORTED ORDER

In the system described thus far, the stable sorted order of the output list is non-increasing going from port Out_(N-1) down to Out_0. A non-decreasing stable sorted order is easily implemented without modifying the system that has already been described. Fig. 8 shows how this is done for a list of 5 values.

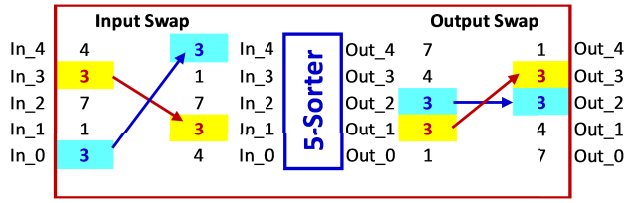


FIGURE 8. Stable non-decreasing output list using non-increasing sorter.

If the non-increasing system is instantiated inside a higher level SV module, an output list with a reverse sorted order is implemented by reverse mapping the output ports of the higher level module versus the output ports of the base non-increasing sorter module. If only the output ports are reverse mapped, the upper level output list would be non-decreasing, but not stable. In order to create a stable non-decreasing output list, the input list also needs to be reverse mapped between the upper level port list and base instantiated port list.

F. RANK ORDER N-FILTER DESIGN

An N-filter can be implemented directly from an N-sorter, simply by removing the unused output ports, and any internal logic that only drives those unused ports. N-median filters, when N is odd, are designed in this manner. Any filter that is created this way will have reduced hardware usage, but will have essentially the same propagation delay as the full N-sorter. All of the comparison result signals needed for a full N-sorter are still needed for any associated N-filter, i.e. none of the comparison result signals can be removed.

N-max and N-min filters can also be implemented by simple removal of unused output ports and associated logic. However, N-max and N-min filters have unique characteristics that allow an alternate methodology to often be used for their design. The unique characteristics are shared between N-max and N-min filters, so the rest of this article will focus only on N-max filters. In general, the N-max filter discussion will serve as an N-min filter discussion as well.

The In_X_goes_to_Out_Y SOP equations for the N-max port have only 1 product term. Also, if the N-max SOP equation for an input contains a particular comparison signal state, the other input which shares that comparison signal will always have an opposite signal state in its N-max equation. This can be seen in the compact 4-max In_X_goes_to_Out_3 table that is shown in SV comments at the top of Fig. 9.

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Signal states for Out_3 (max) single product terms:
//
//                        g g g  g g  g
//                        e e e  e e  e
//                        3 3 3  2 2  1
//                        2 1 0  1 0  0
//                        -------------
//  In_3_goes_to_Out_3  :  1 1 1
//  In_2_goes_to_Out_3  :  @     1 1
//  In_1_goes_to_Out_3  :    @   @   1
//  In_0_goes_to_Out_3  :      @   @ @
//                        -------------
//
//  '1' represents true ; '@' represents false
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

assign Out_3 =

 ( ge32 ? ( ( { ge31, ge30 } == 2'b11 ) ? (               In_3 )
                                        : ( ge10 ? In_1 : In_0 )
          )

        : ( ( { ge21, ge20 } == 2'b11 ) ? (               In_2 )
                                        : ( ge10 ? In_1 : In_0 )
          )
  ) ;
```

FIGURE 9. 4-max compact table; 4-max output assignment using ge* signals.

Because of this behavior, an N-max output port equation can be constructed directly from the ge* comparison result signals, using ternary notation. This type of equation, for a 4-max filter, is shown in the uncommented SV code at the bottom of Fig. 9. The In_X_goes_to_Out_3 equation information shown in the compact table is used to build the ternary equation in the SV RTL, but the In_X_goes_to_Out_3 equations themselves are not implemented in the code.

IV. N-SORTER/N-FILTER FPGA IMPLEMENTATION

The RTL equations developed in accordance with the general N-sorter/N-filter design system, introduced in Section III, can be directly implemented in Xilinx FPGAs using the Vivado synthesis tool. The resultant designs will be fully functional, but they may not maximize speed of operation and minimize resource usage. In this and following sections, the general design system will be modified, as needed, in order to produce speed and resource optimized N-sorter and N-filter designs in FPGAs.

The FPGAs covered here are the devices in the Xilinx 7-Series, Ultrascale, and Ultrascale+ FPGA families. These are the most recently released Xilinx FPGA families and the main families in which designs are currently being implemented. Ultrascale+ products are used in the Amazon

AWS EC2 F1 system as well as Xilinx's own Alveo datacenter acceleration cards.

In these families, the basic logic blocks used for our N-sorter/N-filter designs are **slices**, such as the slice logic block shown in Fig. 10 [19]. The Ultrascale and Ultrascale+ slice block is the full block shown Fig. 10. This slice has 8 6-input lookup tables (LUTs), and a set of 7 2-to-1 multiplexers which are used to combine LUT output signals. The 7-series slice has 4 LUTs and 3 2-to-1 multiplexers [20]. The bottom half of Fig. 10, without the MUXF9, shows the design resources available for 7-series designs. All of the designs detailed in this work only need the resources available in the 4-LUT 7-series slice, so they can be implemented in both 7-series devices and in devices in the Ultrascale and Ultrascale+ FPGA families.

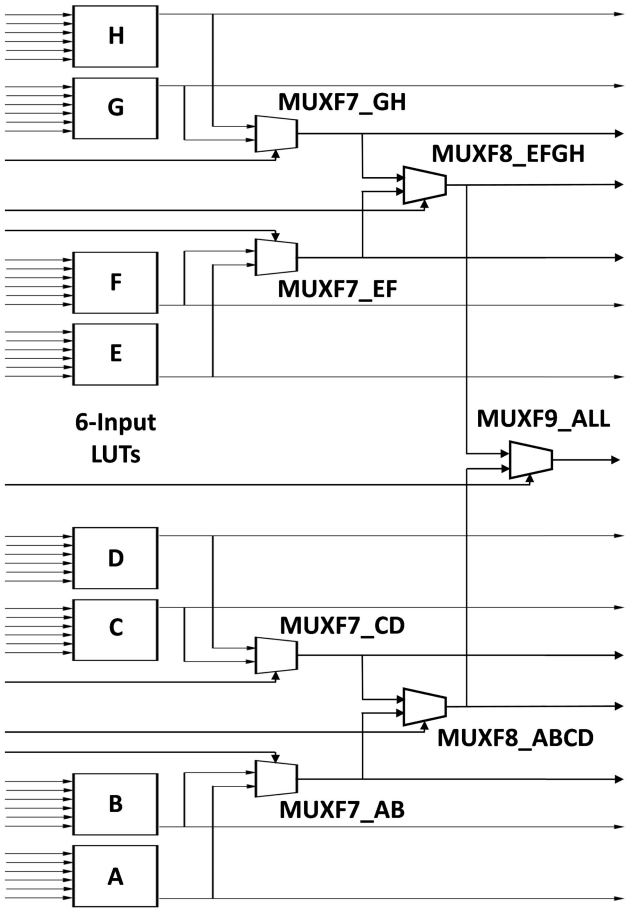


FIGURE 10. Xilinx FPGA slice utilizing 8 6-input LUTs: 7 2-to-1 multiplexers.

There are other hardware structures found in the slice blocks, but not shown in Fig. 10. These additional structures are not normally used in the design of N-sorters and N-filters, and are not discussed here.

The general design and data flow diagram, Fig. 3, is modified for FPGA design and data flow, and is shown in Fig.11. Each block in the Fig.11 diagram represents a group of slices operating in parallel, and the number of slice groups in series

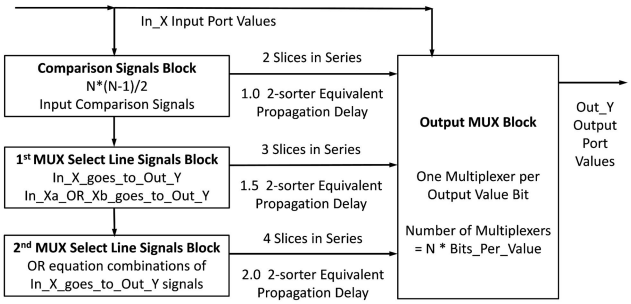


FIGURE 11. FPGA LUT N-sorter design and data flow.

is listed for each of the possible paths that go through the Comparison Signals Block. The possible paths through the Comparison Signals Block are the slowest paths, the paths that determine propagation delay. The fastest sorters are those in which the slowest signals propagate through only 2 slice groups, and the slowest sorters are those in which its slowest signals travel through all 4 slice groups in Fig.11.

Our main objective, when adapting the general N-sorter design methodology for use in the target FPGAs, has been to focus on the speed of the N-sorter operation. This main objective translates into minimizing the number of **series slices** that an N-sorter's slowest signals propagate through. We estimate that the relative propagation delay of hardware block A versus block B is proportional to the number of block A's series slices to that of block B. Starting in Section V, these estimates will be compared to real propagation delay ratios extracted from FPGA synthesis results.

A. TYPICAL N-SORTER FPGA DESIGN MODIFICATIONS

No FPGA modifications are needed for port list RTL, like that shown in Fig. 4 in Section III-B. The port list is a simple mapping of the N-sorter to the input and output lists, and contains no logic in itself.

Also, there is no need to modify the definitions of the comparison result signals, as shown in Fig. 5 in the same section. Each comparison operation is implemented by a hardware vendor's synthesis tool, and requires no input by a designer. All of the comparison result operations are performed in parallel in the Comparison Signals Block in Fig. 11.

FPGA modifications are typically required for the RTL code in the Output MUX Block, versus the standard design example shown in Fig. 6. When there are Output MUX Block changes, changes are required for the output mux select lines as well. The select line signal changes are implemented in Fig. 11's 1st MUX Select Line Signals Block and possibly the 2nd MUX Select Line Signals Block. Discussion of these needed modifications will be covered in the next sections.

B. FPGA N-SORTER IMPLEMENTATION DETAILS

Each bit of an output port value is implemented using the minimal number of Fig. 10 slice resources. The 2-sorter output bit multiplexer only requires 2 input bit values and

1 comparison result signal, so it is easily implemented in a single 6-input LUT. Actually, since 4 input bit values and 1 comparison result signal will fit in a 6-input LUT, the Xilinx Vivado 2018.2 synthesis tool implements 2 output bits in each 2-sorter multiplexer LUT. A 3-sorter multiplexer requires 3 input bit values, and 3 comparison result signals, so it is also easily implemented in a single 6-input LUT.

Since the 2-sorter and 3-sorter output bit multiplexers only require sorter input data and comparison result signals as their inputs, these two sorters have the minimum 2 series slices. Refer to rows 1 to 5 in Table 4 for data covering the number of series slice blocks used by each FPGA N-sorter.

TABLE 4. FPGA N-sorter Series Slices and LUT Usage Data.

Row	Number of Values =>	2	3	4	5	6	7	8	9
1	Comparison Signals Block	X	X	X	X	X	X	X	X
2	1st MUX Select Lines Block			X	X	X	X	X	X
3	2nd MUX Select Lines Block					X	X	X	X
4	Output MUX Block	X	X	X	X	X	X	X	X
5	Total FPGA Series Slices	2	2	3	3	4	4	4	4
6	N-Sorter Equivalent Stages	1	1	1.5	1.5	2	2	2	2
7	In_X_goes_to_Out_Y LUTs			1	1	1	1	2	4
8	Output Mux LUTs per Bit	0.5	1	1	2	2	2	2	4

Row 5 contains the count of series slice block X's in Rows 1 to 4
Row 6 = Row 5 / (Row 5, Column 2) = Row 5 / 2
Data in highlighted rows is used for matching rows in Table 1
The 2-sorter has 2 bits in each Output Bit Mux LUT

It may seem that no In_X_goes_to_Out_Y signals are needed for the 2-sorter and 3-sorter. However, what actually happens is that the appropriate In_X_goes_to_Out_Y signals do exist, but they are implemented inside the output bit multiplexer LUTs.

For any N-sorter larger than a 3-sorter, the simple type of code shown in Fig. 6 will not automatically produce fast and resource efficient FPGA output multiplexers. Therefore, the output multiplexer code for larger sorters requires modification, which in turn requires changes to the associated multiplexer select line signals as well.

The 4-sorter is the first N-sorter that requires output multiplexer modification. If the type of RTL code in Fig. 6 code is used for a 4-sorter, the output bit multiplexer would require 4 data input bits and 3 In_X_goes_to_Out_Y select line signals as inputs, 7 in all. So, this simple design could not be implemented in a single 6-input LUT.

However, a straightforward modification is made so that the 6-input LUT becomes a 4-to-1 multiplexer, with 2 mux select lines, allowing a 4-sorter output multiplexer to be fit into a single LUT. The functionality of the set of 3 In_X_goes_to_Out_Y signals required by the general design system:

In_3_goes_to_Out_Y,
In_2_goes_to_Out_Y,
In_1_goes_to_Out_Y,

is captured in two OR signals:

In_3_goes_to_Out_Y || In_2_goes_to_Out_Y,
In_3_goes_to_Out_Y || In_1_goes_to_Out_Y,

and these two OR signals become the 2 mux select lines for the 4-sorter 4-to-1 single-LUT output multiplexer.

The select line modifications that are needed for 4-sorters and larger sorters all involve the OR'ing of 2 or more In_X_goes_to_Out_Y signals. For the 4-sorter and 5-sorter, the creation of these OR signals is implemented in Fig. 11's 1st MUX Select Line Signals Block, so these two sorters use 3 series slice blocks.

For sorters larger than a 5-sorter, the OR'ing of In_X_goes_to_Out_Y signals is implemented in Fig. 11's 2nd MUX Select Line Signals Block. These largest N-sorters then have 4 series slices.

Row 5 in Table 4 contains the N-sorters' series slice count, and row 6 contains the row 5 data normalized to a 2-sorter's series slice block count, which is 2. Normalizing propagation delay to a 2-sorter's delay allows for direct comparisons between an N-sorter and a comparable network of 2-sorters, which has been the state-of-the-art norm. The equivalent stage data in row 6 provides the data for the row with the same name and highlight color in Table 1, which is where we first estimated the speedup of our proposed N-sorters.

Row 7 in Table 4 lists the number of LUT resources required for each In_X_goes_to_Out_Y select line signal. Not shown in the table, each of the sorters which have select line OR signals generated in the 2nd MUX Select Line Signals Block use at least N additional LUTs.

Row 8 in Table 4 lists data on the LUT resources used for each output bit multiplexer. Rows 7 and 8 emphasize that there is increased resource usage for the larger N-sorters, especially a 9-sorter. This higher resource usage will be discussed in more detail in Section V.

In Sections III-B and III-C, it was noted that the number of hardware resources used by logic in the Fig. 3's Comparison Signal Block and Output MUX Block scales with the bit width of the numbers being sorted, which is also true for those two blocks in Fig. 11. However, the signals generated in Fig. 11's 1st and 2nd MUX Select Line Signals Blocks do not change with bit width. Because of this, the sorters which utilize signals in at least one of these blocks, 4-sorters and larger sorters, will tend to be more resource efficient than 2-sorters and 3-sorters as data value bit widths increase.

C. SINGLE-STAGE FPGA N-MAX FILTERS

As was shown in Fig. 9 in Section III-F, ternary equations which directly use ge* comparison signals are used to define an N-max value, which eliminates the need for intermediate In_X_goes_to_Out_Y signals. The code in Fig. 9 creates a fast 4-max filter. The ge32 signal in Fig. 9 separates the Out_3 assignment into two sections. Each section has 3 ge* signals and 3 input port signals, so each section can be fit into a 6-input LUT. The outputs of the two LUTs become the inputs to a MUXF7, as shown in Fig. 10, and signal ge32 becomes the MUXF7 select line signal.

The 4-max design, as well as the 5-max design, now have 2 series slice blocks, versus 3 for the full 4-sorter and 5-sorter. The 6-max to 8-max designs have 3 series slices, versus 4 for

the associated N-sorter. The reduced series slice values are highlighted in row 5 of Table 5.

TABLE 5. FPGA N-max Series Slices and LUT Usage Data.

Row	N Input Values =>	2	3	4	5	6	7	8	9
1	Comparison Signals Blk	X	X	X	X	X	X	X	X
2	1st MUX Select Line Blk					X	X	X	X
3	2nd MUX Select Line Blk								X
4	Output MUX Block	X	X	X	X	X	X	X	X
5	Total Series Slices	2	2	2	2	3	3	3	4
6	N-Max Equivalent Stages	1	1	1	1	1.5	1.5	1.5	2
7	In_X_goes_to_Out_Y LUTs					2	4	2	4
8	Output Mux LUTs per Bit	0.5	1	2	4	2	2	4	4

Row 5 contains the count of series slice block X's in Rows 1 to 4
Row 6 = Row 5 / (Row 5, Column 2) = Row 5 / 2

D. SINGLE-STAGE FPGA 9-MEDIAN FILTER

Median N-filters can be designed for any odd N, but only 9-median FPGA filters will be discussed here. This focus is primarily due to the ongoing importance in image processing of using the medians of 3×3 pixel squares to filter out noise from these pixel groups.

As was outlined in Section III-F, the single-stage 9-median filter is constructed from the full 9-sorter by removing all of the output ports except for the median port, and all internal logic that only supports the removed ports. However, all $9{*}8/2 = 36$ comparison result signals are still required.

V. FPGA N-SORTER AND N-FILTER RESULTS

In this section, speed and resource usage results are presented and discussed for N-sorters and N-filters implemented in FPGAs, as wells as comparable 2-sorter and 2-max N-networks implemented in the same FPGAs. The data was obtained from synthesis results using Xilinx's Vivado 2018.2 synthesis tool.

For each single-stage and network block design, four sets of unsigned integer synthesis results were obtained:

- 8-bit values in 7-series product xc7z045ffg900-2.
- 32-bit values in the 7-series xc7z045 product.
- 8-bit values in Ultrascale+ xcvu9p-flga2577-3-e.
- 32-bit values in the Ultrascale+ xcvu9p product.

The xcvu9p product is the specific Ultrascale+ device used in the Amazon AWS EC2 F1 system.

In order to obtain accurate propagation delay measurements, the synthesized designs include both input and output register banks. The register banks are not a part of the combinatorial sorting block designs. The input register bank simply provides the drivers for all sorting block inputs, and the output register bank provides the loads for all of the sorting block outputs. The raw synthesis propagation delay results for the single-stage N-sorters and N-filters analyzed in this article are listed in the 4 tables in Appendix C.

Resource usage is measured by the number of LUTs in a design. Resource usage values for a design are not product-specific, but do vary with the bit width of the unsigned integers. For example, an 8-bit 4-sorter design uses the same number of LUT resources in both the xc7z045 and xcvu9p devices. The 32-bit 4-sorter designs use significantly more LUTs than the 8-bit designs, but the 32-bit LUT usage is the same for the two devices.

A. N-SORTERS VS 2-SORTER N-NETWORKS RESULTS

Tables 9, 10, and 11 found in Appendix A contain raw synthesis results for N-sorters and 2-sorter N-networks. In addition, these tables contain N-sorter speedup and resource increase ratios, versus the 2-sorter networks, which are derived from the raw results.

The speedup and resource increase ratios have been used to build the two plots in this section. Fig. 12 shows N-sorter speedup and resource usage increase ratios for 8-bit FPGA designs, in both the 7-series xc7z045 and Ultrascale+ xcvu9p products. Fig. 13 shows the same data for 32-bit designs in both products.

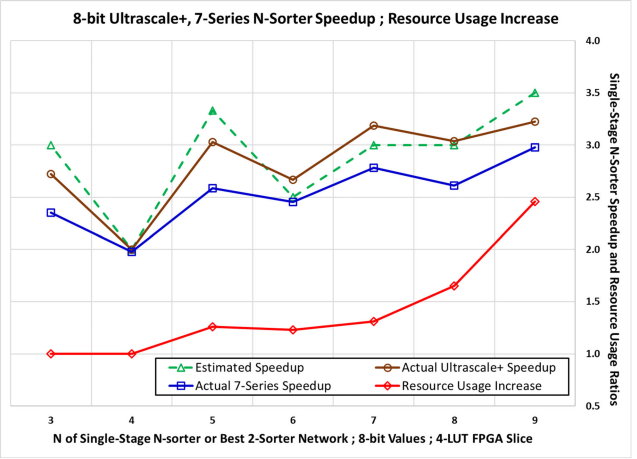


FIGURE 12. 8-bit N-sorter vs N-network speedup and resource increase.

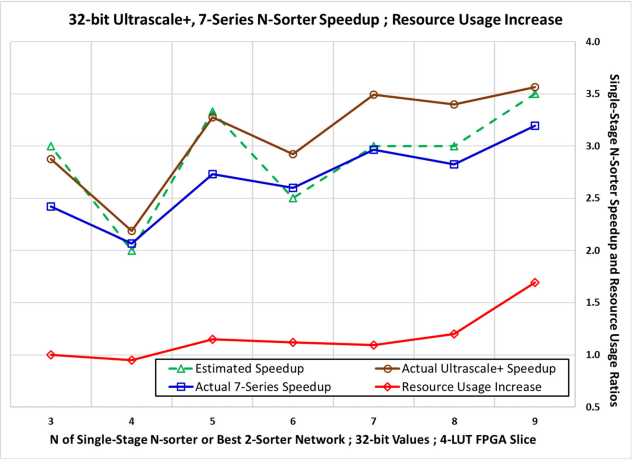


FIGURE 13. 32-bit N-sorter vs N-network speedup and resource increase.

The dashed green curve in both figures is the estimated N-sorter speedup, which was originally listed in Table 1 in the introduction. This data has also been listed Table 4 in

Section IV-B, where the generation of this estimated data was discussed. This green curve is identical in each figure.

The actual speedup values for the xc7z045 are shown in the blue curves in Figs. 12 and 13, and the actual xcvu9p speedups are shown in the brown curves in the figures. The resource increase ratios, which are the same for the xc7z045 and xcvu9p devices, are shown in red in the two figures.

The Ultrascale+ device consistently shows better speedup performance versus the 7-series device, particularly for 32-bit designs. The estimated speedup values in both figures range from 2.0 to 3.5, and the actual speedup values also tend to fall in this same range.

The 32-bit resource usage curve is significantly lower than the 8-bit curve, for the reasons discussed toward the end of Section IV-B. The increase in resource usage ratios as sorter size increases is in line with the data in rows 7 and 8 in Table 4 in Section IV-B. Note that the 32-bit 4-sorter actually uses fewer resources than the comparable 2-sorter 4-network.

Some notable max frequency results, highlighted in yellow in Tables 9 and 10, are worth emphasizing. As shown in these 2 tables, all of the N-sorters complete their sort operation within one period in a 275 MHz 7-series system, and within one 450 MHz period in an Ultrascale+ system. None of the 2-sorter N-networks are able to meet these speed targets.

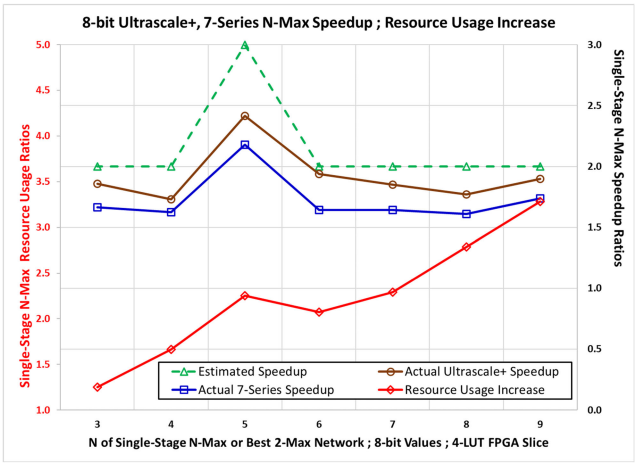
As shown in Appendix A, the data in Tables 9 and 10 are useful for calculating N-sorter throughput values. An Appendix A analysis shows that 10 8-bit 9-sorters, operating in parallel in the slower xc7z045 FPGA, can sort 3 billion 9-input lists in 1 second.

B. FPGA N-MAX VS 2-MAX NETWORK RESULTS

In Appendix B, raw synthesis results and associated ratio calculations for single-stage N-max and 2-max N-network filters are found in Tables 12, 13, and 14. Data from these three tables is used to build the curves plotted in Figs. 14 and 15. These two figures are very similar to Figs. 12 and 13 displayed in Section V-A, except that the resource usage curves in Figs. 14 and 15 are plotted against the new left axis, which has red axis labels to match the red curves.

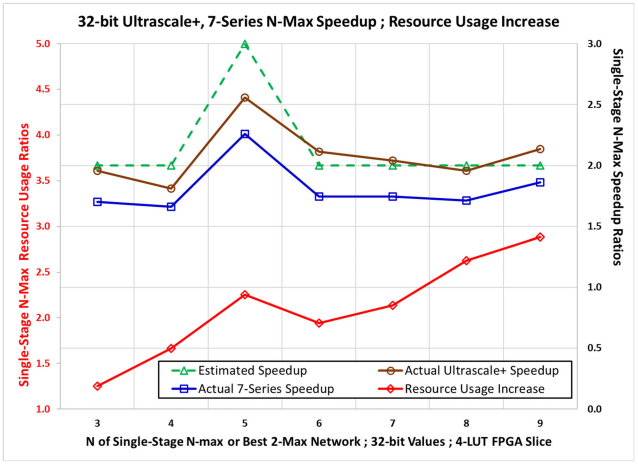
The curves in Figs. 14 and 15, like the earlier ones in Figs. 12 and 13, indicate that the Ultrascale+ product has better speedup than the 7-series device, and 32-bit curves have improved speedup versus 8-bit curves. Once again, the 32-bit resource increase ratios are lower than the 8-bit ratios for larger N-max filters. The single-stage N-max filters in Figs. 14 and 15 do show significant speedup versus the 2-max N-networks. The speedup ratios for the 8-bit and 32-bit designs, for both FPGAs, range from 1.6 to 2.6.

As shown in Appendix B, N-max filter throughput values are easily calculated using the data in Tables 12 and 13. Since max filters are used for max pooling of image filter data, it is also worthwhile to also present max filter throughput values in terms of video frame rates. An analysis



Resource increase ratios are plotted in red against the left axis

FIGURE 14. 8-bit *N*-max vs network filters speedup and resource increase.



Speedups are plotted against the right axis.

FIGURE 15. 32-bit *N*-max vs network filters speedup and resource increase.

in Appendix B indicates that a single 32-bit 9-max filter processes 500 million 9-input lists per second in the xcvu9p FPGA, which equates to a 4K frame rate of 500 frames per second (fps).

C. FPGA 9-MEDIAN RESULTS

Our single-stage 9-median is compared here to two 9-median networks which use a combination of 2-sorter/2-max/2-min blocks. One of the networks is derived from the best 2-sorter 9-network, and the other uses a 3-way merge filter methodology popular in the technical literature [21].

Table 6 lists single-stage vs network 9-median raw and derived synthesis speed data for both the 7-series and Ultrascale+ products, and Table 7 contains the common resource usage data for both products. Our single-stage 9-median filter speedups range from 3.0 to 4.1.

As the xcvu9p 8-bit 9-median can operate at 540 MHz, one 9-median block will process 540 million pixels per second, and 10 9-median blocks operating in parallel will process

TABLE 6. 2-sorter Network vs Single-stage 9-median Speed.

Row	Prop Delay and Speedup	7-Series xc7z045 Single Stage	7-Series xc7z045 Best Ntwrk	7-Series xc7z045 3-way Ntwrk	Ultrascale+ xcvu9p Single Stage	Ultrascale+ xcvu9p Best Ntwrk	Ultrascale+ xcvu9p 3-way Ntwrk
1	FPGA Slices	4	14	16	4	14	16
2	8-b Delay	3.08	9.29	9.93	1.84	6.05	6.88
3	32-b Delay	3.32	10.71	11.55	2.01	7.30	8.31
4	8-b MaxFreq	324	108	101	543	165	145
5	32-b MaxFreq	302	93	87	499	137	120
6	Est. Speedup	1	3.5	4	1	3.5	4
7	8-b Speedup	1	3.01	3.22	1	3.29	3.74
8	32-b Speedup	1	3.23	3.48	1	3.64	4.14

Estimated Speedup: Ntwrk Series Slices / Single-Stage Series Slices
Actual 9-median Speedup: Network Prop Delay / Single-Stage Prop Delay
MaxFreq 7-Series Highlight : sort finishes within one 275 MHz clock cycle
MaxFreq Ultrascale+ Highlight : sort finishes within one 450 MHz cycle

TABLE 7. 2-sorter Network vs Single-stage 9-median LUT resources.

Row	Hardware Resource Data	Single Stage 9-median	Best 2-sorter Network	3-way 2-sorter Network
3	8-bit LUT Resources	210	232	196
4	32-bit LUT Resources	738	928	784
5	8-bit Increase Ratio	1	0.91	1.07
6	32-bit Increase Ratio	1	0.80	0.94

Single-Stage 9-median Increase Ratio : Single-Stage LUTs / Network LUTs

5.4 billion pixels per second. Since 9-median filters are important in image processing, it is also worth presenting throughput numbers in terms of video frames per second.

For a full 4K image size, with 3840 × 2160 pixels, there are less than 9 million pixels per frame. As the xcvu9p 8-bit 9-median can operate at 540 MHz, a single 9-median block will support a 4K frame rate of at least 60 fps, and ten 9-median blocks operating in parallel will then support a 4K frame rate of over 600 fps.

An 8-bit 9-median filter uses 210 LUTs, so 10 such filters will use 2,100 LUTs. These 10 filters use 0.18% of the available 1,182,240 xcvu9p LUT resources. If higher frame rates are needed, additional 9-median filters may be instantiated to operate in parallel. The LUT resource usage is then 210x the total number of 9-median filters, and the minimum frame rate is 60x the number of 9-median filters.

D. FPGA N-SORTER AND N-FILTER VERIFICATION

Verification of the FPGA N-sorters and N-filters introduced here has been implemented using SV simulations performed with the Xilinx Vivado 2018.2 simulation tool. The simulations were performed on the SV RTL code used to define the single-stage blocks.

Verification of N-sorters is a topic that is worthy of further study, but a fairly simple analysis is made here. If a test vector set is created containing all possible N^N permutations of a set of N distinct values, e.g. 1 to N, then the vector set will inherently include all possible input value order permutations. Therefore, passing this vector set guarantees that an N-sorter is correct.

The N^N vector set is easy to create, although the number of vectors in the set quickly becomes very large for increasing values of N. The $N = 2$ vector set has $2^2 = 4$ vectors, the $N = 3$ vector set has $3^3 = 27$ vectors, but the $N = 9$ vector set has $9^9 = 387,420,489$ vectors. All of the FPGA N-sorters from 2-sorters up to 9-sorters introduced in this work have been verified with the appropriate N^N vector set.

Table 8 shows the simulation input and output from a 3-sorter verification run using the 3^3 vector set. All 27 permutations of the 3 input values are applied to the inputs, and the output results show that the input permutation is always correctly sorted. As is shown in the footnotes to Table 8, the 3 values for the 8-bit 3-sorter simulation are chosen so that each bit of each port is tested for correct 0 and 1 operation.

TABLE 8. 3-sorter Simulation Output for $3^3 = 27$ Test Vectors.

Out_2	Out_1	Out_0	|	In_2	In_1	In_0	<	Vec
240	240	240	|	240	240	240	<	0
240	240	15	|	240	240	15	<	1
240	240	0	|	240	240	0	<	2
240	240	15	|	240	15	240	<	3
240	15	15	|	240	15	15	<	4
240	15	0	|	240	15	0	<	5
240	240	0	|	240	0	240	<	6
240	15	0	|	240	0	15	<	7
240	0	0	|	240	0	0	<	8
240	240	15	|	15	240	240	<	9
240	15	15	|	15	240	15	<	10
240	15	0	|	15	240	0	<	11
240	15	15	|	15	15	240	<	12
15	15	15	|	15	15	15	<	13
15	15	0	|	15	15	0	<	14
240	15	0	|	15	0	240	<	15
15	15	0	|	15	0	15	<	16
15	0	0	|	15	0	0	<	17
240	240	0	|	0	240	240	<	18
240	15	0	|	0	240	15	<	19
240	0	0	|	0	240	0	<	20
240	15	0	|	0	15	240	<	21
15	15	0	|	0	15	15	<	22
15	0	0	|	0	15	0	<	23
240	0	0	|	0	0	240	<	24
15	0	0	|	0	0	15	<	25
0	0	0	|	0	0	0	<	26

Non-increasing from Out_2 down to Out_0
integer to 8-bit : 240 – 11110000 ; 15 – 00001111 ; 0 – 00000000

Verification of an N-filter, particularly a single-output filter, presents its own challenges, since there is no output order to check in a single-output filter. Simulation output for a 3-max filter would look like Table 8, except that the Out_1 and Out_0 columns would be missing.

There are two straightforward ways to ensure that N-max simulation results are correct. In one method, a simple software program could be used to check that, for example, the Out_2 output from the 3-max simulation matches the Out_2 output from a full 3-sorter simulation.

Alternatively, a full 3-sorter simulation could be run, but the Out_1 and Out_0 data would not be written out to the log file. The log files from the 3-max and modified 3-sorter simulations are then checked to make sure that they are identical. The single-stage 9-median filter and all of the N-max filters have been verified using this latter method.

VI. CONCLUSION

A methodology has been presented for the design of fast, stable, single-stage N-sorters, with $N \geq 3$, using a novel implementation of Comparison Counting in RTL equations. This methodology has then been adjusted as needed in order to implement fast N-sorters and N-filters in modern FPGA families. When compared to state-of-the-art 2-sorter networks, the single-stage N-sorters are shown to have speedups ranging from 2.0 to 3.5 when sorting up to 9 input values in the target FPGAs. A throughput analysis shows that ten 8-bit 9-sorters, operating in parallel in the slower xcz7045 FPGA, can sort 3 billion 9-input lists in 1 second.

The basic N-sorter system produces stable non-increasing sorted output lists. It has been shown that this base system, without modification, is easily used to implement stable non-decreasing sorted output lists as well.

A modified design system has been presented which produces even faster N-max and N-min filters. The single-stage N-max filters have speedup ratios of 1.6 to 2.6 versus the best existing FPGA max network designs. When a 32-bit xcvu9p 9-max filter is used for max pooling of 4K image filter results, it can process over 500 million non-overlapping 9-input lists in one second, equating to a 4K frame rate of over 500 fps.

Single-stage 9-median speedups range from 3.0 to 4.1, versus the state-of-the-art network implementations. Ten xcvu9p 8-bit 9-median filters, operating in parallel, can produce 5.4 billion output values per second, fast enough to support a 4K frame rate of over 600 fps.

The N-sorters have been verified using comprehensive vector sets, with each set containing N^N test vectors. N-filters have been verified using the same vector sets, and then comparing the N-filter output files to analogous files created during N-sorter verification.

The single-stage designs described in this article all use a 4-LUT slice group, and all are able to be implemented in both 7-series and Ultrascale* devices. Since the Ultrascale* devices have an 8-LUT slice, future efforts will focus on optimizing single-stage N-sorter and N-filter designs which utilize the full 8 LUTs in the Ultrascale* slice.

Although the single-stage sorting blocks are clearly faster than comparable state-of-the-art networks using 2-sorters and 2-filters, the FPGA designs we have introduced process a maximum of 9 input values. The true worth of these single-stage blocks will be realized in the future when they are implemented inside novel multiway-merge sorting networks, enabling the sorting of significantly larger numbers.

APPENDIX A
N-SORTER VS N-NETWORK TABLES

Synthesis results for N-sorters and comparable 2-sorter N-networks are found here in Tables 9, 10, and 11. Table 9 contains 7-series xc7z045 speed data, and Ultrascale+ xcvu9p speed data is found in Table 10. Resource usage data common to both products is found in Table 11. The speedups and resource usage ratios from these three tables have been used to create Figs. 12 and 13 shown earlier in Section V-A.

TABLE 9. 7-Series xc7z045 2-sorter N-network vs N-sorter Speed.

Rw	N: =>	2	3	4	5	6	7	8	9
1	Ntwrk Stages	1	3	3	5	5	6	6	7
2	Ntwrk Slices	2	6	6	10	10	12	12	14
3	8-b Delay	1.62	3.94	4.22	6.71	6.81	7.97	7.97	9.35
4	32-b Delay	1.83	4.55	4.83	7.72	7.83	9.19	9.19	10.77
5	8-b MaxFreq	616	254	237	149	147	125	125	107
6	32-b MaxFreq	548	220	207	130	128	109	109	93
7	Nsrtr Slices	2	2	3	3	4	4	4	4
8	Equiv Stages	1	1	1.5	1.5	2	2	2	2
9	8-b Delay	1.62	1.68	2.13	2.59	2.78	2.87	3.05	3.14
10	32-b Delay	1.83	1.88	2.33	2.83	3.01	3.10	3.25	3.37
11	8-b MaxFreq	616	596	469	386	360	349	328	318
12	32-b MaxFreq	548	532	428	354	332	323	307	296
13	Est. Speedup	1	3	2	3.33	2.5	3	3	3.5
14	8-b Speedup	1	2.35	1.98	2.59	2.45	2.78	2.61	2.98
15	32-b Speedup	1	2.42	2.07	2.73	2.60	2.97	2.82	3.19

2-sorter Network Data in Rows 1-6 ; N-sorter Data in Rows 7-12
Propagation Delay in ns ; Max Frequency in MHz
Estimated N-sorter Speedup : Row 13 = Row 2 / Row 7
Actual N-sorter Speedup : 8-b = Row 3 / Row 9 ; 32-b = Row 4 / Row 10
MaxFreq Yellow Highlight : sort finishes within one 275 MHz clock cycle

TABLE 10. Ultrascale+ xcvu9p 2-sorter N-network vs N-sorter Speed.

Rw	N: =>	2	3	4	5	6	7	8	9
1	Ntwrk Stages	1	3	3	5	5	6	6	7
2	Ntwrk Slices	2	6	6	10	10	12	12	14
3	8-b Delay	0.95	2.67	2.67	4.39	4.39	5.25	5.25	6.11
4	32-b Delay	1.13	3.21	3.21	5.29	5.29	6.33	6.33	7.34
5	8-b MaxFreq	1056	375	375	228	228	190	190	164
6	32-b MaxFreq	888	312	312	189	189	158	158	136
7	Nsrtr Slices	2	2	3	3	4	4	4	4
8	Equiv Stages	1	1	1.5	1.5	2	2	2	2
9	8-b Delay	0.95	0.98	1.34	1.45	1.65	1.65	1.73	1.90
10	32-b Delay	1.13	1.12	1.47	1.61	1.81	1.81	1.86	2.06
11	8-b MaxFreq	1056	1020	749	690	607	607	579	528
12	32-b MaxFreq	888	897	682	620	553	552	537	486
13	Est. Speedup	1	3	2	3.33	2.5	3	3	3.5
14	8-b Speedup	1	2.72	2.00	3.03	2.67	3.19	3.04	3.23
15	32-b Speedup	1	2.88	2.19	3.28	2.92	3.49	3.40	3.56

See Table 9 footnotes, except:
MaxFreq Yellow Highlight : sort finishes within one 450 MHz clock cycle

TABLE 11. 2-sorter N-network and N-sorter LUT Resource Usage.

R	N =>	2	3	4	5	6	7	8	9
1	# of 2-sorters	1	3	5	9	12	16	19	25
2	8-b Ntwrk LUTs	12	36	60	108	144	192	228	300
3	32-b Ntwrk LUTs	48	144	240	432	576	768	912	1200
4	8-b Nsrtr LUTs	12	36	60	136	177	252	376	738
5	32-b Nsrtr LUTs	48	144	228	496	645	840	1096	2034
6	8-b LUTs Ratio	1	1.00	1.00	1.26	1.23	1.31	1.65	2.46
7	32-b LUTs Ratio	1	1.00	0.95	1.15	1.12	1.09	1.20	1.70

2-sorter Network Data in Rows 1-3 ; N-sorter Data in Rows 4-5
Resource Increase Ratio : 8b = Row 4 / Row 2 ; 32b = Row 5 / Row 3

Rows 3 and 4 of the speed tables contain synthesis result propagation delay numbers for the 2-sorter networks, when implemented in the target FPGA, and rows 9 and 10 contain the same type of raw data for corresponding N-sorters. Max Frequency numbers, the inverse of the propagation delay values, are listed in rows 5, 6, 11, and 12.

The highlighting in the Max Frequency rows of Table 9 indicates designs that complete their operation in one 275 MHz clock cycle, and the Table 10 highlighting marks designs that finish their sort in one 450 MHz cycle. All of the N-sorter max frequency values in rows 11 and 12 of both speed tables are highlighted in yellow, while only the 2-sorter

TABLE 12. 7-Series xc7z045 2-max *N*-network vs *N*-max Filter Speed.

Rw	N: =>	2	3	4	5	6	7	8	9
1	Ntwrk Stages	1	2	2	3	3	3	3	4
2	Ntwrk Slices	2	4	4	6	6	6	6	8
3	8-b Delay	1.60	2.74	2.88	4.01	4.01	4.15	4.15	5.29
4	32-b Delay	1.80	3.14	3.28	4.62	4.62	4.76	4.76	6.10
5	8-b MaxFreq	625	365	348	249	249	241	241	189
6	32-b MaxFreq	555	318	305	216	216	210	210	164
7	N-max Slices	2	2	2	2	3	3	3	4
8	Equiv Stages	1	1	1	1	1.5	1.5	1.5	2
9	8-b Delay	1.60	1.64	1.77	1.84	2.45	2.53	2.58	3.04
10	32-b Delay	1.80	1.85	1.97	2.05	2.65	2.73	2.78	3.28
11	8-b MaxFreq	625	608	565	543	409	396	388	329
12	32-b MaxFreq	555	542	507	489	378	367	360	305
13	Est. Speedup	1	2	2	3	2	2	2	2
14	8-b Speedup	1	1.67	1.62	2.18	1.64	1.64	1.61	1.74
15	32-b Speedup	1	1.70	1.66	2.26	1.75	1.74	1.71	1.86

2-max Network Data in Rows 1-6 ; N-max Filter Data in Rows 7-12
Estimated N-sorter Speedup : Row 13 = Row 2 / Row 7
Actual N-sorter Speedup : 8-b = Row 3 / Row 9 ; 32-b = Row 4 / Row 10

TABLE 13. Ultrascale+ 2-max *N*-network vs *N*-max Filter Speed.

Rw	N: =>	2	3	4	5	6	7	8	9
1	Ntwrk Stages	1	2	2	3	3	3	3	4
2	Ntwrk Slices	2	4	4	6	6	6	6	8
3	8-b Delay	0.92	1.76	1.76	2.60	2.60	2.60	2.60	3.44
4	32-b Delay	1.10	2.12	2.12	3.14	3.14	3.14	3.14	4.15
5	8-b MaxFreq	1082	568	568	385	385	385	385	291
6	32-b MaxFreq	907	472	472	319	319	319	319	241
7	N-max Slices	2	2	2	2	3	3	3	4
8	Equiv Stages	1	1	1	1	1.5	1.5	1.5	2
9	8-b Delay	0.92	0.95	1.02	1.08	1.34	1.41	1.47	1.81
10	32-b Delay	1.10	1.08	1.17	1.23	1.48	1.54	1.60	1.95
11	8-b MaxFreq	1082	1055	982	929	745	711	681	552
12	32-b MaxFreq	907	923	853	815	674	651	623	514
13	Est. Speedup	1	2	2	3	2	2	2	2
14	8-b Speedup	1	1.86	1.73	2.42	1.94	1.85	1.77	1.90
15	32-b Speedup	1	1.96	1.81	2.56	2.11	2.04	1.96	2.13

See Table 12 footnotes

TABLE 14. 2-max *N*-network and *N*-max Filter LUT Resource Usage.

R	N =>	2	3	4	5	6	7	8	9
1	# of 2-sorters	1	3	5	9	12	16	19	25
2	8-b Ntwrk LUTs	8	16	24	32	40	48	56	64
3	32-b Ntwrk LUTs	32	64	96	128	160	192	224	256
4	8-b N-max LUTs	8	20	40	72	83	110	156	210
5	32-b N-max LUTs	32	80	160	288	311	410	588	738
6	8-b LUTs Ratio	1	1.25	1.67	2.25	2.08	2.29	2.79	3.28
7	32-b LUTs Ratio	1	1.25	1.67	2.25	1.94	2.14	2.63	2.88

2-max Network Data in Rows 1-3 ; N-max Filter Data in Rows 4-5
Resource Increase Ratio : 8b = Row 4 / Row 2 ; 32b = Row 5 / Row 3

values in rows 5 and 6 are highlighted. In short, all of the N-sorters meet the speed targets, but none of the 2-sorter networks do.

The estimated speedups in row 13 of the speed tables are calculated from the ratio of the network series slices in row 2 to the associated N-sorter series slices in row 7. The actual speedups are calculated as the ratios of network propagation delays to those of the N-sorters.

The tan, pink, and green highlighted rows have identical data in both speed tables, and the data and highlight colors from these rows are used in Table 1 in the introduction. Also, the data in rows 7 and 8 of the speed tables is used for the data in rows 5 and 6 of Table 4 in Section IV-B.

Table 11 contains resource usage data which is shared by both products, as they both implement common designs using 4-LUT slice groups. The data in the tan header row and row 1 of Table 11, like the data in the tan header row and row 1 of Tables 9 and 10, have been known for decades, and are still valid today [5], [6].

The speed data in Tables 9 and 10 can be used for simple throughput calculations. As listed in Table 9, an 8-b xc7z045 9-sorter can operate at 300 MHz, so it will process 300 million 9-input lists in 1 second.

Multiple 9-sorters can be instantiated to operate in parallel in the FPGA. For example, if 10 9-sorters are instantiated in the xc7z045, 3 billion 9-input lists will be sorted in 1 second.

Each 8-bit 9-sorter uses 738 LUTs, as is listed Table 11. Ten 9-sorters then use 7,380 LUTs, which is 3.4% of the 218,600 LUT resources available in the xc7z045. In general, the number of xc7z045 8-bit 9-input lists that can be sorted in one second equals 300e6 times the number of 9-sorters, and the LUT resource usage will be 738x the number of 9-sorters.

APPENDIX B
N-MAX VS 2-MAX NETWORK TABLES

Synthesis result data for single-stage N-max and 2-max N-network filters are listed in Tables 12, 13, and 14. These three tables present the same types of data as did Tables 9, 10, and 11 respectively in Appendix A. The three tables in this appendix provide the data for Figs. 14 and 15 in Section V-B.

As was shown Section V-C and Appendix A, the tables in this appendix can be used for throughput speed and resource usage calculations, this time for single-stage N-max filters. For example, as listed in Table 13, a 32-bit 9-max filter in the xcvu9p device operates within 1.95 ns, less than the 2 ns period of a 500 MHz system. Therefore, a single 32-bit 9-max filter will process 500 million 9-input lists in 1 second.

As mentioned in Section V-C, there are less than 9 million pixels in a 4K frame. If a 9-max filter is used for max pooling of distinct 9-pixel groups in a 4K frame, there will be less than 1 million max pooling result values for each frame. When operating at 500 MHz, a single 9-max filter will then operate on 4K data at over 500 fps.

A single 9-max filter uses 738 LUTs, which is 0.06% of the available xcvu9p LUTs. In general, the total number of 4K 9-max pooling operations per second equals 500 million times the number of instantiated 9-max filters, and the LUT resource usage will be 738x the number of filters.

APPENDIX C
N-SORTER AND N-FILTER SPEED

Raw synthesis result propagation delay values are listed for both N-sorters and N-filters in Tables 15 through 18:

- Table 15 – 8-bit xc7z045 7-series data
- Table 16 – 8-bit xcvu9p Ultrascale+ data.
- Table 17 – 32-bit xc7z045 7-series data
- Table 18 – 32-bit xcvu9p Ultrascale+ data.

These are the speed data values used throughout the results plots, tables, and discussion. The data for full N-sorters is in bold blue font in the tables.

TABLE 15. 7-Series xc7z045 8-bit *N*-sorter and *N*-filter propagation delays.

2 Slices in Series	ns	3 Slices in Series	ns	4 Slices in Series	ns
5-max	1.843	**5-sorter**	**2.591**		
4-max	1.771	8-max	2.577		
3-sorter	**1.677**	7-max	2.525		
3-max	1.644	6-max	2.445		
2-sorter	**1.623**	1.5x 2-sorter	2.435	2.0x 2-sorter	3.246
2-max	1.601	**4-sorter**	**2.132**	**9-sorter**	**3.140**
				9-median	3.082
				8-sorter	**3.052**
				9-max	3.044
				7-sorter	**2.865**
				6-sorter	**2.776**

TABLE 16. Ultrascale+ xcvu9p 8-bit *N*-sorter and *N*-filter propagation delays.

2 Slices in Series	ns	3 Slices in Series	ns	4 Slices in Series	ns
5-max	1.076				
4-max	1.018				
3-sorter	**0.980**	8-max	1.469		
3-max	0.948	**5-sorter**	**1.450**	**9-sorter**	**1.895**
2-sorter	**0.947**	1.5x 2-sorter	1.421	2.0x 2-sorter	1.894
2-max	0.924	7-max	1.406	9-median	1.841
		6-max	1.342	9-max	1.813
		4-sorter	**1.335**	**8-sorter**	**1.728**
				7-sorter	**1.648**
				6-sorter	**1.647**

TABLE 17. 7-Series xc7z045 32-bit *N*-sorter and *N*-filter propagation delays.

2 Slices in Series	ns	3 Slices in Series	ns	4 Slices in Series	ns
5-max	2.046				
4-max	1.973				
3-sorter	**1.881**	**5-sorter**	**2.825**		
3-max	1.846	8-max	2.779		
2-sorter	**1.826**	1.5x 2-sorter	2.739	2.0x 2-sorter	3.652
2-max	1.803	7-max	2.727	**9-sorter**	**3.373**
		6-max	2.647	9-median	3.315
		4-sorter	**2.334**	9-max	3.277
				8-sorter	**3.254**
				7-sorter	**3.098**
				6-sorter	**3.009**

Estimates based on the 2-sorter propagation delay are highlighted in yellow.

TABLE 18. Ultrascale+ xcvu9p 32-bit *N*-sorter and *N*-filter propagation delays.

2 Slices in Series	ns	3 Slices in Series	ns	4 Slices in Series	ns
5-max	1.227				
4-max	1.172				
2-sorter	**1.126**	1.5x 2-sorter	1.689	2.0x 2-sorter	2.252
3-sorter	**1.115**	**5-sorter**	**1.613**	**9-sorter**	**2.059**
2-max	1.103	8-max	1.604	9-median	2.006
3-max	1.083	7-max	1.537	9-max	1.947
		6-max	1.484	**8-sorter**	**1.862**
		4-sorter	**1.467**	**7-sorter**	**1.812**
				6-sorter	**1.808**

Data for full N-sorters is displayed in **bold blue font.**

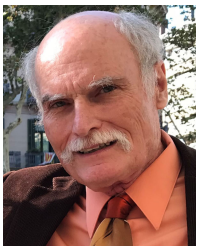
In each table, propagation delay values are listed in 3 columns, based on the hardware block's series slices:

- Leftmost column – 2 series slices.
- Middle column – 3 series slices.
- Rightmost column – 4 series slices.

Since the propagation delay of a hardware block is estimated to be proportional to its series slices, estimated propagation delay values, based on the 2-sorter's propagation delay, are highlighted in yellow and are all placed in the same row. In each column, the values are sorted by speed and are placed relative to the highlighted values, so that their relationship to the simple estimate can be easily seen.

REFERENCES

[1] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS*, New York, NY, USA, 1968, pp. 307–314, doi: 10.1145/1468075.1468121.
[2] I. Skliarova and V. Sklyarov, "Reconfigurable devices and design tools," in *FPGA-BASED Hardware Accelerators*. Cham, Switzerland: Springer, 2019, pp. 1–38.
[3] M. Zuluaga, P. Milder, and M. Püschel, "Streaming sorting networks," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 4, pp. 1–30, May 2016, doi: 10.1145/2854150.
[4] R. Chen and V. K. Prasanna, "Computer generation of high throughput and memory efficient sorting designs on FPGA," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3100–3113, Nov. 2017.
[5] D. E. Knuth, *The Art Computing Programming: Sorting Searching*, vol. 3. London, U.K.: Pearson, 1997.
[6] M. Codish, L. Cruz-Filipe, T. Ehlers, M. Müller, and P. Schneider-Kamp, "Sorting networks: To the end and back again," *J. Comput. Syst. Sci.*, vol. 104, pp. 184–201, Sep. 2019.
[7] D. Koch and J. Torresen, "Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting," in *Proc. 19th ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, New York, NY, USA, 2011, pp. 45–54, doi: 10.1145/1950413.1950427.
[8] A. C. Bovik, *Handbook of Image and Video Processing*. New York, NY, USA: Academic, 2010.
[9] K. Claessen, M. Sheeran, and S. Singh, "The design and verification of a sorter core," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Cham, Switzerland: Springer, 2001, pp. 355–368.
[10] Q. Gao and Z. Liu, "Sloping-and-shaking: Multiway merging and sorting," *Sci. China Ser. E, Technol. Sci.*, vol. 40, no. 3, pp. 225–234, Jun. 1997. [Online]. Available: http://engine.scichina.com/publisher/ScienceChinaPress/journal/ScienceinChinaSeriesE-TechnologicalSciences/40/3/10.1007/BF02916597
[11] F. Shi, Z. Yan, and M. Wagh, "An enhanced multiway sorting network based on n-sorters," in *Proc. IEEE Global Conf. Signal Inf. Process. (GlobalSIP)*, Dec. 2014, pp. 60–64.
[12] R. J. Nelson, "One level sorting network," U.S. Patent 4 628 483, Dec. 9, 1986.
[13] C. Chakrabarti and S. Dhanani, "Median filter architecture based on sorting networks," in *Proc. IEEE Int. Symp. Circuits Syst.*, Oct. 1992, pp. 1069–1072.
[14] C. Chakrabarti, "Sorting network based architectures for median filters," *IEEE Trans. Circuits Syst. II. Analog Digit. Signal Process.*, vol. 40, no. 11, pp. 723–727, Dec. 1993.
[15] R. Maheshwari, S. S. S. P. Rao, and P. G. Poonacha, "FPGA implementation of median filter," in *Proc. 10th Int. Conf. VLSI Design*, 1997, pp. 523–524.
[16] M. A. Mohamed, "Hardware sorter," U.S. Patent App. 11 554 747, May 1, 2008,
[17] E. H. Friend, "Sorting on electronic computer systems," *J. ACM*, vol. 3, no. 3, pp. 134–168, Jul. 1956.
[18] *Ieee Standard for Systemverilog–Unified Hardware Design, Specification, and Verification Language*, Standard 1800-2017 Revision IEEE Std 1800-2012, 2018.
[19] *UltraScale Architure Configurable Log. BlockUser Guide (UG574 Version 1.5)*, Xilinx, San Jose, CA, USA, Feb. 2017. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf
[20] *7 Ser. FPGAs Configurable Log. Block User Guide (UG474 Version 1.8)*, Xilinx, San Jose, CA, USA, Sep. 2016. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf
[21] A. Sanny and V. K. Prasanna, "Energy-efficient median filter on FPGA," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs (ReConFig)*, Dec. 2013, pp. 1–8.



ROBERT B. KENT (Life Member, IEEE) received the B.S. degree in physics from the University of Notre Dame, Notre Dame, IN, in 1970, and the M.S. degree in electrical engineering from The University of Utah, Salt Lake City, UT, in 1983. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, The University of New Mexico.

He has worked for various semiconductor companies: National Semiconductor from 1983 to 1990, Intel Corporation from 1990 to 1998, Philips Semiconductor from 1998 to 1999, and Xilinx, Inc., from 1999 to 2011. He then worked as an Independent Contractor, also in the semiconductor field, from 2012 to 2017. His main research interests include the design of single-stage N-sorters and N-filters in hardware, particularly in FPGAs, and the use of these sorters and filters in sorting networks or other hardware sorting systems.

MARIOS S. PATTICHIS (Senior Member, IEEE) received the B.Sc. degree (Hons.) in computer sciences, the B.A. degree (Hons.) in mathematics, the M.S. degree in electrical engineering, and the Ph.D. degree in computer engineering from The University of Texas at Austin, Austin, in 1991, 1993, and 1998.

He is currently a Professor with the Department of Electrical and Computer Engineering, The University of New Mexico (UNM), Albuquerque. At UNM, he also serves as the Director of the Image and Video Processing and Communications Laboratory (ivPCL). He holds the 2019–2022 ECE Gardner Zemke Professorship for teaching. He is also a Fellow of the Center for Collaborative Research and Community Engagement with the UNM College of Education. His current research interests include digital image, video processing, communications, dynamically reconfigurable computer architectures, and biomedical and space image-processing applications.

Dr. Pattichis was a recipient of the 2016 Lawton-Ellis and the 2004 Distinguished Teaching Awards from the Department of Electrical and Computer Engineering, UNM. For his development of the digital logic design labs at UNM, he was recognized by Xilinx Corporation, in 2003, and by the UNM School of Engineering's Harrison Faculty Excellent Award in 2006. He was the founding Co-PI (with Prof. Christodoulou) of the Configurable Space and Microsystems Innovations and Applications Center (COSMIAC). He was the General Chair of the 2008 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI), where he has served as the General Co-Chair in 2020. He has also served as a Senior Associate Editor for IEEE Signal Processing Letters, an Associate Editor for IEEE Transactions on Image Processing and IEEE Transactions on Industrial Informatics, and a Guest Associate Editor for the IEEE Transactions on Information Technology in Biomedicine. He is currently a Senior Associate Editor of the IEEE Transactions on Image Processing.