Optimal Parallel Hardware K-Sorter and Top$K$-Sorter, with FPGA implementations

Naoyuki Matsumoto, Koji Nakano and Yasuaki Ito
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—This paper presents a FIFO-based parallel merge sorter optimized for the latest FPGA. More specifically, we show a sorter that sorts K keys in latency $K+\log_2 K-1$ using $\log_2 K$ comparators. It uses $\frac{K}{M}+\log_2 K+\log_2 M-1$ memory blocks with capacity M to implement FIFOs. It receives K keys one by one in every clock cycle and outputs the sorted sequence of them from $K+\log_2 K-1$ clock cycles after. Since K clock cycles are necessary to input all K keys, our sorter is almost optimal in terms of the latency. Also, since the total FIFO capacity is only $K+M\log_2 K+M\log_2 M-M$ and at least K keys must be stored in the sorter, our sorter is also almost optimal in terms of the total FIFO capacity if M is small. This paper also presents top$K$-sorter, which outputs top K keys in N input keys for any large N. Our top$K$-sorter runs in latency $N+\log_2 K$ using $\log_2 K+1$ comparators. It uses memory blocks of size M and the total FIFO capacity is only $2K+M\log_2 K+M\log_2 M-2M$. Quite surprisingly, the total FIFO capacity is independent of N. Also, since the latency must be at least N, that of our top$K$-sorter is almost optimal in terms of the latency. Finally, we have implemented our K-sorter and top$K$-sorter in a Xilinx Virtex-7 FPGA using built-in Distributed RAMs and Block RAMs. The implementation results show that our K-sorter reduces the used memory resources by half, and both K-sorter and top$K$-sorter are practical and efficient.**

I. Introduction

An FPGA is a programmable logic device designed to be configured by the customer or designer by hardware description language after manufacturing. Since an FPGA chip maintains relative lower price and programmable features, it is widely used in those fields which need to update architecture or functions frequently such as image processing [1], [2] and education [3]. Latest FPGA architectures consist of an array of Configurable Logic Blocks (CLBs), Block RAMs, DSP Bocks, I/O pads, and interconnects [4], [5]. Since they work in parallel, FPGAs can be used to accelerate the computation.

It is no doubt that sorting is one of the most important tasks in computer engineering, such as database operations, image processing, statistical methodology and so on. Hence, many sequential sorting algorithms have been studied in the past [6]. To speedup the sorting, multiprocessors are employed for parallel sorting. Several parallel sorting algorithms such as parallel merge sort [7], bitonic sort [8], [9], randomized parallel sorting [10], column sort [11], sampling sort [12], and parallel radix sort [13], [14] have been devised. Lately, several parallel sorting algorithm using GPUs has been shown [15]–[17].

It is well known that a FIFO-based merge sorter can sort K keys using $\log_2 K$ comparators and FIFOs with total capacity $2K+\log_2 K-1$ [18]. It uses *standard K'-mergers* ($K'=2^0, 2^1, \ldots, \frac{K}{2}$), each of which merges two sorted sequences of size K' into one sorted sequence of size $2K'$. This sorter that we call *standard K-sorter* sorts K keys given from the input port one by one in every clock cycle, and the resulting sorted K keys are output from $K+\log_2 K-1$ clock cycles after as illustrated in Figure 1. Since the resulting sorted sequence can be output only after all keys are input, the latency must be at least K. Thus, the latency of a standard K-sorter is very close to be optimal. Also, it is well known that $\Omega(K\log K)$ comparisons are necessary to sort K keys [19] . Since K-sorter sorts K keys in $2k+\log K-1$ clock cycles using $\log_2 K$ comparators, it performs $O(K\log K)$ comparisons. Hence, K-sorter is optimal in terms of the total number of comparisons. Although this sorting architecture was presented a long time ago, it is still one of the best sorting architectures in terms of optimality of latency and comparisons.

This paper first presents *K-sorter/M* which is an improvement of a standard K-sorter. We mainly evaluate the performance of hardware sorters using total FIFO capacity and latency, which correspond to the hardware resource and the computing time, respectively. Our K-sorter/M uses memory blocks of size M to implement FIFOs. Since latest FPGAs have a lot of small built-in memory blocks, it makes sense to use memory blocks to implement parallel sorting architectures. Our K-sorter/M sorts K keys in latency $K+\log_2 K-1$ using $\log_2 K$ comparators and FIFOs with $\frac{K}{M}+\log_2 K+\log_2 M-1$ memory blocks. Hence, the total FIFO capacity is only $K+M\log_2 K+M\log_2 M-M$. Since K-sorter/M performs exact simulation of standard K-sorter, it is also almost optimal in terms of the latency and the number of total comparisons. In addition, since at least K keys must be stored in FIFOs, the total FIFO capacity is very close to be optimal if M is small, while standard K-sorter is not optimal.

This paper also presents top$K$-sorter/M, which outputs top K keys in N input keys for any large N. Figure 1 illustrates a timing chart of top8-sorter/M. It first outputs the sorted sequence of the first K keys. After that, it outputs the sorted sequence of the smallest K keys in the first $2K$ keys. In general, at each i-th iteration ($i\geq 1$), it outputs the sorted sequence of the smallest K keys in the first iK keys. Hence, it can find top K keys in $N=iK$ input keys for any large N. Our top$K$-sorter/M runs in latency $N+\log_2 K$ using $\log_2 K+1$ comparators and FIFOs with total capacity $\frac{5}{2}K+M\log_2 K+M\log_2 M$. In Figure 1, top8-sorter outputs top 8 keys out of 24 keys in latency $24+\log_2 8=27$. Since top K keys can be output only after all N keys are input, top$K$-sorter/M is almost optimal in terms of the latency.

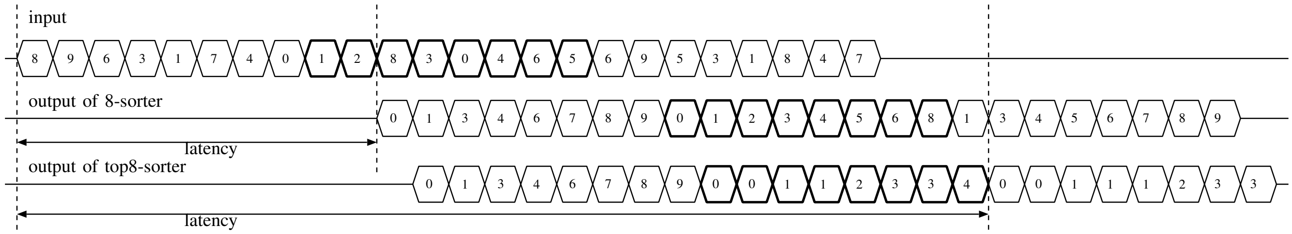

Fig. 1. A timing chart for 8-sorter and top-8 sorter

Top$K$-sorter/M may have many applications in the area of data mining. For example, Apriori algorithm [20], [21], which finds frequent item sets over transaction database, repeatedly finds top K frequently appeared data sets. Hence, it is very important to speed the computation for finding top K keys. We do not discuss implementation of Apriori algorithm, but we expect that our new idea for top$K$-sorter can be used for accelerate Apriori algorithm.

Finally, we have implemented our K-sorter and top$K$-sorter in a Xilinx Virtex-7 FPGA. The implementation results show that our K-sorter/M reduces the used memory resources by half, and both K-sorter/M and top$K$-sorter/M are practical and efficient.

Several sorting architectures based on K-sorter have been presented [22], [23]. In [24], they have presented a top$K$-sorter. Basically, their architecture is an N-input bitonic sorter from which circuit elements unnecessary to find top K keys are removed. Since bitonic sorting needs a lot of comparators, their architecture can find only top 4 key in 256 key.

Table I summarizes the theoretical analysis of performance of mergers, sorters and top$K$ sorters. The FIFO capacity of K-sorter [18] is $2K + \log_2 K - 1$, while that of our K-sorter/M is $K + M \log_2 K + M \log_2 M - M$. Thus, if M is so small that $M \ll K$, then our K-sorter/M reduces by half the total FIFO capacity.

This paper is organized as follows: We first review a standard K-merger in Section II. Section III shows our top$K$-merger and top$K$-sorter, which finds top K keys in any large number of input keys. Section IV presents our K-merger and top$K$-merger, which use FIFOs with fewer total capacity. It also shows K-sorter and top$K$-sorter using them. We show how memory blocks of FPGAs are implemented in V. Section VI shows implementation results of K-sorters and top$K$-mergers in the FPGAs. Section VII concludes our work.

II. MERGER AND SORTER USING FIFOS

This section reviews standard K-merger with two FIFOs of size $K + 1$ and K, respectively, that merges two sorted sequence with K keys each into one sorted sequence with $2K$ keys [18]. We also show that standard K-sorter that sorts K keys can be implemented using multiple mergers and evaluate the performance.

Standard K-merger has one input port and one output port and receives one key from the input port and outputs one key to the output port in every clock cycle. The input sequence is partitioned into subsequences of K keys each and each subsequence is sorted. More specifically, let $X = \langle x_0, x_1, \ldots, x_{N-1} \rangle$ denote N input keys. They are partitioned into subsequences $X_0, X_1, \ldots, X_{\frac{N}{K}-1}$ and each $X_i = \langle x_{i \cdot K}, x_{i \cdot K+1}, \ldots, x_{i \cdot K+K-1} \rangle$ $(0 \leq i \leq \frac{N}{K} - 1)$ is sorted. Standard K-merger merges each pair of adjacent subsequences X_{2i} and X_{2i+1} $(0 \leq i \leq \frac{N}{2K} - 1)$ into one sorted sequence with $2K$ keys.

Standard K-merger has two FIFOs A and B that can store $K+1$ keys and K keys, respectively as illustrated in Figure 2. Initially, both FIFOs are empty. First, all K keys in X_0 are enqueued in FIFO A one by one. After that, all K keys in X_1 are enqueued in FIFO B. Similarly, X_2 is enqueued in FIFO A and then X_3 is enqueued in FIFO B. This enqueue procedure is repeated until all keys in X are enqueued in FIFOs. More specifically, for every i (≥ 0), all keys in X_{2i} are enqueued in FIFO A and then those in X_{2i+1} are enqueued in FIFO B. At the same time, dequeue operation is performed. After the first key x_K in X_1 is enqueued, we start dequeuing one of FIFOs A and B. Two keys in the heads of FIFOs A and B are compared and the smaller one is dequeued and sent to the output port. If FIFO B is empty, then FIFO A is dequeued. Also, if keys stored in the heads of two FIFOs are originated from different pairs, that from earlier pair is dequeued. More specifically, when two keys in X_{2i+1} and X_{2i+2} are compared, that in X_{2i+1} is dequeued even if that of X_{2i+2} is smaller. Once two FIFOs has totally $K + 1$ keys, dequeue operation is performed for one of the FIFOs and enqueue operation is performed for one of the FIFOs. Hence, two FIFOs always have totally $K + 1$ keys until all input keys are enqueued. The readers should refer to Figure 2 that illustrates standard 4-merger, the timing chart and the data movement. We can see that the first two subsequences of 4 keys each are merged into one sorted subsequence of 8 keys. It should be clear that FIFOs A and B may store $K + 1$ and K keys. If all keys in X_0 are larger than those of X_1, then FIFO A will store $K + 1$ keys. Also, if all keys in X_0 are smaller than those of X_1, then FIFO B will store K keys.

Figure 2 illustrates the architecture of a 4-merger. It has FIFOs A and B can store 5 and 4 key each. It also shows the timing chart and the corresponding data movement through two FIFOs. All four keys in the first subsequence are stored in the FIFO A and the first key of the second subsequence is enqueued in the FIFO B. After that, dequeuing procedure, which removes a smaller number of two numbers stored in the heads of the FIFOs, is started. We can see that the first

TABLE I. THEORETICAL ANALYSIS OF MERGERS AND SORTERS

architectures	comparators	# of FIFOs	total FIFO capacity	latency
K-merger [18]	1	2	$2K+1$	$K+1$
K-merger/M	1	$\max(\frac{K}{M}+1,2)$	$\max(K+M,2M)$	$K+1$
K-sorter [18]	$\log_2 K$	$2\log_2 K$	$2K+\log_2 K-2$	$K+\log_2 K-1$
K-sorter/M	$\log_2 K$	$\frac{K}{M}+\log_2 K+\log_2 M-1$	$K+M\log_2 K+M\log_2 M-M$	$K+\log_2 K-1$
top$K$-merger	1	3	$\frac{5}{2}K$	1
top$K$-merger/M	1	$\max(\frac{K}{M}+2,3)$	$\max(\frac{3}{2}K+M,3M)$	1
top$K$-sorter	$\log_2 K+1$	$2\log_2 K+3$	$\frac{9}{2}K+\log_2 K-2$	$N+\log_2 K$
top$K$-sorter/M	$\log_2 K+1$	$2\frac{K}{M}+\log_2 K+\log_2 M+1$	$\frac{5}{2}K+M\log_2 K+M\log_2 M$	$N+\log_2 K$

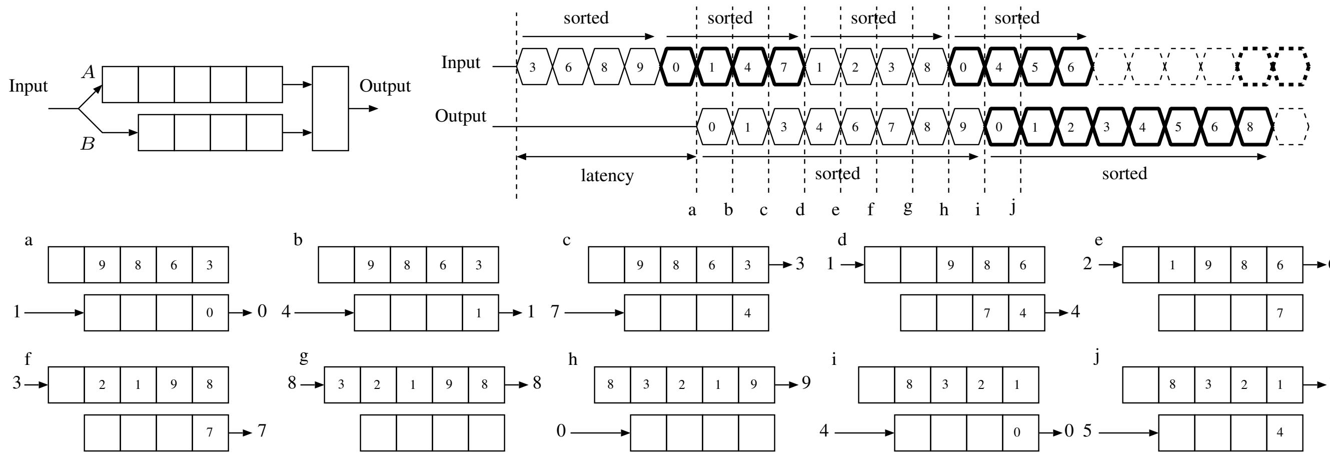

Fig. 2. The architecture of standard 4-merger, the timing chart, and the data movement

two subsequences of 4 key each are merged into one sorted subsequence of 8 key.

Let us confirm that FIFOs A and B may store $K+1$ and K keys, respectively. If all keys in X_0 are larger than those of X_1, then all keys in X_1 are dequeued before those in X_0 are dequeued. Hence, when the first key x_{2K} in X_2 is enqueued, FIFO A still stores all K keys in X_0. Thus, FIFO A will store K keys in X_0 and x_{2K} and $K+1$ keys are necessary and sufficient for the capacity of FIFO A. On the other hand, if all keys in X_0 are smaller than those of X_1, then all keys in X_0 are dequeued before those in X_1 are dequeued. Hence, FIFO B stores all K keys in X_0 when we start enqueueing keys in X_2 in FIFO A. Thus, K keys are necessary and sufficient as the capacity of FIFO B.

Let us evaluate *the latency* of standard K-merger. The latency is the time necessary to obtain the first output after the first input is given. In Figure 2, after the first input 3 is given, the first output 0 is obtained in 5 clock cycles. Hence, the latency of standard 4-merger is 5. In standard K-merger, the first output is obtained after the first key x_K of X_1 is enqueued in FIFO B. Thus, the latency of standard K-merger is $K+1$ and we have,

Lemma 1: Standard K-merger merges two sorted sequence with K keys into one sorted sequence in latency $K+1$ using one comparator and two FIFOs with total capacity $2K+1$.

Using standard 2^0-merger, 2^1-merger, ..., 2^{k-1}-merger, we can construct standard 2^k-sorter that sorts 2^k keys. Figure 3 illustrates standard 8-sorter. We can see that the first 8 keys are sorted correctly. After that, the next 8 keys are also sorted correctly. Let us evaluate the latency of 2^k-sorter. Since the latency of standard K-merger is $K+1$ and it uses 1-merger, 2-merger 4-merger, ..., 2^{k-1}-merger, the latency of 2^k-sorter is $(2^0+1)+(2^1+1)+\cdots+(2^{k-1}+1)=2^k+k-1$. Also, the total capacity of FIFOs used in 2^k-sorter is $(2\cdot 2^0+1)+(2\cdot 2^1+1)+\cdots+(2\cdot 2^{k-1}+1)=2^{k+1}+k-2$. Thus, we have,

Lemma 2: Standard K-sorter can sort K keys in latency $K+\log_2 K-1$ using $\log_2 K$ comparators and $2\log_2 K$ FIFOs with total capacity $2K+\log_2 K-2$.

III. TOPK-MERGER AND TOPK-SORTER

In this section, we first design top$K$-merger, that maintains top K keys given so far. We will design top$K$-sorter using top$K$-merger and standard K-sorter. We assume that a FIFO supports *unenqueue* and *remove-all operations* that removes the tail key and all keys in the FIFO, respectively.

The architecture of top$K$-merger is very similar to that of K-merger. From the input port, sorted sequences of K keys each are given one by one. It always outputs top K keys of the input keys given so far. Top$K$-merger has three FIFOs: FIFOs A and B of size K each and FIFO C of size $\frac{K}{2}$. FIFOs A and B are used to store top K keys and FIFO C is used to buffer input keys. Figure 4 illustrates the architecture of top4-merger, the timing chart, and the data movement. As before, let $x_0, x_1, \ldots$ be input keys and $X_0, X_1, \ldots$ be subsequences of K keys each. First, K keys in X_0 are enqueued in FIFO C and they are dequeued immediately. At the same time, they are enqueued in FIFO A. After that, keys in X_1 are enqueued in FIFO C. After the first key x_K in X_1 is enqueued, the heads

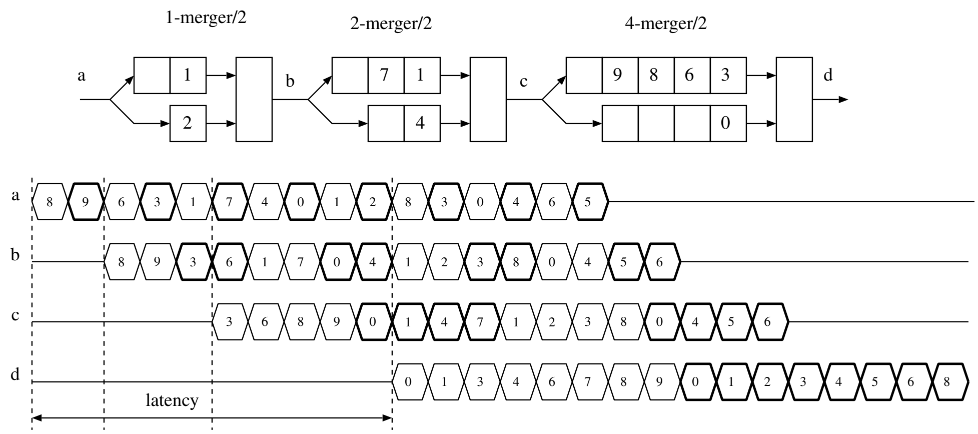

Fig. 3. 8-sorter and the timing chart

of FIFO A and FIFO C, that is, x_0 and x_K are compared, and the smaller one is dequeued and enqueued in FIFO B. This operation is repeated until top K keys of X_0 and X_1 are output and enqueued in FIFO B. Note that top K keys of X_0 and X_1 are output and the remaining K keys should be discarded. Hence, when a key in X_1 in FIFO C is dequeued, FIFO A must be unenqueued, because the tail key in FIFO A cannot be a top K key. In other words, either dequeue operation or unenqueue operation is performed for FIFO A, and the number of keys stored in FIFO A is decreased by one. Also, the remove-all operation is performed FIFO C to discard keys. It is possible that no dequeue operation is performed for FIFO C. Since FIFO C can store up to $\frac{K}{2}$ keys, some keys may not be enqueued. If this is the case, we do not perform such enqueue operation, because such keys cannot be top K keys. When the first key x_{2K} in X_2 is enqueued in FIFO C, FIFO A is empty and FIFO B stores the top K keys of X_0 and X_1. The same operations are repeated for FIFO B and FIFO C, top K keys of X_0, X_1, and X_2 are output, and FIFO A stores them. In this way, top$K$-merger outputs top K keys so far. Since top$K$-merger has three FIFOs of sizes K, K, and $\frac{K}{2}$, respectively, we have,

Lemma 3: Top$K$-merger repeats outputting top K keys of subsequences of sorted K keys each received so far, with latency 1 using 3 FIFOs with total capacity $\frac{5}{2}K$.

Using top$K$-merger and standard K-sorter, we can design top$K$-sorter. By standard K-sorter, each subsequence of length K in an input sequence can be sorted. Top$K$-merger receives them and outputs top K keys so far in latency 1. Hence, after i subsequences of $N = ik$ keys are received, it starts outputting top K keys. Figure 5 illustrates the architecture of top8-sorter, which uses top8-merger and a standard 8-sorter. Also, standard K-sorter has $2\log_2 K$ FIFOs with total capacity $2K+\log_2 K-2$, and top$K$-merger uses three FIFOs with total capacity $\frac{5}{2}K$. Thus, we have,

Lemma 4: Top$K$-sorter outputs top K keys out of N keys in latency $N + \log_2 K$ using $2\log_2 K + 3$ FIFOs with total capacity $\frac{9}{2}K + \log_2 K - 2$.

IV. MEMORY EFFICIENT IMPLEMENTATIONS OF K-SORTER AND TOPK-SORTER

The main purpose of this section is to improve standard K-merger and top$K$-merger by reducing the FIFO capacity. We then show memory efficient implementations of K-sorter and top$K$-sorter.

Let us design *K-merger/M*, which uses FIFOs of size M. We first assume that $M \leq K$. Recall that K-merger can be implemented using two FIFOs A and B of sizes $K+1$ and K. Also, the total number of keys stored in FIFOs A and B is at most $K+1$. Using this fact, we can simulate FIFOs A and B using $S = \frac{K}{M} + 1$ FIFOs $F_0, F_1, \ldots F_{S-1}$, each of which can store M keys. Hence, the total FIFO capacity is $MS = K + M$. Since K keys must be stored in a memory to merge two sorted sequence of length K, the total FIFO capacity cannot be smaller than K. Thus, the total FIFO capacity of $K + M$ is almost optimal. In K-merger/M, the input and output ports of S FIFOs are connected as follows:
(1) the input port of K-merger/M is connected to all S FIFOs,
(2) the output ports of FIFOs F_0 and F_{S-1} are connected to the comparator,
(3) the output port of each FIFO F_i ($1 \leq i \leq S-2$) is connected to the input ports of FIFOs F_{i-1} and F_{i+1}, and
(4) the output port of FIFO F_{S-1} is connected to the input port of FIFO F_{S-2}.
Keys in FIFO A of standard K-merger are stored from FIFO F_0 and those in FIFO B are stored from FIFO F_{S-1}. In other words, if i FIFOs F_0, F_1, …, F_{i-1} are used for keys to be stored in FIFO A, then the remaining $S-i$ FIFOs F_i, F_{i+1}, …, F_{S-1} can be used for simulating FIFO B. If this is the case, the i FIFOs stores at least $(i-1)M+1$ keys to be stored in FIFO A. Hence, the number of keys to be stored in FIFO B is at most $(K+1) - ((i-1)M+1) = M(S-i)$ and these keys can be stored in the remaining $S-i$ FIFOs. Thus, S FIFOs of size M can simulate FIFOs A and B.

Figure 6 (1) illustrates the architecture of 16-merger/4, which has five FIFOs with capacity 4. We can see that 11 keys from 1 to 19 to be stored in FIFO A are stored in FIFOs F_0, F_1, and F_2 and 6 keys from 2 to 12 in FIFO B are stored in FIFOs F_3 and F_4.

Clearly, enqueue, dequeue, unenqueue, remove-all oper-

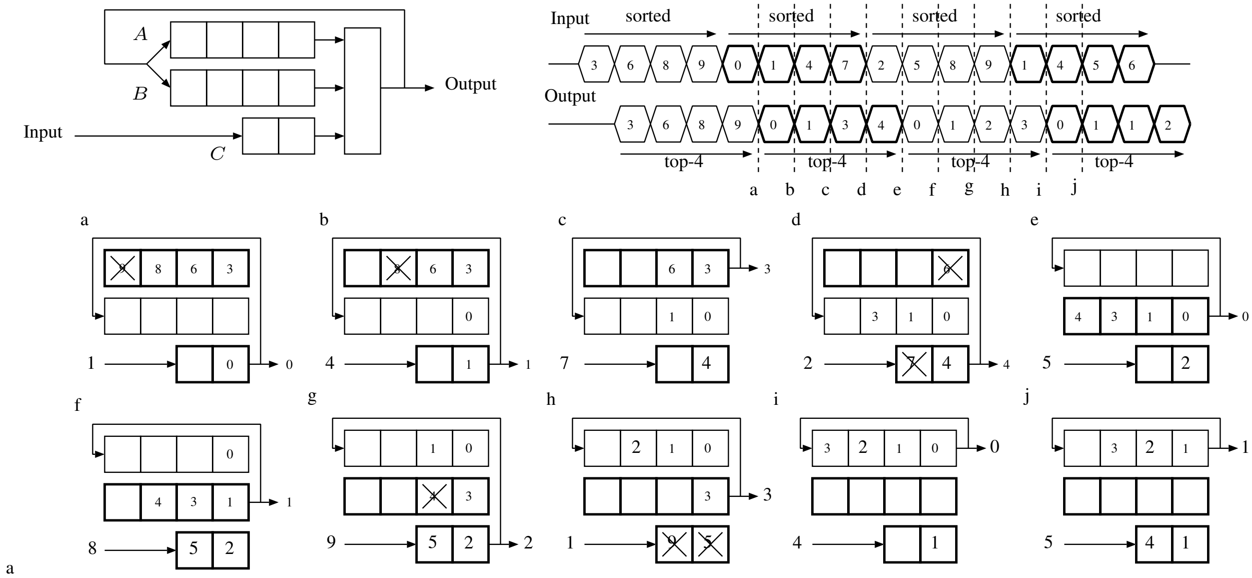

Fig. 4. The architecture of top4-merger, the timing chart, and the data movement

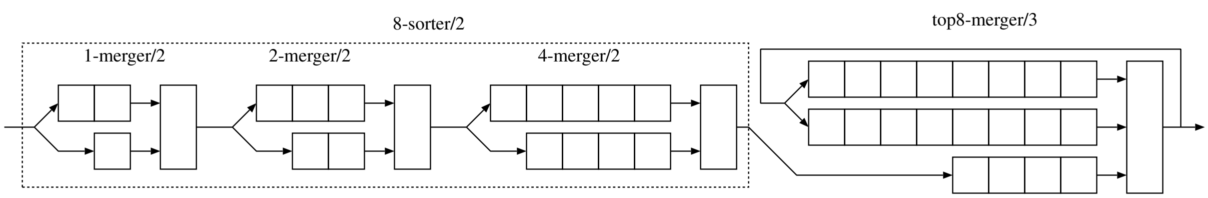

Fig. 5. The architecture of top8-sorter using 8-sorter and top8-merger

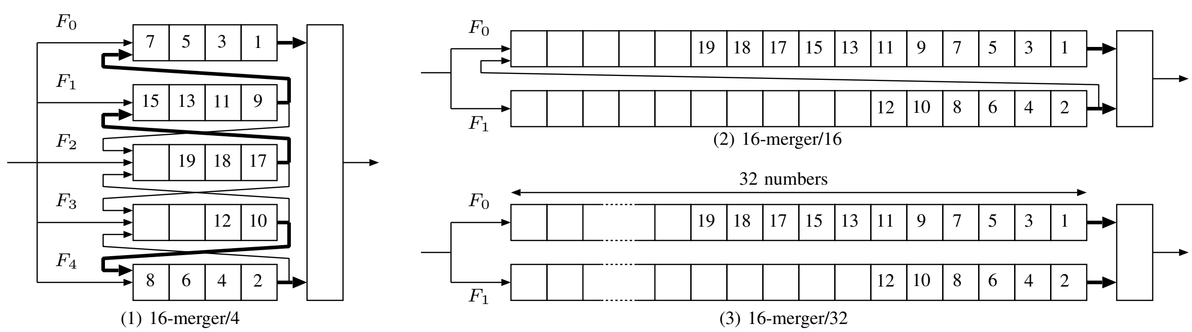

Fig. 6. The architectures of 16-merger/4 and 16-merger/16

ations for FIFOs can be simulated in an obvious way. For example, in Figure 6, enqueue operation for FIFO A can be done by that for FIFO F_2. Dequeue operation for FIFO A can be done by that for FIFOs F_0, F_1, and F_2, and enqueue operation for FIFOs F_0 and F_1.

Also, note that the output port of FIFO F_{S-1} must be connected to the input port of FIFO F_{S-2}, while the output port of FIFO F_0 is not connected to the input port of FIFO F_1. Recall that FIFOs A and B stores at most $K+1$ and K keys, respectively. If $K+1$ keys are stored in FIFO A, the tail key is stored in FIFO F_{S-1} and it will be moved to FIFO F_{M-2} to simulate dequeue operation for FIFO A. On the other hand, FIFO B stores at most K keys, and thus FIFO F_0 never store a key to be stored in FIFO B.

When $K = M$, K-merge/M uses two FIFOs F_0 and F_1 of size M each as illustrated in Figure 6 (2). Since FIFO A may store $K+1 = M+1$ keys, FIFO F_1 is used to store the tail of $K+1$ keys stored in FIFO A. Hence the output port of F_1 must be connected to the input port of F_0 to move the tail stored in FIFO F_1 to FIFO F_0. If $K < M$, then K-merger/M also uses two FIFOs F_0 and F_1 of size M each as illustrated in Figure 6 (3). If this is the case, the connection between the output port of F_1 and the input port of F_0 is not necessary. The reader may think that K-merger/M such that $K < M$

makes no sense because two FIFOs have much larger capacity than the maximum number of keys stored in them. However, Virtex-7 FPGAs have fixed size memory blocks, and we may need to use them to implement K-merger/M for small K.

Consequently, we have,

Lemma 5: K-merger/M merges two sorted sequence with K keys into one sorted sequence in latency $K+1$ using $\max(\frac{K}{M}+1, 2)$ FIFOs with total capacity $\max(K+M, 2M)$.

We can design top$K$-merger/M by the same technique to reduce the total FIFO capacity. Recall that top K merger uses two FIFOs A and B that can store K keys each. Also, they store at most K keys totally. To simulate these two FIFOs, we use $S = \frac{K}{M}+1$ FIFOs $F_0, F_1, \ldots, F_{S-1}$ that can store M keys each. These S FIFOs can simulate FIFOs A and B of top$K$-merger in the same way as the simulation of standard K-merger by K-merger/M. Note that it is not necessary to connect the output port of FIFO F_{S-1} and the input port of FIFO F_{S-2}, because the total capacity of $S-1$ FIFOs F_0, $F_1, \ldots, F_{S-2}$ is K and FIFO F_{S-1} never stores a key to be stored in FIFO A. Figure 7 illustrates top16-merger/4, which has five FIFOs with capacity 4 and FIFO C with capacity 8.

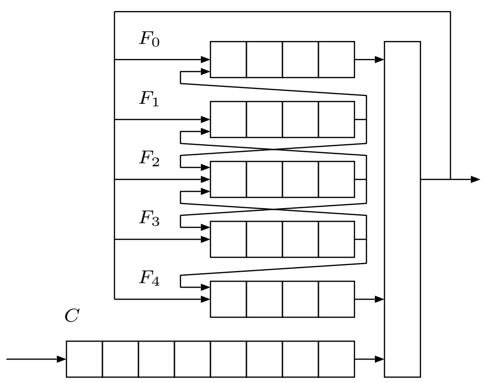

Fig. 7. The architecture of top-16 merger using five FIFOs with 4 keys each and one FIFO with 8 keys

Since we use $S+1 = \frac{K}{M}+2$ FIFOs with total capacity $SM + \frac{K}{2} = \frac{3}{2}K + M$, we have

Lemma 6: Top$K$-merger/M repeats outputting top K keys of subsequences of sorted K keys each received so far, with latency 1 using $\max(\frac{K}{M}+2, 3)$ FIFOs with total capacity $\max(\frac{3}{2}K+M, 3M)$.

We can use 2^0-merger/M, 2^1-merger/M, $\ldots$, 2^{k-1}-merger/M to implement K-sorter. We call the resulting K-sorter, *K-sorter/M*, which has

$$\begin{aligned}\sum_{i=0}^{k-1}\max(\frac{2^i}{M}+1,2) &= \sum_{i=0}^{m-1}2+\sum_{i=m}^{k-1}(\frac{2^i}{M}+1) \\ &= 2m+2^{k-m}-1+k-m \\ &= \frac{K}{M}+\log_2 K+\log_2 M-1\end{aligned}$$

FIFOs, where $2^m = M$. Since the capacity of each FIFO is M, the total FIFO capacity is $K + M\log_2 K + M\log_2 M - M$ and we have,

Theorem 7: K-sorter/M ($K \geq M$) can sort K keys with latency $K + \log_2 K - 1$ using $\log_2 K$ comparators and $\frac{K}{M} + \log_2 K + \log_2 M - 1$ FIFOs with total capacity $K + M\log_2 K + M\log_2 M - M$.

Let *top$K$-sorter/M* denote a sorter obtained using K-sorter/M and top$K$-merger/M. From Lemma 6 and Theorem 7, we have,

Theorem 8: Top$K$-sorter/M ($K \geq M$) outputs top K keys in N keys with latency $N + \log_2 K$ using $2\frac{K}{M} + \log_2 K + \log_2 M + 1$ FIFOs with total capacity $\frac{5}{2}K + M\log_2 K + M\log_2 M$.

V. BUILT-IN MEMORIES IN FPGAS

Virtex-7 FPGAs have built-in memories that can be used to implement FIFOs. This section introduces two types of memories, Block RAMs and Distributed RAMs.

It is well known that a FIFO can be implemented as a ring buffer data structure using a RAM. Elements in a FIFO are stored in a RAM with dual ports for reading and writing. Two pointers, read pointer and write pointer are used to specify the head and the tail of keys. Hence, FIFOs can be implemented using *a simple dual-port RAM*, which has independent writing address input and reading address input. We will show that how RAM can be configured in FPGAs. We assume that keys to be stored in FIFOs have 32 bits.

Virtex-7 FPGAs has a lot of *Block RAMs*, which can be used as ring buffers for FIFOs. For example, XC7VX485T has 1,030 Block RAMs, each of which can be configured as one 36kb Block RAM or two 18kb Block RAMs [5]. A 36kb Block RAM and a 18kb Block RAM can be configured as a 1k×36 and a 512×36 simple dual-port memory as illustrated in Figure 8. They have three input ports for writing data, writing address, and reading address. Writing ports are used to append a key in the tail and reading address port is used to read a key in the head. Thus, FIFOs with 1k and 512 keys with 32 bits can be implemented using 36kb and 18kb Block RAMs, respectively. Also, larger FIFOs can be implemented using multiple Block RAMs in an obvious way.

Virtex-7 FPGAs also have a lot of Configurable Logic Blocks (CLBs), each of which has two slices [4]. For example, XC7VX485T has 37,950 CLBs, that is, 75,900 slices. Each slice is either a SLICEM or a SLICEL, and XC7VX485T has 32,700 SLICEMs and 43,200 SLICELs. Each slice has four 6-input Look-Up Tables (6LUTs), each of which is a $2^6 = 64$-bit memory. Those in a SLICEL is read-only, in the sense that the values stored in 6LUT cannot be updated after the programming of the FPGA. On the other hand, the values stored in a 6LUT in a SLICEM can be changed, and thus, it can be used as a 64×1 RAM. Also, each 6LUT in a SLICEM can be configured as four 5-input Look-Up Tables (5LUTs) such that each 5LUT has 2-bit data input and 2-bit data output. Hence, each 5LUT can be used as a 32×2 RAM. However, address ports of one of the four LUTs in a SLICEM are shared and so it is not possible to use them independently. In particular, since one of the four LUTs has one address input used for specifying both reading and writing addresses, Hence, it cannot be used to implement a simple dual-port RAM. The remaining three LUTs can be used to implement a simple dual-port RAM. As illustrated in Figure 9, four LUTs in a SLICEM

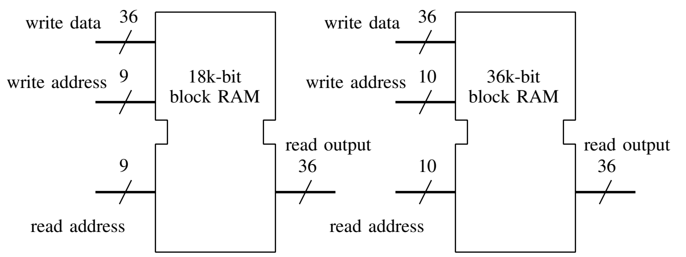

Fig. 8. A 18kb Block RAM and a 36kb Block RAM configured as a 512×36 memory and a 1k×36 memory, respectively

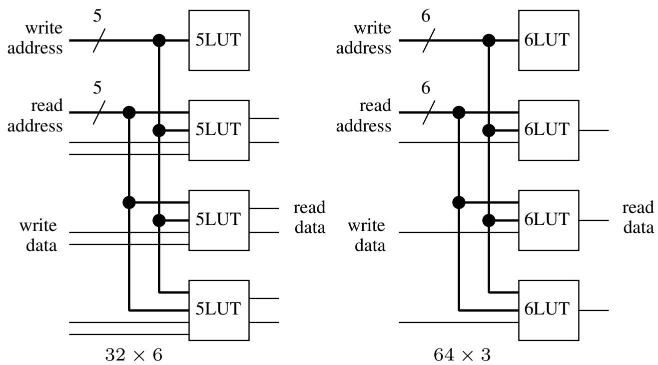

Fig. 9. Four LUTs configured as a 32×6 memory and a 64 × 3 memory

can be configured as either a 32×6 RAM or a 64×3 RAM. Therefore, we can construct a 32×36 RAM using 6 SLICEMs and a 64×33 RAM using 11 SLICEMs. Thus, FIFOs with 32 keys and with 64 keys can be constructed using 6 SLICEMs (i.e. 24 LUTs) and 11 SLICEMs (i.e. 44 LUTs), respectively. RAMs constructed by LUTs are called *Distributed RAMs*.

VI. Implementation results

This section shows implementation results for Virtex-7 FPGA XC7VX485T on the VC707 Evaluation Board [25]. We assume that input keys to be sorted have 32 bits. Input keys to be sorted can be either signed/unsigned 32-bit integers or 32-bit single precision floating-point numbers (IEEE 754 Standard for Floating-Point Arithmetic [26]) , because the comparators for them are the same.

XC7VX485T has 75,900 slices, out of which 43,200 and 32,700 are SLICELs and SLICEMs, respectively. Since 4 LUTs in each SLICEM can be configured as a Distributed RAM, totally $32,700 \times 4 = 130,800$ LUTs can be used for Distributed RAMs. Also, since both SLICELs and SLICEMs can be used for embedded logics, $75,900 \times 4 = 303,600$ LUTs can be used for implementing logics. Further, each slice has 8 flip-flops, and so, we can embed registers with totally $75,900 \times 8 = 607,200$ bits in XC7VX485T. It also has 1,030 Block RAMs, each of which can be configured as either one 36kb Block RAMs or two 18kb Block RAMs. Basically, the tail and head pointers are implemented in embedded registers. State machines for controlling mergers and sorters can be implemented using embedded registers and LUTs in slices. More specifically, registers are used to store the current state and slices are used to compute the next state. Other miscellaneous logics are implemented in either SLICEMs or SLICELs. Distributed RAMs are implemented in SLICEMs. As we have explained in Section V, FIFOs of sizes 32×32 and 64×32 can be implemented using 24 LUTs and 44 LUTs in SLICEMs, respectively.

A. K-merger using Distributed RAMs

We have implemented standard K-merger using Distributed RAMs for various configurations. Table II shows the performance of standard K-merger shown in [18]. It uses two FIFOs of sizes $K+1$ and K implemented using Distributed RAMs on the FPGA. The table shows the numbers of LUTs used for logic (LUT(Logic)) and Distributed RAMs (LUT(Memory)), and the total number of register bits. From $K = 1$ to 32, 32×32 Distributed RAMs implemented in 24 LUTs. When $K = 1$, one FIFO to store 2 keys is implemented using a 32×32 Distributed RAMs, and the other FIFO to store 1 key is implemented in one 32-bit register. From $K = 2$ to 32, two 32×32 Distributed RAMs are used to implement two FIFOs. When $K = 32$, one of the FIFOs need to store 33 keys, one 32-bit register is attached to a 32×32 Distributed RAM. For $K = 64$ and larger, $2 \cdot \frac{K}{64}$ Distributed RAMs of size 64×32 are used to implement two FIFOs of sizes $K+1$ and K each. Also, one 32-bit register is attached to expand the capacity of FIFO A by one. Hence, $2 \cdot \frac{K}{64} \cdot 44$ LUTs are used for Distributed RAMs for $K = 64$ and larger. For large K, the number of LUTs used for FIFOs is dominant. So, we can expect that our idea for reducing the FIFO size works effectively for large merger.

Table III shows the performance of our 64k-merger/M using Distributed RAMs from $M = 32$ to 64k. Recall that K-merger/M uses $S = \frac{K}{M} + 1$ FIFOs of size M each. Thus, 64k-merger/M uses $S = \frac{64\text{k}}{M} + 1$ FIFOs. When $M = 32$, $\frac{64\text{k}}{32} + 1 = 2049$ FIFOs with capacity 32 are used. Since a 32×32 Distributed RAM can be implemented in 24 LUTs, $2049 \times 24 = 49716$ LUTs are used. For $M = 64$ and larger, each FIFO of size M are implemented using 64×32 Distributed RAMs each of which can be embedded in FPGAs using 44 LUTs. Hence, a FIFO with capacity $M \times 32$ can be implemented using $44 \cdot \frac{M}{64}$ LUTs. For example, when $M = 2\text{k}$, a FIFO with capacity $2\text{k} \times 32$ is implemented using $44 \cdot \frac{2\text{k}}{64} = 1408$ LUTs. Since $S = \frac{64\text{k}}{M} + 1 = 33$ FIFOs are used, the total number of LUTs is $1408 \times 33 = 46464$. In general, for $M = 64$ and larger, $44 \cdot \frac{M}{64} \cdot (\frac{64\text{k}}{M} + 1) = \frac{11}{16} \cdot (64\text{k} + M)$ LUTs are used for $\frac{64\text{k}}{M} + 1$ FIFOs. Thus, more LUTs for Distributed RAMs are used when M is smaller. On the other hand, since each FIFO needs LUTs to embed some logic to control it, LUTs used for logic is almost proportional to the number of FIFOs. Therefore, we should find and use the best parameter M that minimizes the total number of LUTs. We can see that, in Table III, the number of LUTs is minimized when $M = 2k$ and $S = 33$.

Table IV shows the performance of our architecture K-merger/M using Distributed RAMs. We have implemented K-merger/M for all possible values of M and selected a parameter M for each K that minimizes the total number of LUTs as we have shown in Table III for $K = 64$k. Recall that our K-merger/M uses $S = \frac{K}{M} + 1$ FIFOs of size M each. From $K = 1$ to 256, the total number of LUTs is minimized when $S = 2$, because FIFOs are small and the

TABLE II. THE PERFORMANCE OF STANDARD K-MERGER USING DISTRIBUTED RAMS

K	1	2	4	8	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k
Registers	41	48	55	62	69	76	83	90	97	104	111	118	125	132	139	146	153
LUTs	140	262	287	288	245	280	336	532	731	1264	2043	3904	7558	14632	28311	57463	114652
LUTs (Logic)	116	214	239	240	197	232	248	356	379	560	635	1088	1926	3368	5783	12407	24540
LUTs (Memory)	24	48	48	48	48	48	88	176	352	704	1408	2816	5632	11264	22528	45056	90112
Clock(MHz)	287	271	270	266	267	268	269	239	220	219	191	170	167	160	163	145	137

TABLE III. THE PERFORMANCE OF 64K-MERGER/M USING DISTRIBUTED RAMS

M (FIFO size)	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k
S (#FIFOs)	2049	1025	513	257	129	65	33	17	9	5	3	2
Registers	24685	14449	8299	4712	2666	1516	878	528	338	236	182	137
LUTs	191746	139817	117036	81114	73728	63678	62684	63527	65668	70705	86310	115986
LUTs(Logic)	142570	94717	71892	35882	28320	17918	16220	15655	14980	14385	18726	25874
LUTs(Memory)	49176	45100	45144	45232	45408	45760	46464	47872	50688	56320	67584	90112
Clock(MHz)	177	196	189	174	167	156	147	138	144	146	135	129

TABLE IV. THE PERFORMANCE OF K-MERGER/M USING DISTRIBUTED RAMS: BEST CONFIGURATION IS SELECTED

K	1	2	4	8	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k
M(FIFO size)	1	2	4	8	16	32	64	128	256	256	256	512	1k	1k	1k	1k	2k
S(#FIFOs)	2	2	2	2	2	2	2	2	2	3	5	5	5	9	17	33	33
Registers	70	17	25	33	41	49	57	65	73	105	146	161	176	269	450	807	878
LUTs	144	211	233	239	254	281	368	570	775	1186	1770	3003	5143	9090	16821	32502	62684
LUTs(Logic)	144	163	185	191	206	233	280	394	423	658	890	1243	1623	2754	4853	9270	16220
LUTs(Memory)	0	48	48	48	48	48	88	176	352	528	880	1760	3520	6336	11968	23232	46464
Clock(MHz)	418	270	269	255	240	242	238	226	227	201	190	180	163	164	164	164	147

logic and registers for controlling FIFOs need more LUTs than Distributed RAMs. So, if we used more than two FIFOs, LUTs for logic would increase and thus the total number of LUTs would also increase. From $K = 512$, the total number of LUTs is minimized when S is more than two. For example, 64k-merger/M has the minimum number of LUTs when $M = 2$k and $S = 33$. We can see that 64k-merger/2k uses 62684 LUTs while standard 64k-merger uses 114652 LUTs. Hence, our 64k-merger/2k uses almost half (54%) LUTs of standard 64-merger.

B. K-merger using Block RAMs

We have also implemented various K-mergers using Block RAMs. Table V shows the performance of standard K-merger [18] using two FIFOs of size $K + 1$ and K. We have implemented a FIFO of size $K + 1$ using a Block RAM of size K and one register, when $K = 512$ and 1k. For K less than or equal to 1k, standard K-merger uses two 18kb Block RAMs, each of which can be configured as a 512×32 memory. From $K = 1$k, K-merger uses $S = 2 \cdot \frac{K}{1k}$ 36kb Block RAMs to implement two FIFOs of size K. Since standard K-merger uses only two FIFOs, the numbers of LUTs and registers to control FIFOs are very small.

Table VI shows the performance of 512k-merger/M for various values of M. It uses 18kb Block RAMs only when $M = 512$. For $M = 1$k and larger, it uses $S = \frac{512\text{k}}{M} + 1$ FIFOs with capacity M. Since FIFOs with capacity M ($M \geq 1$k) can be implemented using $\frac{M}{1k}$ 36kb Block RAMs, it uses $(\frac{512k}{M} + 1)\frac{M}{1\text{k}} = 512 + \frac{M}{1\text{k}}$ Block RAMs. Hence, the number of used Block RAMs increases and the number of LUTs decreases as the value of M increases. The appropriate values of M can be determined by the number of available resources when $K \geq$ 1k. For example, suppose that our first priority is to minimize the number of used block RAMs, and the second priority is to minimize the number of used LUTs. If this is the case, we should select $M = 1$k, because 512k-merger/M uses 513 Block RAMs when $M = 512$ and 1k, and it uses fewer LUTs when $M =$1k. Also, we can confirm that 512k-merger/M can be implemented using 513 block RAMs, while 512k-merger needs 1025 block RAMs. Hence, our 512k-merger/M reduces by half the number of block RAMs.

Table VII shows the best architecture for each K that minimizes the number of Block RAMs. If more than one architecture uses the same number of Block RAMs for some K, we have selected one that uses the minimum number of LUTs. In the table, "4+" in M means that two FIFOs of sizes 5 and 4 are used by standard K-merger is the best. It uses two 18kb block RAMs in one slice for FIFOs A and B. When $K = 512$, both our 512-merger/512 and standard K-merger uses 1 Block RAM. However, our 512-merger/512 uses fewer LUTs, and thus, it is selected. When $K = 1$k and larger, our K-merger/1k uses fewer LUTs than standard K-merger.

C. Top$K$-merger/M

Table VIII show the performance of Top$K$-merger using Distributed RAMs. We have selected parameter M that minimizes the number of Distributed RAMs for each K. From $K = 2$ to 512, two FIFOs F_0 and F_1 of size M are used. If $K \geq$ 1k, architectures with more than two FIFOs minimize the number of used LUTs, because the size of logic to control FIFOs is not dominant.

Tables IX show the performance of Top$K$-merger using Block RAMs. We have selected an architecture that minimizes the number of Block RAMs for each K. If more than one configurations use the same number of Block RAMs, we have selected one with the minimum number of LUTs. From $K = 8$ to 1k, we use 18kb block RAMs. For example, Top1k-merger

TABLE V. THE PERFORMANCE OF STANDARD K-MERGER USING BLOCK RAMS

K	4	8	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k	128k	256k	512k
Registers	121	128	135	142	149	156	163	170	177	184	191	198	205	212	227	228	237	246
LUTs(Logic)	423	307	323	353	476	471	514	578	477	572	621	492	519	582	603	575	623	757
Block RAMs	1	1	1	1	1	1	1	1	2	4	8	16	32	64	128	256	512	1024
18kb Block RAMs	2	2	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0
36kb Block RAMs	0	0	0	0	0	0	0	0	2	4	8	16	32	64	128	256	512	1024
Clock(MHz)	224	205	215	215	215	213	212	220	269	275	292	278	283	299	299	284	281	249

TABLE VI. THE PERFORMANCE OF 512K-MERGER/M USING BLOCK RAMS

M (FIFO size)	512	1k	2k	4k	8k	16k	32k	64k	128k	256k	512k
S (#FIFOs)	1025	513	257	129	65	33	17	9	5	3	2
Registers	54426	28325	14759	7715	4066	2180	1207	705	451	320	233
LUTs(Logic)	187083	96503	52593	25683	14096	7750	4333	2482	1559	1231	806
Block RAMs	513	513	514	516	520	528	544	576	640	768	1024
18kb Block RAMs	1025	0	0	0	0	0	0	0	0	0	0
36kb Block RAMs	0	513	514	516	520	528	544	576	640	768	1024
Clock(MHz)	156	184	209	207	211	212	220	226	225	246	236

TABLE VII. THE PERFORMANCE OF K-MERGER/M USING BLOCK RAMS: BEST CONFIGURATION IS SELECTED

K	4	8	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k	128k	256k	512k
M (FIFO size)	4+	8+	16+	32+	64+	128+	256+	512	1k	1k	1k	1k	1k	1k	1k	1k	1k	1k
S(#FIFOs)	2	2	2	2	2	2	2	2	2	3	5	9	17	33	65	129	257	513
Registers	121	128	135	142	149	156	163	147	155	227	342	569	1011	1898	3669	7188	14281	28325
LUTs(Logic)	423	307	323	353	476	471	514	546	526	821	1275	2083	3681	6622	11721	25593	48492	96503
Block RAMs	1	1	1	1	1	1	1	1	1	3	5	9	17	33	65	129	257	513
18kb Block RAMs	2	2	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0
36kb Block RAMs	0	0	0	0	0	0	0	0	2	3	5	9	17	33	65	129	257	513
Clock(MHz)	224	205	215	215	215	213	212	211	281	234	228	229	226	209	202	197	215	184

uses four FIFOs, F_0, F_1, F_2 and C, each of which stores 512 keys. For $K = 2\text{k}$ and above, we use 36kb block RAMs. For example, top2k-merger uses four FIFOs of size 1k each.

D. K-sorter and top$K$-sorter

Table X shows the performance of K-sorter. Recall that K-sorter can be implemented using 2^i-merger ($0 \leq i \leq \log_2 K - 1$). We succeeded in implementing up to 512k-sorter in XC7VX485T. Since a 18kb block RAM can store 512 keys and we have used distributed RAMs for 2^i-merger such that $2^i \leq 256$ and block RAMs for 2^i-merger such that $2^i \geq 512$. For example, 512-sorter uses 1-merger, 2-merger ,. . ., 256-merger, all of which are implemented using distributed RAMs. We have selected the best configuration for each 2^i-merger shown in Tables IV and VII.

Table XI shows the performance of Top$K$-sorter. Recall that Top$K$-sorter can be implemented using K-sorter and Top$K$-merger. We use K-sorter in Table X and Top$K$-merger in Tables VIII and IX for constructing top$K$-sorter. From $K = 8$ to 256, Distributed RAM implementations for Top$K$-merger shown in Table VIII are used. For $K = 512$ and above, Distributed RAM implementations in Table IX are used. We succeeded in implementing up to top256k-sorter in XC7VX485T.

VII. CONCLUSION

We have presented K-sorter and top$K$-sorter optimized for the latest FPGAs. From theoretical point of view, our K-sorter is almost optimal in terms of the latency, the total number of comparisons, and the total FIFO capacity. Also, our top$K$-sorter is close to be optimal in terms of the latency. We have implemented K-sorter and top$K$-sorter and evaluated the performance in a Virtex-7 FPGA. The implementation results show that our K-sorter and top$K$-sorter are practical and efficient.

REFERENCES

[1] K. Nakano and E. Takamichi, "An image retrieval system using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 811–818, May 2003.

[2] K. Nakano and Y. Yamagishi, "Hardware n choose k counters with applications to the partial exhaustive search," *IEICE Trans. on Information & Systems*, vol. E88-D, no. 7, 2005.

[3] K. Nakano and Y. Ito, "Processor, assembler, and compiler design education using an fpga," in *Proc. of International Conference on Parallel and Distributed Systems*, Dec. 2008, pp. 723–728.

[4] Xilinx Inc., *7 Series FPGAs Configurable Logic Block User Guide*, Nov. 2014.

[5] ——, *7 Series FPGAs Memory Resources User Guide*, Nov. 2014.

[6] D. E. Knuth, *The Art of Computer Programming. Vol.3: Sorting and Searching*. Addison-Wesley, 1973.

[7] M. Jeon and D. Kim, "Parallel merge sort with load balancing," *International Journal of Parallel Programming*, vol. 31, no. 1, pp. 21–33, February 2003.

[8] K. Batcher, "Sorting networks and their applications," in *in Proceedings of the AFIPS Spring Joint Computer Conference 32*, 1968, pp. 307–314.

[9] M. F. Ionescu and K. E. Schauser, "Optimizing parallel bitonic sort," in *in Proceedings of the 11th International Symposium on Parallel Processing*, April 1997, pp. 303–309.

[10] D. R. Helman, D. A. Bader, and J. JáJá, "A randomized parallel sorting algorithm with an experimental study," *Journal of Parallel and Distributed Computing*, vol. 52, pp. 1–23, July 1998.

[11] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin, "Fast parallel sorting under Log P: Experience with the CM-5," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 791–805, August 1996.

TABLE VIII. THE PERFORMANCE OF TOPK-MERGER USING DISTRIBUTED RAMS: BEST CONFIGURATION IS SELECTED

K	2	4	8	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k
M (FIFO size)	2	4	8	16	32	64	128	256	512	256	256	1k	1k	1k	2k	2k
S (#FIFOs)	2	2	2	2	2	2	2	2	2	5	9	5	9	17	17	33
Registers	46	22	29	36	43	50	57	64	71	146	223	176	269	450	489	878
LUTs	204	214	226	243	265	327	506	786	1362	2223	3880	6937	12759	24381	46473	91385
LUTs (Logic)	156	142	154	171	193	215	286	346	482	991	1592	2009	3607	6781	11273	22393
LUTs (Memory)	48	72	72	72	72	112	220	440	880	1232	2288	4928	9152	17600	35200	68992
Clock(MHz)	248	248	248	248	248	255	231	219	199	197	170	160	142	144	138	120

TABLE IX. THE PERFORMANCE OF TOPK-MERGER/M USING BLOCK RAMS: BEST CONFIGURATION IS SELECTED

K	8	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k	128k	256k	512k
M (FIFO size)	8	16	32	64	128	256	512	1k	1k	1k	1k	1k	1k	1k	1k	1k	1k
S(#FIFOs)	2	2	2	2	2	2	2	3	3	5	9	17	33	65	129	257	513
Registers	128	135	142	149	156	163	147	155	227	342	569	1011	1898	3669	7188	14281	28325
LUTs(Logic)	658	559	494	534	859	581	605	867	762	1421	2130	3784	6878	13070	23445	47794	9417
Block RAMs	2	2	2	2	2	2	2	2	4	7	13	25	49	97	193	385	769
18kb Block RAMs	3	3	3	3	3	3	3	4	0	0	0	0	0	0	0	0	0
36kb Block RAMs	0	0	0	0	0	0	0	0	4	7	13	25	49	97	193	385	769
Clock(MHz)	215	216	216	216	215	216	216	190	250	219	209	207	206	198	178	173	158

TABLE X. THE PERFORMANCE OF K-SORTER

K	8	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k	128k	256k	512k
Registers	83	116	185	261	344	434	531	678	855	1081	1423	1989	3000	4898	8556	15742	29978
LUTs	602	796	1003	1257	1605	1994	2847	3161	3709	4378	5482	7190	10334	16615	27947	50076	93280
LUTs(Logic)	482	628	787	993	1253	1466	1967	2281	2829	3498	4602	6310	9454	15735	27067	49196	92400
LUTs(Memory)	120	168	216	264	352	528	880	880	880	880	880	880	880	880	880	880	880
Block RAMs	0	0	0	0	0	0	0	1	3	6	11	20	37	70	135	264	521
18kb Block RAMs	0	0	0	0	0	0	0	2	2	2	2	2	2	2	2	2	2
36kb Block RAMs	0	0	0	0	0	0	0	0	2	5	10	19	36	69	134	263	520
Clock(MHz)	266	255	254	254	254	227	227	211	200	200	200	200	200	199	185	185	182

TABLE XI. THE PERFORMANCE OF TOPK-SORTER

K	8	16	32	64	128	256	512	1k	2k	4k	8k	16k	32k	64k	128k	256k
Registers	111	151	227	310	400	498	701	855	1114	1455	2021	3034	4932	8590	15779	30119
LUTs	783	1019	1264	1570	1974	2764	3240	3788	4467	5553	7433	10512	16848	28172	50445	97381
LUTs(Logic)	591	779	976	1194	1402	1796	2360	2908	3587	4673	6553	9632	15968	27292	49565	96501
LUTs(Memory)	192	240	288	376	572	968	880	880	880	880	880	880	880	880	880	880
Block RAMs	0	0	0	0	0	0	2	4	7	13	24	45	86	167	328	649
18kb Block RAMs	0	0	0	0	0	0	3	3	2	2	2	2	2	2	2	2
36kb Block RAMs	0	0	0	0	0	0	0	2	6	12	23	44	85	166	327	648
Clock(MHz)	248	248	248	254	231	219	216	201	200	200	200	200	195	180	170	167

[12] D. Man, Y. Ito, and K. Nakano, “An efficient parallel sorting algorithm compatible with the standard qsort,” *International Journal on Foundations of Computer Science*, vol. 22, no. 5, pp. 1057–?1072, 2011.

[13] A. Sohn and Y. Kodama, “Load balanced parallel radix sort,” in *in Proceedings of the 12th ACM International Conference on Supercomputing*, July 1998, pp. 305–312.

[14] S. J. Lee, M. Jeon, D. Kim, and A. Sohn, “Partitioned parallel radix sort,” *Journal of Parallel and Distributed Computing*, vol. 62, pp. 656–668, 2002.

[15] P. Kipfer and R. Westermann, “Improved GPU sorting,” in *GPU Gems 2*. Addison Wesley, 2005, pp. 733–746.

[16] E. Sintorn and U. Assarsson, “Fast parallel GPU-sorting using a hybrid algorithm,” *Journal of Parallel and Distributed Computing*, vol. 68, pp. 1381–1388, October 2008.

[17] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberger, “A novel sorting algorithm for many-core architectures based on adaptive bitonic sort,” in *Proc. of International Parallel and Distributed Processing Symposium*, May 2012, pp. 228–237.

[18] S. Todd, “Algorithm and hardware for a merge sort using multiple processors,” *IBM Journal of Research and Development*, vol. 22, no. 5, pp. 509–517, Sept. 1978.

[19] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[20] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules in large databases,” in *Proc. of the 20th International Conference on Very Large Data Bases*, Sept. 1994, pp. 487–499.

[21] R. J. Bayardo Jr., “Efficiently mining long patterns from databases,” in *Proc. of ACM SIGMOD International Conference on Management of Data*, 1998, pp. 85–93.

[22] R. Marcelino, H. C. Neto, and J. M. P. Cardoso, “Unbalanced FIFO sorting for FPGA-based systems,” in *Proc. of International Conference on Electronics, Circuits, and Systems*, Dec. 2009, pp. 431 – 434.

[23] D. Koch and J. Torresen, “FPGASort: A high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting,” in *Proc. of International Symposium on Field Programmable Gate Arrays*, 2011, pp. 45–54.

[24] A. Farmahini-Farahani, A. Gregerson, M. Schulte, and K. Compton, “Modular high-throughput and low-latency sorting units for fpgas in the large hadron collider,” in *Proc. of Symposium on Application Specific Processors*, 2011, pp. 38–45.

[25] Xilinx Inc., *VC707 Evaluation Board for the Virtex-7 FPGA User Guide*, 2014.

[26] IEEE, *754-2008 - IEEE Standard for Floating-Point Arithmetic*, Aug. 2008.