2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip

FACE: Fast and Customizable Sorting Accelerator for Heterogeneous Many-core Systems

Ryohei Kobayashi
Tokyo Institute of Technology
Tokyo, Japan
E-mail: kobayashi@arch.cs.titech.ac.jp

Kenji Kise
Tokyo Institute of Technology
Tokyo, Japan
E-mail: kise@cs.titech.ac.jp

***Abstract*—Performance improvements of a single-core processor relying on high clock rates reached the limit. Instead of a single-core processor, multi-core and many-core processors have been mainstream to accelerate applications by parallel processing. Year by year, the number of cores integrated in a single chip has been increased due to improvements of semiconductor integration technologies depending on Moore's Law. On the other hand, Moore's Law will be ended in the near future. This means that the approaches relying on the increase in the number of cores will be hopeless, thus we have to consider other effective ways. One of them is to design application specific hardware. Several research organizations have explored it and reported its remarkable findings. We focus on such an acceleration approach with dedicated hardware.**

As a case study with dedicated hardware, we present a sorting acceleration method. Sorting is an extremely important computation kernel that should be accelerated in a lot of fields, such as databases, image processing, data compression, etc. We propose a sorting accelerator combining *Sorting Network* and *Merge Sorter Tree*, and detail the design and implementation. Our proposed sorting accelerator is customizable, thus designers can implement a sorting accelerator composed of required hardware resources by means of tuning design parameters. Our experiments show that the proposed hardware achieves up to 10.06x sorting performance, compared with Intel Core i7-4770 operating at 3.4GHz, when sorting 256M 32-bits integer elements. In order to allow every designer to easily and freely use this accelerator, the RTL source code is released as an open-source hardware.

***Keywords*-Fast; Customizable; Sorting; Hardware specialization; Open source;**

I. Introduction

Taking advantage of improvements of clock speeds is no longer promising, and multi-core processors are attractive to accelerate applications by parallel processing. The number of cores has been increased year after year due to improvements of semiconductor integration technologies depending on Moore's Law [1], and many-core processors with up to 256 cores begin to appear [2], [3], [4].

However, Moore's Law will be ended in the near future. This means that the approaches relying on the increase in the number of cores are hopeless. The authors in [5] mention that the trend, which is that semiconductor integration technologies grow exponentially, will be ended, and hardware specialization will be crucial in the future. In [5], the Field-Programmable Gate Array (FPGA) based system to accelerate data center tasks is proposed. This system improves PageRank throughput of "Bing" at twice. [6] presents a dedicated hardware to accelerate stencil computation that is used in lots of field, such as fluid calculation, weather calculation, molecular simulation, etc. The proposed hardware is implemented by using multiple FPGA boards. As a result, the accelerator obtains higher computational performance at up to 13.7 times, compared with Intel Core i7-3930K with six cores operating at 3.2 GHz. Thus, application specific hardware is one of the most desirable ways.

As a case study with dedicated hardware, we focus on integer sorting. Sorting is an extremely important computation kernel that has been accelerated in a lot of fields — databases, image processing, data compression [7], [8], [9], [10], [11].

In [7], [8], [9], FPGA-based systems with *Sorting Networks* are implemented and evaluated in terms of circuit areas, throughputs, and power consumptions. A sorting network is able to sort a fixed number of values, which consists of wires and comparators. This network is easy to be implemented in hardware due to simplicity of the architecture. However, this sorting architecture is unsuitable for larger data sequences. This is because more comparators are required to sort them, and this causes the circuit area increase and the operating frequency degradation. Therefore, the sizes of these data sequences are small. In [7], if the data-sequence size is less than 8, it can be fully sorted only in the sorting network. However if not, the CPU has to merge these sorted portions.

As an approach to merge these portions, *Merge Sorter Tree* is proposed in [10], [11]. The merge sorter tree is a data path that merges sorted portions into a fully sorted data sequence and the data path consists of connecting sorter cells that have 2-inputs and 1-output as a perfect binary tree. We use this sorting architecture as a component to merge the sorted portions. Consequently, larger data sequences can be fully sorted, while the systems with only the sorting network cannot do.

We propose a sorting hardware combining these two sorting architectures, i.e. the sorting network and the merge sorter tree. The proposed sorting hardware is customizable, and designers can implement a sorting accelerator composed of required hardware resources by means of tuning design parameters. In this paper, we detail the design and implementation, and evaluate the proposed hardware in terms of its sorting process time and hardware resource usage. To

978-1-4799-8670-5/15 $31.00 © 2015 IEEE
DOI 10.1109/MCSoC.2015.40

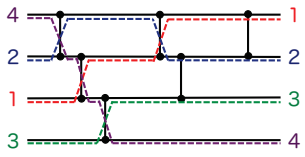

Figure 1. Bubble sort network with 4-inputs and 4-outputs

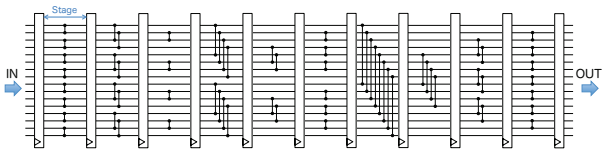

Figure 2. Pipelined synchronous Batcher's odd-even merge sort network with 16-inputs and 16-outputs

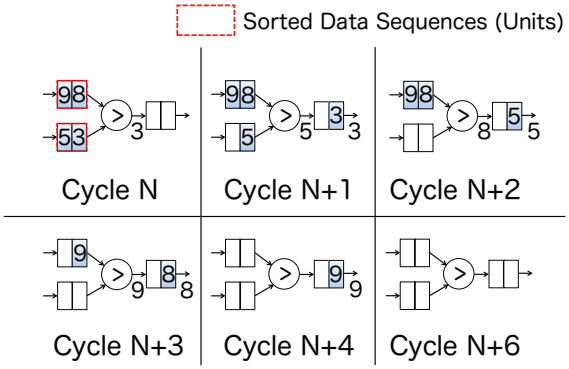

Figure 3. Sorting process in merge sorter tree

allow every designer to easily and freely use this accelerator, the Register Transfer Level (RTL) source code is released as an open-source hardware [12]. Our contributions in this work are:

- We propose a high performance and customizable sorting accelerator with two sorting architectures, which are the sorting network and the merge sorter tree.
- The proposed sorting accelerator is implemented on the Xilinx VC707 evaluation kit [13], and achieves up to 10.06x sorting performance compared with Intel Core i7-4770 operating at 3.4GHz when sorting 256M 32-bits integer elements.
- To allow every designer to easily and freely use this accelerator, we release the RTL source code in Verilog HDL as an open-source hardware. To the best of our knowledge, this is the first open-source sorting accelerator that is high performance and customizable in the world.
- For various embedded systems, designers can customize a sorting accelerator composed of required hardware resources by means of tuning design parameters.

This paper is organized as follows. We describe the sorting architectures that we use in Section II. In Section III, the design and implementation of the proposed sorting hardware are detailed, and Section IV shows the sorting performance, the hardware resource usage, and the discussion about these results. Finally, we conclude this paper in Section V.

II. Sorting Architectures

Our proposed sorting accelerator takes advantage of the sorting network and the merge sorter tree. We describe these sorting architectures.

A. Sorting Network

A sorting network [14] is an algorithm that sorts a fixed sequence of numbers by using a fixed sequence of comparisons. The sorting network consists of two types of items, which are wires and comparators. The wires are running from left to right, carrying values (one per wire) that traverse the network all at the same time. Each comparator connects two wires. When a pair of values, traveling through a pair of wires, encounters a comparator, the comparator swaps the values only if the top wire's value is greater than the bottom wire's value. The sorting network benefits are to sort values in parallel and to be implemented without complicated hardware. That is why the sorting network is a desirable component for building high performance sorting hardware [7], [8], [9].

Fig. 1 shows a sorting network with 4-inputs and 4-outputs. This network realizes bubble sort, since the largest value is carried to the bottom at first. By changing the connection of the comparators, sorting networks can realize lots of sorting algorithms, such as even-odd merge sort, bitonic sort, bubble sort, insertion sort, etc. In [7], authors implement several sorting networks on an FPGA and conclude that Batcher's even-odd merge sort network is the most efficient in terms of hardware resource usage and throughput. Consequently, our proposed hardware uses this sorting network.

Fig. 2 shows Batcher's even-odd merge sort network [15] with 16-inputs and 16-outputs. This sorting network consists of 63 comparators and 10 stages. Although it is possible to be implemented as a purely combinational circuit, this case probably causes performance reduction because of large network delay. To address this problem, it is a common way to implement this network as a pipelined circuit by inserting registers between each stage, which prevents a degradation of the operating frequency and improves the network throughput [7]. This network is embedded in our proposed hardware.

B. Merge Sorter Tree

The merge sorter tree [10] has highly effective performance and good hardware resource usage. The merge sorter tree is a data path that executes merge process and the data path consists of connecting sorter cells as a perfect binary tree. Sorter cells compare two input-values and output one of them, depending on its comparison result.

Fig. 3 shows how elements are sorted in the merge sorter tree. The merge sorter tree in Fig. 3 has two input ports. We define the tree in Fig. 3 as 2-**way** merge sorter tree. If a

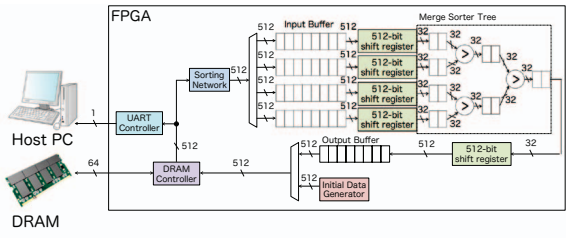

Figure 4. Data path of the proposed sorting accelerator

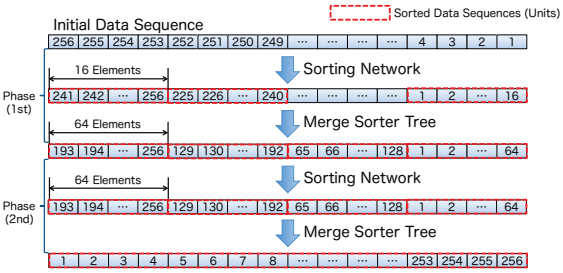

Figure 5. Example: sorting 256 elements from 256 to 1

merge sorter tree has k input ports, the tree is called k-way merge sorter tree. Now, we explain how elements are sorted in this 2-way merge sorter tree.

First, at Cycle N, each way outputs integers of 8, and 3. Then, 8 and 3 are compared. The data sequences in the leftmost FIFOs must be sorted. We define these data sequences as **Units**. In this example, the sorted element 9 and 8 in the upper FIFO is a Unit, and the element 5 and 3 in the lower FIFO is another Unit. The sorter cell outputs the smaller element depending on the comparison result, unless the output FIFO of the sorter cell is full. At Cycle N+1, 3 is emitted from the root. At the same time, 8 and 5 are compared, and then the sorter cell outputs 5. At Cycle N+2, 5 is emitted from the root. At the same time, the sorter cell outputs 8, because the sorter cell receives only one element.

As shown in Fig. 3, the Units are merged in the tree, and then the root of the tree emits the sorted data sequence. In other words, the tree merges the two Units, and then generates the one Unit composed of 3, 5, 8, and 9. If k-way merge sorter tree executes this process, the tree can merge k Units and generate a larger Unit.

III. Proposed Sorting Accelerator

A. Data Path

Fig. 4 shows a data path of the proposed sorting accelerator. We implement it on an FPGA, and verify that it accurately works by using a host PC. We design two modules of Initial Data Generator and UART Controller for the verification and the performance evaluation.

We explain how the hardware sorts data sequences using Fig. 5. For simplicity, the initial data sequence of 256 elements is a reverse-order data sequence from 256 to 1.

A data sequence for sorting is generated from Initial Data Generator. The module can support three data-generation types, which are a random data sequence using Xorshift [16], a sorted data sequence, and a reverse-order sorted data sequence. The data type is 32-bits integer. At first, a data sequence emitted from the module is stored in the external memory via DRAM Controller.

After that initialization, the data sequence in the external memory is loaded and is sent to Sorting Network via DRAM Controller. Sorting Network is Batcher's even-odd merge sort network [15] with 16-inputs and 16-outputs. This means that this network can sort 16 elements. Thus, the initial data sequence turns into 16 sorted data sequences by passed through this network. In other words, the number of Units is 16 and one Unit has 16 elements as shown in Fig. 5.

The data sequence passed through Sorting Network is stored in Input Buffer that consists of FIFO. The stored elements must already be sorted. The data sequence stored in Input Buffer is sent to 512-bit shift register. This shift register breaks down a 512-bits data into 16 elements, and then sends them to Merge Sorter Tree.

For simplicity, we draw 4-way merge sorter tree in Fig. 4. By comparing elements at every sorter cell and storing outputs in the FIFOs in each cycle, the merged data sequence is emitted from the root of the merge sorter tree. After passed through the tree, the data sequence composed of 16 Units turns into 4 Units, each of which has 64 elements.

The data sequence emitted from the root of the merge sorter tree is sent to 512-bit shift register, and then is packed into a 512-bits data. After packed, the data sequence is sent to Output Buffer, and then is stored in the external memory via DRAM Controller. However, the data sequence is not fully sorted yet. Thus, it is read from the memory, and then is sent to Sorting Network again. In this time, the network is a mere data path because portions of the data sequence are already sorted.

The data sequence passed through the network is sent to the tree. In the tree, 4 Units are merged into one Unit and then elements of the Unit are emitted from the root of the tree cycle by cycle. This means that all of the emitted elements are fully sorted. The data sequence is stored in the external memory via DRAM Controller, after passed through 512-bit shift register and Output Buffer.

To verify the result, the fully sorted data sequence in the external memory is loaded and is sent to UART Controller via DRAM Controller. UART Controller sends it to the host PC by serial communication. The transferred data sequence is checked, using typical sorting software.

By passing the data sequence through Sorting Network and Merge Sorter Tree twice in this case, it can be fully sorted. We define the process that passes the data sequence through Sorting Network and Merge Sorter Tree as **Phase**. The number of required Phases for fully sorting the data sequence is given by $\log_{\#\,of\,ways} \frac{\#\,of\,elements}{16}$ where 16 is the number of sorted elements at Sorting Network in the first Phase. For instance, in Fig. 5 the number of required Phases is 2, because the number of ways and elements to be sorted is 4 and 256 respectively.

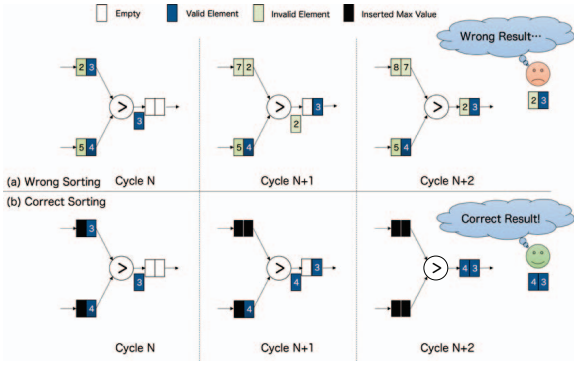

Figure 6. Wrong sorting and correct sorting

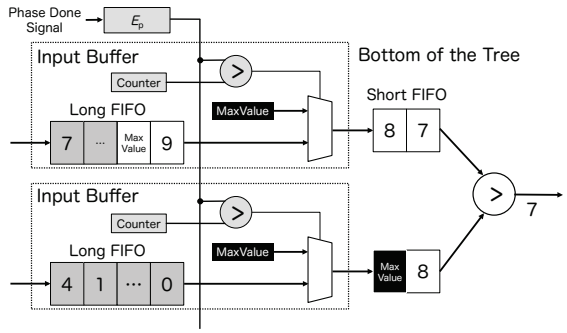

Figure 7. Two input buffers and one sorter cell

B. Control Logic

We describe a control mechanism to sort data sequences.

Depending on the number of ways and elements to be sorted, the number of required Phases is decided. When Units are merged in the merge sorter tree, each Unit needs to be treated separately. If not be separated, that sorting cannot be executed successfully, because invalid elements are mixed into Units.

In Fig. 6 (a), this example is demonstrated. Fig. 6 shows a wrong case (a) and a correct case (b) of merging Units (i.e. merging 3 and 4). We define Valid elements as the elements which should be merged into one Unit in the merge sorter tree (i.e. 3 and 4). At Cycle N+1, 4 should be emitted from the sorter cell, because 4 is a Valid element. However, in (a) 2 is emitted, which is an Invalid element, hence this sorting cannot be done successfully.

To address this problem, we propose that the maximum value, which depends on the bit width of elements, is inserted after Valid elements [17]. Fig. 6 (b) shows how this method is applied. By doing so, this sorting can be executed successfully, because 4 is emitted from the sorter cell at Cycle N+1.

To realize this (Fig. 6 (b)), a circuit that generates the maximum value to separate Units is implemented in Input Buffer as shown in Fig. 7. Each Input Buffer has a counter, which counts the number of emitted elements from this buffer. We define the number of elements in each Unit in Phase p as E_p. When the counter value exceeds E_p, the maximum value is emitted from this buffer, and this buffer keeps holding subsequent Units.

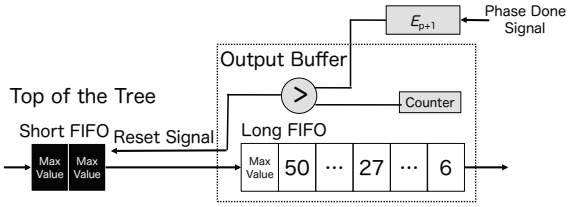

Figure 8. How to generate reset signal from output buffer

Output Buffer also has a counter, which counts the number of stored elements in Output Buffer as shown in Fig. 8. When the counter value exceeds E_{p+1}, all FIFOs in the merge sorter tree, the counter of Input Buffer, and the counter of Output Buffer are reset. After this, the tree begins to merge subsequent Units. Exceeding E_{p+1} means that all elements of a Unit, which is generated in the merge sorter tree, are stored in Output Buffer.

Due to this mechanism, each Unit is treated separately and it can be guaranteed that elements are sorted successfully. In Fig. 5, in the 1st Phase E_p is 16, E_{p+1} is 64, and in the 2nd Phase E_p is 64, E_{p+1} is 256.

C. Performance Model

We analyze theoretical performance of the proposed sorting accelerator. To analyze it, we assume that the DRAM bandwidth is infinity and the latency is 1 cycle. Depending on the number of ways and elements to be sorted, we calculate the number of required cycles to fully sort them. This can be calculated by summation of the number of cycles in each Phase.

As described in Section III-A and Section III-B, multiple Units are merged into one Unit in the merge sorter tree. We call this process **Iteration**. In Fig. 5, 16 Units generated from the sorting network turn into 4 Units by passed through the merge sorter tree in the 1st Phase. This means that four Iterations are executed in the 1st Phase. In other words, the number of Iterations in the 1st Phase is 4, and that is 1 in 2nd Phase. We define the number of Iterations, ways, and elements to be sorted by the proposed system as I, k, and N respectively. In nth Phase, the number of Iterations for nth Phase is given by

$$I_n = \frac{N}{16k^n} \tag{1}$$

where 16 is the number of sorted elements at the sorting network in the 1st Phase.

After Iteration, all FIFOs in the merge sorter tree are reset (Section III-B). Therefore, a few cycles overhead exists between each Iteration. This overhead OH_{iter} is given by

$$OH_{iter} = \log_2 k + 1 \tag{2}$$

and OH_{iter} is equal to the number of stages of the merge sorter tree.

The beginning of each Phase also has an overhead. The merge sorter tree cannot sort data sequences unless elements

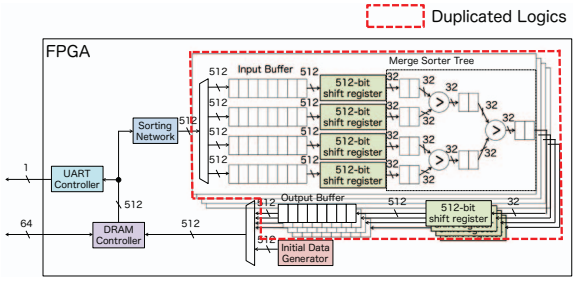

Figure 9. Data path of the proposed sorting accelerator with the duplicated merge sorter trees

are stored in all of the leftmost FIFOs. Elements have to be stored in these FIFOs immediately, because they are empty at the beginning of each Phase. In other words, this is the overhead. We define the number of required cycles for this buffering as α, and then the overhead OH_{phase} is given by

$$OH_{phase} = k\alpha \tag{3}$$

where α is tens of cycles at most.

We define the number of required cycles for nth Phase as C_n. This is given by the following formula.

$$C_n = N + I_n \times OH_{iter} + OH_{phase} \tag{4}$$

We explain this formula in three parts. First, the throughput of the merge sorter tree is one element per cycle. Thus, it takes N cycles to emit all elements from the merge sorter tree. Second, in nth Phase, the number of Iterations is I_n. Thus, the number of cycles for the overhead of all Iterations is $I_n \times OH_{iter}$, because OH_{iter} cycles overhead exists between each Iteration. Third, the beginning of each Phase has OH_{phase} cycles overhead as mentioned. Consequently, C_n can be calculated by summation of the number of these cycles.

Hence, C_{fully}, which is the number of required cycles to fully sort the data sequence, is given by the following formula.

$$C_{fully} = \sum_{i=1}^{n} C_i \tag{5}$$

where n is the number of required Phases. As described in Section III-A, the number of required Phases to fully sort the data sequence is $\log_k \frac{N}{16}$. In other words, C_{fully} can be also given by the following formula.

$$C_{fully} = \sum_{i=1}^{\log_k \frac{N}{16}} \{N + \frac{N}{16k^i}(\log_2 k + 1) + k\alpha\} \tag{6}$$

The sorting process time can be estimated by means of dividing C_{fully} by the operating frequency.

D. Improvement by Duplication of the Merge Sorter Tree

We describe how to improve the proposed sorting accelerator. One of the approaches to achieve this is to improve the sorting logic throughput. We propose duplication of the merge sorter tree. This approach is simple, yet effective for the throughput improvement.

Fig. 9 shows a data path of the sorting accelerator with the duplicated merge sorter trees. The duplicated trees work in parallel. Thus, the more the tree is duplicated, the higher the sorting logic throughput is.

By taking advantage of the performance model described in Section III-C, it is possible to analyze theoretical performance of the sorting accelerator with the duplicated trees. If the number of duplicated trees is defined as P, the number of required cycles for nth Phase is given by $\frac{C_n}{P}$. This is because the duplicated trees sort data sequences in parallel. In the last Phase, the parallelism benefit cannot be obtained. Thus, C_{last}, which is the number of required cycles for the last Phase, is given by

$$C_{last} = N + 1 \times OH_{iter} + OH_{phase} \tag{7}$$

where the number of Iterations for the last Phase is definitely one. Therefore C_{fully_dup}, which is the number of required cycles to fully sort the data sequence by the sorting accelerator with the duplicated trees, is given by the following formula.

$$C_{fully\_dup} = C_{last} + \sum_{i=1}^{(\log_k \frac{N}{16})-1} \frac{C_i}{P} \tag{8}$$

Hence, the sorting process time is estimated, depending on the number of ways, duplicated trees, and elements to be sorted by the sorting accelerator. Besides, designers can implement a sorting accelerator composed of required hardware resources, by means of tuning the number of ways and duplicated trees.

E. Implementation

As a platform for the proposed FPGA accelerator, we use the Xilinx Virtex-7 FPGA VC707 evaluation kit [13]. This kit originally has the Virtex-7 XC7VX485T and 1GB DDR3 SO-DIMM (800MHz/1600Mbps) memory, but we replace this memory with 4GB DDR3 SO-DIMM memory in order to sort larger data sequences.

The sorting logic is implemented in Verilog HDL. To implement DRAM Controller, we use an IP core provided by Xilinx [18]. As a synthesis tool, we use Vivado 2014.4 [19] with the synthesis options default, and the placed and routed logic meets all timing constraints. All implemented logics on the FPGA operate at 200MHz and DRAM operates at 800MHz. Consequently, between the FPGA and the DRAM, the maximum data-transfer speed is 12.8GB/s.

IV. Evaluation

A. Sorting Performance

We compare the sorting performance of the proposed hardware with Intel Core i7-4770 operating at 3.4GHz. The data-sequence size is 256M elements, whose data type is 32-bits integer. We use merge sort and quick sort for the software running on Intel Core i7-4770, implement them in C language, and compile them with gcc 4.8.2 (-O3 optimization). These two applications are executed as single

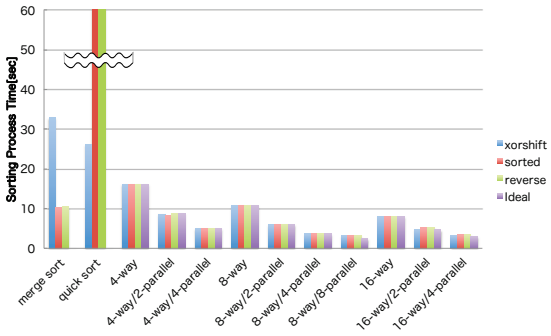

Figure 10. Sorting performance comparison between the software and the proposed sorting accelerator

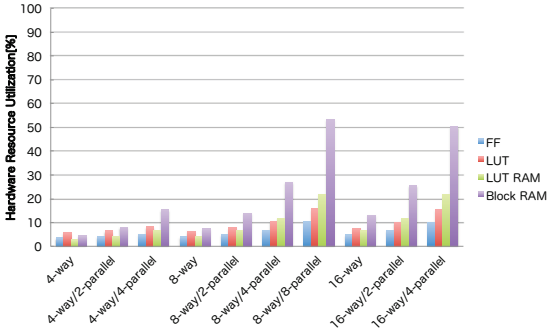

Figure 11. Hardware resource usage of the proposed sorting accelerator

thread of Intel Core i7-4770. For measurement of the sorting process time, in the case of the sorting accelerator we get sorting execution cycles stored in the hardware counter and calculate the time, and we use gettimeofday in case of the applications.

Fig. 10 shows the sorting performance comparison between the software and the proposed sorting accelerator. In Fig. 10, 8-way represents the sorting accelerator with 8-way merge sorter tree and 8-way/2-parallel represents the hardware with two 8-way merge sorter trees. xorshift, sorted, and reverse represent that the initial data-sequence types are a random data sequence, a sorted data sequence, and a reverse-order sorted data sequence respectively. Moreover, Ideal stands for the theoretical performance obtained from the performance model described in Section III-C and Section III-D.

In xorshift, the 4-way performance is 2.03x and 1.61x, compared with merge sort and quick sort respectively. The 8-way performance is 3.05x and 2.42x; the 16-way performance is 4.07x and 3.24x than the two applications. This shows that the more the number of ways is increased, the higher the sorting performance is. This is because the number of required Phases to fully sort the data sequence is decreased since the more the number of ways is increased, the more the number of elements to be sorted in the merge sorter tree is increased.

Besides, the more the merge sorter tree is duplicated, the higher the sorting performance is. For instance, the 4-way/2-parallel performance is 1.88x and the 4-way/4-parallel performance is 3.21x by compared with 4-way. This is because the duplicated trees sort a data sequence in parallel. In this paper, 8-way/8-parallel results in the highest performance that is 10.06x and 8.01x compared with merge sort and quick sort respectively.

Although the increase in the number of duplicated trees leads to higher sorting performance, the required memory bandwidth is increased. In other words, to improve the sorting logic throughput, the required memory bandwidth becomes larger. Using P, the sorting logic throughput is given by

$$200MHz \times 4Bytes \times P \times 2 = 1.6P\,GB/s \qquad (9)$$

where $200MHz \times 4Bytes$ depends on the throughput of the merge sorter tree. The tree operates at 200MHz, and emits one element per cycle from the root, whose data size is 4Bytes. And the constant 2 comes from DRAM read and write.

If the merge sorter tree is not duplicated (4-way, 8-way, and 16-way), the sorting performance is equal to the theoretical one, because the sorting performance is insensitive to the memory bandwidth. In this case, the sorting logic throughput is $200MHz \times 4Bytes \times 1 \times 2 = 1.6GB/s$. The memory bandwidth usage of this hardware is 12.5% of the maximum memory bandwidth. This means that there is sufficient margin in the memory bandwidth, and the sorting logic operates with an efficiency of 100%.

However, the more the merge sorter tree is duplicated, the more the memory bandwidth usage is close to 100%. This means the sorting performance becomes sensitive to the memory bandwidth. In particular, the sorting logic throughput of 8-way/8-parallel is $200MHz \times 4Bytes \times 8 \times 2 = 12.8\,GB/s$, which is equal to the maximum memory bandwidth. However, the practical memory bandwidth is lower than the maximum due to the overhead of DRAM Controller. Thus, in 8-way/8-parallel the memory bandwidth bottleneck degrades the sorting performance, which is 70.2% of the theoretical one.

As shown in Fig. 10, the performance of xorshift, sorted, and reverse of the sorting accelerator are almost same. This means that the sorting accelerator is independent on the data-sequence type. On the other hand, the software considerably depends on it. Especially, the results of sorted and reverse of quick sort clearly show this aspect because of the worst-case complexity of $O(n^2)$.

B. Hardware Resource Usage

Fig. 11 shows the hardware resource usage of the sorting accelerator. In Fig. 11, FF, LUT, LUT RAM, and Block RAM represent a flip-flop (FF), a lookup table (LUT) for combinational logic, LUT for distributed memory, and an internal memory (hard macro) of the FPGA.

As shown in Fig. 11, the logic usage except Block RAM is almost within 20%. Because Block RAM is used to implement Input Buffer and Output Buffer, the more the

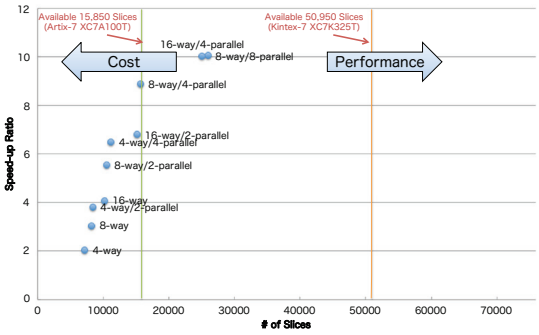

Figure 12. Relationship between the performance and the hardware resource usage

number of ways and duplicated trees is increased, the larger the Block RAM usage is. However, it is possible to mitigate the usage by means of tuning the number of FIFO entries, since Input Buffer and Output Buffer consist of them.

C. Discussion

We evaluated the sorting accelerator in terms of the sorting process time and the hardware resource usage. These results show that the more the number of ways and duplicated trees is increased, the higher the sorting performance is, although it needs more hardware resources. Well then, if same hardware resource usage, which customized hardware achieves the highest sorting performance?

Fig. 11 shows that the hardware resource usage is almost same among three configurations, which are 4-way/4-parallel, 8-way/2-parallel, and 16-way. This means that if the total number of Input Buffers is same, the hardware resource usage is almost same. In the three customized hardware, the results of 4-way/4-parallel, 8-way/2-parallel, and 16-way are 5.05sec, 5.93sec, and 8.08sec. Thus, 4-way/4-parallel achieves the highest sorting performance. This result means that if same hardware resources are used, the hardware that has more duplicated trees can achieve higher performance. This is due to the sorting logic throughput described in Section IV-A.

Fig. 12 shows relationship between the performance and the hardware resource usage. The x-axis and y-axis stand for the number of consumed slices and the speed-up ratio compared with merge sort. A Slice is a term used by Xilinx, and it is a logic component including several LUTs and FFs. As mentioned before, if the total number of Input Buffers is same, the hardware resource usage is almost same — (4-way/2-parallel and 8-way), (4-way/4-parallel, 8-way/2-parallel, and 16-way), (16-way/2-parallel and 8-way/4-parallel), and (16-way/4-parallel and 8-way/8-parallel). However, It is found that the hardware, which has more duplicated trees, consumes slightly more slices. This is because control logics for the duplicated trees are also duplicated.

We draw three areas in Fig. 12. The left is for cost aware systems. We define the borderline of this area as the number of available slices on the Artix-7 XC7A100T that is used in the Digilent Nexys4 board [20]. The middle is for cost-performance aware systems. We define the upper border of this area as the number of available slices on the Kintex-7 XC7K325T that is used in the Xilinx Kintex-7 FPGA KC705 Evaluation Kit [21]. If more performance-aware systems are required, they need larger devices like the Virtex-7 FPGAs. As shown in Fig. 12, the designs except 16-way/4-parallel and 8-way/8-parallel are within the left area, and the other designs are within the middle area. This means that most of the presented designs in this paper can be implemented on low-end devices and our proposed accelerator is available on various environments depending on constraints of the cost and performance. We release the RTL source code as an open-source hardware. Hence, designers can customize a sorting accelerator composed of required hardware resources by means of tuning the number of ways and duplicated trees. However, note that if the number of duplicated trees is increased, it is crucial to consider the memory bandwidth usage.

8-way/8-parallel achieves up to 10.06x speed-up compared with single thread of Intel Core i7-4770 operating at 3.4GHz. In comparison with the systems using the sorting network [7], [8], [9], our proposed system can sort more elements at higher speed. However, our system performance is about one-seventh of the system proposed in [11]. One of the reasons is that the merge sorter tree in [11] can handle 6 elements per cycle, while our system can do only one element per cycle.

However, the merge sorter tree, which can handle multiple values per cycle, is truly difficult to be operated at high clock frequency. We also implement the tree that can handle 4 elements per cycle, and then in the RTL simulation the sorting accelerator with the tree works successfully. However, the logic can operate at most 140MHz according to the post-place and route timing report. Therefore, to realize the method proposed in [11], it definitely needs high-level optimization techniques. Unlike [11], our sorting accelerator is simple, relatively high speed, customizable, and the RTL source code is released as an open-source hardware. These are big differences with the prior work and we have never seen such a sorting hardware.

Our future work is to realize higher performance hardware than the prior work. For achievement of this, although it is necessary to improve the sorting logic throughput, the memory bandwidth usage has to be considered. The higher the sorting logic throughput is, the more the memory bandwidth becomes the bottleneck of the entire system performance, like 8-way/8-parallel. Therefore, it is necessary to improve the memory bandwidth usage, while keeping the operate frequency high. This is our next challenge.

V. Conclusion

In this work, we presented the acceleration approach for sorting application. Our proposed accelerator has two sorting architectures that are the sorting network and the merge sorter tree, and we detailed the design and implementation. The most characteristic point of the proposed system is customizable. Designers can implement a sorting accelerator

composed of required hardware resources by means of tuning the number of ways and duplicated trees, depending on constraints of the cost and the performance. In order to allow every designer to easily and freely use this accelerator, the RTL source code is released as an open-source hardware. To the best of our knowledge, there is no such a sorting accelerator. This is the first open-source sorting accelerator that is high performance and customizable in the world.

To evaluate the system performance, we compared the sorting process time of the system with single thread of Intel Core i7-4770 operating at 3.4GHz when they sort 256M 32-bits integer elements. The more the number of ways is increased, the more the number of elements to be sorted in the merge sorter tree is increased. This means that the number of required Phases to fully sort data sequences is decreased, and then the sorting performance becomes higher. Hence, in 4-way, 8-way, and 16-way, 16-way achieved the highest sorting performance, which is 4.07x and 3.24x compared with merge sort and quick sort respectively.

Besides, the more the merge sorter tree is duplicated, the higher the sorting performance is. As a result, 4-way/4-parallel achieved 3.21x performance compared with 4-way. This is because the duplicated trees sort data sequences in parallel. This means that the sorting logic throughput is improved by the duplication of the tree. However, note that if the number of duplicated trees is increased, it is crucial to consider the memory bandwidth usage.

Most importantly, the hardware resource usage is almost same if the total number of Input Buffers is same, and if same hardware resource usage, the hardware that has more duplicated trees can achieve higher performance. The slight differences of the hardware resource usage are due to duplicated control logics and differences of the number of the Output Buffers. In this paper, 8-way/8-parallel configuration results in the highest performance, which is 10.06x and 8.01x compared with merge sort and quick sort respectively.

Our future work is to realize higher performance hardware than the prior work. To do this, we have to develop the merge sorter tree that can handle multiple values at high operating frequency, and to look for approaches which can improve the memory bandwidth usage while keeping the operating frequency high.

REFERENCES

[1] G. E. Moore, "Readings in computer architecture," M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, ch. Cramming More Components Onto Integrated Circuits, pp. 56–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=333067.333074

[2] "Intel Xeon Phi," http://www.intel.com/.

[3] "TILE-Mx," http://www.tilera.com/.

[4] "MPPA MANYCORE MPPA 256," http://www.kalray.com/.

[5] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, G. Jan, G. Michael, H. S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Yi, and X. D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665678

[6] K. Sano, Y. Hatsuda, and S. Yamamoto, "Multi-fpga accelerator for scalable stencil computation with constant memory bandwidth," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 3, pp. 695–705, March 2014.

[7] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on fpgas," *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1007/s00778-011-0232-z

[8] V. Sklyarov and I. Skliarova, "High-performance implementation of regular and easily scalable sorting networks on an fpga," *Microprocess. Microsyst.*, vol. 38, no. 5, pp. 470–484, Jul. 2014. [Online]. Available: http://dx.doi.org/10.1016/j.micpro.2014.03.003

[9] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 240–249. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689068

[10] D. Koch and J. Torresen, "Fpgasort: A high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting," in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 45–54. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950427

[11] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, ser. FPGA '14. New York, NY, USA: ACM, 2014, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/2554688.2554787

[12] "FACE," https://github.com/monotone-RK/FACE.

[13] "Xilinx Virtex-7 FPGA VC707 Evaluation Kit," http://www.xilinx.com/products/boards-and-kits/ek-v7-vc707-g.html.

[14] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[15] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968, pp. 307–314. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468121

[16] G. Marsaglia, "Xorshift rngs," *Journal of Statistical Software*, vol. 8, no. 14, pp. 1–6, 7 2003. [Online]. Available: http://www.jstatsoft.org/v08/i14

[17] T. Usui, R. Kobayashi, and K. Kise, "A challenge of portable and high-speed fpga accelerator," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, K. Sano, D. Soudris, M. Hubner, and P. C. Diniz, Eds. Springer International Publishing, 2015, vol. 9040, pp. 383–392. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16214-0_34

[18] "Memory Interface Generator (MIG)," http://www.xilinx.com/products/intellectual-property/mig.html.

[19] "Vivado Design Suite," http://www.xilinx.com/products/design-tools/vivado.html.

[20] "Nexys4 DDR Artix-7 FPGA Board," https://www.digilentinc.com/.

[21] "Xilinx Kintex-7 FPGA KC705 Evaluation Kit," http://www.xilinx.com/products/boards-and-kits/ek-k7-kc705-g.html.