PAPER

A High Performance FPGA-Based Sorting Accelerator with a Data Compression Mechanism

Ryohei KOBAYASHI[†a)] ***and*** **Kenji KISE**[††b)], ***Members***

SUMMARY Sorting is an extremely important computation kernel that has been accelerated in a lot of fields such as databases, image processing, and genome analysis. Given that advent of Internet of Things (IoT) era due to mobile technology progressions, the future needs a sorting method that is available on any environment, such as not only high performance systems like servers but also low computational performance machines like embedded systems. In this paper, we present an FPGA-based sorting accelerator combining *Sorting Network* and *Merge Sorter Tree*, which is customizable by means of tuning design parameters. The proposed FPGA accelerator sorts data sent from a host PC via the PCIe bus, and sends back the fully sorted data sequence to it. We also present a detailed analytical model that accurately estimates the sorting performance. Due to these characteristics, designers can know how fast a developed sorting hardware is in advance and can implement the best one to fulfill the cost and performance constraints. Our experiments show that the proposed hardware achieves up to 19.5x sorting performance, compared with Intel Core i7-3770K operating at 3.50GHz, when sorting 256M 32-bits integer elements. However, this result is limited because of insufficient memory bandwidth. To overcome this problem, we propose a *data compression mechanism* and the experimental result shows that the sorting hardware with it achieves almost 90% of the estimated performance, while the hardware without it does about 60%. In order to allow every designer to easily and freely use this accelerator, the RTL source code is released as open-source hardware.

key words: *sorting, hardware accelerator, data compression, open source*

1. Introduction

Sorting is one of the most fundamental computation kernels in data management, and lots of approaches to accelerate the kernel have been proposed [1]–[8]. These approaches offer significant results, but mostly these studies utilize SIMD instructions of Intel processors [1], [7], [8] to exploit data-level parallelism or experiment on rich hardware environments such as supercomputers [5] or clusters [7]. It is unclear that these approaches are available on low computational performance machines like embedded systems. Besides, Internet of Things (IoT) era is about to seriously begin due to mobile technology progressions, and large amounts of information are more and more generated from mobile devices, wireless sensors, and others. Therefore, the future needs a sorting method that is available on any environment from embedded systems to high performance systems like servers.

To address the problem, we propose an FPGA-based sorting hardware called FACE [9], which combines *Sorting Network* and *Merge Sorter Tree*. The proposed sorting hardware is customizable by means of tuning design parameters, and we also provide an analytical model that accurately estimates the sorting performance depending on the hardware configuration. In other words, due to these characteristics designers can estimate sorting accelerator performance in advance and can implement the best one to meet constraints of the cost and performance. We have presented the basic concept of the proposed sorting accelerator in [9], and in this paper we summarize the proposed sorting accelerator in terms of the design, the implementation, and the evaluation of the sorting performance and the hardware resource usage. To allow every designer to easily and freely use this accelerator, the Register Transfer Level (RTL) source code is available at [10].

Our proposed sorting accelerator can be high performance by tuning design parameters, in that case, not only the hardware resource usage but also the memory bandwidth has to be considered. In fact, the highest performance configuration in [9] suffers from the memory bandwidth limitation. To address this problem, in this paper we propose a *data compression mechanism* for the sorting accelerator. Among lots of data compression algorithms, we use an algorithm using the relative difference between values of continuous locations, which is based on [11]. As sorting is proceeded, the relative difference between them becomes smaller. This means that the algorithm is quite suitable for sorting. Besides, the algorithm can be implemented by a simple vector subtraction and addition. That is why we introduce this algorithm, and the data compression mechanism can alleviate the memory bandwidth limitation while keeping the operating frequency high.

Our contributions in this work are:

- We propose a high performance and customizable sorting accelerator with two sorting architectures, and also propose a detailed analytical model. Therefore, designers can estimate sorting accelerator performance in advance and can implement the best one to fulfill constraints of the cost and performance.
- To mitigate the bandwidth limitation of accessing the off-chip memory, we propose a data compression mechanism for the sorting accelerator. Experimental results show that the sorting accelerator with the mechanism offers better performance than without it.

Manuscript received September 6, 2016.
Manuscript publicized January 30, 2017.

[†] The author is with University of Tsukuba, Tsukuba-shi, 305–3577 Japan.
[††] The author is with Tokyo Institute of Technology, Tokyo, 152–8552 Japan.
a) E-mail: kobayashi@cs.tsukuba.ac.jp
b) E-mail: kise@cs.titech.ac.jp

DOI: 10.1587/transinf.2016EDP7383

- 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 in the world that is high performance, is customizable, and addresses the memory bandwidth limitation.

This paper is organized as follows. We describe the sorting architectures that we use in Sect. 2. In Sect. 3, the design and the analytical model of the proposed sorting hardware are detailed, and we explain the data compression mechanism for the proposed sorting accelerator in Sect. 4. Section 5 shows the implementation of the sorting hardware, the sorting performance and the hardware resource usage with and without the data compression mechanism, and we discuss these experimental results. In Sect. 6, we present several related studies, and we finally conclude this paper in Sect. 7.

2. Sorting Architectures

Our proposed sorting accelerator takes advantage of the sorting network and the merge sorter tree. We describe these sorting architectures. In order to avoid ambiguity, all sorters described in this paper target at the ascending order and 32-bits integer elements.

2.1 Sorting Network

A sorting network [12] 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 [13]–[15].

Figure 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 [13], 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.

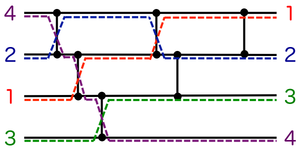


Fig. 1 Bubble sort network with 4-inputs and 4-outputs

Figure 2 shows Batcher's even-odd merge sort network [16] 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 [13]. This network is embedded in our proposed hardware.

2.2 Merge Sorter Tree

2.2.1 Overview

The merge sorter tree [17] offers 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.

Figure 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 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 2, and 1. Then, 2 and 1 are compared. The data sequences in the left-

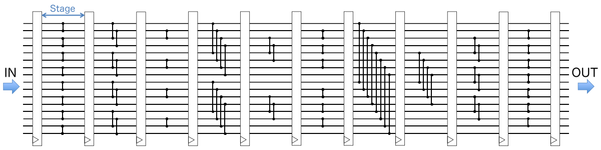


Fig. 2 Pipelined synchronous Batcher's odd-even merge sort network with 16-inputs and 16-outputs

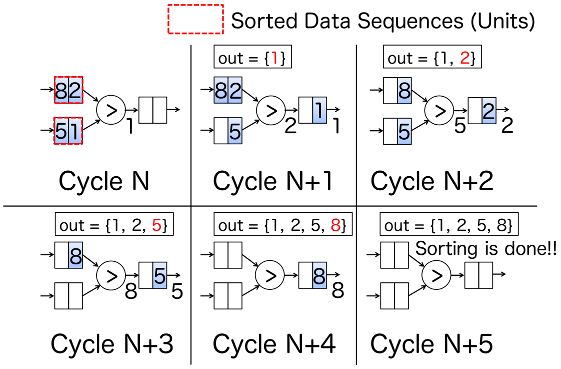
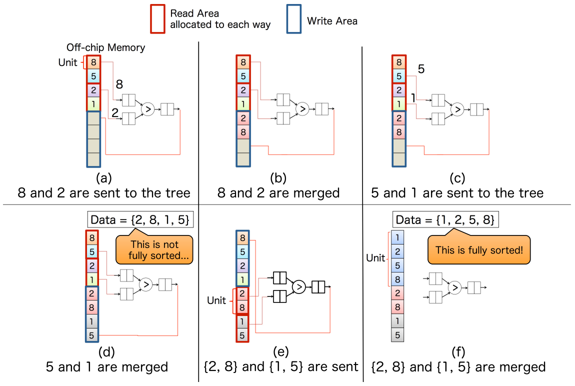


Fig. 3 Sorting process in merge sorter tree



(a) 8 and 2 are sent to the tree (b) 8 and 2 are merged (c) 5 and 1 are sent to the tree

(d) 5 and 1 are merged (e) {2, 8} and {1, 5} are sent (f) {2, 8} and {1, 5} are merged

Fig. 4 The merge sorter tree and off-chip memory. The tree sorts the initial data sequence {8, 5, 2, 1} by using the memory.

most FIFOs must be sorted. We define these data sequences as **Units**. In this example, the sorted element 8 and 2 in the upper FIFO is a Unit, and the element 5 and 1 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, 1 is emitted from the root. At the same time, 2 and 5 are compared, and then the sorter cell outputs 2. At Cycle N+2, 2 is emitted from the root. At the same time, the sorter cell outputs 5 depending on the comparison result between 8 and 5.

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 1, 2, 5, and 8. If k-way merge sorter tree executes this process, the tree can merge k Units and generate a larger Unit.

However, if the number of the Units to be merged is more than k, the data sequence passed through the tree is not fully sorted yet. If so, the tree uses a buffer like off-chip memory in order to store the data sequence. We explain this using a simple example shown in Fig. 4.

In Fig. 4, the off-chip memory is divided into two areas, which are Read Area and Write Area. Read Area is used to hold the data sequence that is sent to the merge sorter tree. If k-way merge sorter tree is used, this area is further divided into k areas and each divided area is allocated to each way. Here k is 2, therefore Read Area is divided into two areas. Write Area is used to buffer the data sequence emitted from the tree.

In Fig. 4 (a), Read Area has an unsorted data sequence, which is {8, 5, 2, 1}. This data sequence consists of 4 Units shown in Fig. 4 (a). As mentioned above, Read Area is divided into two areas. Hence, one has {8, 5}, the other has {2, 1}. First, 8 and 2 are sent to each way. These elements are merged into one Unit, and then the Unit {2, 8} is written into the head of the Write Area, as shown in Fig. 4 (b).

After that, in Fig. 4 (c), 5 and 1 are sent and merged. Then, as shown in Fig. 4 (d), the Unit {1, 5} is stored in Write area. This means that the entire data sequence in Read Area is passed through the merge sorter tree and stored in Write Area. As shown in Fig. 4 (d), the data sequence {2, 8, 1, 5} in Write Area is not fully sorted, and has to be passed through the tree again. That is why if the number of the Units to be merged is more than k, the data sequence passed through the tree is not fully sorted yet.

In Fig. 4 (e) the 2 Units, {2, 8} and {1, 5}, are sent to the ways and are merged in the tree. The operation of the tree is same as Fig. 3. As shown in Fig. 4 (f), the data sequence emitted from the tree is fully sorted, which is {1, 2, 5, 8}.

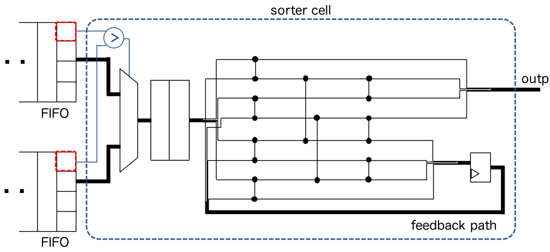


Fig. 5 A sorter cell that can emit four elements per cycle

2.2.2 Enhancement of the Merge Sorter Tree

The merge sorter tree takes N cycles to emit N elements, because the sorter cell can emit only one element. In order to improve the tree throughput, it is important for the cell to be able to handle multiple elements.

Figure 5 shows a logical overview of how a sorter cell emits four sorted elements at a cycle [18]. Similar to Sect. 2.2.1, the data sequences in the both FIFOs must be sorted. Each cycle, the smallest element of each input are compared and four elements with the smaller smallest element are ejected from the input FIFO. These four elements are passed through the multiplexer and the internal FIFO, and then are merged with the largest four elements, which are from the previous cycle, at the network in the cell. The four smallest elements resulting from the merge are guaranteed to be smaller than any other element yet to be handled, because any elements smaller than the four elements have already been ejected. However, the largest four values **may be** larger than and therefore must be fed back and merged with the next set of input elements. In this way, four sorted elements are emitted and four elements are ejected from one of the input FIFOs each cycle.

3. Proposed Sorting Accelerator

3.1 Data Path

Figure 6 shows a data path of the baseline sorting accelerator. We implement it on an FPGA, and verify that it accurately works by using a host PC. We explain how the hardware sorts data sequences using Fig. 7. For simplicity, the initial data sequence of 256 elements is a reverse-order data sequence from 256 to 1.

At first, the host PC generates the initial data sequence and sends it to the FPGA via the PCI Express (PCIe) bus, and the data received in the FPGA is passed through Sorting

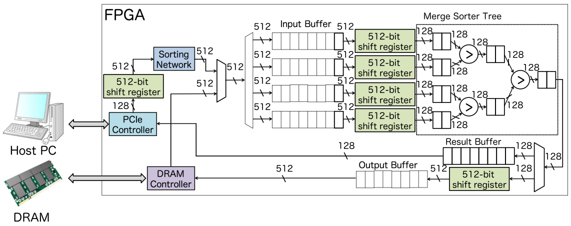


Fig. 6 Data path of the baseline sorting accelerator

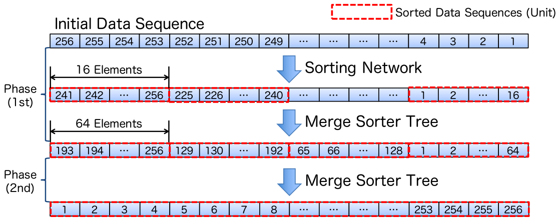


Fig. 7 Example: sorting 256 elements from 256 to 1

Network after packed into a 512-bits data. This network is Batcher's even-odd merge sort network [16] with 16-inputs and 16-outputs, which 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. 7.

The data sequence passed through Sorting Network is stored in Input Buffer that consists of FIFO. The stored elements must already be sorted, and that is why the network is used. 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 with the sorter cell that can emit 4 sorted elements in Fig. 6. Every sorter cell works in parallel and ejected elements from it are enqueued into an output FIFO each cycle, and eventually the merged data sequence is emitted from the root of the 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 tree is sent back to the host PC if it is fully sorted. However, as shown in Fig. 7, this process is not done yet and it has to be passed through the merge sorter tree again as described in Sect. 2.2.1. Therefore, the emitted data sequence 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.

To complete the sorting process, the stored data is load from the off-chip memory and is sent to the merge sorter 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. Therefore, the data sequence is streamed into Result Buffer and is sent back to the host PC. To verify the result, the received data sequence is checked by using typical sorting software.

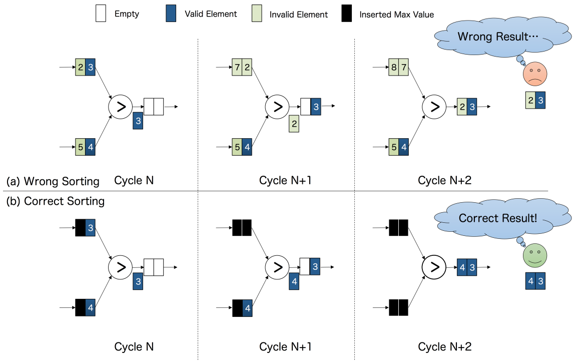


Fig. 8 Wrong sorting and correct sorting

By passing the data sequence through Sorting Network and Merge Sorter Tree, 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 $\log_{\#\text{ of ways}} \frac{\#\text{ of elements}}{S}$ where S is the number of sorted elements at Sorting Network in the first Phase. For instance, in Fig. 7 the number of required Phases is 2, because S and the number of ways and elements to be sorted are 16, 4, and 256 respectively.

3.2 Control Logic

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. 8 (a), this example is demonstrated. Figure 8 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 proposed that the maximum value, which depends on the bit width of elements, is inserted after Valid elements [19]. Figure 8 (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. 8 (b)), a circuit that generates the maximum value to separate Units is implemented in Input Buffer as shown in Fig. 9. 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.

Output Buffer also has a counter, which counts the

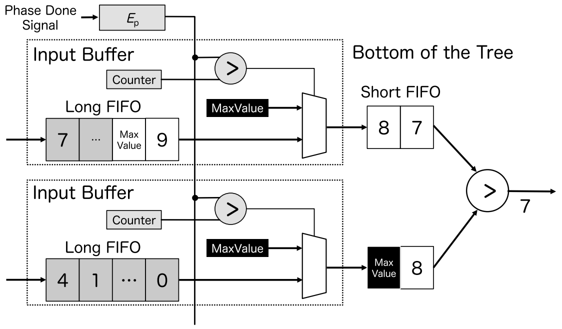


Fig. 9 Two input buffers and one sorter cell

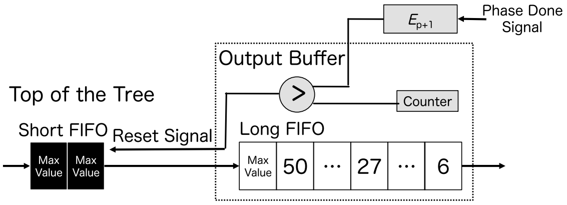


Fig. 10 How to generate reset signal from output buffer

number of stored elements in Output Buffer as shown in Fig. 10. 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. 7, 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.

3.3 Performance Model

In this section, in order to accurately analyze 100% performance of the proposed sorting accelerator itself without any effect like memory latency or bandwidth, we assume an ideal external memory whose bandwidth and latency are infinity and 1 cycle respectively. The bandwidth and latency come from FPGA internal memory performance like Block RAM. The sorting performance can be estimated by calculation of the number of required cycles to complete the entire process. This can be calculated by summation of the number of cycles in each Phase.

As described in Sect. 3.1 and Sect. 3.2, multiple Units are merged into one Unit in the merge sorter tree. We call this process **Iteration**. In Fig. 7, 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

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

where S 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 (Sect. 3.2). Therefore, a few cycles overhead exists between each Iteration. This overhead $\mathrm{OH}_{\mathrm{iter}}$ is given by

$$\mathrm{OH}_{\mathrm{iter}} = 3\log_2 k + 1 \tag{2}$$

because at the beginning of each Iteration, it takes three cycles to pass through a sorter cell handling multiple elements.

The beginning of each Phase also has an overhead. The merge sorter tree cannot sort data sequences unless elements 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 $\mathrm{OH}_{\mathrm{phase}}$ is given by

$$\mathrm{OH}_{\mathrm{phase}} = k\alpha \tag{3}$$

where α is tens of cycles at most.

We define the number of elements emitted from the tree at a cycle as M. The number of required cycles for nth Phase, C_n, is given by the following formula.

$$C_n = \frac{N}{M} + I_n \times \mathrm{OH}_{\mathrm{iter}} + \mathrm{OH}_{\mathrm{phase}} \tag{4}$$

We explain this formula in three parts. First, the throughput of the merge sorter tree is M element per cycle. Thus, it takes $\frac{N}{M}$ 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 $I_n \times \mathrm{OH}_{\mathrm{iter}}$, because $\mathrm{OH}_{\mathrm{iter}}$ cycles overhead exists between each Iteration. Third, the beginning of each Phase has $\mathrm{OH}_{\mathrm{phase}}$ 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

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

where n is the number of required Phases. As described in Sect. 3.1, the number of required Phases to fully sort the data sequence is $\log_k \frac{N}{S}$. In other words, C_{fully} can be also given by the following formula.

$$C_{\mathrm{fully}} = \sum_{i=1}^{\log_k \frac{N}{S}} \{\frac{N}{M} + \frac{N}{S\,k^i}(3\log_2 k + 1) + k\alpha\} \tag{6}$$

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

3.4 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.

Figure 11 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 analytical model described in Sect. 3.3, 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

$$C_{\mathrm{last}} = \frac{N}{M} + 1 \times \mathrm{OH}_{\mathrm{iter}} + \mathrm{OH}_{\mathrm{phase}} \tag{7}$$

where the number of Iterations for the last Phase is definitely one. Therefore, $C_{\mathrm{fully\_dup}}$, 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.

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

Hence, the number of required total cycles is estimated, depending on the number of ways, duplicated trees, elements that the sorting network can handle, elements to be emitted from the tree at a cycle, 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 these configuration parameters.

As mentioned before, the higher the sorting logic throughput is, the higher performance the accelerator achieves. However, as the sorting logic throughput is higher, the sorting performance becomes sensitive to the memory bandwidth. This means that the memory bandwidth becomes the performance bottleneck. Therefore, it is truly important to consider approaches which can alleviate the memory bandwidth limitation while keeping the operating frequency high. We present an effective way to realize this in the next section.

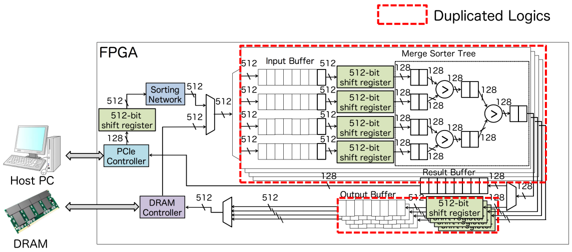


Fig. 11 Data path of the proposed sorting accelerator with the duplicated merge sorter trees

4. Data Compression for the Sorting Accelerator

4.1 Algorithm

To mitigate the bandwidth limitation of accessing the off-chip memory, we adopt *data compression*. Data compression has been successfully adopted in a number of different contexts in modern computer systems as a way to conserve storage capacity and/or data bandwidth (e.g., downloading compressed files over the Internet or compressing off-chip memory) for several decades. Many data compression algorithms are proposed in previous studies, and it is necessary to decide the most appropriate algorithm according to data types, applications, and hardware.

In general, data compression algorithms take advantage of redundancy in the data used by applications [20]–[22]. However, the data handled in sorting is generally random, and there is little redundancy in the data used by the application. Therefore, such algorithms like [20]–[22] are not effective against sorting. Well then, which algorithm is promising for the application?

We focus on a data compression algorithm based on [11], which uses the relative difference between values of continuous locations. As sorting is proceeded, the relative difference between them becomes smaller. That is why the algorithm is quite suitable for sorting. The data compressed by the algorithm is represented in a compact form using a common *base* value and an array of relative differences (*deltas*).

Figure 12 shows the example diagram of the compression and decompression method described in [11]. As shown in Fig. 12, the compression and decompression method can be implemented by a simple vector subtraction and addition. In Fig. 12 (a), the compressed data is represented in Base V_0 and the array of $\Delta_1 \sim \Delta_3$, using 7Bytes

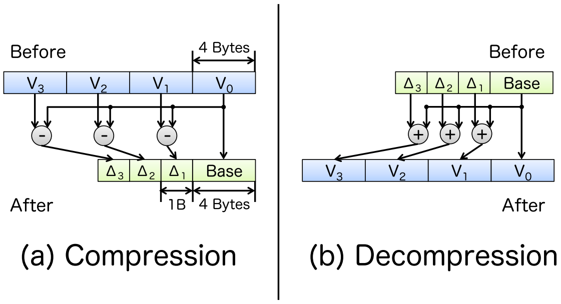


Fig. 12 Example of the compressor and decompressor method that is described in [11]. In this example, 4Bytes values are compressed into a 4Bytes base and an array of 1Byte Δ.

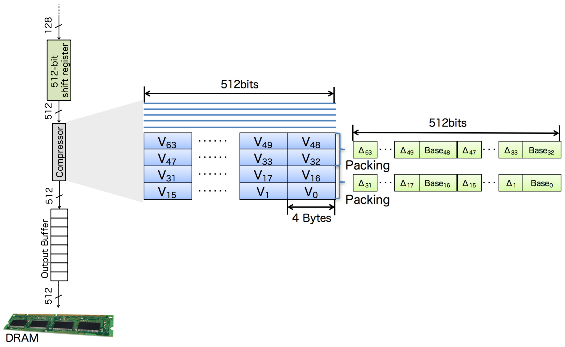


Fig. 13 Adoption of the data compression

instead of 16Bytes. This results in saving 9Bytes of the originally used space. The compressed data can be easily decompressed by the addition of each delta to Base shown in Fig. 12 (b).

4.2 Adoption of the Data Compression against the Proposed Sorting Accelerator

Figure 13 shows the adoption of the data compression against the proposed sorting accelerator. As described in Sect. 3.1, 512-bit shift register packs 32-bits elements emitted from the root of the merge sorter tree into a 512-bits data. In Fig. 13 if two compressible 512-bits data are successive, the two data are packed into a 512-bits data. For instance, if all 512-bits data packed by the shift register are compressible, the data amount transferred to the external memory is a half of the original one. This means that it is possible to theoretically obtain double performance efficiency if the memory bandwidth is the performance bottleneck. In other words, it is possible to estimate the performance improvement ratio by calculation of the compression ratio against all 512-bits data packed by the shift register under such situation. We define the process to pack two compressible 512-bits data into a 512-bits data as 2x compression if the two data are successive. The 2x compression ratio is given by

$$\text{2xCompRatio} = 1.0 + (2.0 - 1.0) \times \text{incidence} \qquad (9)$$

where “incidence” is the occurrence rate of consecutive two compressible 512-bits data against all 512-bits data packed by the shift register.

4.3 Data Path

Figure 14 shows the data path of the proposed sorting accelerator with the compressor and decompressor. The compressor packs two compressible 512-bits data into a 512-bits data like Fig. 15, and the decompressor unpacks compressed data read from the external memory.

The encoding format for the 2x compression consists of four parts, which are Base, Compressed, Void, and Flag shown in Fig. 15. Base and Compressed represent a base value and an array of deltas. The region of Base and Compressed stands for a compressed data shown in Fig. 13. The data emitted from the shift register consists of 16 sorted elements. For instance, in Fig. 13 the 16 elements from V_0 to V_{15} are sorted, and V_0 is the smallest and V_{15} is the largest in the 16 elements. The most simple way is to choose the smallest one as Base and is that the other 15 elements are converted into the 15 13-bits deltas if Δ_{ls}, which is a difference between the largest and the smallest, ≤ 0x1fff. And then, if a subsequent 512-bits data is also compressible, the compressor converts the two original data into a 512-bits data shown in Fig. 15.

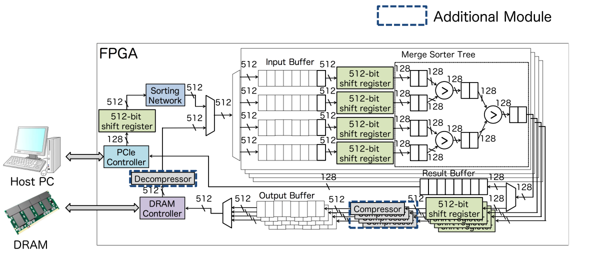


Fig. 14 Data path of the proposed sorting accelerator with the compressor and decompressor

Name	***Description***
Base	A base value
Compressed	An array of 15 deltas. The delta size is 13bits.
Void	Unused space
Flag	The marker to check whether or not this data is compressed. The marker is 0x0000_0000_1

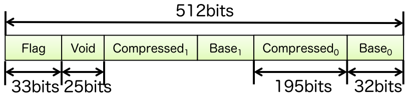


Fig. 15 The encoding format for the 2x compression

On this occasion, Flag is set to 0x0000_0000_1 in order to identify that the 512-bits data is encoded. Here, we explain why the flag needs 33 bits. It is assumed that the flag is the Most Significant Bit (MSB) of a 512-bits data, which means that ‘1’ stands for that the 512-bits data is compressed. For instance, if a 512-bits data, which is composed of 0xffff_ffff, 0x0000_000e, 0x0000_000d, 0x0000_000c, . . . , and 0x0000_0000, is emitted from the shift register, the compressor cannot compress the data because the delta is beyond 0x1fff. If the compressor cannot convert two original data into a 512-bits data, this module outputs the two original data one by one. However, because of the MSB value, the decompressor identifies that the data is compressed despite the fact that the data is NOT compressed, and generates incorrect data. In order to prevent this and to deal with all data patterns, we present a usage of a flag 0x0000_0000_1, which requires 33 bits from the MSB. The region corresponding to Flag of the two original data blocks is NEVER 0x0000_0000_1, because the elements in the two data are sorted in descending order shown in Fig. 13. Due to this, the decompressor can identify whether or not the data read from the external memory is encoded by

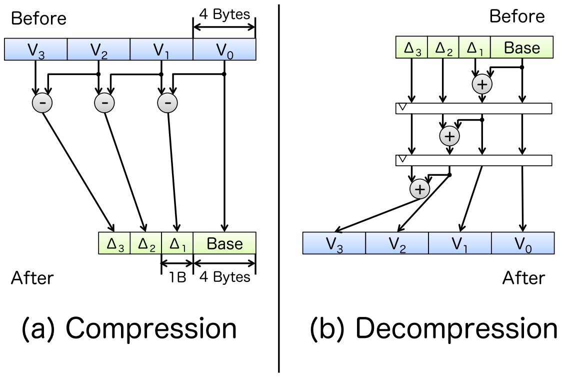


Fig. 16 Modified compression and decompression designs based on the prior work. In this diagram, 4Bytes values are compressed into a 4Bytes base and an array of 1Byte Δ.

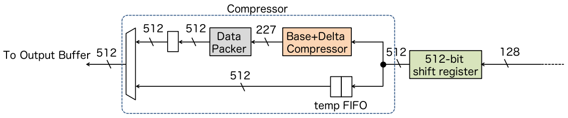


Fig. 17 Data path of the compressor

checking the region. We define the remained part in the encoding format as Void. In this paper, Void is unused space.

If a gap between the largest and the smallest is too large, the simple way to generate Compressed cannot efficiently work. To address this problem, we propose modified compression and decompression designs based on the prior approach, which is shown in Fig. 16. Unlike Fig. 12 (a), the compression design subtracts the neighbor value from each value in order to generate the array of $\Delta_1 \sim \Delta_3$. Due to this, in the 16 elements from V_0 to V_{15} shown in Fig. 13, V_0 is Base and the other 15 elements are converted into the 15 13-bits deltas if all deltas ≤ 0x1fff. Because each delta is smaller than Δ_{ls}, this approach has more likelihood to generate Compressed than the simple way. In the decompression, the pipelined addition is executed unlike the simple vector addition shown in Fig. 12 (b). Although the decompression design can be implemented as a purely combinational circuit, this probably causes the operating frequency reduction due to the large delay. That is why we choose the pipelined design like the sorting network described in Sect. 2.1.

Figure 17 shows the data path of the compressor. This module consists of three components, which are Base+Delta Compressor, Data Packer, and temp FIFO. Base+Delta Compressor executes a simple vector subtraction to compress a 512-bit data emitted from the 512-bit shift register. At the same time, the original data is stored in temp FIFO. If the original data is not compressible, temp FIFO outputs all stored data immediately. Data Packer generates a formatted 512-bits data shown in Fig. 15 if two compressible 512-bits data are successive. On this occasion, temp FIFO is reset.

Figure 18 shows the data path of the decompressor. This module has three components, which are FIFO, temp FIFO, and Base+Delta Decompressor. The FIFO consists of internal memory resources (hard macros) of the FPGA, and the data read from the external memory is stored in the component. The stored data in the FIFO is sent to temp FIFO, and then this component keeps holding the data unless the dequeue signal is asserted. If the data is compressed one, the data splits into two parts, and then Base+Delta Decompressor picks up and decompresses the parts one by one, at the same time the dequeue signal is asserted. If not, the data is sent to Input Buffer directly, and the dequeue is done simultaneously. As mentioned before, the decompressor can identify whether or not the data is compressed by checking Flag shown in Fig. 15.

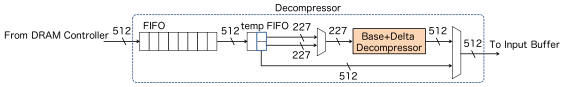


Fig. 18 Data path of the decompressor

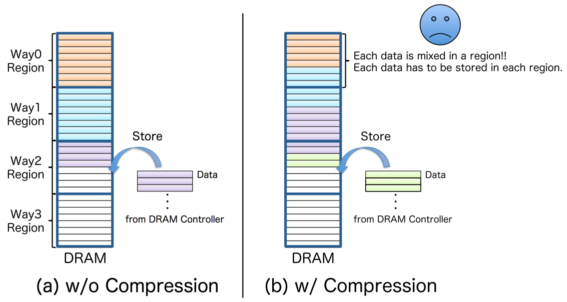


Fig. 19 The data emitted from the merge sorter tree is sequentially written into the head of the Write Area, without (a) and with (b) the data compression.

4.4 Control Logic

As described in Sect. 2.2.1, the data emitted from the merge sorter tree is sequentially written into the head of the Write Area like Fig. 19 (a). The figure shows that the data emitted from 4-way merge sorter tree is written into the external memory via DRAM Controller. The buffered data is sent to each way of the tree in the next Phase. Without the data compression in Fig. 19 (a), the data sent to each way in the next Phase is correctly written into each region of the memory by tuning data size of each way and grain size of data written into the memory. With the data compression in Fig. 19 (b), simple sequential write can mix each region data that should be sent to each way, because each region data can be non-uniform due to the data compression. In that case, sorting cannot be accurately performed because incorrect data is sent to each way.

To address this problem, we present a mechanism named *Throttling*, which tunes grain size of the data written into the external memory. Figure 20 shows the overview of Throttling. DRAM Controller writes the data emitted from the tree into the head of the Write Area. At the last of completion of writing data sent to a way in the next Phase, DRAM Controller gradually throttles the grain size. By doing this, emitted data is correctly written into a correspond-

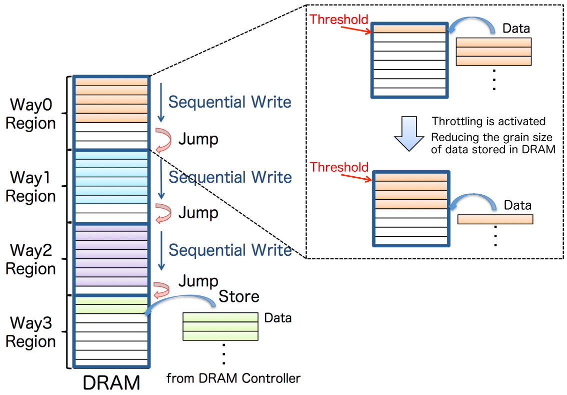


Fig. 20 Writing the data emitted from the merge sorter tree by Throttling

ing region without mixing each region data. The flag to alert that writing data almost finishes is asserted when the writing data address crosses Threshold of a region.

After writing data into a corresponding region finishes, DRAM Controller sets the writing address as the head of the next region and then sequentially writes the data emitted from the tree. By repeating this process, all of emitted data is correctly written into the external memory. When setting the writing address as the head of the next region, the writing address is preserved. The address is used as a pointer to identify how much data each way should read in the next Phase.

5. Evaluation

5.1 Implementation

As a platform for the proposed FPGA accelerator, we use the Xilinx Virtex-7 FPGA VC707 evaluation kit [23]. This kit has the Virtex-7 XC7VX485T, 1GB DDR3 SO-DIMM (800MHz/1600Mbps) memory, and PCIe Gen2 x8 connector. We replace that memory with 4GB DDR3 SO-DIMM memory in order to sort larger data sequences. This kit is connected to a desktop PC with Intel Core i7-3770K operating at 3.50GHz and 16GB DDR3-1600 memory via the PCIe slot.

The sorting logic is implemented in Verilog HDL. To implement DRAM and PCIe Controller, we use an IP core provided by Xilinx [24] and an open-source framework for communicating data using Direct Memory Access (DMA) transfers between the host PC and FPGA via the PCIe bus, which is called RIFFA [25]. The data widths from the sorting logic to the both controllers are 512 bits and 128 bits as shown in Fig. 6, and we set S and M to 16 and 4 respectively.

As a synthesis tool, we use Vivado 2015.4 [26]. The placed and routed logic meets all timing constraints, and all implemented logics on the FPGA operate at 200MHz. We measured effective memory and PCIe bandwidths, and they are 7.58GB/s (the harmonic mean of the read and write memory bandwidths) and 3.20GB/s (both of from the host PC/FPGA) respectively.

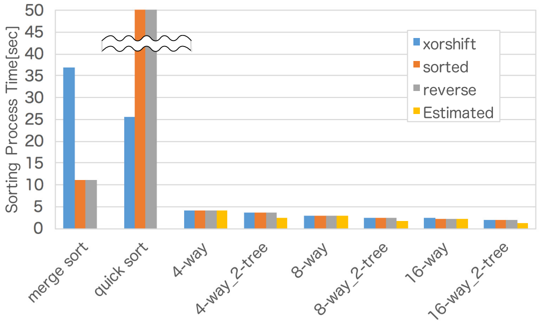


Fig. 21 Sorting performance comparison between the software and the proposed sorting accelerator

5.2 Sorting Performance without Data Compression

We run our experiments on the host PC with the proposed hardware sorter without the data compression mechanism. The data-sequence size is 256M (M = 2^{20}) elements, whose data type is 32-bits integer. As described in Sect. 3.1, the initial data sequence is generated and sent to the FPGA via the PCIe bus by the host PC, and then the proposed accelerator sorts it and sends back the fully sorted data sequence to the host PC. We measured the elapsed time by using gettimeofday. To compare the proposed sorter performance, we implement popular sorting algorithms that are merge sort and quick sort in C language, compile them using gcc 4.8.4 with -Ofast optimization, and run them on the host PC itself. These two sorting are executed as single thread of Intel Core i7-3770K.

Figure 21 shows the experimental results. In the figure, 4-way represents the sorting accelerator with 4-way merge sorter tree and 4-way_2-tree represents the hardware with two 4-way merge sorter trees. xorshift, sorted, and reverse represent that the initial data-sequence types are a random data sequence using Xorshift [27], a sorted data sequence, and a reverse-order sorted data sequence respectively.

As shown in Fig. 21, 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 $O(n^2)$.

In xorshift, the 4-way performance is 8.77x and 6.07x, compared with merge sort and quick sort respectively. The 8-way performance is 12.8x and 8.89x; the 16-way performance is 16.0x and 11.1x than them. 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. Moreover, the sorting accelerator achieves almost 100% performance efficiency. Estimated in Fig. 21 stands for the theoretical performance obtained from the analytical model described in Sect. 3.3. As shown in the figure, these sorting performance are almost same as the theoretical ones. This is because these sorting logic throughputs are not higher than both of the effective memory and PCIe bandwidths. Using the operating frequency F, the sorting logic throughput in nth Phase, T_n, is given by the following formula.

$$T_n = \frac{N \times 4\text{Bytes}}{\frac{C_n}{F}} \tag{10}$$

According to this formula, T_1 of 4-way, 8-way, and 16-way are 2.23GB/s, 2.44GB/s, and 2.66GB/s respectively, and in the last Phase their throughputs are 3.20GB/s. The sorting logic throughput against the external memory in nth Phase except the first and last Phases is given by $2T_n$ where the constant 2 comes from DRAM read and write. Therefore, it can be seen that the sorting logic throughputs of 4-way, 8-way, and 16-way exceed neither of the both bandwidths.

As described before, the more the merge sorter tree is duplicated, the higher the sorting performance is, since the sorting logic throughput is improved. For instance, the 4-way_2-tree performance is 1.15x and the 8-way_2-tree performance is 1.17x and the 16-way_2-tree performance is 1.22x by compared with the performance of 4-way, 8-way, and 16-way. The sorting accelerator with 16-way offers 19.5x and 13.5x performance compared with merge sort and quick sort, but the tree duplication faces the memory and PCIe bandwidth limitation. The sorting throughput using the duplication technique in nth Phase is given by PT_n, and those of 4-way_2-tree, 8-way_2-tree, and 16-way_2-tree in the first Phase are 4.45GB/s, 4.88GB/s, and 5.32GB/s. This means that the performance efficiencies in the first Phase are limited by the PCIe bandwidth. On the other hand, their throughputs in the last Phase are 3.20GB/s, which means that the sorting process and the PCIe data transfer in the last Phase can be overlapped. This is because single tree operates only in the last Phase. We modify the analytical model described in Sect. 3.4 so that the sorting performance including effect of the PCIe bandwidth can be estimated. The sorting process time, Time, can be estimated the following formula.

$$\text{Time} = \frac{N \times 4\text{Bytes}}{3.2\text{GB/s}} + \Big(\sum_{i=2}^{(\log_k \frac{N}{S})-1} \frac{C_i}{P} + C_{\text{last}}\Big) \times \frac{1}{F} \tag{11}$$

We draw the estimated performance of the three configurations in Fig. 21. The performance efficiencies of 4-way_2-tree, 8-way_2-tree, and 16-way_2-tree are 44.7%, 54.4%, and 58.2% respectively. These relatively low performance results are due to the memory bandwidth limitation. In the next section, we evaluate how the proposed data compression mechanism mitigates it.

Figure 22 shows the hardware resource usage of the sorting accelerator. In Fig. 22, FF, LUT Logic, 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. FF is used for the control logics of the sorting accelerator, LUT Logic and RAM are mostly for the merge sorter tree, and Block RAM is used to implement Input Buffer and Output Buffer. The Block RAM usage can be mitigated by tuning the number of FIFO entries of Input Buffer and Output Buffer.

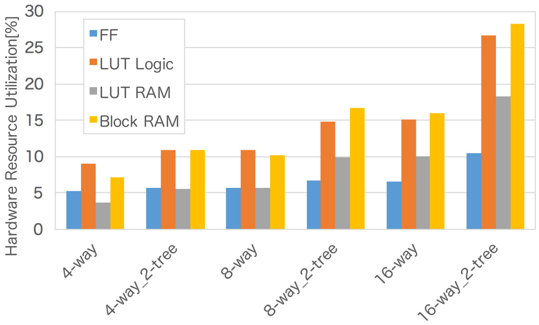


Fig. 22 Hardware resource usage of the proposed sorting accelerator

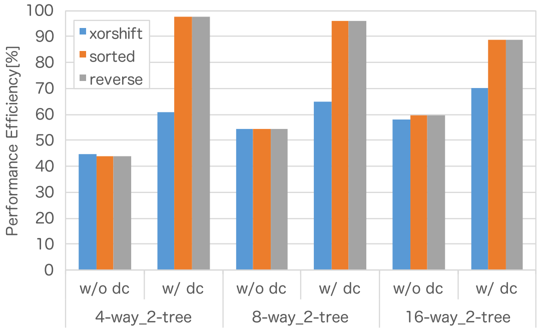


Fig. 23 The performance efficiency of the sorting accelerator with and without the data compression mechanism

5.3 Sorting Performance with Data Compression

Figure 23 shows that performance efficiencies of the sorting accelerator with and without the data compression mechanism. The data set is same as Sect. 5.2. Without it, as the number of ways is larger, the performance efficiency becomes higher and gets close to the ratio of the memory bandwidth to the average memory bandwidth required from the sorting accelerator that is the harmonic mean of the sorting logic throughput against the external memory among Phases except the first and last. This is because the sorting accelerator with a fewer ways has to access to the external memory more frequently, which means that it is prone to suffer from the memory bandwidth limitation.

It can be seen that the performance efficiency of the sorting accelerator with the data compression mechanism is improved in all data-sequence types. In sorted and reverse, all data is compressed because each delta is very small. The performance efficiencies of 4-way_2-tree, 8-way_2-tree, and 16-way_2-tree are 97.5%, 95.9%, and 88.2% respectively, because of alleviation of the memory bandwidth limitation. The reason why the efficiencies go down along with the

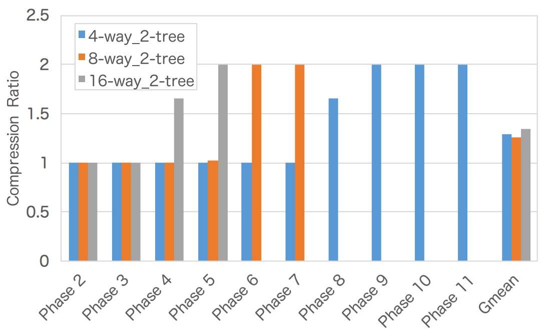


Fig. 24 The data compression ratio in each Phase except the first and last

increased in the number of ways is Throttling overhead. The sorting accelerator with more ways has to do Throttling more frequently, and the process gradually reduces the grain size of the data, which leads to reduction of the memory bandwidth utilization.

In xorshift, the performance efficiencies of them are slightly better, which are 16.1%, 10.7%, 11.9% improvements. In order to investigate the reason why the improvement ratio is low, we implement a software simulator to evaluate the data compression ratio in each Phase. Figure 24 shows that result. The number of required Phases of them are calculated by using the formula described in Sect. 3.1. The compression ratio shown in Fig. 24 is calculated by using the formula described in Sect. 4.2, for instance if all data is compressible, the compression ratio is 2. Those in the first and last Phase are not measured, because the sorting logic throughput is limited by the PCIe bandwidth in the both Phases. Gmean shown in Fig. 24 represents the geometric mean of all compression ratios. As shown in Fig. 24, while no data is compressed in early Phases, the compression ratio is improved as sorting is proceeded. Gmean of 4-way_2-tree, 8-way_2-tree, and 16-way_2-tree are 1.29, 1.26, and 1.35. It is clear that in xorshift the efficiency improvements of them are due to the data compression in the latter half of the Phases. In other words, because the average compression ratio is low, the efficiencies are not improved well. Besides, Throttling is executed in all Phases even if no data is compressed. That is why the efficiency improvement of 16-way_2-tree is relatively low while the average compression ratio is higher than the others.

Figure 25 shows that the hardware resource usage of the sorting accelerator with and without the data compression mechanism. It can be seen that the usage increases of FF, LUT Logic, LUT RAM, and Block RAM are quite small. The increase rates of FF and LUT Logic are due to implementation of control logics like Throttling for the data compression mechanism, and those of LUT RAM and Block RAM depend on buffers of the compressor and decompressor.

5.4 Discussion

We discuss that the proposed sorting accelerator can be implemented on any other FPGA. Figure 26 shows the number of Slices used for the sorting logic itself. A Slice is a term used by Xilinx, and it is a logic component including several LUTs and FFs. We draw three areas in Fig. 26. The down 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 [28]. 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 [29]. If more performance-aware systems are required, they need larger devices like the Virtex-7 FPGAs. As shown in Fig. 26, the designs except 16-way_2-tree with and without the data compression mechanism are within the down 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 open-source hardware. Hence, designers can implement a sorting accelerator composed of required hardware resources by means of tuning the configuration parameters.

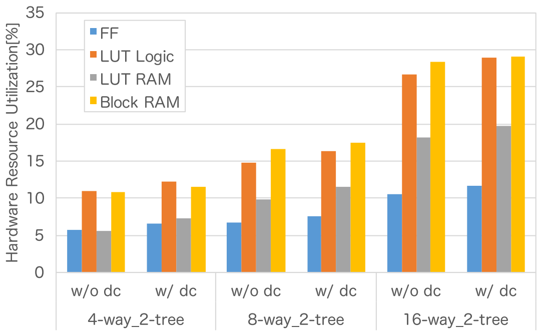


Fig. 25 The hardware resource usage of the sorting accelerator with and without the data compression mechanism

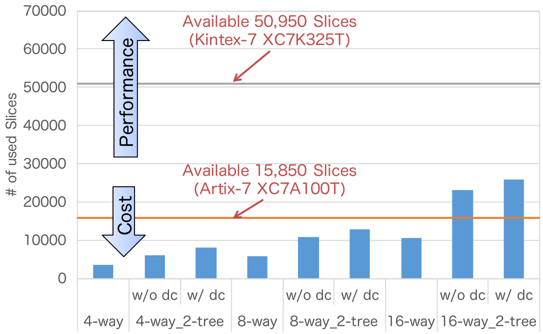


Fig. 26 The number of Slices used for the sorting logic itself

6. Related Work

In recent years, FPGAs have benefited from technology process advances to become significant alternatives to ASICs, and lots of companies and research institutes have been interested in them. Due to the trend, several studies have proposed sorting hardware with FPGAs [13]–[15], [17], [18], [30].

The sorting network is one of the most famous sorting architectures, and most studies focus on it [13]–[15], [18], [30]. In [13]–[15], FPGA-based systems with sorting networks are implemented and evaluated in terms of circuit areas, throughputs, and power consumptions. [30] proposes a Domain Specific Language (DSL) and a compiler to automatically generate sorting networks with optimized throughput and area efficiency. As mentioned before, a sorting network is easy to be implemented in hardware due to simplicity of the architecture, but 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 [13], if the data-sequence size is less than eight, it can be fully sorted only in the sorting network. However, if not, the CPU merges these sorted portions.

In addition to the sorting network, the merge sorter tree is proposed in [17], [18]. In particular, [18] proposes a special merge sorter tree that can handle six elements per cycle, and we also use the special one handling four elements at a cycle. Thanks to this, our proposed accelerator can offer much higher sorting performance than that of the host PC itself.

As mentioned before, as the sorting logic throughput is higher, it is truly important consider approaches which can address the memory bandwidth limitation while keeping the operating frequency high. However, in [18], the proposed hardware has massive memory bandwidth and the authors do not consider that problem. In contrast to [18], we propose a data compression mechanism for the sorting accelerator to mitigate the bandwidth limitation of accessing the off-chip memory. The experimental results show that the sorting accelerator with the mechanism achieves better performance than without it. To the best of our knowledge, no related work proposes data compression mechanisms for sorting hardware and evaluates the effectiveness. Besides, our sorting accelerator is customizable and the RTL source code is released as an open-source hardware. These are significant differences with the prior work.

7. Conclusion

In this work, we presented the acceleration approach for sorting application. Our proposed accelerator uses two sorting architectures, the sorting network and the merge sorter tree. It sorts data sent from a host PC via the PCIe bus and sends back the fully sorted data sequence to it. In this paper, we detailed the design and implementation, and evaluated the sorting performance and hardware resource utilization.

As its most characteristic point, the proposed system is customizable, and we also provided a detailed analytical model that accurately estimates the sorting performance depending on the hardware configuration. Due to these characteristics, designers can estimate sorting accelerator performance in advance and can implement the best one to meet the cost and performance constraints.

Our proposed accelerator offers significantly high sorting performance, but the performance efficiencies are limited due to the insufficient memory bandwidth. To address this problem, we proposed the data compression mechanism based on the algorithm using a base value and an array of deltas. As a result, the sorting accelerators with it improved the performance efficiencies in all data-sequence types, and the performance efficiencies are almost 90% if all data is compressed.

In order to allow every designer to easily and freely use this accelerator, the RTL source code is released as open-source hardware. To the best of our knowledge, this is the first open-source sorting accelerator in the world that is high performance, is customizable, and mitigates the memory bandwidth limitation. All the code used to obtain the results in this paper is also available at https://github.com/monotone-RK/FACE.

References

[1] H. Inoue and K. Taura, "Simd- and cache-friendly algorithm for sorting an array of structures," Proc. VLDB Endow., vol.8, no.11, pp.1274–1285, July 2015.

[2] M. Cho, D. Brand, R. Bordawekar, U. Finkler, V. Kulandaisamy, and R. Puri, "Paradis: An efficient parallel algorithm for in-place radix sort," Proc. VLDB Endow., vol.8, no.12, pp.1518–1529, Aug. 2015.

[3] O. Polychroniou and K.A. Ross, "A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort," Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, New York, NY, USA, pp.755–766, ACM, 2014.

[4] B. Chandramouli and J. Goldstein, "Patience is a virtue: Revisiting merge and sort on modern processors," Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, New York, NY, USA, pp.731–742, ACM, 2014.

[5] H. Sundar, D. Malhotra, and G. Biros, "Hyksort: A new variant of hypercube quicksort on distributed memory architectures," Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, ICS '13, New York, NY, USA, pp.293–302, ACM, 2013.

[6] C. Balkesen, G. Alonso, J. Teubner, and M.T. Özsu, "Multi-core, main-memory joins: Sort vs. hash revisited," Proc. VLDB Endow., vol.7, no.1, pp.85–96, Sept. 2013.

[7] C. Kim, J. Park, N. Satish, H. Lee, P. Dubey, and J. Chhugani, "Cloudramsort: Fast and efficient large-scale distributed ram sort on shared-nothing cluster," Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, New York, NY, USA, pp.841–850, ACM, 2012.

[8] N. Satish, C. Kim, J. Chhugani, A.D. Nguyen, V.W. Lee, D. Kim, and P. Dubey, "Fast sort on cpus and gpus: A case for bandwidth oblivious simd sort," Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, New York, NY, USA, pp.351–362, ACM, 2010.

[9] R. Kobayashi and K. Kise, "Face: Fast and customizable sorting accelerator for heterogeneous many-core systems," Embedded Multicore/Manycore SoCs (MCSoc), 2015 IEEE 9th International Symposium on, pp.49–56, Sept 2015.

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

[11] G. Pekhimenko, V. Seshadri, O. Mutlu, P.B. Gibbons, M.A. Kozuch, and T.C. Mowry, "Base-delta-immediate compression: Practical

data compression for on-chip caches," Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT '12, New York, NY, USA, pp.377–388, ACM, 2012.

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

[13] R. Mueller, J. Teubner, and G. Alonso, "Sorting networks on fpgas," The VLDB Journal, vol.21, no.1, pp.1–23, Feb. 2012.

[14] 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, July 2014.

[15] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '15, New York, NY, USA, pp.240–249, ACM, 2015.

[16] K.E. Batcher, "Sorting networks and their applications," Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring), New York, NY, USA, pp.307–314, ACM, 1968.

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

[18] J. Casper and K. Olukotun, "Hardware acceleration of database operations," Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays, FPGA '14, New York, NY, USA, pp.151–160, ACM, 2014.

[19] T. Usui, R. Kobayashi, and K. Kise, "A challenge of portable and high-speed fpga accelerator," Applied Reconfigurable Computing, ed. K. Sano, D. Soudris, M. Hübner, and P.C. Diniz, Lecture Notes in Computer Science, vol.9040, pp.383–392, Springer International Publishing, 2015.

[20] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," Information Theory, IEEE Transactions on, vol.23, no.3, pp.337–343, May 1977.

[21] D. Huffman, "A method for the construction of minimum-redundancy codes," Proceedings of the IRE, vol.40, no.9, pp.1098–1101, Sept. 1952.

[22] P. Deutsch, "Deflate compressed data format specification version 1.3," 1996.

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

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

[25] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "Riffa 2.1: A reusable integration framework for fpga accelerators," ACM Trans. Reconfigurable Technol. Syst., vol.8, no.4, pp.22:1–22:23, Sept. 2015.

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

[27] G. Marsaglia, "Xorshift rngs," Journal of Statistical Software, vol.8, no.14, pp.1–6, 2003.

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

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

[30] M. Zuluaga, P. Milder, and M. Püschel, "Computer generation of streaming sorting networks," Proceedings of the 49th Annual Design Automation Conference, DAC '12, New York, NY, USA, pp.1245–1253, ACM, 2012.

Ryohei Kobayashi received the M.E degree and the Ph.D. degree from Tokyo Institute of Technology, Japan in 2013 and 2016. He is currently an assistant professor of Center for Computational Sciences, University of Tsukuba, Japan. His research interests include FPGA systems for high performance computing. He is a member of IPSJ.

Kenji Kise received the B.E. degree from Nagoya University in 1995, the M.E. degree and the Ph.D. degree from the University of Tokyo in 1997 and 2000 respectively. He is currently an associate professor of the Graduate School of Information Science and Engineering, Tokyo Institute of Technology. His research interests include computer architecture and parallel processing. He is a member of ACM, IEEE, and IPSJ.