Electronic Preprint for Journal of Information Processing Vol.30

Regular Paper

An Open-source FPGA Library for Data Sorting

Ryohei Kobayashi[1,a)] Kento Miura[2] Norihisa Fujita[1] Taisuke Boku[1] Toshiyuki Amagasa[1]

Received: February 4, 2022, Accepted: June 17, 2022

Abstract: Field-programmable gate arrays (FPGAs) have garnered significant interest in research on high-performance computing because their flexibility enables the building of application-specific computation pipelines and data supply systems. In addition to the flexibility, toolchains for the development of FPGAs in OpenCL have been developed and offered by FPGA vendors that reduce the programming effort required. However, the high level of abstraction in the OpenCL-based development approach is a disadvantage, making it difficult to perform fine-grained performance tuning. In this paper, we present one of the methodologies to achieve both the reduction of FPGA programming cost and the provision of high performance. We focus on data sorting, which is a basic arithmetic operation, and we introduce a sorting library that can be used with the OpenCL programming model for FPGAs. Our sorting library has so far only supported integer data, but in this paper, we propose a new method that supports floating-point data. It consumes at least twice as many hardware resources compared to the merge sort restructured for the OpenCL programming model for FPGAs. However, its operating frequency is 1.08x higher and its sorting throughput is three orders of magnitude greater than the baseline. The source code of our sorting library is open source, and it can be used by application developers around the world.

Keywords: FPGA, Sorting, Intel FPGA SDK for OpenCL

1. Introduction

Field-programmable gate arrays (FPGAs) have been used in the development of various applications because their flexibility allows designers to freely implement and customize their application-specific computation pipelines and data supply systems. However, the traditional development of applications using FPGAs has been mainly based on highly specialized register transfer level (RTL) designs, and programming cost and program portability have not been emphasized. As a result, even though FPGA-based applications' performance is higher than non-FPGA implementations, only limited hardware engineers could develop them due to the high hardware knowledge requirement with the result that application developers tend not to use FPGAs.

High-level synthesis (HLS) techniques have emerged and evolved considerably in the last decade for mitigating the FPGA implementation cost which removes that barrier. In particular, the Xilinx Vitis and Intel FPGA SDK for OpenCL are an all-in-one development environment that hide from the user the fundamental parts of FPGA implementation, such as external memory controllers, the PCIe interface, and device drivers, enabling implementation of FPGA applications without the user being aware of them. The Intel FPGA SDK for OpenCL provides a mechanism for controlling the peripherals for the FPGA fabric from OpenCL, and we previously proposed a framework [1] using the mechanism to assist application developers in developing HPC applications in the OpenCL programming layer. Therefore, we believe that the Intel FPGA SDK for OpenCL is currently the best solution for application developers when using FPGAs.

However, whereas register transfer level (RTL)-based FPGA implementations allow designers to define to a fine degree the appropriate architecture for their applications, OpenCL-based programming models are limited in their expressiveness. In addition, OpenCL-based programming models often suffer from place and route problems that limit the maximum frequency or entirely prevent a successful synthesis [2]. Therefore, to develop high-performance FPGA applications with low programming cost, it is necessary to optimize general-purpose computation kernels for the practical application development for FPGAs and to make them available to users. For example, the authors in Ref. [2] proposed an OpenCL implementation of Cannon's matrix multiplication algorithm optimized for Intel Stratix 10 FPGAs, assuming that it could serve as a building block for algorithms that are built upon the base functionality of matrix multiplications.

Based on the background, we have introduced a sorting library that can be used with the OpenCL programming model for FPGA [3]. Sorting is a basic arithmetic operation used in the development of various applications, and to accommodate its wide range of uses, libraries are provided in the form of `std::sort()` for C++ and `thrust::sort()` for CUDA. We developed a hardware engine in RTL to perform the sorting process and call it a library from the OpenCL kernel code. The engine is built by combining the following three hardware sorting algorithms: the sorting network, the high-bandwidth merge sorter tree, and the virtual merge sorter tree. These algorithms are designed to run at high operating frequencies so that the engine called from the OpenCL kernel does not become the critical path and degrade the overall performance of the application. In addition, the engine's

[1] Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki 305–8577, Japan
[2] Rakuten, Inc., Setagaya, Tokyo 158–0094, Japan
[a)] kobayashi@cs.tsukuba.ac.jp

configuration can be set by parameters, and the application developer can determine the best configuration for the sorting engine according to our derived performance model while remaining at the OpenCL abstraction level.

In this paper, we describe in detail the specifications of the sorting library, which could not be described in Ref. [3] due to space limitations, and propose a sorting method for floating-point data as a difference from our previous work. We evaluated the performance of our proposed sorting library for floating-point data and show that it can achieve the same performance as integer types with only a small amount of hardware resource consumption. And we confirmed that its sorting throughput for floating-point data was three orders of magnitude greater than that of the merge sort algorithm restructured for the OpenCL-based implementation. Our sorting library is open source, and it is available in the GitHub repository here: https://github.com/ac2-prod/fpga_sort.

Our contributions in this paper are:

- We present one of the methodologies and its implementation in detail to achieve both the reduction of FPGA programming cost and the provision of high performance.
- We propose a sorting method for floating-point data that can be integrated into the sorting engine, and quantitatively demonstrated that it can achieve almost the same performance as sorting integer data with a small amount of hardware resources.
- We have open-sourced our sorting library and made it freely available to application developers around the world.

This paper is organized as follows. In Section 2, we describe the prerequisite knowledge needed to understand the behavior of the sorting engine developed in this study. And based on that, we introduce a hardware sorting engine implemented in an FPGA and show how to perform floating point arithmetic sorting in Section 3 and describe the sorting method for floating-point data in Section 4. Our proposed sorting engine can be called from OpenCL kernel code and Section 5 describes how to create an OpenCL library that contains the sorting function. In Section 6, we perform experimental evaluations, and show hardware resource consumption and sorting performance. Section 7 shows the latest researches related to this study, and finally, this paper is concluded in Section 8.

2. Hardware Sorting Algorithm

Hardware sorting algorithms are hardware models that embody a processing procedure suitable for data sorting by hardware. In particular, a sorting network [4] and merge sorter tree [5] are suitable for FPGAs in terms of parallelism, ease of control, and operating frequency, and numerous related studies that use them have been conducted [6], [7], [8], [9], [10]. In this section, we introduce these basic behaviors as a background for understanding our developed sorting engine. To avoid ambiguity in the description, the hardware sorting algorithm described in this study sorts data records with integer keys in ascending order.

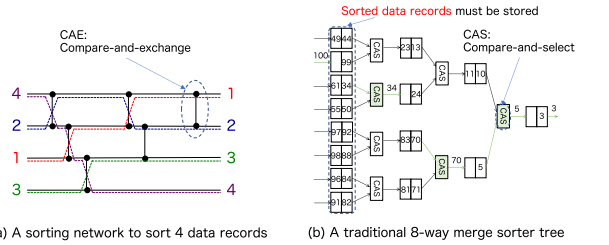


Fig. 1 Underlying hardware sorting algorithms: (a) a sorting network and (b) a merge sorter tree.

The sorting network is a hardware model for sorting sequences of numbers that consist of wires and a compare-and-exchange (CAE) unit. **Figure 1** (a) shows its behavior. The wires are responsible for propagating records, and the CAE unit has two wires for input and output. When two records are input, it outputs a record with the smaller key to one and the other with the larger key to the other. This hardware model can sort the records in parallel without the need for complex control logic.

The merge sorter tree is a hardware model for merging multiple sorted data records and is composed of a set of compare-and-select (CAS) units laid out in a binary tree. The CAS unit is a combinational circuit that compares the keys of two input data records and selects one of them according to the comparison result. Figure 1 (b) shows a traditional 8-way merge sorter tree and its behavior at a certain clock cycle. At each cycle, the keys are compared in all CAS units, and the data records that are output according to the comparison results are stored in first-in/first-out buffers (FIFOs), and eventually, sorted data records are ejected from the root of the merge sorter tree. FIFOs located at the input port of the tree must always contain sorted data records and, therefore, the sorting network is often used to produce them [6].

3. FPGA-based Sorting Engine

Figure 2 shows our sorting engine that is built by combining three hardware sorting algorithms: the sorting network, the high-bandwidth merge sorter tree, and the virtual merge sorter tree. In this section, we first explain the mechanism of each hardware sorting algorithm and then describe the sorting engine's behavior.

3.1 Batcher's Odd-even Sorting Network

We use Batcher's odd-even sorting network [11], where **Fig. 3** shows an example with eight inputs and eight outputs. This sorting network consists of 19 CAE units with six stages. In Ref. [12], sorting networks on FPGAs are discussed in detail in terms of hardware resource usage and throughput, and the authors conclude that odd-merge sorting is the most efficient sorting network. For example, a bitonic sorting network with eight inputs and eight

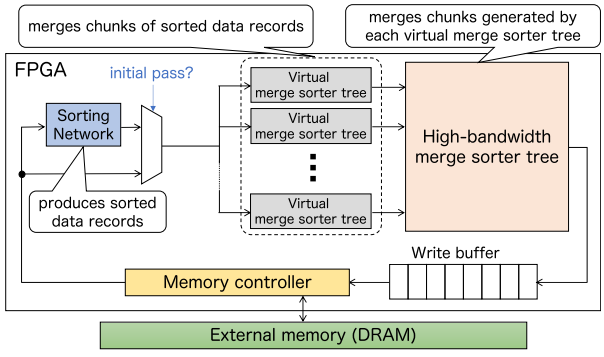


Fig. 2 Our developed sorting engine.

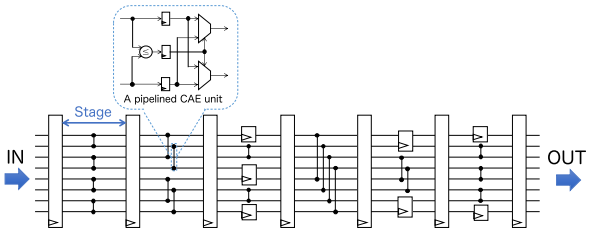


Fig. 3 Pipelined Batcher's odd-even sorting network with eight inputs and eight outputs.

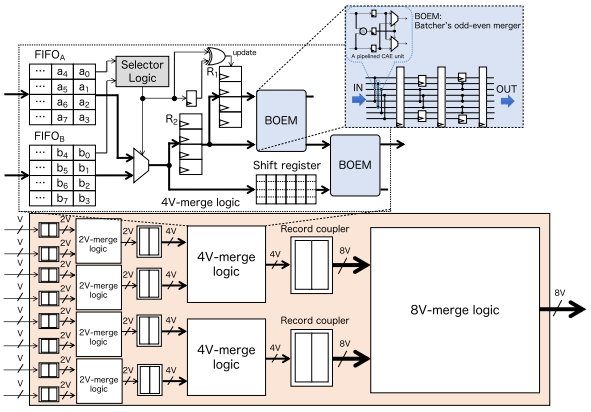


Fig. 4 High-bandwidth merge sorter tree [8].

outputs consists of the same six stages as the odd-even sorting network, but the former consumes more hardware resources than the latter because it requires 24 CAE units.

The sorting network can be implemented as a purely combinational circuit, but its naive implementation causes a performance degradation due to the large network delay. To avoid degradation of the operating frequency and to improve the network throughput, this network is commonly implemented as a pipelined circuit by inserting registers between each stage. In addition, the sorting network used in the sorting engine employs a two-stage pipelined CAE unit [7]. This allows the critical path to be only one comparator, thereby increasing the operating frequency.

3.2 High-bandwidth Merge Sorter Tree

The traditional merge sorter tree described in Section 2 can output only a maximum of one record per cycle, and at least n cycles are required to output n records. If the tree can output m records per cycle, the number of cycles required to output all the records can be reduced to a maximum of $\frac{n}{m}$, and several methods to achieve that throughput have been proposed in recent years [6], [7], [8]. We call these merge sorter trees high-bandwidth merge sorter trees and incorporate the one proposed in Ref. [8] into our sorting engine.

Figure 4 shows a high-bandwidth merge sorter tree proposed in Ref. [8]. This tree is built by connecting the merging logic for multiple data records in the form of a binary tree. As the number of input ports of the tree increases, the number of data records output per cycle from the root of the tree increases, as when several tributaries join to form a wide river. In Fig. 4, the number of input ports is eight, and the incoming eight data streams are merged into a single data stream, which is then output with a throughput of a maximum of eight data records per cycle.

We briefly explain the merging logic underlying this tree's behavior. With the logic of merging four data records taken as an example, it can be seen that two Batcher odd-even mergers (BOEMs) are connected in series. In this manner, the feedback path [6], which is necessary to merge multiple data records constantly, is successfully eliminated, and the logic can be pipelined at a high operating frequency. The most outstanding point of the proposed method in Ref. [8] is that it efficiently solves the tie-record issue [7] by developing timelined bundle ordering, in which R1 and R2 are connected in series, and by selecting BOEM as the merger instead of the bitonic merger used in Ref. [7]. Because of this mechanism, the tree proposed in Ref. [8] exhibited the best merging performance at the stage when we were developing the specification of the sorting engine and, therefore, we decided to adopt it as a component of the engine.

Because the tree in Ref. [8] is also a merge sorter tree, increasing the number of input ports not only increases the number of data records output per cycle from the root of the tree, but it also increases the number of sorted data records in the output data stream. However, as the number of input ports of the tree is increased, the hardware resource usage increases linearly. Therefore, the size of the tree that can be implemented in an FPGA is limited to a certain extent, and the number of sorted data records in the output data stream generated by passing through the tree does not scale. We describe a method to overcome this problem in the section that follows.

3.3 Virtual Merge Sorter Tree

In Refs. [9], [10], the authors proposed hardware-efficient multi-way merge sorter trees that are logically equivalent to the traditional merge sorter tree described in Section 2. We call those trees virtual merge sorter trees and describe their mechanism as follows.

In the traditional merge sorter tree, the CAS unit located at the root selects and outputs only one record on either side at a time. Therefore, in each stage of the tree, only one CAS unit as well as one FIFO dequeue request and one FIFO enqueue request are active. This is highlighted in Fig. 1 (b). When this characteristic is taken advantage of, the virtual merge sorter tree proposed in Refs. [9], [10] reduces hardware resource usage from $O(W)$ to $O(\log_2 W)$ by placing only one CAS unit in each stage, thus working in a time sharing manner and integrating the FIFOs between CAS units into a single buffer layer implemented in the FPGA's Block RAM, where W is the number of leaves in the tree. In our sorting engine, the virtual merge sorter tree is replicated as many times as the number of ports of the high-throughput merge sorter tree described in Section 3.2, and the output port of each virtual merge sorter tree is connected to each input port of the high-throughput merge sorter tree, which builds a merge sorter tree having both high throughput and a wide input width.

However, the virtual merge sorter trees proposed in Refs. [9], [10] have the disadvantage whereby their operating frequencies are less than half that of the high-bandwidth merge sorter tree proposed in Ref. [8]. Therefore, the performance of the high-bandwidth merge sorter tree cannot be maximized by simply connecting the virtual merge sorter tree proposed in Refs. [9], [10].

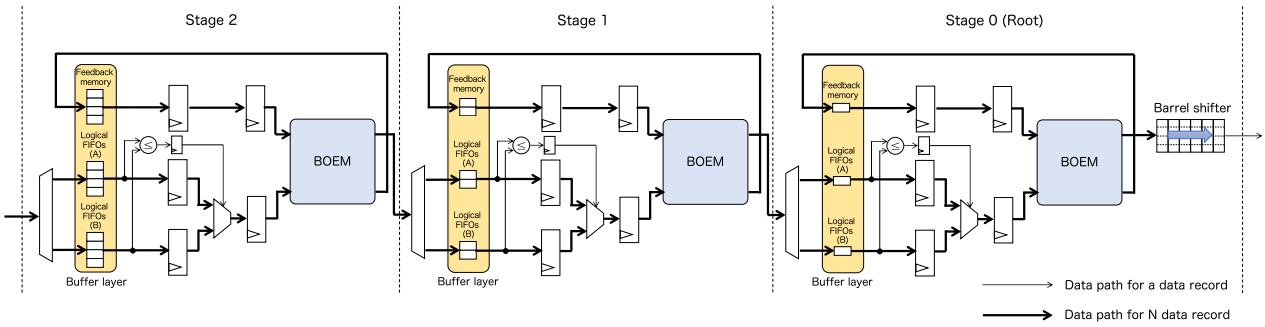


Fig. 5 Multi-cycle virtual merge sorter tree.

To solve this problem, we developed a multi-cycle virtual merge sorter tree, as shown in **Fig. 5**.

The developed tree outputs N data records per N cycles, whereas the virtual merge sorter tree proposed in Refs. [9], [10] outputs only one data record per cycle. To output multiple sorted data records, we replaced the CAS units in each stage with the BOEM used in Ref. [8] and added a memory area to the buffer layer to store the data records fed back from the BOEM. Similar to the CAS unit in Refs. [9], [10], the BOEM works in a time sharing manner. It merges the data read from the logical FIFOs in the buffer layer corresponding to the request coming from the upper stage and selected by the selector with the data read from the feedback memory. It then outputs the smaller data records to the upper stage and writes the larger data records back to the corresponding area in the feedback memory. Then, the sorted data records output from the BOEM at the root are extracted as a single data record at a clock cycle by the barrel shifter and input to the high-bandwidth merge sorter tree. Therefore, the number of data records ejected per cycle of the developed tree is the same as that of the virtual merge sorter trees proposed in Refs. [9], [10]. However, the critical path of the developed tree is only one comparator and, consequently, it can operate at a frequency as high as that of the high-bandwidth merge sorter tree. In addition, the datapath in the BOEM used in the developed tree is optimized by using techniques in Ref. [8] such as redundant CAS unit elimination and datapath decoupling for payload, resulting in a hardware-efficient implementation.

3.4 Sorting Engine Behavior

Because our design concept of the sorting engine is basically the same as that in Ref. [6], it completes the sorting process by performing multiple memory round-trips of the data. In this section, we explain how the sorting engine sorts 2,048 unsorted data records. Here, we assume the width of the sorting network, number of leaves in the virtual merge sorter tree, and number of input ports in the high-bandwidth merge sorter tree to be 8, 4, and 4, respectively.

Similar to Ref. [6], during the initial pass of the sorting process, the data for input can come from anywhere. Therefore, the 2,048 unsorted data records stored in the external memory are read out sequentially by the memory controller and sent to the virtual merge sorter trees through the sorting network, as shown in **Fig. 6** (a). This is because the virtual merge sorter trees expect their inputs to be chunks of sorted data records, and a single chunk contains eight sorted data records at this point. These chunks are merged in each virtual merge sorter tree and then sent to the high-bandwidth merge sorter tree. The merge sorter tree then merges the chunks generated by each virtual merge sorter tree at a data rate of four data records per cycle and enqueues sorted data records output from the high-bandwidth merge sorter tree into the write buffer. The data records stored in the write buffer are then sequentially written back to the external memory by the memory controller, as shown in Fig. 6 (b). The sorting engine uses double buffering to allow both writing and reading from memory simultaneously. In this initial pass, 16 chunks of sorted data records are generated by the sorting engine and written to the external memory. At this point, each chunk contains 128 sorted data records.

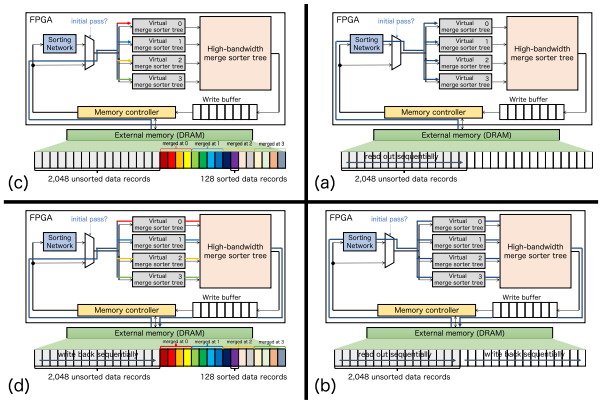


Fig. 6 The sorting engine's behavior to sort 2,048 unsorted data records.

On passes after the initial pass, the virtual merge sorter tree must retrieve data from a particular chunk that matches the tree input of the request, which is similar to Ref. [6]. Therefore, the first four chunks of the 16 chunks are merged in virtual merge sorter tree 0, and the next four are merged in virtual merge sorter tree 1, as shown in Fig. 6 (c). Each virtual merge sorter tree outputs 512 sorted data records, which are merged by the high throughput merge sorter tree. Then, the fully sorted data records are sequentially overwritten back to where initial data records are stored in the external memory, as shown in Fig. 6 (d).

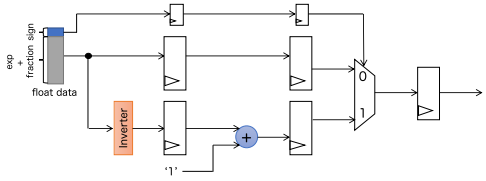
Let N_{pass} be the number of passes. It can be formulated as $\lceil \log_{W \times E} \frac{N_{data}}{P} \rceil$, where N_{data} is the number of data records to be sorted, P is the width of the sorting network, W is the number of leaves in the virtual merge sorter tree, and E is the number of input ports in the high-bandwidth merge sorter tree. In addition,

theoretical peak throughput of the sorting engine can be formulated as $\frac{E \times B \times F}{N_{pass}}$, where F is the operating frequency and B is the data size of a data record. Please note that this formula is based on the assumption that the memory bandwidth is infinite and the active rate [8] is 1.

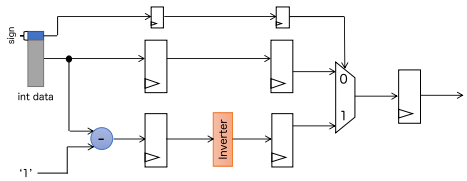
4. Sorting Floating-point Data

Our proposed method targets floating-point data conforming to the IEEE754 standard. The IEEE754 consists of a sign part, an exponent part, and a fraction part to represent floating-point data. The exponent part is represented by a bias record instead of two's complement, and it is placed to the left of the fraction part to simplify the comparison between large and small floating-point data. In other words, if the whole bit represented by the IEEE754 is regarded as just a signed integer, the relation between the size of the data is the same as the relation between the size of the original floating-point data, and therefore, the signed-integer comparator can identify the size of the floating-point data. However, when the floating-point data is a negative number, the larger the value excluding the sign bit (absolute value), the smaller the number becomes, so the relationship between the size of the data when the whole bit is regarded as just a signed integer is the exact opposite of the relationship between the size of the original floating-point data. In this case, by taking the two's complement for the value excluding the sign bit and setting the MSB to 1, it is possible to compare the data as a negative number of integer data.

Using this property, our proposed method converts the floating-point data stored in external memory into signed integer data that can be compared in size when reading out, sorts the signed integer data with the sorting engine, and writes the sorted data back to the external memory after converting back to the original floating-point data. **Figure 7** shows the hardware logic to convert floating-point data into signed integer data that can be compared in size, and the hardware logic to convert the signed integer data back to the original floating-point data.



(a) FLOAT2INT: Converting floating point data to signed integer data that allows for comparison



(b) INT2FLOAT: Converting signed integer data to its original floating-point data

Fig. 7 Hardware logic for sorting floating-point data.

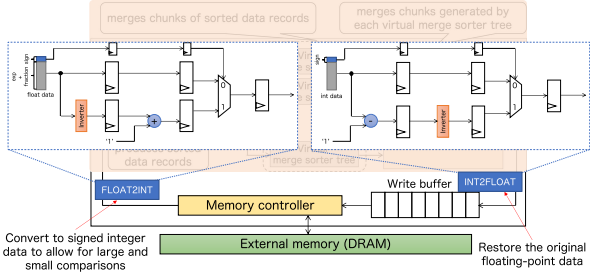


Fig. 8 Illustration of applying hardware logic for sorting floating-point data.

As shown in the figure, these logics consist of a three-stage pipeline. In FLOAT2INT, the first stage inverts the value without the sign bit, the second stage adds 1, and the third stage selects the signed integer data to be sorted according to the sign of the original floating-point data. In INT2FLOAT, the first stage subtracts 1, the second stage inverts the bits, and the third stage selects the floating-point data to be written back to memory according to the sign of the signed integer data. This configuration requires only one adder/subtracter for the critical path, and thus reduces the frequency degradation of the sorting engine.

Figure 8 shows overview of the hardware logic for sorting floating-point data incorporated into the sorting engine. As shown in the figure, it is possible to support floating-point data by adding only a few modules to the original sorting engine. Application developers can then decide whether to add these modules or not by changing the parameter value.

5. Creating an OpenCL Library for Data Sorting

As mentioned in Section 1, we focus on providing the sorting engine to application developers so that they can develop their FPGA codes easily, and we believe that the OpenCL programming model for FPGA is currently the best solution for that purpose. In this section, we describe how the RTL-tuned sorting engine can be used in an OpenCL programming environment.

The Intel FPGA SDK for OpenCL includes a feature [13] that enables the user's own RTL modules to be an OpenCL library that can be called from the OpenCL kernel code. **Figure 9** shows the programming model of the Intel FPGA SDK for OpenCL using the generated OpenCL library. The kernel code that includes the OpenCL library is compiled using an Intel FPGA OpenCL compiler, which is offered by the Intel FPGA SDK for OpenCL. The compiler is used to convert into synthesizable Verilog HDL files, which are then used in Quartus Prime to generate an `.aocx` file that includes FPGA configuration information. The host code is for programming the host application; it runs on a host PC and manages an FPGA device at runtime using a set of common

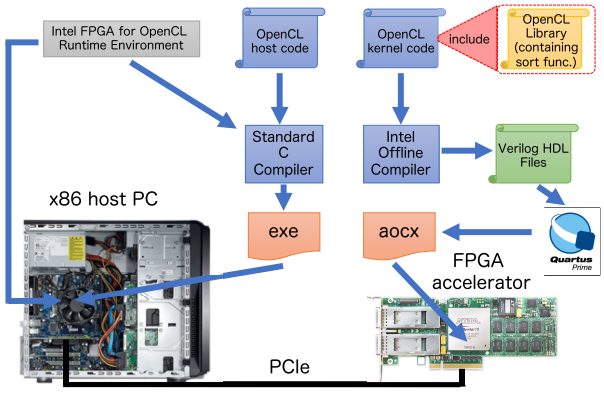


Fig. 9 Intel FPGA SDK for OpenCL programming model with our sorting library.

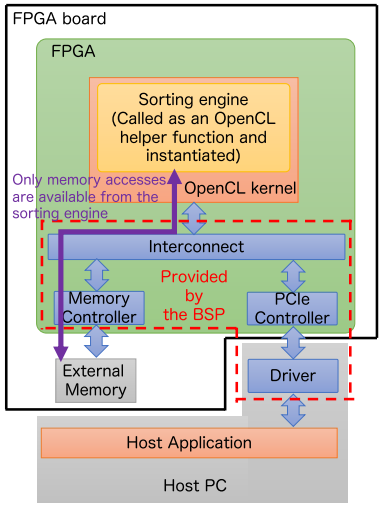


Fig. 10 Schematic of the Intel FPGA SDK for OpenCL platform with our sorting library.

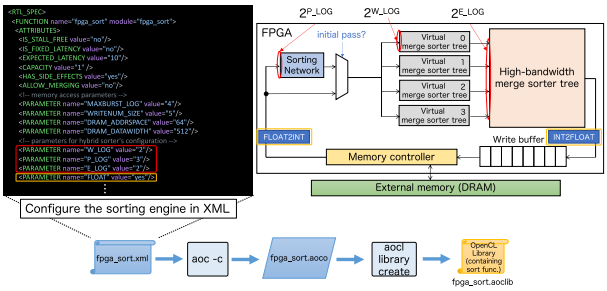


Fig. 11 Overview of OpenCL library creation that contains the sorting function.

application programming interfaces (APIs). To generate a host binary, the code is compiled using a standard C compiler such as GCC on Linux, and its object file is linked to the Intel FPGA SDK for OpenCL runtime libraries. At a runtime of the host application, the `.aocx` file is downloaded to the FPGA using the OpenCL APIs; any data required for kernel execution and any data generated are transferred via the PCIe bus.

The FPGA computing realized by the above programming model is illustrated in **Fig. 10**. The host application is implemented using the OpenCL host code, and the application-specific pipelined hardware including the sorting engine is generated from the OpenCL kernel code. The sorting engine called as a helper function of the OpenCL kernel is instantiated as a hardware module and automatically connected to the hardware generated by the OpenCL kernel (the caller) during kernel code compilation. As a restriction of the RTL module called as a helper function, the sorting engine can only access external memory via the Avalon-MM interface (e.g., it is impossible to directly sort incoming data from an optical link without going through external memory). The external memory controller, PCIe controller, and PCIe driver, which are essential for the OpenCL programming model, are bundled in the board support package (BSP) provided by the FPGA board vendors. Therefore, application developers basically need to focus on developing these two codes (OpenCL host and kernel codes) to implement FPGA-based applications.

Figure 11 shows an overview of the library creation process that contains the sorting function. First, an intermediate object

```
1  #include "fpga_sort.h"
2
3  __kernel void fpga_sort_test(
4                               __global float *restrict tmp,
5                               __global float *restrict src,
6                               const uint numdata,
7                               __global uint *restrict ret
8                              )
9  {
10   // Do sort
11   *ret = fpga_sort(tmp, src, numdata);
12 }
```

Fig. 12 OpenCL kernel code snippet of the data sorting.

file (`.aoco` file) is generated from the XML file that defines the configuration of our sorting engine using the `aoc -c` command. The parameter values of W, P, and E described in Section 3.4 are defined in this XML file, and application developers can determine the optimal configuration of the sorting engine while considering the amount of target FPGA resources and required sorting performance. They can also determine whether to perform sorting of floating-point data by changing the value of the `FLOAT` parameter to '`yes`' or '`no`'. Then, an OpenCL library (`.aoclib` file) that contains the sorting function is created from the generated `.aoco` file using the `aocl library create` command.

An OpenCL kernel code snippet with the sorting library is shown in **Fig. 12**. Initially, it is necessary to include the header file (`fpga_sort.h`) that declares the signatures of the sorting function offered from the engine. The sorting function in the library is called in Line 11 and requires three arguments: `tmp`, `src`, and `numdata`. As described in Section 3.4, our sorting engine requires two memory areas because reading and writing data from/to memory occur simultaneously, and initial data records are stored in an `src[]` array. These arguments are sent to the sorting engine by the Avalon-ST protocol, and it starts the sorting process. Access to the external memory is performed through the Avalon-MM interface using the address to which the pointer pointed and that is passed by the arguments as the base address. Please note that the Intel FPGA SDK for OpenCL does not allow you to pass local memory pointers to RTL modules that can be called in the OpenCL kernel code. When the sorting process is complete, the sorting engine returns a value indicating in which array (`tmp[]` or `src[]`) the sorted data records are stored.

6. Evaluation

6.1 Experimental Settings

We implemented the sorting engine in Verilog HDL and created the OpenCL library that contains the sorting function with Intel FPGA SDK for OpenCL, the version for which was 19.4.0 Build 64 Pro Edition. Our solution was deployed on a BittWare 520N board featuring an Intel Stratix 10 FPGA (1SG280HN2F43E2VG). In this study, we focus on the fact that the sorting engine can be used in OpenCL kernel code as an OpenCL library and, therefore, we evaluated its hardware resource usage and sorting performance itself as offered from the OpenCL library. This means that this evaluation did not consider data transfer between the CPU and FPGA. We used the `clock_gettime()` function with `CLOCK_MONOTONIC` argument to measure the execution time for sorting data records, which means that the overhead of launching the OpenCL kernel is in-

Table 1 Comparison results between our sorting library ($W = 4$, $P = 8$, and $E = 4$) and the restructured merge sort algorithm [14] for 2^{29} records. The resource usage excludes resources for the OpenCL BSP.

Integer										
	ALMs	(%)	Registers	(%)	M20Ks	(%)	DSPs	(%)	fmax [MHz]	Throughput [MB/s]
Our sorting library	76,644	8%	219,000	6%	350	3%	2	0.03%	**380.66**	**118**
The merge sort [14]	23,642	3%	48,755	1%	147	1%	0	0%	352.6	0.32

Floating point										
	ALMs	(%)	Registers	(%)	M20Ks	(%)	DSPs	(%)	fmax [MHz]	Throughput [MB/s]
Our sorting library	79,543	9%	229,761	6%	350	3%	2	0.03%	**380.66**	**116**
The merge sort [14]	18,263	2%	50,962	1%	147	1%	0	0%	351.0	0.32

cluded in the sorting execution time. For this evaluation, we used 64-bit data records composed of 32-bit payload fields and 32-bit key fields.

6.2 Comparison to Merge Sort Algorithm Restructured for HLS

We first compared the proposed sorting library with a sorting algorithm that assumes the use of HLS. In order to accurately compare sorting performance, the dataset to be sorted, the data type, and the execution hardware must be identical. Although there are many studies on FPGA sorting, there are subtle differences, such as the use of FPGAs from other manufacturers or older generations, the use of only keys for sorting, and the use of only integer data types, which may lead to incorrect conclusions if the results are easily compared with those presented in previous studies. Therefore, for an accurate comparison of FPGA sorting performance, it is necessary to actually run the FPGA sorting algorithm to be compared on the hardware platform and it is desirable that the source code is open source and publicly available. *Parallel Programming for FPGAs* [14] introduced a merge sort algorithm restructured for the Xilinx Vivado HLS and its source code is written in the book. We reimplemented it for the Intel FPGA SDK for OpenCL and run on the BittWare 520N board. For this comparison, we used 2^{29} data records that were randomly generated and uniformly distributed. The data size was determined according to our target application (i.e., regular path query against large labeled graphs [15] in which sort plays a crucial role in improving the overall performance). The keys used in this comparison were positive integers and real numbers in the range `-FLT_MAX` to `FLT_MAX`.

Table 1 shows the comparison results. The parameter values of W, P, and E for the sorting engine were the same as those described in Section 3.4. The only difference between the sorting libraries in the "Integer" table and the "Floating point" table is whether the `FLOAT` parameter is set to '`no`' or '`yes`' in the compilation. Our sorting library consumes at least twice as many hardware resources compared to merge sort [14], but its operating frequency is 1.08x higher and its sorting throughput is three orders of magnitude greater. By default, a control logic for global memory interleave that is automatically generated by the `aoc` has become a critical path. We removed it by disabling global memory interleave and selecting a ring topology as a global memory interconnection by means of flags in the `aoc` as described in Ref. [2]. This optimization is applied to both our method and the comparator, and both of which operate at frequencies above 350 MHz. However, the difference in sorting throughput by more than three orders of magnitude is nevertheless due to the highly pipelined architecture of our sorting engine. Even if the data to be sorted is floating point, our sorting engine can achieve almost the same performance compared to the integer case because the pipeline is still built on the FPGA. The 1.7% drop in throughput compared to the integer case is due to the overhead of filling the pipeline including FLOAT2INT and INT2FLOAT modules. The hardware resource usage increased by about 1% due to the inclusion of these modules.

For the operating frequency, our sorting library itself can operate at 494.07 MHz for integer data and can operate at 464.04 MHz for floating-point data, but they are down to 380.66 MHz that is the maximum operating frequency achievable by the actual combination of the OpenCL toolchain and the BSP offered from BittWare. Therefore, if the OpenCL environment infrastructure is further improved, sorting performance should be achieved.

6.3 Comparison to Data Sorting on CPU/GPU

We compared the sorting performances for floating-point data of the FPGA and those of a CPU and GPU. As in the experiment in the previous section, this comparative evaluation also used data records that were randomly generated and uniformly distributed. The keys were real numbers in the range `-FLT_MAX` to `FLT_MAX`. In this comparison, we used the parameters $W = 64$, $P = 32$, and $E = 4$, which typically achieve the highest performance and efficiency. The hardware resource utilization was 64% for ALMs, 54% for Registers, 30% for M20Ks, and 0.02% for DSPs. The buffer layer of the multi-cycle virtual merge sorter tree suppressed the M20K utilization as expected, but the logic utilization (ALMs) became dominant. The main reason is that hardware resources are allocated to pipeline registers of each stage of the virtual merge sorter tree. The utilization of ALM can be reduced by reducing the width of BOEM and making the pipeline shallower but doing so can degrade the throughput of a 1-record/cycle (the barrel shifter at the root becomes empty). Therefore, this parameter needs to be carefully determined by looking at the balance between the required sorting performance and hardware resources. The fmax is 286.28 MHz because of routing delay affected by the increased hardware resource utilization.

Figure 13 shows the comparison results. The CPU performance was assessed by compiling the OpenMP-versioned radix sort based on Ref. [16] using Intel C++ Compiler 18.0.1 with `-O3` and `-xHost` options, and the GPU performance was assessed by using the Thrust library (CUDA 11.2.152). The data layout in

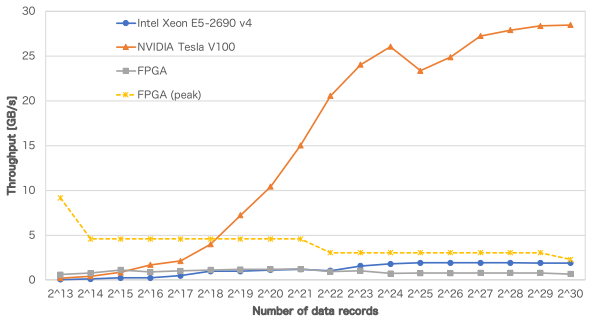


Fig. 13 Comparison of sorting performances for floating-point data based on data size.

the CPU/GPU sort is structure of arrays (SoA) in order to maximize the performance of each device, which means that we used the `thrust::sort_by_key()` function for the data sorting on the GPU. Please note that our sorting library does not support SoA data layout, and the comparison in the previous section is based on array of structures (AoS) data layout. This is because the cost of mapping the sorted keys to the values in the SoA data layout is high in FPGAs compared to the other two devices. On the other hand, in AoS, the key field can be extracted from the record and be input to the comparator, and according to the result of the comparison, the record can be directly flowed through the pipeline. Therefore, the cost of key-value mapping is zero, and our library supports only AoS data layout. When application developers use our sorting library, they need to use a vector type such as `uint2` or `float2` or define their own structure.

The FPGA (peak) was derived using the performance model described in Section 3.4, and the effective performance of the FPGA showed the same performance trend. It was observed that the sorting performance of the FPGA for data of small size was better than those of the other two devices. This was because the sorting engine is a pipelined architecture, and the effect of data size is minimal. In addition, the data size is too small for the number of computing cores of the CPU and GPU (particularly the GPU). The ability to process small problem sizes at high speed is necessary to achieve strong scaling for FPGA-centric applications, and sorting is required in nearly all algorithms, particularly in data intensive applications (e.g., statistics (percentile), search (k-nearest neighbor query), index construction (inverted index), and our target (graph processing)). We believe that our study is worthwhile for developing a hardware logic to accelerate that fundamental process and enabling its ease of use as a library.

The ability to process small problem sizes at high speed is also necessary for the PIC simulation method that is frequently used in the HPC field. The PIC is an ab-initio simulation technique for collisionless plasmas applied to astrophysics and laboratory plasma physics. As a well-known optimization technique for PIC simulations, periodic sorting of particles is used to increase the spatial locality of the overall computational data. Dorobisz et al. [17] experimented with 600×10^6 particles, but according to this reference, the latest application deals with a large number of particles, up to 10^{11}. The data set is divided into domains, and each computational resource has less than 10^6 data per domain as its computational unit. Therefore, a fast sorting method for small problem sizes is desired to accelerate PIC simulations, and our proposed sorting library is expected to be one of the options for this purpose.

From Fig. 13, it can also be read that it is important to use the most appropriate device for the application in the right place, taking into account the nature and extent of the sorting process based on the overall application data size. The main focus of this paper is to facilitate the efficient development of FPGA-centric applications that use sorting, not necessarily to propose FPGA sorting that is more powerful than GPUs. Figure 13 quantitatively indicates that only large-scale sorting should be offloaded to the GPU. When parallelizing under strong scaling conditions, the data size per computing unit will be small, in which case FPGA offloading will be effective. It is also effective for applications such as regular path querying for labeled graphs [15], which is currently our target application. It requires frequent conditional branches that are difficult for GPUs to efficiently work and the sorting is a large part of the overall process. In these cases, our proposed library is expected to serve as a building block for the efficient implementation of the application. This is because our proposed sorting library can be used to perform optimized sorting for FPGAs with function calls similar to `std::sort()` for C++ and `thrust::sort()` for CUDA. If designers try to optimize FPGA sorting with OpenCL, they will need to write at least tens of lines of code, and with RTL, thousands of lines. In contrast, our library does not require the user to implement the sort core, and only one function call is needed to use the RTL-tuned sort engine.

There is an argument that since FPGAs are dedicated circuits, they do not suffer from the cold-start cache misses like CPUs and GPUs do, and therefore, they might be able to outperform CPU and GPU by cold-starting the applications. However, since being good at cold start is a side effect, it is nonsense to make the application cold start just for this reason. It is important to use the most suitable device for the application's acceleration in the right place at the right time. Based on our previous research [18], we recommend offloading the computation parts that the GPU is not good at to the FPGA. The sorting library proposed in this paper is expected to function as a building block to facilitate efficient implementation of the offloading part.

6.4 The Parameters of the Sorting Library

We evaluated how the parameters W, P, and E affect hardware cost and sorting performance. The result is shown in **Table 2**. The keys used in this evaluation were real numbers in the range `-FLT_MAX` to `FLT_MAX` and the number of records was 2^{20} that is the same sorting scale as in Ref. [17]. As discussed in Section 3.4, as the parameters W, P, and E are increased, the number of passes through the sorting engine, N_{pass}, is reduced and sorting performance is improved. Table 2 quantitatively demonstrates that the actual sorting performance follows this principle.

Next, we discuss parameter combinations that have the same N_{pass}, but lead to differences in sorting performance. For example, increasing P improves sorting performance even though N_{pass} is the same, because the throughput of the virtual merge sorter tree approaches 1-record/cycle, which improves the pipeline fill rate of the entire sorting engine. However, there is a tradeoff: increasing P doubles the hardware resource usage of

Table 2 Hardware cost and sorting performance for 2^{20} records when changing the parameters W, P, and E. The resource usage excludes resources for the OpenCL BSP.

W	P	E	N_{pass}	ALMs	(%)	Registers	(%)	M20Ks	(%)	DSPs	(%)	fmax [MHz]	Throughput [MB/s]
4	8	4	5	79,543	9%	229,761	6%	350	3%	2	0.03%	380.66	166
4	16	4	4	130,284	14%	408,782	11%	574	5%	2	0.03%	380.66	365
4	32	4	4	248,398	27%	831,657	22%	990	8%	2	0.03%	380.66	680
4	16	8	4	256,406	27%	830,313	22%	1102	9%	2	0.03%	342.70	597
8	8	4	4	95,155	10%	284,560	8%	506	4%	2	0.03%	380.66	248
8	16	4	4	167,065	18%	538,984	14%	886	8%	2	0.03%	360.36	419
8	32	4	3	330,416	35%	1,159,420	31%	1614	14%	2	0.03%	369.54	914
8	16	8	3	320,677	34%	1,061,483	28%	1726	15%	2	0.03%	348.06	870
16	8	4	3	111,224	12%	338,392	9%	662	6%	2	0.03%	380.66	266
16	16	4	3	202,508	22%	661,952	18%	1198	10%	2	0.03%	380.66	503
16	32	4	3	416,001	45%	1,593,039	43%	2238	19%	2	0.03%	346.26	877
16	16	8	3	393,038	42%	1,322,329	35%	2350	20%	2	0.03%	348.06	1162
32	8	4	3	127,052	14%	393,287	11%	818	7%	2	0.03%	380.66	259
32	16	4	3	238,438	26%	788,176	21%	1510	13%	2	0.03%	374.25	718
32	32	4	3	507,568	54%	1,689,130	45%	2862	24%	2	0.03%	362.45	1110
32	16	8	2	470,266	50%	1,564,389	42%	2974	25%	1	0.02%	323.83	1365
64	8	4	3	149,818	16%	449,364	12%	974	8%	1	0.02%	379.79	393
64	16	4	2	280,766	30%	916,404	25%	1822	16%	1	0.02%	372.30	805
64	32	4	2	594,237	64%	2,029,028	54%	3486	30%	1	0.02%	286.28	1228
64	16	8	2	556,882	60%	1,855,722	50%	3598	31%	1	0.02%	370.37	1414

the BOEM and buffer layer of the virtual merge sorter tree, in addition to the sorting network. The sorting performance improves as E is increased as well. Because the throughput of the sorting engine is determined by the throughput of the high-bandwidth merge sorter tree, increasing E improves sorting performance because the throughput of the high-bandwidth merge sorter tree is increased by a factor of 2. However, there is also a tradeoff: the virtual merge sorter tree connected to the input port of the high-bandwidth merge sorter tree is duplicated twice as often, and the hardware resource usage of the high-bandwidth merge sorter tree increases. Finally, focusing on W, we see a slight improvement in sorting performance (see $W = 16$, $P = 16$, $E = 4$ and $W = 32$, $P = 16$, $E = 4$). When the sorting engine with $W = 32$, $P = 16$, and $E = 4$ sorts 2^{20} records, it merges four chunks on the final pass. If the number of chunks to merge on the final pass is less than or equal to the parameter E, the sorting performance is improved because the virtual merge sorter tree is bypassed and those chunks are directly merged in the high-bandwidth merge sorter tree. However, if the chunks cannot be directly merged in the high-bandwidth merge sorter tree on the final pass (see $W = 16$, $P = 8$, $E = 4$ and $W = 32$, $P = 8$, $E = 4$), the sorting performance is slightly worse due to the penalty of a longer sorting engine pipeline.

In summary, the guideline for determining the parameters of the sorting engine is as follows.

(1) Increase W to reduce N_{pass} as much as possible.

(2) Increase P or E, depending on the available FPGA resources.

The hardware cost of increasing W is less than that of increasing P or E. This is because the virtual merge sorter tree can reduce hardware resource usage from $O(W)$ to $O(\log_2 W)$, as explained in Section 3.3. Therefore, the standard practice is to first increase W to reduce N_{pass} as much as possible. After that, it is desirable to increase P or E to achieve the required sorting performance, taking into account the available FPGA resources.

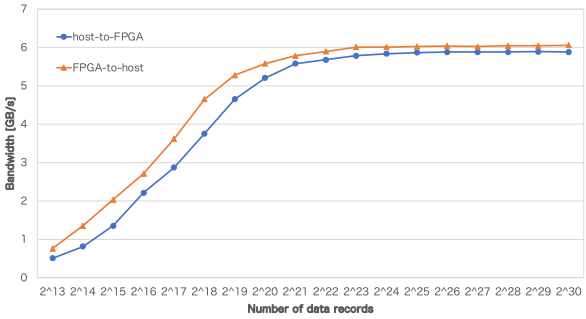


Fig. 14 Communication bandwidth between host and FPGA.

6.5 Data Transfer between Host and FPGA

In this section, we discuss the performance of the sorting library based on the cost of data communication between the host and FPGA. **Figure 14** shows the communication bandwidth when using OpenCL APIs such as `clEnqueueWriteBuffer()` and `clEnqueueReadBuffer()`. The BittWare 520N FPGA board is physically connected via PCIe Gen3 x16, but the theoretical peak of data transfer is 8 GB/s because the PCIe IP core in the OpenCL Board Support Package (BSP) supports up to PCIe Gen3 x8. Therefore, the communication bandwidth between host and FPGA saturates around 6 GB/s, which is 75% of the theoretical performance.

The sorting engine used as a function call from the OpenCL kernel code can only access the FPGA's external memory and cannot directly read data input from PCIe. Therefore, in addition to the sorting execution time, the data transfer time between the host and FPGA independently exists, and the cost of offloading the sorting is the sum of these times.

Our proposed method assumes that the sorting engine is used as a function call from the OpenCL kernel code. However, since the source code of the sorting engine is pure Verilog HDL, it is also possible to incorporate only that part into a dedicated system implemented in RTL. If that dedicated system is such that the sorting engine can directly receive data transferred from the host and send the sorting results back to the host via PCIe com-

munication without going through the FPGA's external memory, then the host-FPGA communication and the FPGA Since sorting can overlap, the cost of offloading sorting is $(B \times N) \div \min(C, T)$ where B is the record's data size, N is the number of data records, C is the communication bandwidth between host and FPGA, and T is the sorting throughput, respectively.

7. Related Work

In this study, we proposed a sorting library that can be handled in OpenCL programming, referring to the results of related studies, such as Refs. [4], [5], [6], [7], [8], [9], [10], [11], [12]. There has been a long line of research on FPGA-based sorting and several studies on it have been reported in recent years [19], [20], [21].

Chen et al. [19] proposed a method for sorting large data sets using a samplesort algorithm on a server with a PCIe-connected FPGA. They implemented the prototype system in Verilog HDL using Amazon Web Services (AWS) FPGA instances equipped with Xilinx Virtex UltraScale+ FPGAs and demonstrate that the system can sort 2^{30} key-value records 37.4x faster than GNU parallel sort running on an CPU with 8 threads. However, their sorting method assumes that the CPU and FPGA work together, and the sorting performance is eventually limited by the PCIe bandwidth, which is a speed of 7.2 GB/s in Ref. [19]. Due to its structure, their proposed method is not suitable for implementing FPGA-centric applications that use sorting.

Samardzic et al. [20] proposed Bonsai, a comprehensive model and sorter optimization strategy that allows the adapting of the sorter design to the available hardware while taking into account the off-chip memory bandwidth and the amount of on-chip resources. The sorter targeted by their proposed method is based on merge sorter trees, and its architecture is similar to our sorting engine. Therefore, the optimization method presented in Ref. [20] can be applied to our sorting engine, which can further improve the sorting performance. The difference between our approach is that they evaluated the sorting performance on integer datasets and did not report on sorting floating-point data. Also, their sorter is implemented in Verilog HDL, but they do not show how it is used for building practical applications.

Mizutani et al. [21] proposed a counting sorting method of in-memory column-oriented data using multiple FPGAs tightly coupled by an optical network, and demonstrate that it is 5.3x faster than that using eight CPU servers. This study is very interesting because there are few reported research cases on distributed sorting using multiple FPGAs. However, as in Refs. [19], [20], their proposed method is also out of scope of research on how application developers can use it.

Although related studies reported in recent years could achieve a higher sorting performance compared to CPUs and GPUs, they did not consider how application developers use them and did not cover floating-point data, which defines the uniqueness of our work. And the validity of our research direction is supported by Ref. [22]. That study conducted a comparative analysis of OpenCL and RTL for sort-merge primitives on FPGA, and reported that while requiring comparable development effort, RTL implementations of critical primitives used in the algorithm achieve 4x better performance while using half as much of the FPGA resources. Based on the experimental results, the authors of Ref. [22] recommended a hybrid implementation, using RTL for performance-critical modules and OpenCL for other parts. Their approach is in the same direction as our design concept, where the RTL-tuned sorting engine is made available in the OpenCL environment for FPGA application development. In our future work, we will apply our sorting library to FPGA applications where sorting is a bottleneck and aim to improve the overall application performance.

8. Conclusion

To achieve both low FPGA programming cost and high sorting performance tuned by RTL, we had developed a sorting library that can be used with the OpenCL programming model for FPGA. In this paper, we proposed a sorting method for floating-point data that can be integrated into the sorting engine that is built by combining three hardware sorting algorithms. Our sorting library consumes at least twice as many hardware resources compared to the merge sort restructured for the OpenCL programming model for FPGA. However, its operating frequency is 1.08x higher and its sorting throughput is three orders of magnitude greater. We also derived the performance model for data sorting to assist application developers in determining the optimal configuration of the sorting engine while considering the number of target FPGA resources and the required sorting performance.

Our sorting library is currently designed with Intel FPGA SDK for OpenCL. However, since the sorting engine is implemented in Verilog HDL, it can work on Xilinx Vitis by changing the interface logic to external memory and Block RAM IP to those for Xilinx FPGAs. In addition, Intel is currently focusing on promoting the oneAPI programming model, and it is important to make our sorting library available in its toolkit. Similar to the Intel FPGA SDK for OpenCL, the oneAPI toolkit also supports the feature of using RTL modules as libraries, making it possible to use our sorting library in the oneAPI environment. We will perform these tasks in the future and make the source code available in the same GitHub repository (https://github.com/ac2-prod/fpga_sort) as the sorting library proposed in this study.

Acknowledgments This work was supported in part by the "Next Generation High-Performance Computing Infrastructures and Applications R&D Program" (Development of Computing Communication Unified Supercomputer in Next Generation) of MEXT. This research was also supported in part by the Multidisciplinary Cooperative Research Program in CCS, University of Tsukuba, and JSPS KAKENHI Grant Number 19K20276. We also thank the Intel University Program for providing hardware and software.

References

[1] Fujita, N., Kobayashi, R., Yamaguchi, Y., Ueno, T., Sano, K. and Boku, T.: Performance Evaluation of Pipelined Communication Combined with Computation in OpenCL Programming on FPGA, *2020 IEEE International Parallel and Distributed Processing Symposium Workshops* (*IPDPSW*), pp.450–459 (online), DOI: 10.1109/IPDPSW50202.2020.00083 (2020).

[2] Gorlani, P., Kenter, T. and Plessl, C.: OpenCL Implementation of Cannon's Matrix Multiplication Algorithm on Intel

Stratix 10 FPGAs, *2019 International Conference on Field-Programmable Technology* (*ICFPT*), pp.99–107 (online), DOI: 10.1109/ICFPT47387.2019.00020 (2019).
[3] Kobayashi, R., Miura, K., Fujita, N., Boku, T. and Amagasa, T.: A Sorting Library for FPGA Implementation in OpenCL Programming, *Proc. 11th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*, *HEART '21*, Association for Computing Machinery (online), DOI: 10.1145/3468044.3468054 (2021).
[4] Knuth, D.E.: *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*, Addison Wesley Longman Publishing Co., Inc. (1998).
[5] Koch, D. and Torresen, J.: FPGASort: A High Performance Sorting Architecture Exploiting Run-Time Reconfiguration on Fpgas for Large Problem Sorting, *Proc. 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, *FPGA '11*, pp.45–54, Association for Computing Machinery (online), DOI: 10.1145/1950413.1950427 (2011).
[6] Casper, J. and Olukotun, K.: Hardware Acceleration of Database Operations, *Proc. 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, *FPGA '14*, pp.151–160, Association for Computing Machinery (online), DOI: 10.1145/2554688.2554787 (2014).
[7] Saitoh, M., Elsayed, E.A., Chu, T.V., Mashimo, S. and Kise, K.: A High-Performance and Cost-Effective Hardware Merge Sorter without Feedback Datapath, *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines* (*FCCM*), pp.197–204 (online), DOI: 10.1109/FCCM.2018.00038 (2018).
[8] Saitoh, M. and Kise, K.: Very Massive Hardware Merge Sorter, *2018 International Conference on Field-Programmable Technology* (*FPT*), pp.86–93 (online), DOI: 10.1109/FPT.2018.00023 (2018).
[9] Usui, T., Van Chu, T. and Kise, K.: A Cost-Effective and Scalable Merge Sorter Tree on FPGAs, *2016 4th International Symposium on Computing and Networking* (*CANDAR*), pp.47–56 (online), DOI: 10.1109/CANDAR.2016.0023 (2016).
[10] Manev, K. and Koch, D.: Large Utility Sorting on FPGAs, *2018 International Conference on Field-Programmable Technology* (*FPT*), pp.334–337 (online), DOI: 10.1109/FPT.2018.00067 (2018).
[11] Batcher, K.E.: Sorting Networks and Their Applications, *Proc. Spring Joint Computer Conference*, *AFIPS '68* (*Spring*), pp.307–314, ACM (online), DOI: 10.1145/1468075.1468121 (1968).
[12] Mueller, R., Teubner, J. and Alonso, G.: Sorting Networks on FPGAs, *The VLDB Journal*, Vol.21, No.1, pp.1–23 (online), DOI: 10.1007/s00778-011-0232-z (2012).
[13] Intel FPGA SDK for OpenCL, available from ⟨https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/archives/aocl_programming_guide-19-4.pdf⟩.
[14] Kastner, R., Matai, J. and Neuendorffer, S.: Parallel Programming for FPGAs, ArXiv e-prints (2018).
[15] Cruz, I.F., Mendelzon, A.O. and Wood, P.T.: A Graphical Query Language Supporting Recursion, *Proc. 1987 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '87*, pp.323–330, Association for Computing Machinery (online), DOI: 10.1145/38713.38749 (1987).
[16] Satish, N., Kim, C., Chhugani, J., Nguyen, A.D., Lee, V.W., Kim, D. and Dubey, P.: Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort, *Proc. 2010 ACM SIGMOD International Conference on Management of Data*, *SIGMOD '10*, pp.351–362, Association for Computing Machinery (online), DOI: 10.1145/1807167.1807207 (2010).
[17] Dorobisz, A., Kotwica, M., Niemiec, J., Kobzar, O., Bohdan, A. and Wiatr, K.: The Impact of Particle Sorting on Particle-In-Cell Simulation Performance, *Parallel Processing and Applied Mathematics*, Wyrzykowski, R., Dongarra, J., Deelman, E. and Karczewski, K. (Eds.), pp.156–165, Springer International Publishing (2018).
[18] Kobayashi, R., Fujita, N., Yamaguchi, Y., Boku, T., Yoshikawa, K., Abe, M. and Umemura, M.: Multi-Hybrid Accelerated Simulation by GPU and FPGA on Radiative Transfer Simulation in Astrophysics, *Journal of Information Processing*, Vol.28, pp.1073–1089 (online), DOI: 10.2197/ipsjjip.28.1073 (2020).
[19] Chen, H., Madaminov, S., Ferdman, M. and Milder, P.: FPGA-Accelerated Samplesort for Large Data Sets, *Proc. 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, *FPGA '20*, pp.222–232, Association for Computing Machinery (online), DOI: 10.1145/3373087.3375304 (2020).
[20] Samardzic, N., Qiao, W., Aggarwal, V., Chang, M.-C.F. and Cong, J.: Bonsai: High-Performance Adaptive Merge Tree Sorting, *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture* (*ISCA*), pp.282–294 (online), DOI: 10.1109/ISCA45697.2020.00033 (2020).
[21] Mizutani, K., Yamaguchi, H., Urino, Y. and Koibuchi, M.: Accelerating Parallel Sort on Tightly-Coupled FPGAs enabled by Onboard Si-Photonics Transceivers, *2021 Optical Fiber Communications Conference and Exhibition* (*OFC*), pp.1–3 (2021).
[22] Moghaddamfar, M., Färber, C., Lehner, W. and May, N.: Comparative Analysis of OpenCL and RTL for Sort-Merge Primitives on FPGA, *Proc. 16th International Workshop on Data Management on New Hardware*, *DaMoN '20*, Association for Computing Machinery (online), DOI: 10.1145/3399666.3399897 (2020).

Ryohei Kobayashi received his Ph.D. degree from Tokyo Institute of Technology, Japan in 2016. He is an assistant professor of Center for Computational Sciences, University of Tsukuba, Japan. His research interests include GPU–FPGA-accelerated computing and FPGA systems for high performance computing. He is a member of the IPSJ, IEICE, IEEE, and ACM.

Kento Miura received B.E. and M.E. from the Collage of Information Science and the Department of Computer Science, University of Tsukuba in 2019 and 2021, respectively. Since 2021, he has been with Rakuten, Inc.

Norihisa Fujita received Ph.D. degrees (Doctor of Engineering) from Graduate School of Science and Technology, University of Tsukuba in 2016. From 2016 until 2019, he was a post-doctoral researcher in the center. Since 2019, he is an assistant professor in the center. His research interests are accelerators in high-performance computing and interconnection in HPC systems. He is a member of IPSJ.

Taisuke Boku received his M.S. and Ph.D. degrees from the department of electrical engineering, faculty of science and technology, Keio University. He joined to Center for Computational Sciences (former Center for Computational Physics) at University of Tsukuba where he is currently the director and the HPC division leader. He has been working there more than 25 years for HPC system architecture, system software, and performance evaluation on various scientific applications. In these years, he has been playing the central role of system development on CP-PACS (ranked as number one in TOP500 in 1996), FIRST, PACS-CS, HA-PACS and Cygnus as the representative supercomputers in Japan. He is currently the Director of Center for Computational Sciences, University of Tsukuba, and the Vice Director of JCAHPC (Joint Center for Advanced HPC) which is a joint organization by University of Tsukuba and the University of Tokyo. He is also the President of HPCI Consortium Japan from 2020. He received ACM Gordon Bell Prize in 2011, Paper Awards at IPSJ in 2004 and 2005, Best Paper Award at HPC Symposium of IPSJ in 2003 and 2016, Best Paper Award at HEART Symposium in 2014, and SCA Asia HPC Leadership Award in 2020.

Toshiyuki Amagasa received B.E., M.E., and Ph.D. from the Department of Computer Science, Gunma University in 1994, 1996, and 1999, respectively. He is currently a full professor at the Center for Computational Sciences, University of Tsukuba. His research interests cover database systems, data mining, and database application in scientific domains. He is a senior mpplicationember of IPSJ, IEICE, and IEEE, a board member of DBSJ, and a member of ACM.