Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2018

MSc THESIS

FPGA-Based High Throughput Merge Sorter

Xianwei Zeng

CE-MS-2018-02

Abstract

As database systems have shifted from disk-based to in-memory, and the scale of the database in big data analysis increases significantly, the workloads analyzing huge datasets are growing. Adopting FPGAs as hardware accelerators improves the flexibility, parallelism and power consumption versus CPU-only systems. The accelerators are also required to keep up with high memory bandwidth provided by advanced memory technologies and new interconnect interfaces. Sorting is the most fundamental database operation. In multiple-pass merge sorting, the final pass of the merge operation requires significant throughput performance to keep up with the high memory bandwidth. We study the state-of-the-art hardware-based sorters and present an analysis of our own design. In this thesis, we present an FPGA-based odd-even merge sorter which features throughput of 27.18 GB/s when merging 4 streams. Our design also presents stable throughput performance when the number of input streams is increased due to its high degree of parallelism. Thanks to such a generic design, the odd-even merge sorter does not suffer throughput drop for skewed data distributions and presents constant performance over various kinds of input distributions.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

FPGA-Based High Throughput Merge Sorter

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL ENGINEERING

by

Xianwei Zeng
born in GUANGDONG, CHINA

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

FPGA-Based High Throughput Merge Sorter

by Xianwei Zeng

Abstract

As database systems have shifted from disk-based to in-memory, and the scale of the database in big data analysis increases significantly, the workloads analyzing huge datasets are growing. Adopting FPGAs as hardware accelerators improves the flexibility, parallelism and power consumption versus CPU-only systems. The accelerators are also required to keep up with high memory bandwidth provided by advanced memory technologies and new interconnect interfaces. Sorting is the most fundamental database operation. In multiple-pass merge sorting, the final pass of the merge operation requires significant throughput performance to keep up with the high memory bandwidth. We study the state-of-the-art hardware-based sorters and present an analysis of our own design. In this thesis, we present an FPGA-based odd-even merge sorter which features throughput of 27.18 GB/s when merging 4 streams. Our design also presents stable throughput performance when the number of input streams is increased due to its high degree of parallelism. Thanks to such a generic design, the odd-even merge sorter does not suffer throughput drop for skewed data distributions and presents constant performance over various kinds of input distributions.

Laboratory : Computer Engineering
Codenumber : CE-MS-2018-02

Committee Members :

Advisor:	Prof. Dr. H. P. Hofstee, CE, TU Delft
Chairperson:	Prof. Dr. H. P. Hofstee, CE, TU Delft
Member:	Dr. Ir. R.F. Remis, CAS, TU Delft
Member:	Prof. Dr. Ir. A.J.H. Hidders, CS, VUB

Dedicated to my family and friends

Contents

List of Figures

List of Tables

List of Acronyms

AFU Acceleration Function Unit

FPGA Field-Programmable Gate Array

HBM High Bandwidth Memory

GPU Graphics Processing Unit

CPU Central Processing Unit

OpenCAPI Open Coherent Accelerator Processor Interface

PCIe Peripheral Component Interconnect Express

DMA Direct Memory Access

MUX Multiplexer

MSD Most Significant Digit

LSD Least Significant Digit

Acknowledgements

I want to express my sincere gratitude to my supervisor Prof. Peter Hofstee, who welcomed me to work on my MSc research under his guidance since January 2017, for his patience, enthusiasm, and immense knowledge. He kindly guided me into the field of computer architecture and database acceleration and offered many helpful suggestions when I encountered problems. Most thankfully, Peter even spent much of his spare time to have meetings with us and proofread this dissertation.

I thank my daily supervisor, Jian Fang. Over the last year, he has always been willing to help me with the project. He also taught me a proper way to demonstrate problems and to think independently. His passion for academic research has also inspired me a lot.

I thank the committee members, Rob Remis and Jan Hidders, for reviewing this dissertation carefully. I thank Jinho Lee for his kind support at many of our weekly meetings. I am also thankful to Yvo Mulder for his good illustration of the multi-stream buffering interface.

I thank Kangli Huang and Yang Qiao. We helped each other over the master study period in the Netherlands. I have benefited a lot from our discussions on both the courses and the master thesis. I also appreciate the spare time we spent together.

I thank Yixin Shi, Xiaoran Li, Jiayi Wang, Zhebin Hu, Xuefei You, Yang Liu, Mengyu Zhang, Xin Zhan, Jiang Gong, Bo Jiang and He Zhang for all the fun we have had over the last two years.

I thank Wuyin for her love and encouragement when I was depressed. I thank my parents and sister for their love, for their support, and for their inspiring me to pursue an MSc degree overseas.

Xianwei Zeng
Delft, The Netherlands
January 23, 2018

1 Introduction

1.1 Motivation

In big data analysis, large-scale database systems are widely used and they often occupy a huge number of database tuples, for example, several terabytes. The large-scale database systems require strong computation ability which is now normally achieved by highly parallel multi-core processing systems. Scalability of multiple CPUs is limited due to the great penalty of communication cost and bus resource utilization. A technology trend is to investigate heterogeneous systems to perform the acceleration.

Disk-based database systems used to rely on disk performance greatly and I/O interconnect. Now due to the increasing capacities of main memory, the available amount of main memory is often large enough to contain the entire large-scale database. Thus, instead of relying on I/O performance, database acceleration nowadays is bounded by memory and its interconnect.

Apart from the growing capacitance, the available memory bandwidth on a chip also rises. General purpose processing units are not able to fully utilize all the bandwidth that a chip can support. A significant advancement is the introduction of Open Coherent Accelerator Processor Interface (OpenCAPI)[4] which can provide a high bandwidth up to 25Gb/s per lane. More importantly, the total OpenCAPI bandwidth on a socket rivals that of main memory on a socket.

Sorting is one of the most fundamental operations in database systems. A sorting algorithm is to arrange a sequence of tuples in a certain order, descending or ascending. A tuple in database normally consists of a key and a payload, and the sorting operation aims to arrange a tuple according to the key value. An effective sorting algorithm can greatly contribute to the optimization of many complex applications such as search and join which require their input to be sorted sequences.

A hardware-based sorting algorithm often features high parallelism and high throughput. Instead of serial sorting, which produces sorted tuples one after another and takes $O(nlogn)$ to accomplish, parallel sorting can produce several tuples at a time. Some popular parallel sorting algorithms are a sorting network, sample sort, radix sort, etc. They all have the potential to feature remarkable performance on parallel architectures like FPGA.

As a design choice with low power consumption and high parallelism, dedicated hardware like FPGA is often used to accelerate computing tasks due to their efficient computing ability and flexibility as shown in Fig. 1.1 [1].

A significant challenge we are currently facing when designing an acceleration function unit (AFU), is to deal with large-scale in-memory databases and keep up with the high bandwidth expected of an in-memory database. In this thesis, we have provided a

[1] Source: Bob Broderson, Berkeley Wireless group.

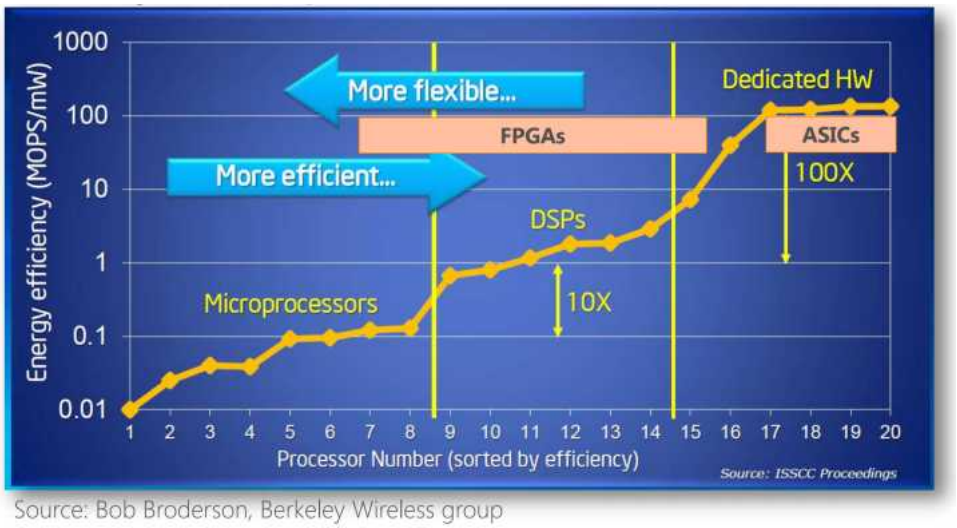

Figure 1.1: The efficiency and flexibility of different architectures.

design of the AFU that performs merge sort algorithm on an FPGA platform. The work described in this thesis is part of our Database group project. One of the other projects aims at feeding the high-bandwidth FPGA accelerators with a designed AFU protocol layer[9]. Another two projects involve different database operations such as hash-join and decompression[10, 11].

This chapter serves as the introduction of this thesis. Section 1.2 states the main contributions of this thesis while section 1.3 lists the outline of the rest part of this dissertation.

1.2 Thesis Contribution

The goal of this thesis is to implement an efficient acceleration function unit (AFU) to perform high-performance merge sort on FPGA platform, with a focus on achieving high throughput, large scale in-memory handling of datasets and effective resource utilization.

The main contributions of this thesis include:

- We survey the state-of-art FPGA-based acceleration on sort algorithms.
- We analysis design choices for parallel sort algorithm to achieve high throughput and to deal with the throughput drop upon skewed dataset distributions.
- We present the architecture of an FPGA-based odd-even merge sorter which can ensure stable high throughput while handling multiple streams to-be-merged simultaneously.
- We present the validation of our odd-even merge sorter on an FPGA platform*.
- We analyze the performance of our prototype and explore the trade-off between resource usage and bandwidth requirements.

- We explore the scalability of our prototype and compare that with the state-of-art regarding the number of streams and individual tuple size.

1.3 Thesis Outline

In chapter 1, we present the motivations for this thesis. We also provide a brief introduction of the technology trends in database hardware acceleration, the interconnect interface and sorting algorithms. The thesis contributions are also shown here.

In chapter 2 we give an essential explanation of the background. Database systems and some popular operations are introduced. The advantages of FPGA as a good design choice for accelerators are also extensively explained. We also present some existing high-bandwidth interconnect technologies, especially OpenCAPI. We mainly focus on the interface composition and bandwidth performance of OpenCAPI. More importantly, we present several of the most popular sorting algorithms which can be implemented on hardware accelerators. Both their principles and hardware performance are presented and compared.

In chapter 3 we focus on the hardware-based sorting algorithms. We present the performance metrics of a single sort engine: problem size, latency and throughput. How these parameters interact with each other and their effect on the performance of hardware-based sorter are also discussed. We also target on the bottleneck of merging multiple sorted streams when dealing with extremely large problem sizes. By adopting an HBM, we provide a complete sketch of multiple pass merge sorter and present the benefit of applying an HBM on AFU design. The total number of passes required, the capacity of on-chip buffers, and the number of trips to main memory the data need to make are also illustrated. More importantly, we provide the essential principle for a multiple-input merge unit by introducing the concept of feedback elements. We also present an analysis based on the initial dataset distribution. Skewed data distributions can result in significant performance drop and thus a merge sorter with stable throughput performance is needed.

In chapter 4 we propose the odd-even merge sorter. We start with setting the throughput target and the corresponding parameters of our design. We explain the overall principle of our design by providing an architectural overview of how the odd-even merge sorter interacts with its environment. We then illustrate the main data path of this merge sorter and its behaviors in odd and even cycles. By introducing the definition of odd and even cycles, we present the main components of this design and look at how they respond to the alternatively changing cycle status. When we come to the swap unit, we compare the traditional sorting network and our tag sorting network on both working principles and latency performance. More importantly, the design choice for the lowest swap unit is also illustrated.

In chapter 5 we present the validation of our proposal. The simulation and synthesis platform are introduced, as well as the target FPGA chip. The simulation results demonstrate the behavior of our odd-even merge sorter cycle by cycle. The waveform shows the exact cycles when the engine is initialized and when the output sequence is effective and ready to be collected. The performance evaluation on the odd-even merge sorter is based on three metrics: resource utilization, frequency and throughput. We

present several synthesis results showing the performance metrics versus different numbers of input streams. The evaluation results indicate stable throughput performance and flexible scaling of our odd-even merge sorter.

In chapter 6 we present our conclusions and illustarte the future work.

2 Background

In this chapter, we give a brief introduction to sort algorithm and the hardware acceleration of it. Section 2.1 introduces the database and presents some accelerators that are widely studied and used. The high bandwidth interconnection plays an important role in hardware acceleration applications and is introduced in section 2.2. In section 2.3.1, some popular sorting algorithms are also illustrated. We talk about their operating functions and hardware performance.

2.1 Database Acceleration

2.1.1 Database

A database is a collection of data where data is organized, stored and managed inside a storage device. The data within a database system can be shared, analyzed and operated. Various applications take advantage of various kinds of database systems from simple relational databases to large-scale warehouse systems. Database systems are designed typically to support different operation upon the data.

A relational database is a popular kind of database that is built on the relational model of data[12]. It consists of a set of relations from which data can be accessed or reassembled in many different ways without having to reorganize the database tables. A relation, or called as a table, is a set of tuples that represents a single object and information about this object.

Applications usually access datasets by specifying queries using operations such as *select* to identify desired tuples, and *join* to combine multiple relations together. Within a table, each column corresponds to an attribute or field, the data categories within each row represent tuples or records.

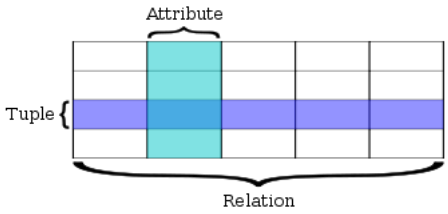

Figure 2.1: Relational database terminology[1].

2.1.2 Hardware Acceleration

Hardware acceleration is to perform some certain functions or algorithms more efficiently than in general purpose processing units, programmed by software. Remarkable applications of hardware acceleration are the use of GPUs to speed up the processing of graphs[13], and FPGA arrays utilized in Microsoft's Bing search engine[14]. This hardware that is an individual unit different from CPU is called the accelerator. GPUs and FPGAs are widely used accelerators, and studies of them are still an active area of research.

Traditional single-core or even multi-core CPU systems are less effective when the computing tasks involve a high level of parallelism. Limited by its Von Neumann architecture, a CPU sequentially performs instructions and relies on instruction fetch. The benefit of hardware accelerators is that they allow more freedom in instruction level parallelism and is potential for higher concurrency than traditional homogeneous systems. Accelerators are free from the sophisticated control logic and generalized data paths, so the overhead of each instruction is greatly reduced.

GPU stands for "graphics processing unit". A GPU is very efficient at image processing due to its highly parallel structure. With thousands of cores integrated inside a GPU, it can perform massively parallel tasks such as floating-point operations at a relatively small granularity.

FPGA stands for the field-programmable gate array. It is an integrated circuit with re-configurable resources such as logic blocks, I/O cells and interconnect resources. The logic blocks can be programmed to simple logic units, like AND and NOR, and are also able to build complex function units to perform instructions with a high degree of concurrency. FPGAs also contain memory resources such as flip-flops and block RAMs.

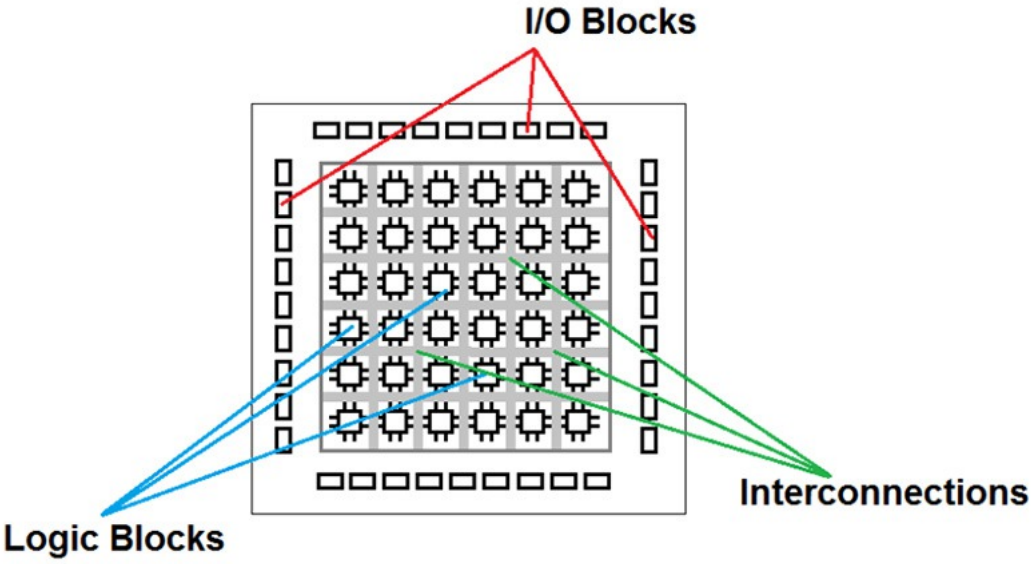

Figure 2.2: Architectural overview of an FPGA[2]

Increasingly a choice for accelerators, one of the FPGA's advantages is the flexibility due to its reconfigurable resources. Although the general-purpose GPU can handle various acceleration applications, an FPGA can be handily reconfigured into dedicated hardware for a specific application. It can save some unnecessary logic or instructions compared with that of GPU. Consequently, an FPGA is more power-efficient than a GPU as shown in Fig. 2.3.

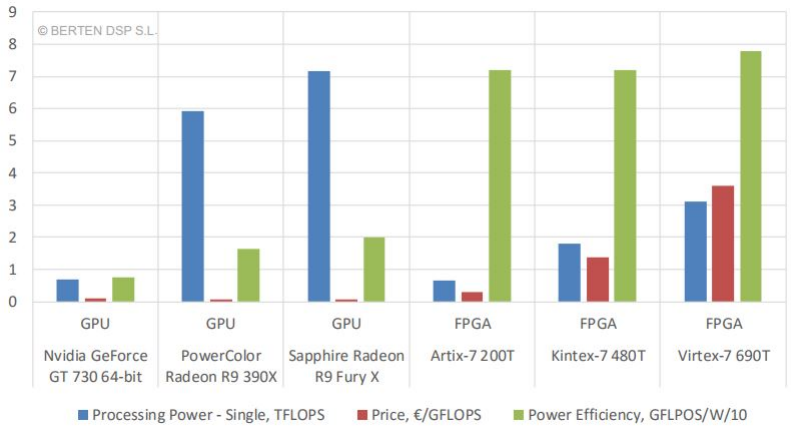

Figure 2.3: GPU vs FPGA performance comparison.[3]

Latency is another important issue for many applications, such as network synchronization and control applications. Take the Microsoft's Bing search engine[14] as an example, to return search results as soon as possible we have to reduce the latency for every instruction. Due to its pipelining and dedicated design, FPGAs can process data within a couple of cycles. It even gets boosted performance when applying high bandwidth interconnections such as the OpenCAPI interface (illustrated in section 2.2.1).

We now look at the FPGA performance in computation-intensive applications, such as matrix operation, compress, sort, etc. FPGAs can handle a much higher workload than CPUs when configured with deep pipelines and configured to fetch data in parallel. For bandwidth-intensive applications, FPGAs benefit much more because its interconnection can be configured to meet a throughput over 100GB/s[15].

Although ASICs perform best in throughput, latency and power consumption, an ASIC is often not an appropriate alternative for hardware acceleration. One important reason is that it takes more time and more cost to develop an ASIC which is not acceptable in a data center where the computing tasks are changing rapidly and the flexibility to apply hardware to many different tasks is key to achieving good utilization.

2.2 High Bandwidth Connection

High bandwidth connections are required to provide both high bandwidth and low latency to meet the requirement of high-performance computing systems (HPCS).

2.2.1 OpenCAPI

OpenCAPI[4] stands for the Open Coherent Accelerator Processor Interface and is a new bus stand. It is designed for heterogeneous systems to leverage the advantages of accelerators. Such an architecture allows highly effective connections between processors and accelerators while promising greatly improved coherent memory access by placing the accelerators in user-level to provide fine-grained interaction.

2.2.2 Interface Composition

Fig. 2.2.1 shows an overview of the OpenCAPI protocol stack with multiple layers from the host processor side to OpenCAPI device side. In our case, we regard the OpenCAPI device as an acceleration function unit (AFU). Transaction Layer(TL) can be seen on both the host and accelerator side and is responsible for the conversion between requests and commands. In host side, TL performs the interpretation from host specific protocol requests into TL commands and the inverse operation. Similarly, Transaction Layer on accelerator side (TLx) performs the interpretation of AFU protocol requests into TLx commands and the inverse operation.

The Physical Layers, Data Link Layers, and Transaction Layers are established on both host and AFU side in a symmetric structure, and the OpenCAPI presets them.

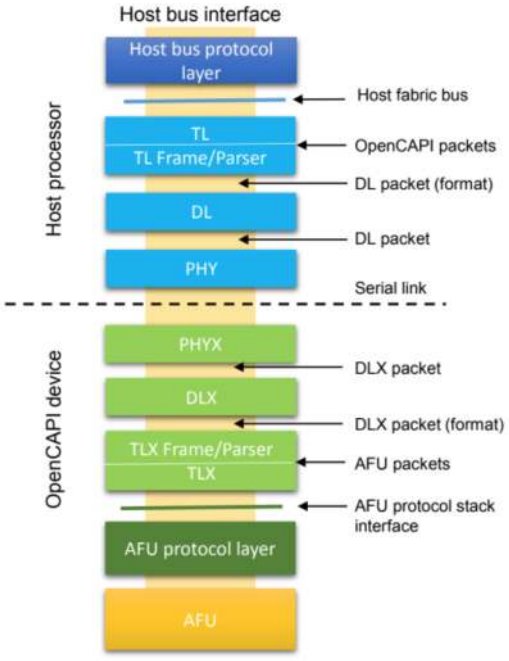

Figure 2.4: An overview of the OpenCAPI protocol stack[4]

2.2.3 Memory Bandwidth Performance

The Data Link Layer shown in Fig. 2.2.1 can support a maximum bandwidth of 25GB/s if utilizing 8 lanes. The available throughput is much higher than that in PCIe gen 3 which is around 8 GB/s. Also, 32 lanes in total can be supported by POWER9 architecture, of which the bandwidth can reach a higher performance[16].

Apart from accelerators, other devices such as coherent network controllers, advanced memory and coherent storage controllers in heterogeneous systems are also enabled by OpenCAPI. Among them, the advanced memory access is guaranteed by read/write or user-level DMA semantics with very low latency. The circuit complexity of accelerators is also reduced in OpenCAPI by placing more of the complexity on the host side.

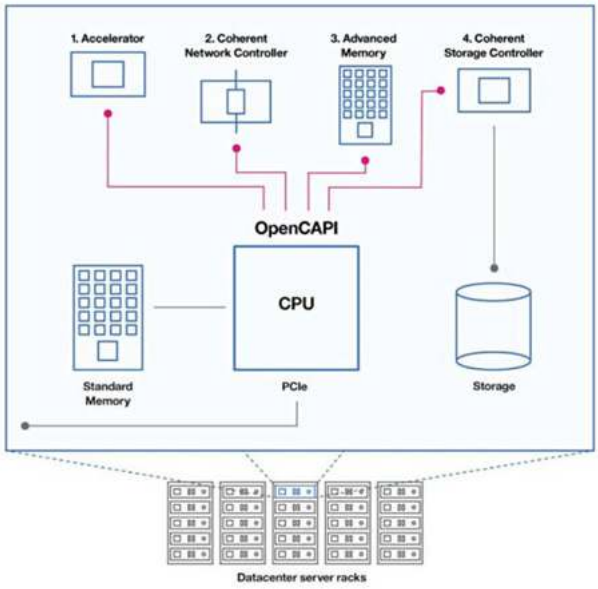

Figure 2.5: Server internal overview[4]

2.3 Sort Algorithm

2.3.1 Stability and Adaptation of Sort Algorithms.

There are two important characteristics of sort algorithms. One is stability, and the other is the adaptability to the input dataset. A sort algorithm is stable when the produced sorted lists preserve the initial relative order of identical values. A tuple to be sorted usually consists of a key and a payload. The key is the reference we access and count on when we perform a sort operation upon several tuples while the payload represents the attached information that is not taken into consideration as part of the sort key. A conflict happens when several tuples carry with the same value of keys, and they will be regarded as identical tuples. Thus, apart from the key and payload of a tuple, we should also be aware of the initial order of the tuples. If several tuples appear to be identical,

the output order should be the same with the relative order of them as indicated in the input sequence. A stable sort algorithm is said to satisfy this property.

The adaptive sort algorithm is to be aware of the initial status, or the distribution of the dataset of interest, and take advantages of the initial distribution to perform corresponding processes. A sort algorithm is adaptive when the running time of it differs for inputs with different initial distributions. For example, if a sequence is partially sorted itself, an adaptive sort algorithm can sort it faster than a random sequence. Some smart and adaptive sort algorithms can sort a reversely sorted input in short running time by recognizing the ascending order in Fig. 2.6, but a non-adaptive sort algorithm can take much longer time to sort such a list.

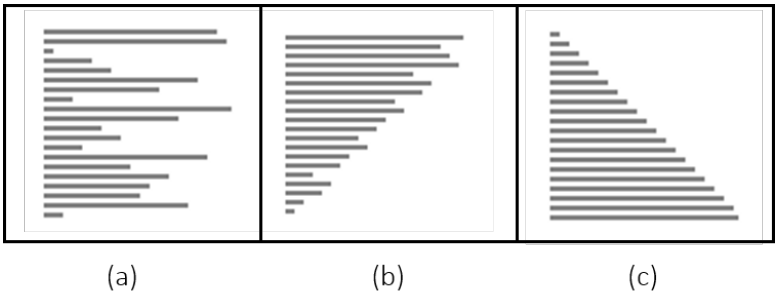

Figure 2.6: Random(a), partially sorted(b) and Reversed(c) distributions.

In following subsections, we present several popular sort algorithms from their function to their running performance. These sort algorithms have been widely studied for accelerators due to their flexibility in scaling and advantages in parallelism. The two characteristics mentioned above are stated along with their descriptions.

2.3.2 Radix Sort

The radix sort[17] starts with the reinterpretation of a k-bit key as a composition of many d-bit digits, given that d is smaller than k. A radix sort splits the k bits of keys into smaller d-bit digits to acquire a small enough radix $r = 2^d$. By iterating over all the keys, we can partition the entire data set into r buckets according to the unique value of the particular d-bit digits.

The interpretation of k-bit keys into d-bit digits can be performed by either from the most-significant (MSD) digit to the least-significant digit(LSD) or the inverse operation. We take the MSD as the starting digit in the following explanation.

The partition of all the keys into r distinct buckets is done by using a counting sort. The counting sort is performed to compute a histogram over the most-significant digit, which represents the number of keys over all the possible MSD values. Calculated from the histogram, a prefix sum indicates the required memory offsets for each of the r buckets and the number of keys should be placed into each r buckets. Every key within the same bucket shares the same MSD value.

After the first partition by the most-significant digit, the buckets resulted from the partition are either further partitioned into many sub-buckets by another counting sort, or to be sorted by applying a local sort. When presenting another partition, which

is another pass, the digit according to which the counting sort partitions the existing buckets into a set of sub-buckets is now shifted towards the LSD, normally by one. The partition pass is finished either until the current referring digit is the LSD, or the partitioned sub-buckets are small enough to be sorted out by a local sort.

A local sort here is actually to sort all the keys within a bucket within on-chip shared memory, which is greatly limited to the size of the key and the hardware resources. Thus, a local sort will only be performed when a bucket contains a small enough number of keys. An example applying the radix sort is shown in table 2.1.

Table 2.1: A radix sort example: sorting 16 keys of k=4 and d=2.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MSD	23	02	03	11	13	22	33	30	10	02	01	13	12	21	10	02
histogram	5	6	3	2												
prefix-sum	0	5	11	14												
	bucket 0					bucket 1						bucket 2			bucket 3	
Next digit	02	03	02	01	02	11	13	10	13	12	10	23	22	21	33	30
histogram	0	1	3	1		2	1	1	2							
prefix-sum	0	0	1	4		0	2	3	4							
	b1	b2			b3	b0		b1	b2	b3		local sort			local sort	
	01	02	02	02	03	10	10	11	12	13	13	21	22	23	30	33

It is easy to ensure stable sorting by preserving the initial offsets of each key before the first partition. However, this radix sort does not adapt to the initial status of inputs since the partition process shuffles the keys around.

2.3.3 Sample Sort

The sample sort normally relies on using a set of samples of the input data sets to partition them into unique sub-problems that can be sorted independently. The set of samples, also called the splitters, are randomly selected from the entire input data set. The $p-1$ splitters are then sorted into a sequence $S = \{s_1, s_2, ..., s_{p-1}\}$ which represents the distribution of key values since they are sampled from the entire data set. The more splitters we collect, the more information about the entire dataset we can know.

Once sorted, the sequence S defines p buckets, and their ranges are also indicated by the splitters. For example, bucket i has a range of $[s_i, s_{i+1})$. Having defined the p buckets, iterate over the input elements and place them into correct buckets, of which the ranges can cover the key values. Then the final step is to sort each of all the buckets.Each individual bucket is usually created for one individual processor so that each bucket can be sorted in parallel.

The performance of a sample sort depends greatly on the bucket size. A smaller bucket size can predict a relatively shorter running time while keeping the total number of input elements the same, but it also requires more sort engines for each bucket. Besides, variation in bucket sizes can lead to an unbalanced workload for each sort engine so that the sort engines have to be created for the worst case bucket size.

To obtain desired buckets, the splitters are expected to be large and representative enough. Oversampling technology[18] can be applied here to create a set of more effective

splitters by introducing an oversampling ratio which is taken into account when selecting the splitters.

Now the sample sort is often used as a partition method rather than a sorter since it has unstable bucket performance as illustrated above. [19] applies sample sort as part of its sort algorithm and utilizes it in distributed memory systems.

The sample sort is adaptive since it interprets the distribution of the inputs as a sequence of splitters. For uniform inputs, sample sort can perform small enough bucket sizes with small deviation when selecting splitters correctly, but it can be difficult to obtain the desired bucket sizes if the initial distribution of dataset is not easy to be sampled. Stable output can also be ensured by tagging the duplicates before they are sorted inside each bucket.

2.3.4 Sorting Network

The sorting network is a traditional sort algorithm that consists of multiple inputs and is comparator-based. The basic construction unit of a sorting network is a 2-input and 2-output comparator in Fig. 2.7. This unit compares the key values of these two inputs, and it collected output0 is the smaller one of the 2 inputs while output1 is the larger one. The outputs can be reversed in ascending order but we present descending order in the rest of the dissertation.

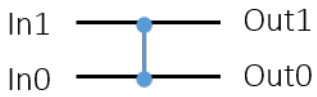

Figure 2.7: A typical 2-input and 2-output comparator.

We assume that the running time of each comparator is the same constant and the running time of one comparator is a unit time. In Fig. 2.8, each vertical line, C1, C2,..., C5 represents one comparator. The C1 and C2 in Fig. 2.8 (a) operate simultaneously so C1 and C2 can be considered as a depth of 1 with a total running time of one unit time. It is the same for C3 and C4. We call the combination of C1 and C2 as layer 1 and the combination of C3 and C4 as layer 2. Therefore, layer 1 and layer 2 are constructed in cascade mode. Another configuration of a sorting network with a depth of 3 is also shown in Fig. 2.8 (b).

We define the latency of a sorting network as the time for one specific input to travel from the first comparator layer to the last layer. It is determined by the largest number of comparators that any input has to pass through.

We distinguish sorting networks by the organization of comparators. A classical way to organize a sorting network is put forward by Batcher[20] which is a bitonic sort. It applies in the first step to transfer a n-number sequence in arbitrary distribution into a bitonic sequence with half in descending order and another half in ascending order. More discussion on bitonic sequences can be found in section4.4.2. The bitonic sequences obtained above will then be merged by $logn$ stages where each stage consists of shuffle, compare and unshuffle as shown in Fig. 2.9[5].

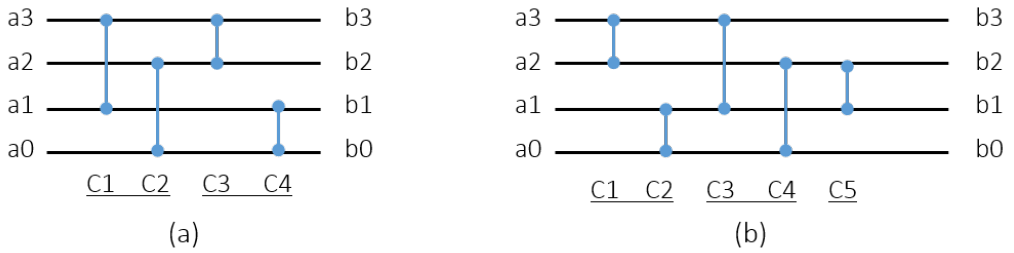

Figure 2.8: (a) A 4-input, 4-output sorting network with a depth of 2. (b) A 4-input, 4-output sorting network with a depth of 3.

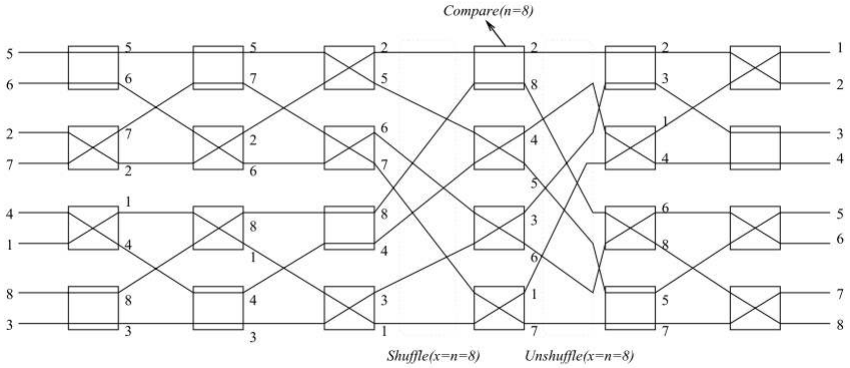

Figure 2.9: A Batcher's merge stages[5]

There are also many other ways to build a sorting network, such as a depth-optimal sorting network in [21] and odd-even merging network[22].

Its component comparators can guarantee the stability of a sorting network. If two inputs have identical keys, then comparator will not perform the swap operation and preserve output0/1 as input0/1. Sorting network algorithms can be viewed as adaptive sort algorithms since different configurations of the network can be selected upon the data distribution, but in general, sorting networks are created to handle arbitrary distributions.

2.3.5 Merge-Tree Sort

A merge-tree is a log-structure consisting of many leaves. In the beginning, one element is placed at each leaf. Two leaves meet each other, and the smaller element will slip to the edge and is placed at the top of another leaf. A set of leaves that operate simultaneously is called a level, and we name two leaves as a merge node. It can be predicted that for each level, the elements produced are the currently smallest.

An example is shown in Fig. 2.10. The input sequence is {1,3,4,2,7,6,9,5}, each of which is placed on the top leaves. It will then become {1,2,6,5} at level 2 and eventually {1} will be produced at the final node. One drawback of the tree-structure is only one element can be produced each time and only one can be fed into it as well.

The tree-structure is a very simple implementation of merge sort which aims to merge

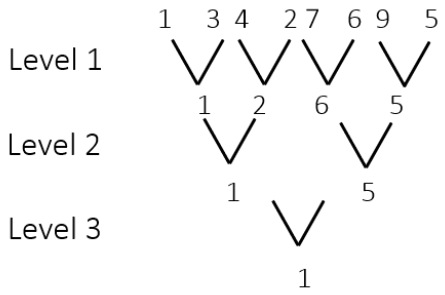

Figure 2.10: Log-structured merge tree

one or more pairs of sequences into one sequence. The performance of a tree-structure can be improved by elaborating the individual merge nodes[23]. The enhanced merge node performs complex merging of multiple elements by combining other sort algorithms together. Another way to improve the tree-structure is to enable high throughput at the top of each leaf[15]. It is a way to utilize high bandwidth to perform merge sort.

A discussion of the enhanced tree-structure is performed in section 3.4.2 with multiple outputs at the final node.

2.3.6 Hardware Performance of Prior Work

Although the hardware implementation of sorting algorithms can easily achieve performance several times faster than a software implementation, a hardware-based sorter is typically required to perform massively parallel data fetch as well as deliver both high bandwidth and stable throughput.

In [23] a parallel tree-structure merge sort is presented. It merges 32 input streams and reaches a maximum throughput at around 24.6 GB/s. It greatly increases the parallelism of FPGA-based merge sorters but the throughput is not stable and will suffer a significant drop especially in an extreme case when the data distribution among the 32 input streams is badly skewed.

Another unstable throughput performance can be seen in[24]. It also performs a sort algorithm to handle many input streams simultaneously by using a hybrid design of a merge tree and a bitonic merge network, but the merge nodes presented also suffer from throughput drop upon skewed distributions.

A stable throughput method is studied in [15] by establishing a high bandwidth sort merger to utilize the previous sort merge passes and enhance the throughput performance. A maximum throughput of 19.2 GB/s is obtained. The drawback here is that the design is not easy to scale for a much larger amount of streams.

In summary, the design specifications to satisfy are on both algorithm side and hardware performance side. The desired sort algorithm is expected to be stable upon duplicates and also able to handle the overhead upon various initial input distributions. When implemented on hardware, the resources should be configured and utilized effectively, and the presented dedicated architecture should perform advantages on parallelism, operating frequency, and throughput. The sort algorithm design choice and the implementation

are shown in Chapter 3 and Chapter 4, while the validation can be found in Chapter 5.

3 Hardware Merge Sorter

The previous chapter 2 introduced several popular sequential and parallel sort algorithms and the highlights in hardware acceleration. Chapter 3 presents the performance analysis of the merge sort algorithm in hardware. Section 3.1 illustrates the single-pass and multi-pass sorter. Section 3.2 provides a complete multi-pass sort proposal based on the utilization of high-bandwidth memory and analyzes the buffer usage to cover communication latency with memory. Section 3.3 illustrates some essential rules for a merge unit to merge multiple streams. Finally, in section 3.4, the relation between throughput performance and input randomization in hardware-based sorters is addressed.

3.1 Performance of Hardware-based Sorter

3.1.1 Problem Size and Overall Latency

The throughput of accelerators for in-memory database operations also indicates the interconnection bandwidth utilization. In the past when the bandwidth was low, a throughput of producing one sorted element per cycle, a few gigabytes per second, if performed by a general purpose processing unit, was enough to keep up with the memory bandwidth. However, because the interconnect technologies can support significantly higher bandwidth, the design of hardware-based database operations should be revisited and more attention should be put on throughput performance.

The overall throughput is the volume of sorted data produced within a unit amount of time. In equation 3.1, T represents the overall throughput. D is the problem size, the total amount of data to be sorted and t is the overall latency, the time from the first tuple leaves the memory to the time all tuples are sorted.

$$T = \frac{D}{t} \tag{3.1}$$

In equation 3.2, t_{FPGA} is the execution time spent on the FPGA accelerator, and t_{memory} is the time spent on trips to access external host memory via the interconnect. $t_{overlap}$ is the overlapping amount of latency when the FPGA and the host memory operate concurrently.

$$t_{overall} = t_{FPGA} + t_{memory} - t_{overlap} \tag{3.2}$$

Let d be the width of a sort engine. It represents the maximum number of input ports of a sort engine at a time. In an 8-input sorting network case as an example, d is 8. We can then estimate the execution time of a sort engine by equation 3.3 where $t_{average}$ is the average execution time for one iteration.

$$t_{FPGA} = \log_d D \times t_{average} \tag{3.3}$$

We can now see the impact of a large problem size on the overall throughput performance from the equations above. For a small problem size that can fit into the FPGA on-chip memory, a peak throughput can be achieved when t_{FPGA} dominates. As the problem size grows and exceeds the capacity of on-chip memory, extra trips to host memory are necessary which results in a large t_{memory}. The overall throughput achieved drops for large problem sizes. A plot from [6] indicates the trade-off between throughput and input size.

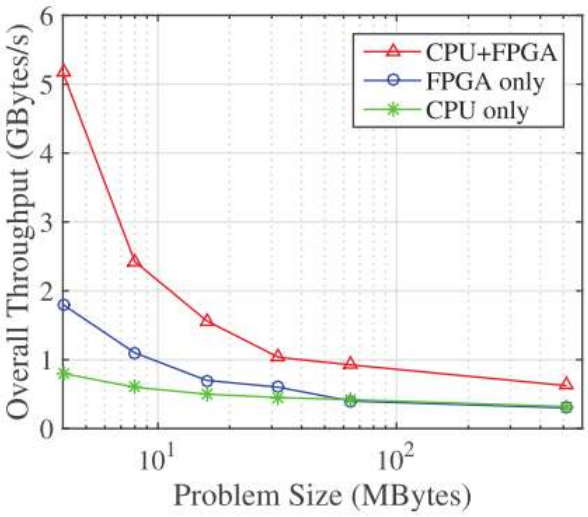

Figure 3.1: Throughput comparison for various input size [6]

To avoid significantly decreasing throughput, one way is to establish a strong sort engine with a large width to reduce the execution times. Another method is to partition a large-size problem into many small partitions, sort them in parallel and merge them together. Studies[19, 22, 25] employ different methods to divide the large-scale dataset into many partitions in a smaller size and sort them using individual sort engine to avoid performance drop in throughput.

3.1.2 Throughput of A Single Engine

We now consider the throughput performance of an individual sort engine. A straightforward calculation for the throughput of a single engine is shown in equation 3.4. E is the number of sorted elements produced every cycle, s is the element size, and F is the operating frequency of the FPGA accelerator.

$$T_{engine} = E \times s \times F \tag{3.4}$$

In this thesis, we focus on improving E, since increasing the number of output elements per cycle is attracting great research interest recently. It is an effective way to improve the throughput performance of the hardware-based sorter. Multiple elements can be processed simultaneously thanks to the great flexibility and parallelism of FPGAs. Thus we can explore algorithms with the potential to feature massively parallel processing.

From a generic perspective, most of the sort algorithms for large-scale problem size have to perform a merge operation after sorting many partitions. We call them the hardware merge sorter, which performs the merge operation upon two or more already sorted streams.

3.1.3 Merging Multiple Streams

A typical merge engine to perform a merging operation is shown in Fig. 3.2 where N streams are merged into one larger stream. In large-scale database systems with thousands of partitions, a N-stream merge engine is usually unable to read all the top elements from every sequence simultaneously. Therefore, either more merge engines are cascaded, or the merge process will take more iterations over the merge engine until the entire dataset is merged. The number of iterations is determined by the width N of the merge engine as $\log_N S$, where S is the total number of streams to be merged. Therefore, if we have a large merge engine, the number of iterations required grows more slowly versus the increasing problem size and allows larger datasets to be handled.

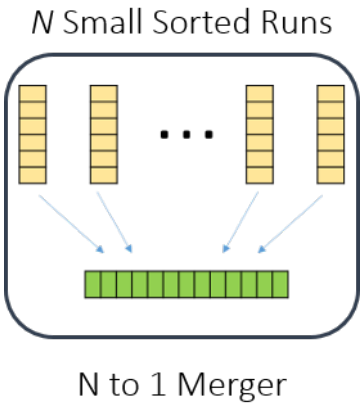

Figure 3.2: One-pass Merge Sorter, which merges several sorted runs in small problem size into one whole large run.

It takes more than one iteration over every partition to finish the merge. Therefore, multiple iterations (at least two) will be necessary when dealing with large datasets. Datasets will go through the sort or merge engine more than once. The first pass basically involves partitioning and sorting. Datasets can be partitioned by radix bits [17] or by sampling [26]. The sort engine here can be applied in various ways (in parallel or in serial) as long as it performs well enough to sort the partition fed to it.

Fig. 3.3 indicates the multi-pass merge engine design methodology from our work [7]. Fig. 3.3 (b) explains our thoughts for the multi-pass merge engine. We have more engines at each pass to work in parallel, which can reduce the number of iterations needed to access the memory and eventually save much time for each previous pass before the final pass. The produced sorted runs of each engine become larger at later passes.

In general, what we obtain after accomplishing the first pass are many data sequences in a small size that are sorted. It may take another pass to sort the data sequences or

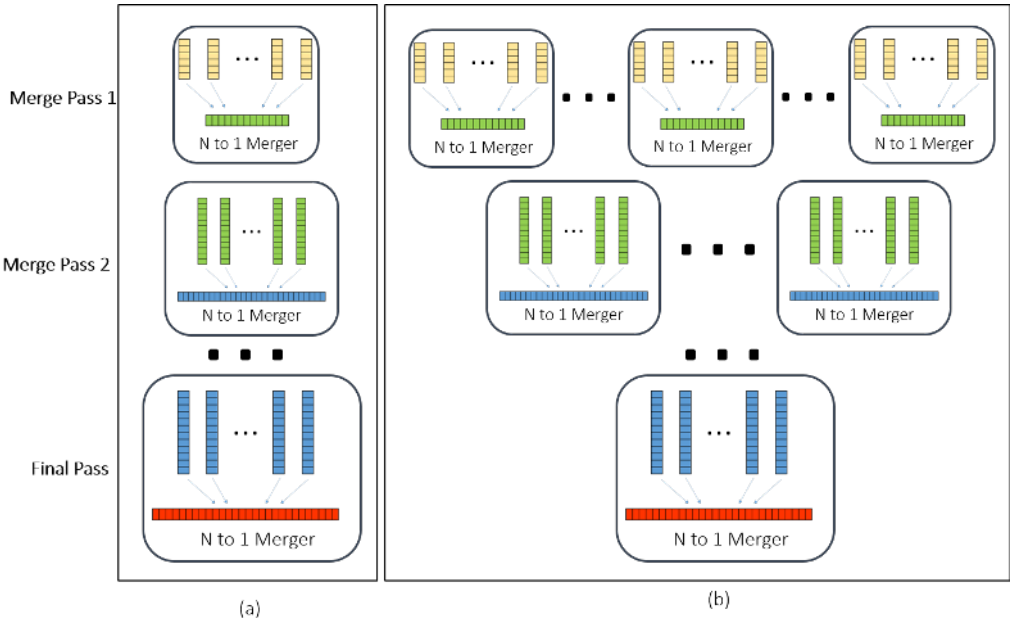

Figure 3.3: multi-pass merge sort from our work [7]. (a) explains how the multiple passes of a merge algorithm interact with each other (b) is our high performance multi-pass merger where each pass has multiple engines to handle more runs to save the execution time of previous passes before the final one.

even more passes depending on the ability of the merge engine. Therefore, the passes after the first pass consist of merging only. Several ordered sequences are merged into one large ordered sequence. Small sequences are combined into large streams. However, in both cases the final pass is doing the same job, that is merging N runs into one final run after which no further merging will be done. This indicates us that the final merge engine is now the bottleneck of the throughput performance. The final merge pass has to guarantee all the remaining streams are combined one single and final sorted stream. This requires a single stronger merge engine instead of more. The architecture and implementation of such a strong merge engine will be presented in later chapters.

3.2 Adopting HBM for High Performance Merge Sort

3.2.1 High-Bandwidth Memory

High-Bandwidth Memory (HBM) is a new type of memory chip with low power consumption and ultra-wide communication lanes. It is the new-generation memory solution for bandwidth-hungry applications. The most remarkable characteristic of HBM is the high bandwidth performance which is more than 3 times of that of a GDDR5. Figure 3.4 indicates that the first generation of HBM was announced with a bandwidth of 128GB/s in 2013 while the HBM2 was demonstrated with a 256GB/s bandwidth in 2016 [8].

Fig. 3.5 shows how the multi-pass merge sort interacts with the host memory. Given that the maximum available memory bandwidth is 25GB/s, every time we across a sort/

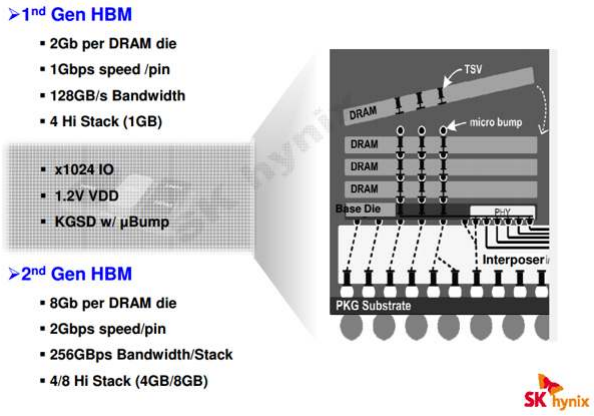

Figure 3.4: HBM overall specification from [8]

merge engine, a read and write operations are necessary to access the host memory in a certain latency. We can have this improved by adopting an HBM.

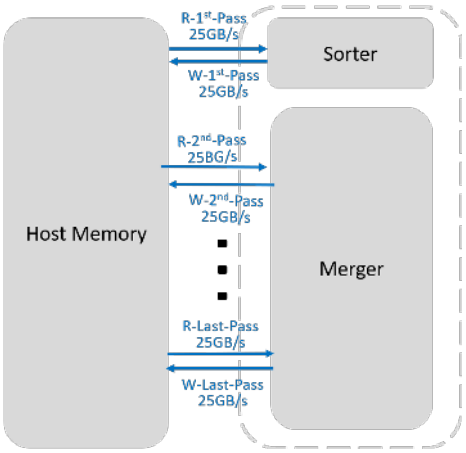

Figure 3.5: A Multiple pass merge sort interacting with the host memory.

A construction of multi-pass merge sorter involving the utilization of HBM is sketched in Fig. 3.6. Given that the unsorted datasets are stored in host memory, the sorter and merger are implemented together as an AFU, and the HBM is also integrated on the same chip where our AFU is located. The sketch presents a typical 5-pass merge sorter. As the OpenCAPI is introduced in section 2.2.1, the bandwidth between host memory and AFU is considered to be 25GB/s while the bandwidth between HBM and AFU is considered to be 200GB/s. In the 1st pass, we fetch data from host memory in small partitions, sort them and write to HBM as sorted runs. The merge starts from the second pass where the top elements (the smallest) are read from several runs and merged into a new but larger set of runs. Consequently, the third and fourth passes are similar merge operation with the second pass but the generated runs are getting larger. At the final pass, all the remaining runs have to be merged into one individual final run. All the runs are read with their top elements from the HBM, and the merge results are written

to the main memory. Therefore, since all these three intermediate passes are performed via the HBM but not the host memory, we can regard this whole structure as a one-pass design since we only have to read from and write to the host memory once, which are the very beginning and the final end.

In extremely large-scale cases when the storage room needed by the intermediate generated runs exceed the capacity of HBM, more passes through the host memory will be necessary. The HBM will not be able to hold all the intermediate passes. Therefore, we have to do a first pass as shown and then a final merge from host memory.

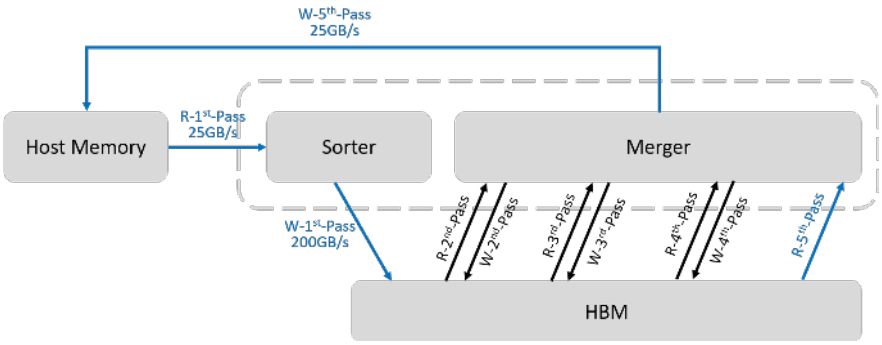

Figure 3.6: A multi-pass merge sorter with HBM utilized.

The read and write execution time reveal the advantages of HBM. In the first pass, we have a read speed of 25GB/s and a write speed to HBM of 200GB/s, so the dominant execution here is the read which is much slower than the write. This pass performs the same with and without HBM. Fortunately, things are different in later passes. The read and write in the second, third and fourth passes shown in Fig. 3.6 are all executed through the HBM at 200GB/s. Assuming that read and write are executed in parallel, these three intermediate passes are 8 times as fast as the 1st pass. By regarding the entire design as a one-pass design, we can consider that the latency of multiple memory access is greatly saved.

When it comes to the final pass, which consists of read from HBM and write to host memory, the overall execution time is determined by the write execution. To fully leverage the advantages of HBM, the throughput of the last pass is supposed to catch up with 25GB/s. The 25GB/s throughput requirement also indicates the performance of the final-pass merge engine is essentially significant.

3.2.2 Latency and Buffer

For a typical FPGA with a clock frequency of 200MHz, each cycle takes 5ns. We assume that the latency of data communication from main memory takes 1us and that from HBM takes 100ns. When performing in-memory database operations, the latency of fetching data from memory is longer than one FPGA cycle. Therefore, a compensation buffer is necessary to cover the latency from memory. The compensation buffer size can be obtained by equation 3.5. B is the on-chip buffer capacity, t_{memory} is the latency of memory access, s is the element size and E is the number of output elements of the

merge engine.

$$B = t_{memory} \times s \times E \tag{3.5}$$

The limited capacity of on-chip buffer is also one of the reasons why multiple passes are preferred when sorting large-scale dataset. If we have the merging done within one pass, extra buffers are needed. The extra buffers are needed to store the output of one stage and then provide input for the next stage. The previous stage merges small sorted partitions and generates larger sorted partitions. Given that the first stage merges x inputs and the second merge stage handle y inputs, the buffer size required at later passes increases by a factor of xy. The buffer here is to store y times output of the partitions from the previous merge stage. Thus, the size of such an intermediate buffer is determined by the overall consumed data size of the next merge stage.

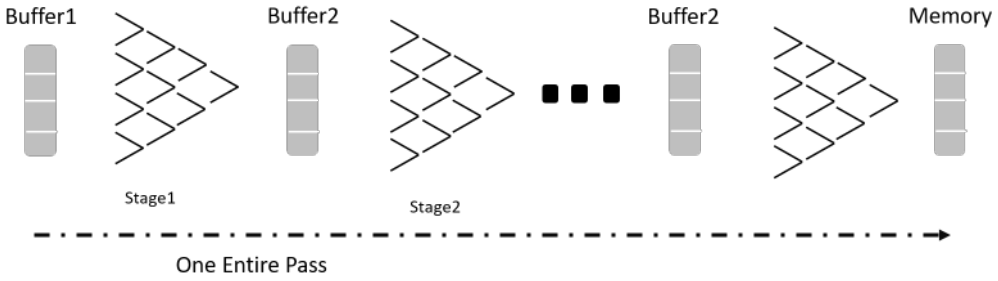

Figure 3.7: Buffer1 is for latency compensation and buffer2 to store intermediate partitions.

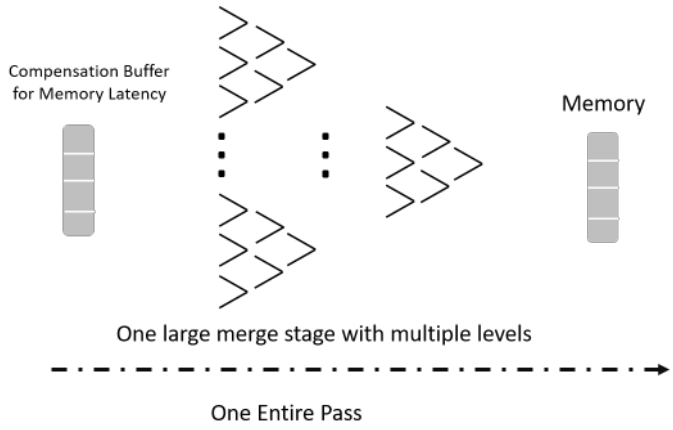

Figure 3.8: Apply a larger compensation buffer with a large merge unit

A simple assumption is that with a 2-stage merge, shown in fig. 3.7 where each stage consumes 4 inputs, and the throughput of the tree is only one element per cycle. Considering the case that with 8B key and 8B payload, the buffer size required at the 1st/2nd/3rd/4th/5th pass is 256B/4KB/64KB/1MB/16MB, which indicates that the intermediate buffer size determines how much input each stage can deal with.

For the sake of dealing with the growing buffer demands brought by large problem sizes, it is more effective to implement one single large merge stage without establishing

several stages as in fig 3.8. Getting rid of intermediate buffer requires more buffer to compensate the latency.

3.3 Merge Unit

As indicated in the previous section a merge unit combines two or more sorted sequences into one single sorted sequence and is usually performed as a later processing step after sorting small partitions. Studies on merge usually focus on two significant performance improvements: to handle more streams[23] or to handle more elements per stream[15]. Both of these approaches have to deal with a general issue, how to fetch new elements in the streaming pattern.

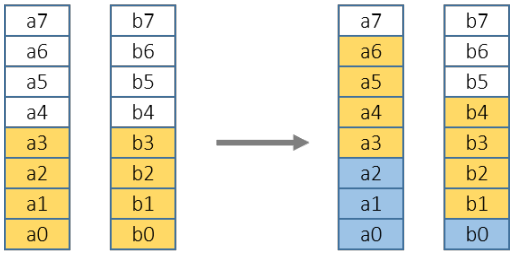

Figure 3.9: Input columns for 2-stream merging. The right arrow means one cycle later. Yellow blocks in left columns are input while the yellow one in right are the input for next cycle. Blue blocks are what have been output.

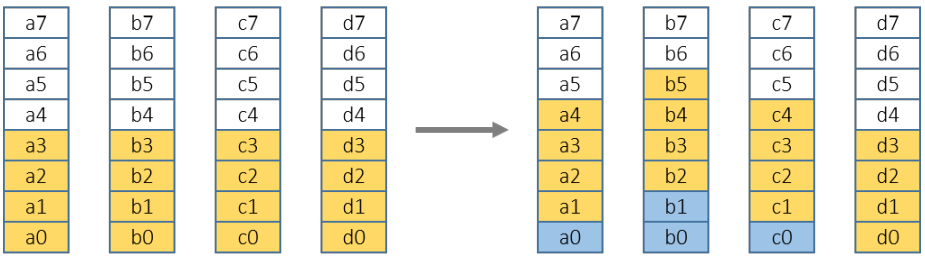

Figure 3.10: Input columns for 4-stream merging. Yellow blocks in left columns are input while the yellow one in right are the input for next cycle. Blue blocks are what have been output.

Figure 3.9 shows how the inputs are fetched and fed when merging two streams. Column a and b are both sorted sequences in descending order where a0 and b0 are the smallest entries in each column. At the very first we will take the smallest 4 elements from both columns and go through a merge unit. Given that $b_0 < a_3$ and $a_2 < b_0$, we will have $\{a_0, a_1, a_2, b_0\}$ as the first series of output. Thus we will see that $\{a_3, b_1, b_2, b_3\}$ will not be sent to output because they are fetched and fed into the merge unit to ensure that the generated $\{a_0, a_1, a_2, b_0\}$ is the smallest 4 elements between the entire 2 streams. Therefore the difference between the number of inputs and outputs is apparent since we have 8 inputs in total but only 4 outputs are produced. At the next cycle, together

with $\{a_3, b_1, b_2, b_3\}$, we fetch a new element set of $\{a_4, a_5, a_6, b_4\}$ so as to generate new 4 outputs.

When merging more streams we see the same effect. In fig 3.10 totally 16 elements from 4 streams were fed into the merge unit to generate the smallest 4 elements among all the streams. Again, we have some elements involved in the merge but not sent out. We call them the feedback elements. They will be updated every cycle and remain inside the merge unit.

In summary, if a merge unit is intended to output N elements every cycle from M stream, it has to deal with $M \times N$ elements every cycle while $(M \times N - N)$ elements are the feedback elements stored inside the merge unit. Only N elements should be fetched and fed into the merge unit every cycle since the feedback elements have been taken in previous cycles. The new elements fetching is data dependent. Which next N elements should be fetched depends on what were produced in the previous cycle. To implement this, we build a simple chain table. A few extra bits are attached to every individual element representing the stream number it comes from. In the outgoing N elements, if n ($n <= N$) elements are from stream i ($i <= M$), then the next input elements consist of the remaining smallest n elements from stream i. No read request will be executed upon streams without elements sent out.

3.4 Throughput Performance and Input Randomization

3.4.1 Skewed Data Distribution

In the design of our sorter, we have to consider a variety of key distributions, as various kinds of distribution can occur in unsorted datasets when they are collected from real life databases.

A popular assumption is for the dataset distribution to be uniform or ideally random, where in a general overview of the entire dataset, both relatively smaller and larger values occur in any column at equally the same probability. If we partition such a uniform distribution beforehand, we can obtain small partitions in which sequences have relatively the same distribution in the number of small and large values. It also indicates that the workload for each processing unit to sort different small partitions are also the same.

However, the distribution of the to-be-sorted dataset is usually not ideal. Skewed dataset distributions often occur especially when dealing with datasets in large scale or originating from natural applications like physics, biology or geoscience. We regard a distribution to be skewed when its distribution function is not uniform, like having a collection of data with highly similar or identical values more towards one side of the scale than the other, creating an asymmetrical distribution.

Fig. 3.11 provides two classical skewed distributions. The tapering sides in each graph are called tails, as they taper differently from left to right.

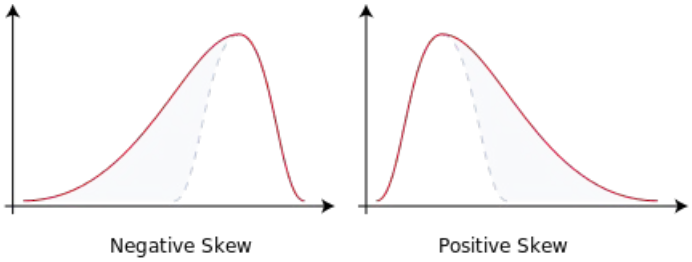

Figure 3.11: Left is the negative skew and right is the positive skew.

3.4.2 Performance Drop upon Skewed Distribution

The skewed distribution can result in an unbalanced partition at the beginning of the sort. For content-based partition, like radix sort [17], the bucket sizes may differ a lot depending on the skewness of the distribution, so that some buckets take significantly more time to process while some take much less. For equal partition, each sub-dataset is divided to be in the same size with the same number of elements, but some subsets can contain most of the relatively larger values while some only collect the relatively smaller ones.

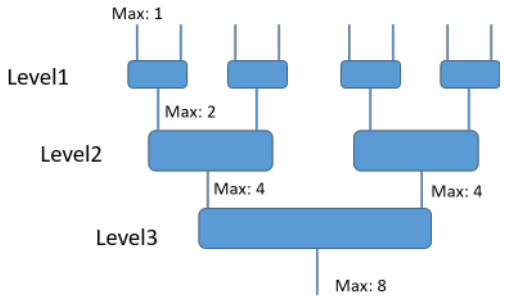

Figure 3.12: An improved merge-tree structure with increasing data rate at each level.

We model a typical merge-tree like merge engine to illustrate the performance drop resulted from skewed distribution. Fig. 3.12 shows an 8-input merge-tree with each input leaf connected to an input FIFO where the presorted sub-dataset in equal size is stored. Ideally, when the dataset distribution is uniform or globally random, each input FIFO can ensure a stable rate x and contribute to the final output rate y of multiple elements. Here we set the data rate at input leafs to be 1 element per cycle and the final output rate to be theoretically 8 elements per cycle. It can be expected that upon uniform distribution, each merge node at level 1 is expected to perform at a stable output rate of 2 elements per cycle, since ideally both upper input leafs are providing 2 elements in total to it. By cascading the leafs and merge nodes together, the final output rate is expected to be stable 8 elements per cycle.

The expecting number of elements per cycle at the output is not 8. We provide a simple deduction here based on Fig. 3.12 again. Assuming that for each level n, the tree can be fed by a maximum number of 2^{n-1} elements and the output of this level, coming from the upper two leafs equally, can produce $k = 2^n$ elements at most. By

summarizing from the true value tables in Appendix A in which list all the possibilities, the probability of fetching i elements from one of the two leafs is represented by P_i:

$$P_i = \frac{1}{2^k} * C_k^i \tag{3.6}$$

And the output expectation is

$$E_n = \sum_{0}^{k} (P_i * (\frac{k}{2} + i)) \tag{3.7}$$

It can be obtained by equations above that the expected output in the random case is around 6.9 elements per cycle.

However, if the dataset is skewed, the output rate is no longer stable and even worse. The throughput of the entire tree is the output rate of the final node at the bottom of the tree. It is expected that each upper node provides a stable input rate to the next node and contribute to a relatively high rate, but this is only satisfied when the dataset distribution among all the input FIFOs is uniform, so that each input FIFO are ensured to have an effective data rate to compromise the next node. Thus, in skewed distributions, it can be estimated that bottleneck can be one of the other nodes of the tree instead of only the final node. For example, if one of the input FIFO contains all the lowest elements, then only this particular FIFO will be working while all the rest FIFOs have to stall until the entire skewed elements are pushed out since most of the final multiple outputs are from that particular input FIFO. The mismatch between input FIFO rate and output processing rate will occur. Since the input FIFO is normally set to feed into the merge-tree at 1 element per cycle, it will take at most 8 cycles for this particular FIFO to produce all the desired elements. In the meantime, since the rest FIFOs are not in demand, they will be in idle doing nothing but waiting during this period. Such an extremely skewed case leads to a significant throughput drop to only one element per cycle rather than multiple elements.

Some existing design choices are made to handle skewed distributions. It either contains a shuffle operation[25] to randomize the distribution of datasets to ensure stable input rate, or applies an adaptive sort algorithm[19] by being aware of skewness and taking advantages of the original distribution function of given datasets. Both designs can get rid of the negative impact of skewness on throughput performance. In contrast, we provide a generic design choice to handle skewed distribution by promising the equal input with output data rate in Chapter 4.

4 Odd-Even Merge Sort

As indicated in previous chapters, a strong merge engine is essential in the final pass of the merge-sort. In this chapter, we provide the architecture of such a merge engine from how it interfaces to its environment to how it is constructed inside. Section 4.1 states the external architecture and the core construction of the merge sorter. Constructions of different components within the engine and their operating functions are presented in following sections.

4.1 Architectural Overview

The Odd-Even Merge Sorter aims to be a strong merge engine while handling multiple streams and promising a high throughput. We set the throughput target to be 25GB/s which is equal to the maximum bandwidth that an 8-lane OpenCAPI channel can provide. Given that the clock frequency of our AFU is targeting 200MHz, the throughput target is 128 bytes per cycle, or 8 elements with 8-Byte keys and 8-Byte values per cycle. An input Latch and an output FIFO in Fig. 4.1 are configured to ensure that read and write are executed at the desired bandwidth since the actual data fetching and producing rates within the merge sorter can be different. Streams to be merged are stored in the BRAM with a data streaming rate of 128 bytes per cycle to the input Latch.

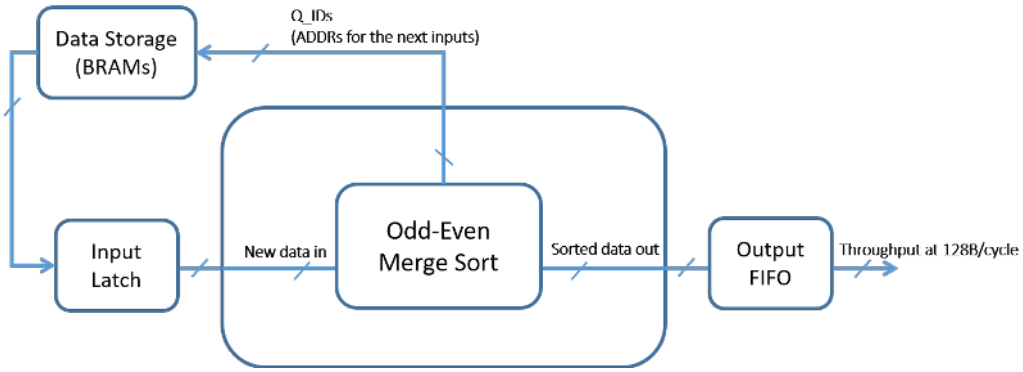

Figure 4.1: The odd-even merge sorter and its external environment.

Every time the odd-even merge sorter produces a series of outputs, the Queue-IDs (Q-IDs) are produced as well. The Q-ID here is some extra bits attached to an element indicating which stream it comes from. The Q-IDs are known in each even cycle as illustrated in section 4.4.5. Q-IDs will be interpreted as read requests sent to BRAM, so the next input elements are known. Elements sorted out will be sent to the output FIFO to ensure a throughput at 128 bytes per cycle. Given that each element consists

of an 8-Byte key and an 8-Byte record, at least eight elements produced every cycle is required.

The main components we build in the internal construction of the odd-even merge sorter are tuple buffers, swap units, and control units. Tuple buffers are where the feedback elements are placed, and they are stacked together one by one as shown in Fig. 4.2. Swap units deal with these feedback elements. Two adjacent tuple buffers send their stored elements to one swap unit, after which two sets of sequences both in order are sent back into these two buffers after comparisons and swaps. It will also be guaranteed that elements received by the upper buffer are all greater than those of the lower buffer. Control units interpret the clock signal into control signal representing current cycle status as odd or even and deliver it to other components to make sure the entire merge sorter operates under odd and even mode alternately.

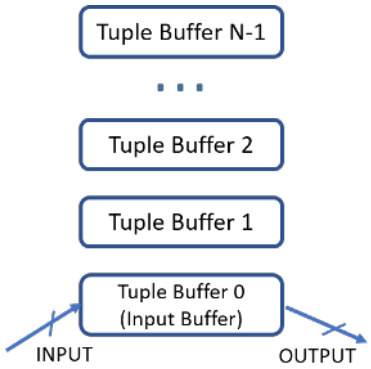

Figure 4.2: Tuple buffers piping up mechanism. The lowest one is labeled as buffer 0, and the uppermost one is labeled as buffer $N-1$

The odd-even merge sorter operates in two phases as shown in Fig. 4.3, the filling phase, and the streaming phase. The filling phase is the initialization of the engine. All the feedback elements inside tuple buffers are reset with their keys to be 0. We call them dummy elements. During the first input cycle, after the reset signal is deasserted, all the input elements are be stored in the feedback elements since they are definitely greater than the dummy elements. Thus, the first output run of the engine will be dummy elements. By continuing to feed the engine with new elements, all the dummy elements that previously stored inside the engine will eventually be replaced and sent out. After the dummy elements are cleared, the streaming phase begins, and we can start collecting output elements to the output FIFO.

The streaming phase operates the same with filling phase except there are no dummy elements anymore. Feedback elements are stored in tuple buffers regarding sequences each with N elements. In general, relatively small elements will drop to tuple buffers in lower positions so that the lowest tuple buffer always stores the smallest N elements among the entire engine.

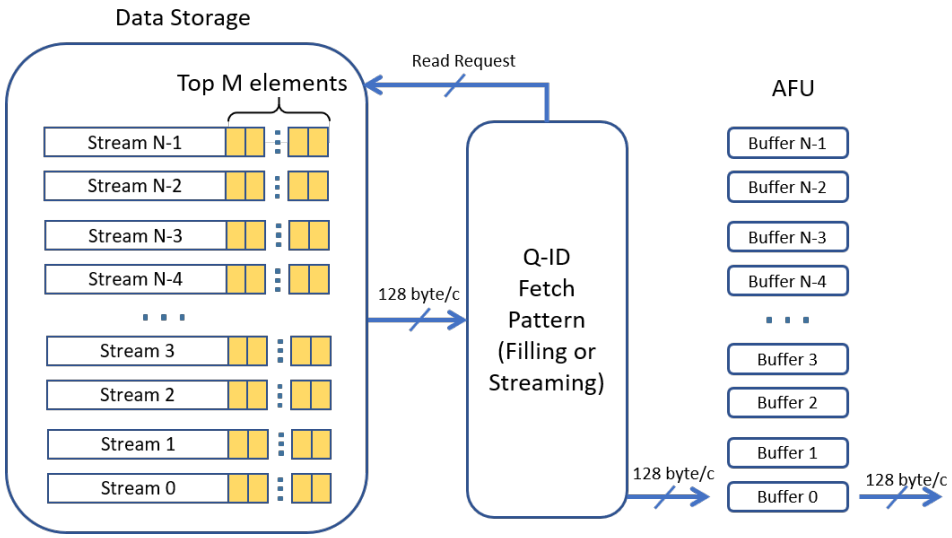

Figure 4.3: Data flow from data storage to AFU. The arrows here represent only one output port on the data storage side and one input port on AFU side at data rate of 128-byte per cycle.

4.2 Cycle Count

The reason we name this an odd-even merge sorter is that this engine operates in two different modes, the odd mode, and the even mode, which corresponds to the cycle status named as the odd cycle and the even cycle. The cycle status changes at every rising edge of the clock signal. The change between odd cycles and even cycles is continuously and alternatively.

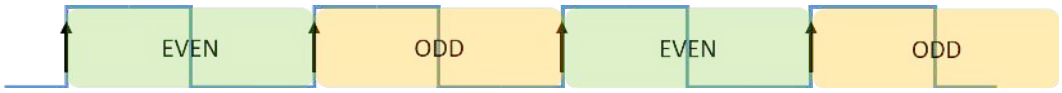

Figure 4.4: The merge sorter operates in even and odd cycles alternatively.

In the even cycle, swap units operate in their even mode while the input buffer receives new elements from input latch. In the odd cycle, swap units operate in their odd mode while the input buffer sends out sorted elements. The Q-IDs are generated after the swap is accomplished in the even cycle and assigned to the BRAM in next odd cycle.

In every cycle, no matter even or odd, the swap units are always running. What differs from even to odd mode is the input received by the swap unit, which is accomplished by a multiplexer. To achieve high-frequency performance, the running time of the swap unit should be as short as possible.

The control unit which determines the cycle status is shown in Fig. 4.5. It is a simple counter which starts counting cycles since the reset signal is wiped. The cycle count logic delivers the CYCLE-COUNT signal to all the other internal components of the entire odd-even merge sorter.

As we mentioned in section 4.1, given that the clock frequency of the AFU is 200MHz,

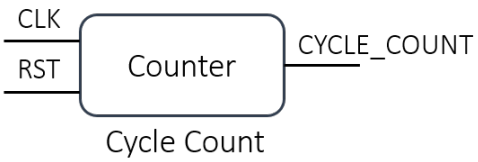

Figure 4.5: Architectural view of the cycle count.

at least eight elements, 8-Byte key and 8-Byte payload, are supposed to be produced every cycle to satisfy the 25GB/s throughput needed. Since the odd-even merge sorter operates in two modes and it only produces output in even cycles, the number of elements to be produced every even cycle is 16. Therefore, the output FIFO is an asynchronous FIFO which is filled with 16 new elements every two clock cycle and sends out eight elements every clock cycle. The input Latch is to store the 16 input elements and send them to the merge engine every two clock cycles, which is every odd cycle.

4.3 Tuple Buffer

Tuple buffers are used to store the feedback elements, but the number of feedback elements rises as the stream number grows. When merging M ($M > 2$, assuming that M is even) streams, the essential price of obtaining the current smallest N elements is to take all the top N elements from each stream into comparison. Consequently, $(M \times N - 1)$ feedback elements are unavoidable. Besides, sorting out N elements from $M \times N$ elements every cycle consumes enormous resources and time. We implement the tuple buffers in a way not saving the number of tuple buffers but reducing the penalty for each single sort operation. Instead of sorting N elements from $M \times N$ elements, we sort N from $2N$ in parallel. Tuple buffers shown in Fig. 4.2 are stacked up in equal amount M to the number of to-be-merged streams.

The tuple buffers are labeled from 0 to $M-1$, and we assume M to be even. In even cycles, elements stored in buffer 0 will be compared with those in buffer 1. Buffer 2 will compare with buffer 3 and accordingly buffer i (given that i is even) will compare with buffer $i+1$. $\frac{M}{2}$ swap units are in demand to accomplish these comparisons. However, different comparisons happen in odd cycles. In odd cycles, buffer j (given that j is odd) compares with buffer $j+1$. It is obviously learned from Fig. 4.6 that in even cycles, the swap starts from buffers labeled with an even number while in odd cycles it starts from buffers with an odd number. When operating in odd cycles, two buffers are not involved by any swap units. One is the buffer $N-1$, and the other is the lowest buffer which we name it the input buffer.

However, the input buffer is not really in idle. It sends out the elements stored itself in the previous even cycle. Besides, the Q-IDs are sent along with the outputs as well. As illustrated in section 4.1, new elements enter the input buffer in filling phase. In the first even cycle of filling phase, what is stored in the input buffer swaps with that in buffer 1, which is the dummy elements. The result is obvious that dummy elements are shifted to the lowest buffer while those real elements are delivered to buffer 1. Now we

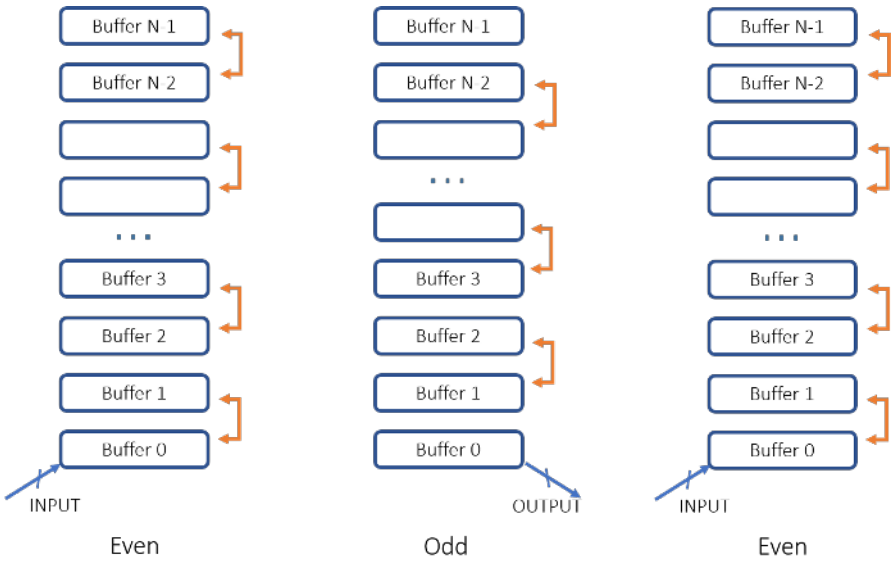

Figure 4.6: Swap operations in even cycles and odd cycles.

can see that in the following odd cycle, the input buffer will send the dummy elements to the output while another set of dummy elements will be delivered to buffer 1 as the result of the swap happens in this cycle. Therefore, since we have the same amount of buffers to the streams, all the dummy elements will be clear in the Mth odd cycle.

Fig. 4.7 illustrates the implementation of the tuple buffers which are configured by multiplexers and registers. As mentioned before, the swap unit is always running not matter in even or odd cycles, but the tuple buffers involved are different. Port *Din0* and *Din1* are for receiving swap results from the swap units as *IN* ports. For each tuple buffer, one *IN* port connects to the swap unit it interacts with in even cycles while the other *IN* port connects to the other swap unit it interacts in odd cycles. A control signal representing whether the current cycle is odd or even will determine the output of the multiplexer. Afterwards, the selected output from the multiplexer will be captured by and stored in the register.

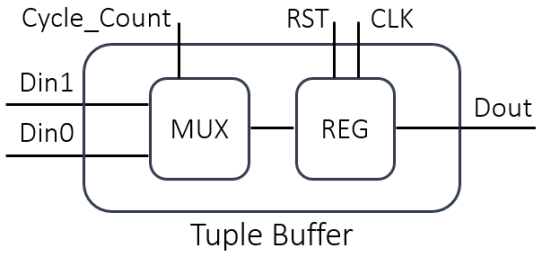

Figure 4.7: Architectural view of the tuple buffer.

4.4 Swap Unit

In this section, we first illustrate the logic function of swap units and the basic architecture of it. Then in the rest of this section, two possible configurations of the swap unit are presented.

4.4.1 Logic Function of Swap Unit

A swap unit aims to merge 2 sequences in descending order each with N elements and generate another 2 sequences also in descending order. Elements in one of the generated sequence are all at least smaller than all those in the other sequence. We label the 2 input sequences as *Din0* and *Din1* while the 2 output sequences are *Dout0* and *Dout1*.

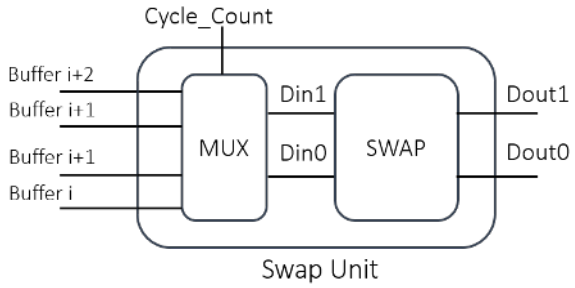

Figure 4.8: Architectural view of the swap unit.

In even cycles, we can see tuple buffers labeled with even number interact with its upper adjacent tuple buffer while the inverse interaction happens in odd cycles. For the sake of saving some hardware resource, we attach a 4-to-2 multiplexer into each swap unit so that they can be continuously utilized in both even cycles or odd cycles. The cycle status serves as the select signal of the multiplexer in Fig. 4.8 which determines which 2 sets of elements will enter the SWAP. Therefore, only $M/2$ swap units are in demand for merging M streams. We now label the lowest swap unit as swap unit 0, so the uppermost one is swap unit $M/2-1$.

Now we look at the first even cycle of the filling up phase again. In even cycle, the multiplexer selects buffer i and buffer $i+1$ as its output (given that i is even). Therefore in swap unit 0, *Din0* corresponds to the input buffer and *Din1* is the dummy elements in buffer 1. Swap unit 0 sends *Dout0* (dummy elements) back to the input buffer and *Dout1* (effective elements) back to buffer 1. Then in the coming odd cycle, dummy elements within input buffer are delivered to output FIFO and buffer 1 will interact with buffer 2 in swap unit 0. All the dummy elements are clear by alternatively swapping as indicated in Fig. 4.9. Eventually, after all the dummy elements are clean, effective elements will take over all the tuple buffers.

It can be foreseen that the swap unit will take the longest data path among the entire odd-even merge sorter. The swap unit is constructed to perform a parallel sort. Following subsections will discuss the characteristics of its input and output sequences, as well as the inner structure topology.

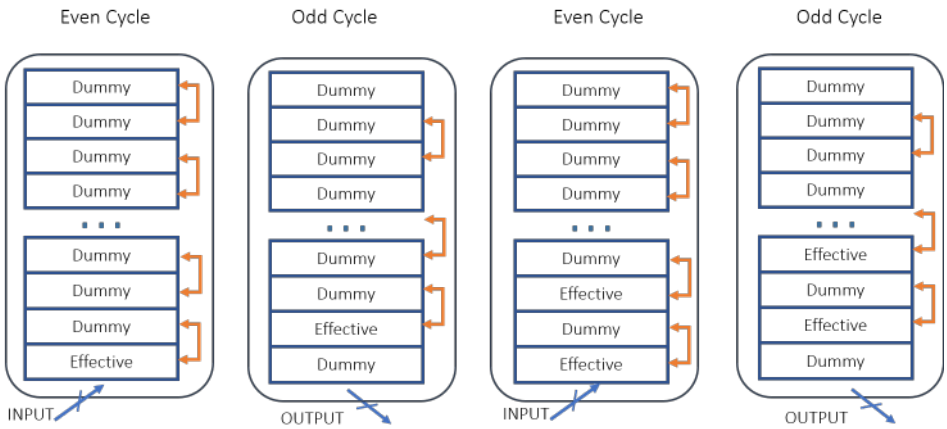

Figure 4.9: The first 4 cycles of filling up phase are shown here. Dummy elements are cleared gradually. The orange double-arrow represents swap units.

4.4.2 Bitonic Sequence

The bitonic sequence is a sequence that monotonically increases and then monotonically decreases, or can be circularly shifted to become monotonically increasing and then monotonically decreasing[27]. A circular shift here stands for moving the first element of a sequence to the final position while shifting all the rest elements to the next position, or the inverse operation. Simply we can have some bitonic sequence examples like {1,3,9,6,4} and {6,8,10,9,7,2,3,4}. It is also bitonic if an entire sequence is monotonically increasing or monotonically decreasing. Particularly a sequence with dummy elements is a bitonic sequence as well since every element here is equal to each other.

A half-cleaner is a particular kind of comparison network of depth 1. For a sequence with N elements, the half-cleaner compares element i with element $i + N/2$ (for $i = 1$, 2,..., $N/2$, assuming that N is even) and swaps them if element i is greater than the other. The reason why it is named by "half" and "clean" is shown in Fig. 4.10 where two half-cleaners are both fed with a bitonic sequence of 0's and 1's as the application of lemma 4.1.

Lemma 4.1 *If the input to a half-cleaner is a bitonic sequence of 0's and 1's, then the output satisfies the property: both the top half and the bottom half are bitonic, every element in the bottom half is at least as small as every element of the top half, and at least one half is clean(consisting of either all 0's or all 1's)[27].*

Lemma 4.1 indicates that a half-cleaner can at least clean half of the bitonic sequence which contains only 0 and 1. If the input to a half-cleaner is a bitonic sequence with not only 0's and 1's but also other values, we can still obtain the output satisfying an significant property: every element in the bottom half is at least as small as every element of the top half. Fig. 4.11 applies this general case.

Now take the feedback elements stored in tuple buffers to consider. For buffer i and buffer $i + 1$, they store sequences as {a15,..., a1, a0} and {b15,..., b1, b0} both in descending order. Combining them together we have a bitonic sequence {b15,..., b1, b0,

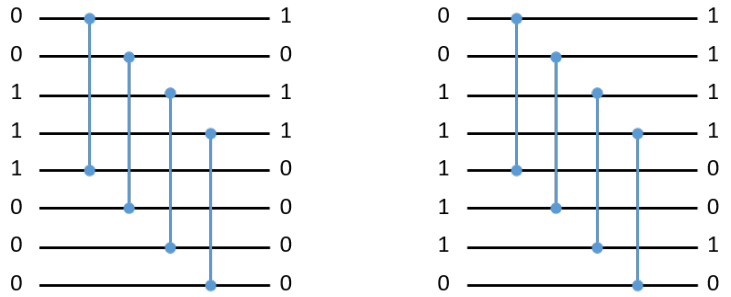

Figure 4.10: A Half-Cleaner comparison network with 8 inputs.

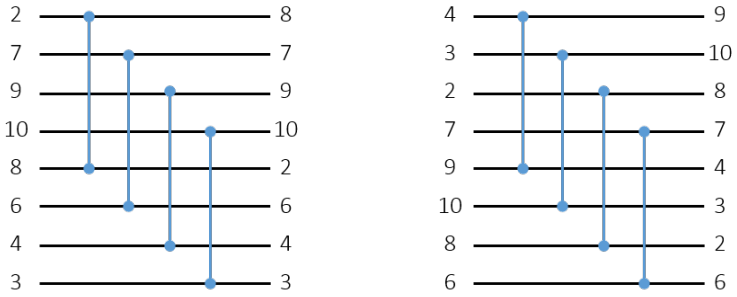

Figure 4.11: A half-cleaner dealing with a bitonic sequence with general values.

a0, a1,..., a15} which monotonically decreases and then monotonically increases. Then we apply this combined sequence into a half-cleaner to obtain a bottom half and a top half while the bottom half is at least smaller than the top half. Both these two half are not in order, but they can be fed to two sorting networks in parallel and save resources by adopting a sorting network with fewer inputs.

4.4.3 Sorting Network

After the half-cleaner, we split the 32-element sequence into two 16-element sequences. Since the bottom half is the smaller half, it is sure that the smaller 16-element sequence will eventually leave the swap unit through the *Dout0* port while the other sequence will be mapped with port *Dout1*. Therefore, as shown in Fig. 4.12, the next step is to sort these two sequences in parallel. A reasonable choice is a sorting network.

As illustrated in Chapter 2, the sorting network is one of the most popular parallel sorting algorithms. We first consider the sorting network which consists of only comparators and lots of comparisons can be performed simultaneously.

A basic comparator is a device with two inputs and two outputs which performs the following function:

output(0)=min(input 0,input 1)
output(1)=max(input 0,input 1)

The actual processing upon the two inputs is a compare and a swap operation. Since

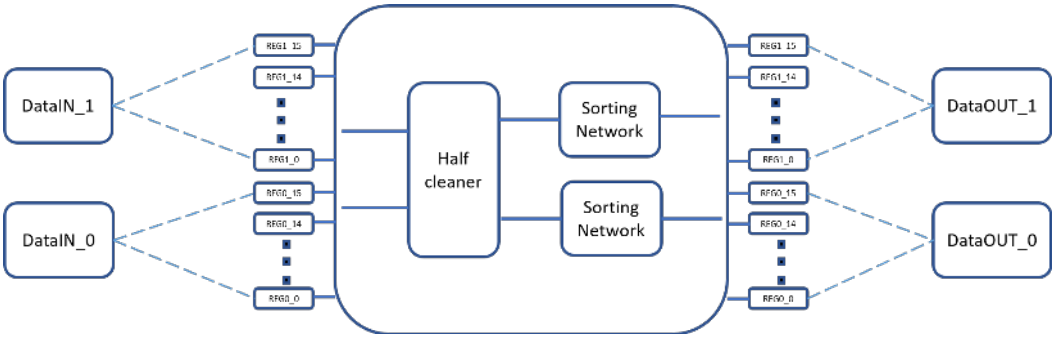

Figure 4.12: What is configured inside the SWAP of Fig. 4.8.

the sorting network is constructed by consecutive and cascaded comparators, we can foresee that each element of an input sequence will go through the sorting network one layer by another.

After the half-cleaner we acquire two 16-element sequences in unpredictable order, so we need a generic sorting network to handle all possible data distributions to ensure stable performance. Besides, since one of the two sequences is all at least smaller than the other, we can perform these two sorting network in parallel.

So we have to be careful about the configuration of our sorting networks. The depth of a half-cleaner is 1 and the running time of it is one unit time which should also be taken into account the timing budget. In a sorting network with x input, the larger x is, the larger the minimum number of comparator layers in demand is. In our odd-even merge sorter case, an individual sorting network has 16 inputs. If we apply Batcher's Merge-Exchange method[20] to configure the sorting network, at least 10 comparator layers will be needed, which also means the minimum depth of a sorting network is 10, and it consumes 10 unit time. Apart from Batcher's method, we will also apply several classical sorting networks that mentioned in section2.3.4 to validate the trade-off between a number of comparators and depth. Experiment results in Chapter 5 perform the actual timing performance of these classical sorting network methods.

4.4.4 Tag Sorting Network

For the sake of high throughput and high-frequency performance, the running time of the sorting network in our swap unit should be as short as possible, which demands a smaller depth. The depth can be greatly reduced by paying the price of more comparators, which is to establish a large comparator layer and perform all the comparisons simultaneously. The resource utilization of this configuration of introducing more comparators is discussed in Chapter 5. We name the sorting network with huge comparator layer the tag sorting network.

The tag sorting network consists of three sets of processing logic, first of which is the comparator layer. The comparators in a tag sorting network are different from what we have in the last subsection. The output of each comparator, the Tag, indicates the result of this comparison. They have two inputs but only have one output. This new comparator satisfies the following function:

$$\text{Tag} = 1 \text{ when input } 0 > \text{input } 1$$
$$\text{Tag} = 0 \text{ when input } 0 \leq \text{input } 1$$

For a non-ordered sequence with N elements, we name the elements of it as {a(0), a(1), a(2),..., a($N-1$)}. In our proposed comparator layer, element a(i) compares itself with all the other $N-1$ elements and collect the Tag values generated by each comparison. By collecting all the $N-1$ Tag values and adding them together, we can know from the sum that how many elements are greater than element i. Every element will obtain its own sum of Tag values which we call it Flag. It may happen that some elements equal in key value will obtain identical Flags. By taking the original order (range from 0 to $N-1$) of each element into account, we are also able to deal with this conflict. The detailed structure of such a tag sorting network is shown in Fig.4.13

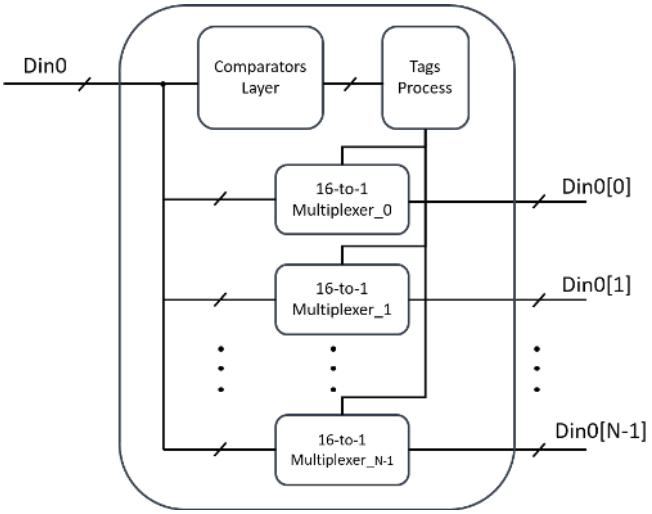

Figure 4.13: Internal components of the tag sorting network.

Fig. 4.14 shows the configuration of the comparator layer with 8 inputs. Since the new comparators only produce a Tag with a value of 0 and 1, no swap operation is performed. Consequently, all the comparison can be performed simultaneously within one unit time. The effective depth of this configuration is 1.

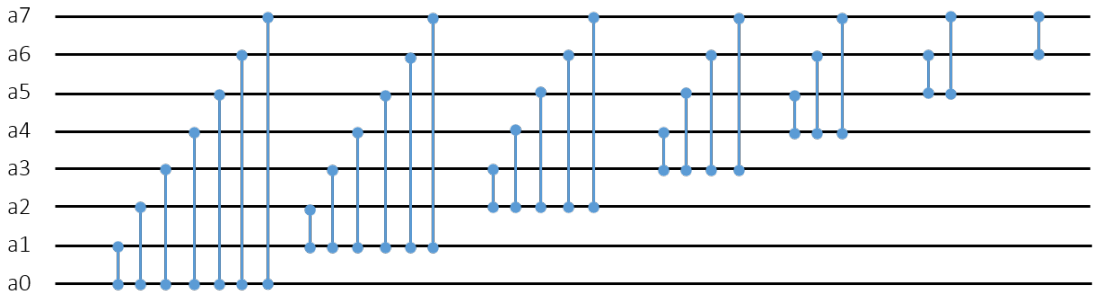

Figure 4.14: The comparison network in tag sorting network with 8 inputs and 8 outputs

To save comparator resource, the number of comparators used here is $\frac{N\times(N-1)}{2}$. For a comparison between element a(x) and a(y), we say that the generated Tag is a(x)(y). In Fig. 4.14, the Flag value of a0 is exactly the sum of all the 7 Tags {a(0)(1), a(0)(2),..., a(0)(7)} generated by the comparator layer. However, when calculating the Flag value of a1, apart from the sum of 6 Tags, the Tag of a(0)(1) should also be considered.

After having the Flags ready for each N input elements, which are unique and range from 0 to $N-1$, all the input elements will go through N N-to-1 multiplexers. The output will be picked up from these N inputs according to the Flags which serve as the select signal of the multiplexers. We can foresee that such a tag sorting network, configured by one comparator layer, some adders, and multiplexers, will perform a relatively high frequency in hardware implementation.

4.4.5 Lowest Swap Unit

Q-IDs are generated in the lowest swap unit in each even cycle. When the two 16-element sequences are produced by the half-cleaner, it is sure that the sequence with smaller elements will be sent to the output FIFO after the sorting network. The Q-IDs are collected from this smaller sequence since it has the stream number information we want. Therefore, in each even cycle, we present the Q-IDs intermediately in the lowest swap unit.

In the coming odd cycle, the BRAM receives the Q-IDs as the read requests and will response with data in a rate of 128-Byte per cycle which is 8 elements per cycle. It will take two cycles for the BRAM to perform all the 16 elements, so we will have them ready at the next odd cycle.

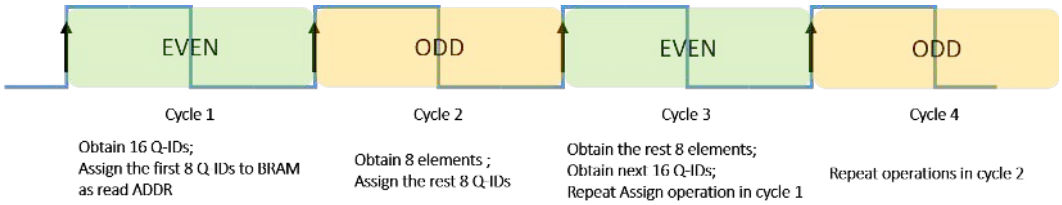

Figure 4.15: Inputs fetch pattern based on Q-IDs.

Fig. 4.15 shows how inputs are fetch based on Q-IDs. In cycle 1 we obtain a set of 16 Q-IDs and assign the first 8 of them to BRAM as a read address. In cycle 2 we obtain 8 elements from BRAM and assign the rest 8 Q-IDs. In cycle 3 we obtain the rest 8 elements, so all the 16 inputs are ready. We also obtain next set of 16 Q-IDs, first 8 of which will be assigned to BRAM as the same in cycle 1. In cycle 4 we repeat the operations in cycle 2. In summary, by obtaining Q-IDs and assigning them to BRAM alternatively, we have the 16 inputs in every odd cycle.

The input sequence with 16 elements is in unpredictable and unsorted order, which has to be sorted before entering the half-cleaner. Based on previous subsections, two different configurations of this particular lowest swap unit are shown in Fig. 4.16 and Fig. 4.17.

Another possible implementation is that we presort these new input elements outside

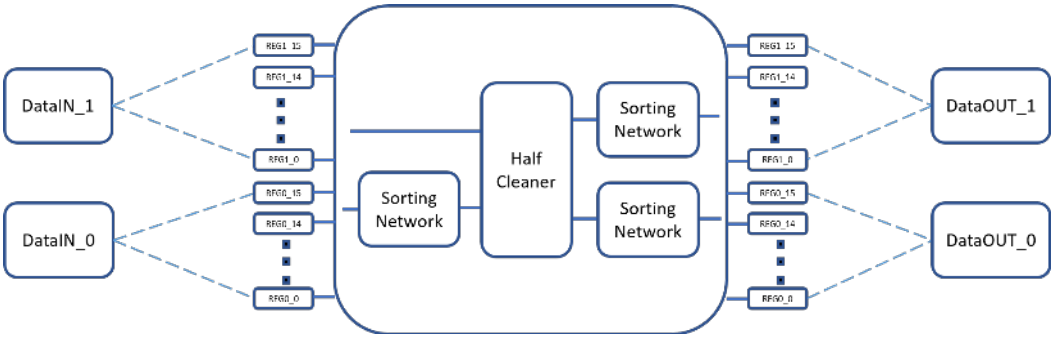

Figure 4.16: Architectural view of Swap Unit 0 with additional 16-input sorting network.

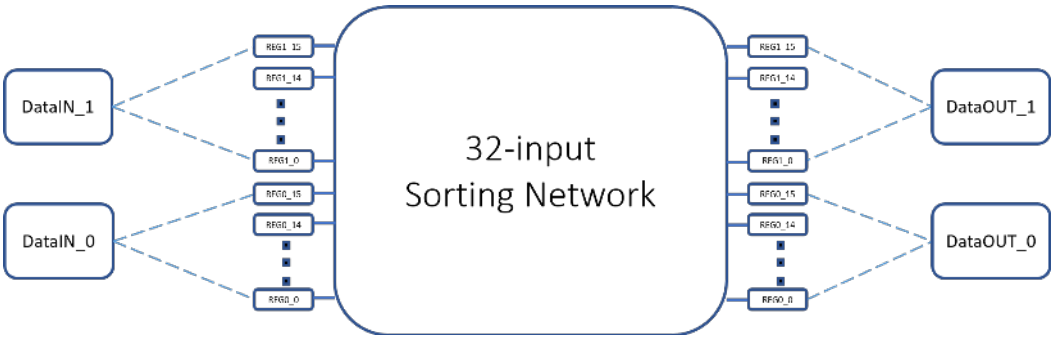

Figure 4.17: Architectural view of Swap Unit 0 with a 32-input sorting network

the odd-even merge sorter. When the input elements are fetched, we sort them with the same sorting network in section 4.4.3 or section 4.4.4. Thus, the lowest swap unit can adopt the same structure with the other swap units, and a 16-input sorting network will be performed apart from the odd-even merge sorter engine. In addition, a drawback is that the presorting will reduce the timing slack to meet the timing request before when the 16 inputs enter the input buffer in the coming odd cycle.

All these three methods to implement the lowest swap unit can ensure that the entire odd-even merge sorter can work properly. However, we have to pick up the one with good enough throughput performance and small enough resources utilization. The throughput and resources of them after implementation will be discussed in Chapter 5.

5 Validation and Experiments

In Chapter 4 we illustrated the design details of our odd-even merge sorter. Chapter 5 presents the validation and experiment results. Section 5.1 introduces the simulation tool, the synthesis platform and the FPGA chip we used. Section 5.2 provides the simulation result for the odd-even merge sorter. Section 5.3 presents the evaluation results of our odd-even merge sorter regarding the performance metrics. Section 5.4 provides evaluation results of the swap units with different implementation.

5.1 Platform Introduction

5.1.1 Simulation and Synthesis Tools

The proposed odd-even merge sorter is written in VHDL. The behavior simulation platform is Modelsim 10.4 and the synthesis tool we use is Vivado 2017.1.

5.1.2 Target FPGA

We indicate in Chapter 2 that the high bandwidth interconnect plays an important role in FPGA accelerators. In this chapter, we build our design based on FPGAs that are integrated on commercial FPGA acceleration cards featuring the PCIe or OpenCAPI interface.

ADM-PCIE-9V3[28] is an FPGA acceleration card from Alpha Data. It features the VU3P FPGA in Xilinx Virtex Ulatrascale family.

The Nallatech 250S+[29] is a high-performance PCIe-based Flash SSD with localized FPGA acceleration capability. Its processing unit is the KU15P FPGA in Xilinx Kintex Ultrascale family.

In this thesis, we use the KU15P-ffve1760-3-3 as the target FPGA chip. It features 668 I/O pins, 523k LUTs and 1M flip-flops.

5.2 Simulation

5.2.1 behavioral simulation

The behavioral simulation is a cycle by cycle representation of the RTL design. We perform the simulation on Modelsim, a widely used simulation platform. When we have the compiled RTL code written in VHDL, a test-bench is added to provide a streaming input to the odd-even merge sorter.

The test-bench initiates the odd-even merge sorter by a global “reset” signal which is effective in high-level logic. Most of the Xilinx FPGAs are adopting active-high logic to serve as a synchronous reset signal as stated in its white paper WP272[30]. When the

"reset" signal is active, all the tuple buffers are initialized with dummy elements which every bit of them is '0'.

The filling phase starts at the first clock rising edge after the reset signal is released. We determine the first cycle to be the first odd cycle, and the next is the first even cycle. Taking the case with eight to-be-merged streams ($N = 8$) as an example, the total number of cycles required to finish the filling phase is shown in equation 5.1, which is 16 cycles in the 8-stream case.

$$t_{fill} = N \times 2 \tag{5.1}$$

In the waveform in Fig. 5.1, after we deassert the 'rst' signal, the counting of odd-even cycles starts. We mark the odd cycle with a low-level value '0' of 'cycle-count-s' and the even cycle with a high-level value '1'. The counting starts from the first rising edge of 'clk' after the 'rst' signal is released. First the merge sorter performs an odd cycle, and after that an even cycle.

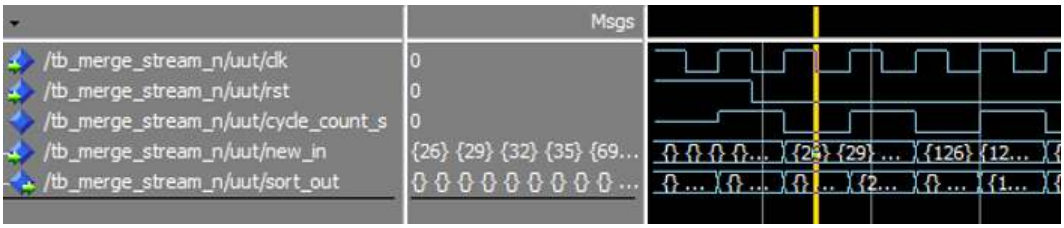

Figure 5.1: Waveform for the odd-even merge sorter at first few cycles.

In the first odd cycle, a set of dummy elements are produced in port 'sort-out'. The next coming cycle is the even cycle, where the input is captured by the input buffer and sent to the swap unit. Alternatively, the first few odd cycles perform a series of dummy sequences while the input elements are captured and swapped among the upper stacks of buffers.

Figure 5.2: Waveform at cycles when the first effective outputs are captured.

In the first even cycle we have the first input effective sequence as {26, 29, 32, 35, 69, 64, 59, 54, 3, 4, 5, 6, 15, 17, 19, 21} and we can the sorted sequence after 16 cycles as {69, 64, 59, 54, 35, 32, 29, 26, 21, 19, 17, 15, 6, 5, 4, 3}. It is the first effective output we can collect at the output port.

Smoothly, 16 sorted elements can be collected stably every two cycles. The entire merge sorter operates in odd and even cycles alternatively according to the global signal 'cycle-count-s' which indicates the cycle status. When it comes to the end of streams when less than 16 elements are left, we compensate the sequence to 16 elements with another kind of dummy elements of which each bit is set to be '1'. Therefore, the merge

sorter can always operate on 16 elements, and the output FIFO only has to filter out the dummy elements for the final output.

5.3 Performance Evaluation

To evaluate our odd-even merge sorter, we focus on these performance metrics: frequency, resource utilization, and throughput. The evaluations present the performance metrics based on different numbers of sorted streams to merge. The number of elements E produced by the merge sorter per cycle is 8.

As we discussed in section 4.4.5, we can have different design choices for the swap unit 0. Here we first present the performance metrics of the structure with a 32-input tag sorting network to be the swap unit 0. The tuple size of each element is 136-bit, with a 64-bit key and a 64-bit value, together with an 8-bit Q-ID. We compare the performance variation among five cases with a different number of streams. The evaluations are shown in table 5.1.

Table 5.1: Evaluation results of our odd-even merge sorter.

Structure	Number of streams	Tuple size	LUT	Registers	Frequency (MHz)	Period (ns)	Throughput (GB/s)
Swap0: 32TagSN, Rest: Half Cleaner +16TagSN	4	8B Key, 8B Value, 1B ID	66569 (12.74%)	4609 (0.44%)	212.36	4.71	27.18
	8		112522 (21.53%)	8705 (0.83%)	210.39	4.75	26.93
	16		209666 (40.11%)	16897 (1.62%)	209.51	4.77	26.82
	24		306725 (58.68%)	25089 (2.4%)	209.07	4.78	26.76
	32		403644 (77.22%)	33281 (3.18%)	208.76	4.79	26.72

1. *Resource utilization analysis:*

   Table 5.1 shows the resource utilization of our odd-even merge sorter. The overhead on resource utilization of merging more streams is mainly the larger stacks of tuple buffers and swap units. To merge more streams requires more tuple buffers and also more swap units which are configured from LUTs and Registers.

2. *Frequency analysis:*

   We evaluate the design in five cases: N= 4, 8, 16, 24 and 32. The result shows that we have achieved frequencies over 200MHz on average which meets the initial target we set for efficient utilization of the OpenCAPI bandwidth.

   The operating frequency only suffers a slight drop when the number of streams to be merged N increases. This is because the extra tuple buffers to store more

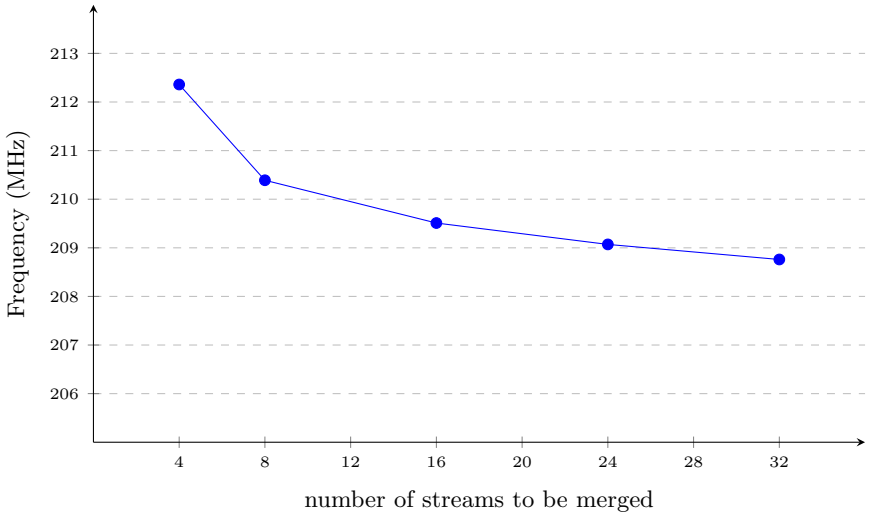

Figure 5.3: Frequencies in five cases: N=4, 8, 16, 24, 32.

feedback elements and the extra swap units due to an increasing N all operate in parallel. More components are stacked up, but no extra logic is added to the critical path. Fig. 5.3 shows the frequency versus the increasing N only decreases slightly.

3. *Throughput analysis:*

   We calculate the throughput based on the equation 3.4. The throughput of our odd-even merge sorter is mainly determined by the operating frequency. According to the previous analysis on frequency, we can achieve a stable and high throughput over a different number of streams. The highest throughput we achieve when merging four streams is 27.18 GB/s.

5.4 Evaluations on Different Swap Units

We also evaluate the different design choices for the swap unit 0. The above evaluation results apply an 32-input tag sorting network as the lowest swap unit.

According to previous analysis in section 4.4.5, the half-cleaner demands the input sequence to be bitonic. Since the input elements in each odd cycle come from different streams and are not in order, an extra sorting network has to be applied to sort these inputs before they enter the half-cleaner. As shown in Fig. 4.16, an additional 16-input tag sorting network is added to the swap unit 0.

Table. 5.2 shows the effect of the extra sorting network on frequency performance. It introduces more logic into the critical path and results in a longer period.

We also evaluate if we apply all the swap units with a 32-input tag sorting network as shown in table 5.2. Both these two structures feature high frequency and throughput performance, but the case without half-cleaner requires more LUT usage, which is not easy to be applied into multiple-stream cases.

Table 5.2: Evaluation results of different implementations of the swap units.

Structure	Number of streams	Tuple size	LUT	Registers	Frequency (MHz)	Period (ns)	Throughput (GB/s)
Swap0: 32TagSN, Rest: Half Cleaner +16TagSN	8	8B Key, 8B Value, 1B ID	112522 (21.53%)	8705 (0.83%)	210.39	4.75	26.92
Extra 16TagSN in Swap0 Rest: Half Cleaner +16TagSN	8		102827 (19.67%)	8705 (0.83%)	131.89	7.58	16.88
All Swap Units: 32TagSN	8		173496 (33.19%)	8705 (0.83%)	186.67	5.36	23.89

The three methods to construct swap units for the odd-even merge sorter present different advantages and disadvantages. Regarding resource utilization, all of them use the same amount of registers since the number of tuple buffers are the same. However, the structure with an extra 16-input tag sorting network greatly increases the time period of the merge sorter and presents a frequency of only 131.89 MHz. The 32-input tag sorting network can handle the non-bitonic input sequences but it also introduce extra latency when applied to all the swap units. In summary, regarding fewest resource utilization and best throughput performance, it is most efficient to implement a 32-input tag sorting network to the swap unit 0 and implement the rest swap units with a half-cleaner, which also proves the analysis in section 4.4.5.

6 Conclusions and Future Work

This chapter concludes the thesis and shows our future work. Section 6.1 presents the conclusions we have. Section 6.2 illustrates our future work to improve the presented FPGA-based merge sorter.

6.1 Conclusions

In this thesis, we aim at designing an FPGA-based AFU to perform database acceleration and leverage the advantages of high bandwidth interconnects. We present a novel and high-performance FPGA-based odd-even merge sorter. The proposed odd-even merge sorter features a high throughput and the performance is very stable over various numbers of to-be-merged streams. Synthesis results show that our odd-even merge sorter operates at 212.36 MHz and presents a significant throughput of 27.18 GB/s when merging 4 streams and suffers only a small drop in throughput when the number of input streams increases. More importantly, the performance of our odd-even merge sorter is independent of the data distribution and specifically it can deliver peak performance for skewed data distributions. We draw the following conclusions.

1. A strong merge engine to merge multiple streams in multi-pass merge sort algorithms is essential. Based on our analysis, hardware-based sorters suffer degrading performance when sorting large-scale datasets due to the increasing numbers of trips to access main memory. Large-scale datasets are usually partitioned into many small streams and merged after they are sorted individually. Our analysis also concludes that for the final pass accessing the main memory, the merge engine has to feature a throughput high enough to keep up with the available memory bandwidth. It also has to merge as many as streams as possible for the sake of saving memory access latency. The proposal architecture uses an odd-even merge sorter, which acts as a multi-stream merger that can produce multiple tuples every cycle.

2. Initial data distributions in worst cases should be taken care of since they can result in significant performance drop to the hardware-based sorters. Although many of the previous hardware-based sorters present relatively high throughput on average, their throughput are bounded by the data distributions. Our proposed odd-even merge sorter present sustained high throughput over all the possible data distributions. We achieve this constant performance by utilizing the multi-stream buffering interface and our input fetch pattern based on Q-IDs.

3. We improve the sorting network and reduce its latency. A sorting network based on compare-swap units takes a long time to sort multiple inputs, negatively affecting

cycle time and throughput. Our tag sorting network compares every two elements in the first step, and use the generated tags to arrange correct orders for each input. It features less latency and greatly reduces the critical data path delay.

4. Our proposal presents great potential in scaling. In our evaluation results, when we try to merge more streams simultaneously, the operating frequency remains to be over 200 MHz. We merge 32 streams in this thesis, and it can still keep up with our target memory bandwidth. The odd-even merge sorter can be easily extended into more streams like 64 or even 128. The only requirement is to stack up more tuple buffers for feedback elements and more swap units, which can be accomplished on an FPGA with more resource.

6.2 Future Work

1. Place and Route

   The current evaluations of odd-even merge sorter are based on the synthesis result from Vivado 2017.1. The place and route are not finished yet but we anticipate that by adding sufficient placement constraints to the design the design should be routable.

2. Connect with the multi-stream buffering interface.

   The proposed odd-even merge sorter requires a muliti-stream buffering interface, which is designed by Yvo Mulder, to feed the input data at the desired bandwidth. The next step will be combining them together.

3. A full-pass sorter from the initial partition to the final merge.

   In this thesis, we present an odd-even merge sorter as a design proposal for the final pass of merge operation since we illustrate that it ensures a high utilization of the memory bandwidth. A future improvement can work on a complete case to perform the detailed design choice in the initial partition and sort operation. Therefore they can combine with the merge sorter we proposed.

Bibliography

[1] Wikipedia, "Database — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Database&oldid=818710553, 2018, [Online; accessed 05-January-2018].

[2] S. Imaging, "Introduction to fpga acceleration," https://www.stemmer-imaging.co.uk/en/technical-tips/introduction-to-fpga-acceleration/, 2018.

[3] B. D. S.L., "White paper: Gpu vs fpga performance comparison," http://www.bertendsp.com/pdf/whitepaper/BWP001_GPU_vs_FPGA_Performance_Comparison_v1.0.pdf, 2018.

[4] J. Stuecheli, "Opencapi - a new standard for high performance memory, acceleration and networks." in *HPC Advisory Council - Swiss Conference 2017, 2017*, 2017.

[5] N. M. Amato, R. Iyer, S. Sundaresan, and Y. Wu, "A comparison of parallel sorting algorithms on different architectures," *Technical Report TR98-029, Department of Computer Science, Texas A&M University*, 1996.

[6] C. Zhang, R. Chen, and V. Prasanna, "High throughput large scale sorting on a cpu-fpga heterogeneous platform," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 148–155.

[7] Jian., "Progress with power systems and capi." Exploiting Accelerator Diversity for Cognitive Workloads - Workshop at MICRO'50. [Online]. Available: https://ibm.ent.box.com/v/OpenPOWERWorkshopMicro50/file/239719608792

[8] J. Kim and Y. Kim., "Hbm vs ddr3 comparison." Hot Chips 2014. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc26/HC26-11-day1-epub/HC26.11-3-Technology-epub/HC26.11.310-HBM-Bandwidth-Kim-Hynix-Hot%20Chips%20HBM%202014%20v7.pdf

[9] Y. Mulder, "Feeding high-bandwidth streaming-based fpga accelerators," Master's thesis, Delft University of Technology, Delft, 2018.

[10] K. Huang, "Multi-way hash join based on fpgas," Master's thesis, Delft University of Technology, Delft, 2018.

[11] Y. Qiao, "An fpga-based snappy decompressor-filter," Master's thesis, Delft University of Technology, Delft, 2018.

[12] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[13] M. Pharr and R. Fernando, *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[14] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 13–24.

[15] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proceedings of the 2014 ACM/SIGDA international symposium on Field-programmable gate arrays*. ACM, 2014, pp. 151–160.

[16] I. S. Myron Slota, POWER Systems, "Power processor technology overview," https://indico-jsc.fz-juelich.de/event/55/session/4/contribution/0/material/slides/0.pdf, 2017, [Online; accessed 05-January-2018].

[17] E. Stehle and H.-A. Jacobsen, "A memory bandwidth-efficient hybrid radix sort on gpus," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 417–432.

[18] Wikipedia, "Samplesort — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Samplesort&oldid=762743195, 2018, [Online; accessed 04-January-2018].

[19] B. Dong, S. Byna, and K. Wu, "Sds-sort: Scalable dynamic skew-aware parallel sorting," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 57–68.

[20] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968, pp. 307–314.

[21] M. Codish, L. Cruz-Filipe, T. Ehlers, M. Müller, and P. Schneider-Kamp, "Sorting networks: to the end and back again," *Journal of Computer and System Sciences*, 2016.

[22] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core simd cpu architecture," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1313–1324, 2008.

[23] W. Song, D. Koch, M. Luján, and J. Garside, "Parallel hardware merge sorter," in *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*. IEEE, 2016, pp. 95–102.

[24] A. Srivastava, R. Chen, V. K. Prasanna, and C. Chelmis, "A hybrid design for high performance large-scale sorting on fpga," in *ReConFigurable Computing and FPGAs (ReConFig), 2015 International Conference on*. IEEE, 2015, pp. 1–6.

[25] J. Jiang, L. Zheng, J. Pu, X. Cheng, C. Zhao, M. R. Nutter, and J. D. Schaub, "Tencent Sort," pp. 1–11, 2016.

[26] E. Solomonik and L. V. Kale, "Highly scalable parallel sorting," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[27] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, "Introduction to algorithms (Chapter 27: Sorting Networks)," 2001. [Online]. Available: http://euler.slu.edu/{~}goldwasser/courses/loyola/comp363/2003{_}Spring/handouts/course-info.pdf

[28] A. DAta, "Adm-pcie-9v3 user manual," https://www.alpha-data.com/pdfs/adm-pcie-9v3%20user%20manual.pdf, 2017.

[29] Nallatech, "Nallatech 250s+ datasheet," http://www.nallatech.com/wp-content/uploads/Nallatech-250S-Product-Brief-v0-1-7.pdf, 2017.

[30] X. F. Ken Chapman, "Get smart about reset: Think local, not global," https://www.xilinx.com/support/documentation/white_papers/wp272.pdf, 2008.

Appendix

A

Table A.1: True value table for all the level 1 of Fig. 3.12

Level1

Q0	Q1		probability	value
2	0	P_2	$\frac{1}{2^2} \times C_2^2$	0.25
1	1	P_0	$\frac{1}{2^2} \times C_2^1$	0.5
0	2	P_1	$\frac{1}{2^2} \times C_2^2$	0.25
Expectation			P_0*2+P_1 *1+P_2 *1	1.5

Table A.2: True value table for all the level 2 of Fig. 3.12

Level2

Q0	Q1		probability	value
0	4	P_0	$\frac{1}{2^4} \times C_4^0$	$\frac{1}{2^4}$
1	3	P_1	$\frac{1}{2^4} \times C_4^1$	$\frac{4}{2^4}$
2	2	P_2	$\frac{1}{2^4} \times C_4^2$	$\frac{6}{2^4}$
3	1	P_3	$\frac{1}{2^4} \times C_4^3$	$\frac{4}{2^4}$
4	0	P_4	$\frac{1}{2^4} \times C_4^4$	$\frac{1}{2^4}$
Expectation			P_0*2+P_1 *3+P_2 *4+P_3*3+4*2	3.25

Table A.3: True value table for all the level 3 of Fig. 3.12

Level3

Q0	Q1		probability	value
0	8	P_0	$\frac{1}{2^8} \times C_8^0$	$\frac{1}{2^8}$
1	7	P_1	$\frac{1}{2^8} \times C_8^1$	$\frac{8}{2^8}$
2	6	P_2	$\frac{1}{2^8} \times C_8^2$	$\frac{28}{2^8}$
3	5	P_3	$\frac{1}{2^8} \times C_8^3$	$\frac{56}{2^8}$
4	4	P_4	$\frac{1}{2^8} \times C_8^4$	$\frac{70}{2^8}$
5	3	P_5	$\frac{1}{2^8} \times C_8^5$	$\frac{56}{2^8}$
6	2	P_6	$\frac{1}{2^8} \times C_8^6$	$\frac{28}{2^8}$
7	1	P_7	$\frac{1}{2^8} \times C_8^7$	$\frac{8}{2^8}$
8	0	P_8	$\frac{1}{2^8} \times C_8^8$	$\frac{1}{2^8}$
Expectation				6.9