UCLA
UCLA Electronic Theses and Dissertations

Title
Customized Computing: Acceleration of Big-Data Applications

Permalink
https://escholarship.org/uc/item/4pk3897j

Author
Qiao, Weikang

Publication Date
2022

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
Los Angeles

Customized Computing: Acceleration of Big-Data Applications

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Electrical and Computer Engineering

by

Weikang Qiao

2022

© Copyright by

Weikang Qiao

2022

ABSTRACT OF THE DISSERTATION

Customized Computing: Acceleration of Big-Data Applications

by

Weikang Qiao

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Los Angeles, 2022

Professor Mau-Chung Frank Chang, Co-Chair

Professor Jingsheng Jason Cong, Co-Chair

Customized computing is gaining ever-increasing popularity in today's data center to meet the demand for high computational capabilities and energy efficiency. Among the existing customized processors, field-programmable gate array (FPGA) is one of the most promising candidates due to its flexible re-programmability and low power consumption. However, it is still unclear how to design efficient accelerators that entirely exploit the hardware features and deliver high performance for many big-data applications.

This dissertation focuses on using FPGAs to accelerate fundamental applications in data centers, specifically, sorting and compression, with the aim of providing high-performance sorting and compression solutions that can be scaled to fully take advantage of the hardware resources such as on-chip computing resources, off-chip memory bandwidth or I/O bandwidth to meet various application requirements.

The first part of the thesis explores how merge tree sort can be implemented and optimized on FPGAs to work efficiently with different problem sizes ranging from MB

to TB as well as various memory hierarchies such as DDR DRAM, high-bandwidth memory (HBM) and solid-state drive (SSD). On DRAM-based FPGAs, we show that a single merge tree can be optimally configured to saturate the off-chip memory bandwidth and minimize the number of merge passes. On HBM-based FPGAs, the performance bottleneck of the sorting acceleration is the limited on-chip resources. To alleviate the resource bottleneck, a two-phase merge tree sorter with resource sharing between the two phases is presented. Finally, to deal with the case that the problem size is larger than what can be held in the DRAM, we couple the merge tree with the emerging in-storage computing devices. The proposed in-storage sorter contains two separately optimized phases and can be re-programmed at runtime to switch between the phases.

In the second part of the thesis, we investigate the FPGA-accelerated lossless compression. We first present a scalable high-throughput lossless compression accelerator design that achieves state-of-the-art compression speed. The core of this accelerator is a multi-way parallel architecture where each way represents a well-optimized and fully-pipelined streaming compression engine. The compression engines are further equipped with proper data feeding mechanism and chained hash tables to avoid the potential degradation of the compression quality. Moreover, we also explore the possibility of accelerating high-quality lossless compression, the core of which performs Burrows-Wheeler Transform (BWT) on a data window containing hundreds of kilobyte symbols. We find the anti-sequential suffix sorting algorithm could be a good fit for hardware implementation. We utilize the abundant on-chip memory resources of the FPGA to construct a linked list and propose several optimization to reduce the searching time of the list. The proposed design is the first FPGA-based BWT solution that supports BWT window sizes of hundreds of kilobytes.

The dissertation of Weikang Qiao is approved.

Glenn D. Reinman

Nader Sehatbakhsh

Jingsheng Jason Cong, Committee Co-Chair

Mau-Chung Frank Chang, Committee Co-Chair

University of California, Los Angeles

2022

To my parents.

TABLE OF CONTENTS

1 Introduction **1**

2 Background and Related Work **7**

2.1 Algorithms 7

2.1.1 Merge Sort 7

2.1.2 Deflate 8

2.1.3 Burrows-Wheeler Transform 11

2.2 Hardware Platforms 13

2.2.1 DRAM-based FPGAs 13

2.2.2 HBM-based FPGAs 14

2.2.3 In-Storage Computing Devices with FPGAs 16

2.2.4 Tightly-Coupled CPU-FPGA Platforms 17

2.3 Related Work 19

2.3.1 Sorting Accelerators 19

2.3.2 High-Throughput Lossless Compression Accelerators 22

2.3.3 BWT Implementations on FPGAs 24

3 Sorting Acceleration on DRAM-Based FPGAs **25**

3.1 Merge Tree Sort Architecture 26

3.1.1 Streaming Hardware Merge Unit 26

3.1.2 Merge Tree 29
3.2 Modeling of Merge Tree Sort 30
3.2.1 Characterization of DRAM-based FPGA Sorting 31
3.2.2 Performance Model 33
3.2.3 Resource Model 33
3.2.4 Optimal Configurations 34
3.3 Implementation Details 35
3.3.1 Data Loading Units 35
3.3.2 Record Dispatching and Packing 37
3.3.3 Presorter 38
3.4 Experimental Evaluation 39
3.4.1 Experimental Setup 39
3.4.2 Model Validation 40
3.4.3 DRAM Sorting Results 41
3.4.4 Bandwidth Efficiency 42
3.4.5 Scalability in Record Width 44
3.5 Summary 45

4 Sorting Acceleration on HBM-Based FPGAs 46
4.1 Scalability Analysis of A Single Merge Tree 49
4.2 Architecture of TopSort 50
4.2.1 Two-Phase Merge Tree Sort 50

4.2.2 Logic Reuse Between The Two Phases 51
4.2.3 Architecture to Support Logic Reuse 53
4.2.4 Tuning Sorted Sequence Size from Phase 1 55
4.2.5 Memory Write Pattern in Phase 2 57
4.3 Analysis of TopSort 58
4.3.1 Performance Model 58
4.3.2 Resource Model 60
4.3.3 Optimized configuration on Xilinx U280 Board 60
4.4 HBM-Specific Profiling and Optimization 62
4.4.1 Resource Overhead From AXI Interface 63
4.4.2 Data Layout Consideration 63
4.4.3 Choice of Burst Size 64
4.4.4 Floorplanning Optimization 67
4.5 Evaluation 69
4.5.1 Experimental Setup 69
4.5.2 Merge Tree Configuration & Resource Utilization 70
4.5.3 Design Layout & Frequency 71
4.5.4 Overall Sorting Performance 72
4.5.5 Sorting Performance with Different Burst Sizes 75
4.5.6 Comparison with DRAM-based FPGA Sorters 76
4.5.7 Comparison with Scaled Single Merge Tree 78

4.5.8 Sensitivity to the Input Data 79
4.5.9 Comparison with GPU Sorter 81
4.6 Summary 82

5 Sorting Acceleration on In-Storage Computing Devices 83
5.1 System Architecture of FANS 85
5.1.1 Sorting Kernel 85
5.1.2 Sorting Phase Design 86
5.1.3 Merging Phase Design 87
5.2 Performance Modeling 91
5.2.1 Modeling of The Sorting Phase 92
5.2.2 Modeling of The Merging Phase 93
5.2.3 Optimized Configuration for Samsung SmartSSD 94
5.2.4 Architectural Insights 97
5.3 Evaluation 98
5.3.1 Experimental Setup 98
5.3.2 Design Configuration 99
5.3.3 Performance Breakdown 100
5.3.4 Impact of the DRAM 101
5.3.5 Comparison with Related Work 102
5.3.6 Benefits from In-Storage Acceleration 103
5.4 Summary 104

6 Acceleration of High-Throughput Lossless Compression 105

6.1 Single-Engine Fully-Pipelined Compressor 107

6.2 Optimization Associated with Scaling Engines 115

6.2.1 Multi-Engine Data Feeding Method 117

6.2.2 Hash Chain 119

6.2.3 Increase Bank Number 120

6.2.4 Switch Optimization 121

6.3 Evaluation 122

6.3.1 Experimental Setup 122

6.3.2 System Integration 123

6.3.3 Comparison of Deflate Accelerators 125

6.3.4 Scaling Effect 127

6.3.5 Design Tradeoff Evaluation 128

6.3.6 End-to-End Compression Throughput 130

6.4 Summary 132

7 FPGA-Based BWT Acceleration 133

7.1 Algorithm Selection 134

7.1.1 Suffix Sorting 134

7.1.2 Antisequential Suffix Sorting 137

7.2 Antisequential Suffix Sorting in Hardware 139

7.2.1 Direct Suffix List Design 139

7.2.2 Two-Level Suffix List Design 142
7.3 Evaluation 145
7.3.1 Experimental Setup 145
7.3.2 Resource Utilization 146
7.3.3 Performance Evaluation 147
7.3.4 Comparison of the Compression Quality 148
7.4 Summary 149

8 Conclusion 150

References 152

LIST OF FIGURES

2.1 Example of the Deflate algorithm, which consists of the LZ77 compression part and the Huffman encoding part. 9
2.2 Example of the BWT on a string "BANANA$". 11
2.3 The average compression ratio of the Calgary Corpus benchmarks when using the bzip2 standard with various BWT window sizes. 12
2.4 Architecture layout for modern DRAM-based FPGAs, there could be multiple FPGA dies and multiple DDR DRAM channels. 14
2.5 Connections from user logic to HBM channels on an FPGA. AXI Cvt is short for AXI Rate Converter, which is the AXI-based memory controller sub-module. Lateral connections between nearby crossbars enable AXI to access HBM channels that are not located in the same group. 15
2.6 High-level diagram of Samsung SmartSSD. 16
2.7 Intel-Altera HARP and HARPv2 CPU-FPGA architecture. 18
3.1 An example of 4-rate bitonic merge unit: each vertical line that connects two dots is a compare-swap element and the compare-swap elements in the same box are processed in the same cycle. c_{0-3} will be the outputs. . 27
3.2 The topology of a 4-rate streaming merge unit. 28

3.3 An example of the merge tree. 1-M and 2-M represent the 1-rate merge unit and the 2-rate merge unit. Different input sequences are stored in the different input FIFOs. The couplers are used to match the output rate of the lower-level merge units to the input rate of the merge units at the next level. 30

3.4 Data movement flow of DRAM-based merge tree sort. 31

3.5 The merge tree system implemented on the AWS EC2 F1 instance. The host can configure the merge tree kernel and prepare the data to be sorted through the PCIe DMA channels. 36

3.6 Sorting time per GB of various merge tree configurations. 40

3.7 Sorting time per GB of various merge tree configurations. 40

3.8 LUT utilization of various merge tree configurations. 41

4.1 Merge tree decomposition: four $p/4$-rate merge trees can be combined to build a larger p-rate merge trees. 52

4.2 Detailed dataflow of the merge tree reuse in the proposed two-phase approach: the four demuxes (trapezoids in the figure) are controlled by the same sel signal, which indicates whether the current phase is phase one or phase two. 53

4.3 The overall architecture of the two-phase merge tree sort. The merge trees rely on AXI interfaces to access HBM channels. The role of demux is covered in Section 4.2.5. 54

4.4 The active rate of one reused merge tree in phase 2, if the sequence in each channel is completely sorted. Initially, only the leaves feeding sub sequence 0 from each channel are active, then are the leaves feeding sub sequences 1, and so on. Leaves marked as yellow are actively feeding. Merge units partially marked mean they are idle half of the time. 56

4.5 The memory write behavior of phase 2. Writes from 4 AXI interfaces are used to match the tree's throughput. 57

4.6 Search space of different configurations when sorting (32-bit key, 32-bit value) pairs on the Xilinx U280 board. Each dot represents a possible configuration of k, (p_1, l_1), (p_2, l_2). The X-axis indicates the LUT utilization and the Y-axis indicates the theoretical performance of each configuration. The dot marked by a red circle is the actual configuration we choose. 62

4.7 Usage of the first 4 AXI interface. The same behavior is repeated for rest of the 12 AXI interfaces. 65

4.8 HBM channel performance of inter-crossbar & intra-crossbar behaviors when varying the AXI burst size. 66

4.9 The floorplan of the two-phase merge sorter on the Xilinx U280 FPGA. Each Super Logic Region (SLR) is a single FPGA die slice in the Stacked Silicon Interconnect (SSI) FPGA device. 68

4.10 The actual layout of the design. Number 0-15 label the 16 merge trees of phase 1. The orange part in the bottom die represents the extra logic used to form the wider merge tree of phase 2. 72

4.11 Sorting performance with different data sizes. The overall performance is calculated through dividing the data size by the total sorting execution time. 73

4.12 Overall sorting performance with the same AXI burst size in phase one but different AXI burst size in phase two. 76

4.13 Overall sorting performance when sorting (32-bit key, 32-bit value) pairs and 32-bit keys. The performance variation is due to the difference in the design frequencies. 79

4.14 Overall sorting performance when sorting 4 GB data of (32-bit key, 32-bit value) pairs with different key distributions. 80

5.1 The internal architecture of the sorting kernel. 86

5.2 The double buffering scheme with the host-side multi-threading in the sorting phase. 88

5.3 Illustration of using the DRAM as buffers for the flash in the merging phase. Here we only show the input buffers and omit the output buffer for simplicity. 89

5.4 Illustration of the double buffering scheme in the merging phase. There are three threads working in parallel seen from the host side. At any time of the process, the kernel thread is invoking the FPGA merging kernel to read from a set of read buffers and to write to the write buffer, while the p2p read thread and the p2p write thread are working with the other set of read buffers and the write buffer. 90

6.1 The architecture of a single-engine fully-pipelined Deflate compressor. . . 107

6.2 Current and next window of input string to be compressed. 109

6.3 Example of hash conflicts in a VEC=4 design. 110

6.4 Memory bank mapping to input strings. Here we perform string matching after selecting the right candidate strings from the 128 bit-width multiplexers. 111

6.5 Match selection examples: The numbers in the box represent the match length from each position. Black arrows are head matches (tail matches extending from the previous window); red arrows are the potential tail matches, and the position with a circle is the actual tail match: (a) the head does not overlap with the current tail; (b) if the head is covering the start of the tail match, the tail must be trimmed to start from the head pointer's position; (c) after performing the trimming in (b) the length is 1, so use a no-match at the head pointer's location; (d) after performing the trimming in (b) the length is 2, which is not allowed by the Deflate algorithm, so a no-match length 1 is used instead, and tail reach is changed; (e) similar to (d). (d) and (e) are the only cases in which a trimmed match will have length 2 and needs special handling. 113

6.6 Illustration of the parallel bit packing. 115

6.7 Cyclic data feeding. 117

6.8 Block data feeding. 118

6.9 Hash memory chain for each hash history. 119

6.10 Reduce hash conflicts by doubling the hash banks. 120

6.11 Optimized memory bank mapping to reduce resource utilization, here we use 5-to-1 multiplexers after string matching. Cand. X is short for the potential match candidate for Substring X. 121
6.12 Overview of the multi-way parallel and fully-pipelined Deflate accelerator implementation. 124
6.13 Flow chart of CPU-FPGA system execution. 125
6.14 Compression ratio when scaling from 1 engine to 4 engines. 127
6.15 Compression throughput for different files of Calgary Corpus datasets. . 131

7.1 Example of the BWT using suffix sort. 135
7.2 Example of the BWT using antisequential suffix sort. 138
7.3 High-level diagram for direct suffix list design. 140
7.4 Illustration of the direct suffix list. 141
7.5 Accelerating the search time by using two levels of list. 142
7.6 The overall flow of the two-level suffix list design. 143
7.7 Performance comparison for various implementations. 147

LIST OF TABLES

2.1 A summary of the existing FPGA-based sorter designs. The Complete column indicates if the work completely sorts the data and the Data size column shows the actual size of the data it sorts. The Analysis column lists whether the work does comprehensive analysis on either the sorting performance or the resource utilization. The efficiency is defined as the overall sorting performance in GB/s divided by the LUT utilization in the unit of 100K LUTs. 20

2.2 A summary of the existing FPGA-based high-throughput lossless compression accelerator designs. 23

2.3 A summary of the existing BWT implementations on FPGAs. 24

3.1 Parameters that impact the performance of Bonsai 32

3.2 Resource utilization breakdown of the best-performing DRAM sorter on AWS F1. 42

3.3 Comparison of the sorting time when sorting 32 GB data using different DRAM-based solutions. 43

3.4 LUT utilization and throughput of building-block elements. 44

4.1 The design frequency, the number of HBM Channels used and the effective HBM bandwidth utilized in prior HBM-based FPGA accelerators. 47

4.2 Parameters associated with TopSort 59

4.3 Correspondence between each user side AXI and the HBM channels it accesses, i=0-3. 64
4.4 Resources utilization and percentage of TopSort. 70
4.5 Resources utilization of individual merge trees. Here we pick one merge tree from each type since they have slightly different resource consumption. 71
4.6 Comparison with existing FPGA based DRAM sorters. 77
4.7 Comparison of resource utilization for various FPGA based in-memory sorters. The resource efficiency of the sorter is defined as the overall sorting performance in GB/s divided by the LUT utilization in the unit of 100K LUTs. 77
4.8 Comparison with the latest NVIDIA A100 GPU sorter. The unit of Perf./Watt is GB/s/W. 82

5.1 Extra parameters considered in FANS. 92
5.2 Detailed Configuration for Samsung SmartSSD. 99
5.3 Performance breakdown for each phase. 101
5.4 Comparison between FANS and [JXA17] 102
5.5 Effective communication bandwidth between the FPGA and the flash without P2P transfers. 104

6.1 Comparison of FPGA Deflate accelerators. 126
6.2 Scaling effect on the performance, compression ratio and resources. 128
6.3 Impact of changing hash chain depth 129
6.4 Double Clocks v.s. Double Banks 129

6.5 End-to-end compression throughput on HARP and HARPv2 130

7.1 Resources utilization 146

7.2 Comparison of the compression speed and the compression ratio of the high-quality BWT-based compression design and the high-throughput Deflate-based compression design in Chapter 6. 148

ACKNOWLEDGMENTS

During my six years' research in Center for Customizable Domain-Specific Computing (CDSC) at UCLA, I have worked with many smart and kind people and got countless support from them. This dissertation would not have been possible without them and it is my honor to recognize their contributions.

First, I would like to express my sincere gratitude to my two advisors, Professor Mau-Chung Frank Chang and Professor Jason Cong, for their continuous trust, support and guidance during my Ph.D study. On the one hand, as a new graduate student with little experience in customized computing, I was initially not confident nor comfortable in solving the complicate problems that require domain-specific knowledge. It is Professor Cong that directed me into the right research area and taught me how to identify the research opportunities from the architectural point of view and abstract the problem with the analytical modeling. On the other hand, Professor Chang constantly encouraged and guided me to overcome challenges ever since I was admitted into the Ph.D program. He always shared with me his unique vision on the research trend, educated me to analyze the research problem from a system perspective and inspired me to deliver first-class results. I feel extremely fortunate to be co-advised by these two distinguished professors and to receive joint training from two different areas, engineering and computer science.

I also want to express my appreciation to my doctoral committee members, Professor Glenn D. Reinman and Prof. Nader Sehatbakhsh, for their serving on my dissertation committee and constructive suggestions on this dissertation.

Many of my fellow researchers have contributed to the dissertation. Especially, I thank:

- Professor Zhenman Fang, for the extensive collaboration in the sorting and compression projects, as well as his help during my early Ph.D stage.
- Jieqiong Du and Michael Lo, for the collaborative effort in exploring the characteristics of CPU-FPGA communication.
- Nikola Smardzic, for our collaboration in enabling the complete sorting acceleration on FPGAs and making Bonsai presented in ISCA.
- Licheng Guo, for many helpful discussions in how to improve the design floorplan, the timing closure and the usage of HBM as well as being a nice roommate.
- Young-kyu Choi and Jihun Oh, for our collaboration and technical discussion in HBM-based sorting designs.
- Yan Zhao, Richard AI Hadi and Jia Zhou, for our collaboration in several sensor projects and their help in my Ph.D life.

I also want to thank my other fellow researchers—Peng Wei, Cody Hao Yu, Peipei Zhou, Zain Zhenyuan Ruan, Zhe Chen, Jessie Xu, Tianhe Yu, Mrunal Patel, Wan-Hsuan Lin, Jie Wang, Yuze Chi, Jason Lau, Atefeh Sohrabizadeh, Karl Marrett, Bochen Daniel Tan, Suhail Basalama, Stephane Pouget, Chengdi Cao, Neha Prakriya, Jason Kimko, Lorenzo Ferretti, Boyu Hu, Yan Zhang, Rulin Huang, Andrew Liu, Christopher Chen and Zhengrong Wang. I especially thank Jie and Zhengrong for being long-time companion and maintaining valuable friendship in my Ph.D life.

I want to appreciate Jason Lau and Yuze Chi for maintaining our server and providing reliable technical support.

I want to thank Professor Milos D. Ercegovac for offering me sincere and helpful suggestions in both research and career.

I want to express my thankfulness to Alexandra Luong, Janet Lin and Iris Li for their help with my graduate student paper work.

I also want to thank Janice Martin-Wheeler and Marci Baun for proofreading and editing my research papers.

I would like to thank Xuebin Yao, Hingkwan Huen, Han Chen, Nigel Gulstone and Danny Marquette, for their help during my internships at Samsung Memory Solution Lab and Amazon Web Services

I wish to greatly appreciate my girlfriend Bing Han, for always standing behind me and giving me encouragement, support and her unconditional love.

Last but most importantly, I would like to reserve my greatest gratitude to my parents — Xianxiang Qiao and Yuling Dai, for their incomparable love, care and support since I was born. Thank you for being so patient with me and understanding every choice that I have made during this long Ph.D journey.

The research studies in this dissertation are partially supported by the Intel Corporation with matching funds from the NSF under the Innovation Transition (InTrans) Program; fundings from the CDSC industrial partners, including Samsung and Siemens Mentor Graphics; CRISP, one of the six centers of the Joint University Microelectronics Program (JUMP); AWS Research Credits on Amazon EC2 services; and the Xilinx Adaptive Computer Clusters (XACC) program.

VITA

2011–2015	B.E., Information & Communication Engineering, Zhejiang University, Zhejiang, China.
2015–2017	M.S., Electrical Engineering, University of California, Los Angeles, U.S.A.
2017–present	Graduate Student Researcher., Electrical and Computer Engineering, University of California, Los Angeles, U.S.A.
2020	System Architect Intern, Samsung Memory Solution Lab, San Jose, CA
2021	RTL ASIC Design Intern, Amazon Web Services, East Palo Alto, CA

PUBLICATIONS

Weikang Qiao*, Licheng Guo, Zhenman Fang, Mau-Chung Frank Chang, Jason Cong, TopSort: A High-Performance Two-Phase Sorting Accelerator Optimized on HBM-based FPGAs (Poster), *The 30th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022.

Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, Jason Cong, RapidStream: Parallel Physical Implementation of FPGA HLS Designs, *The 2022 ACM/SIGDA International Symposium on FPGAs (FPGA)*, 2022. **(Best paper award)**

Weikang Qiao*, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, Jason Cong, FANS: FPGA-Accelerated Near-Storage Sorting, *The 29th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021.

Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, Jason Cong, HBM Connect: High-Performance HLS Interconnect for FPGA HBM, *The 2021 ACM/SIGDA International Symposium on FPGAs (FPGA)*, 2021.

Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, Jason Cong, AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs, *The 2021 ACM/SIGDA International Symposium on FPGAs (FPGA)*, 2021. **(Best paper award)**

Nikola Samardzic, Weikang Qiao*, Vaibhav Aggarwal, Mau-Chung Frank Chang, Jason Cong, Bonsai: High-Performance Adaptive Merge Tree Sorting, *The 47th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2020.

Weikang Qiao*, Zhenman Fang, Mau-Chung Frank Chang, Jason Cong, An FPGA-based BWT Accelerator for Bzip2 Data Compression, *The 27th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019.

Weikang Qiao*, Jieqiong Du, Zhenman Fang, Mau-Chung Frank Chang, Jason Cong, High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms, *The 26th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018.

CHAPTER 1

Introduction

The amount of data stored and processed in today's data center is growing at an unprecedented rate. Meanwhile, the performance advancement of general-purpose processors has slowed down significantly due to the end of Dennard scaling [DGY74]. To meet the demand for high computational capabilities and energy efficiency, customized computing has been introduced where one can customize the processor design to match the specific workloads using specialized accelerators [CGG12, RSW13, DTH20, WGC21]. Among the existing customized processors, field-programmable gate array (FPGA) is especially gaining popularity in the data center, as its reprogrammability enables flexible acceleration of the ever-evolving workloads [GMZ22] and its low power consumption results in impressive performance/watt gain [CSR10]. Nowadays, most of the cloud service providers have deployed FPGAs in their data center infrastructures. For example, Amazon Web Services (AWS) is integrating FPGAs into its storage to accelerate certain types of database queries [Bar21]. Microsoft has adopted FPGAs to accelerate the Bing search engine [PCC14].

While FPGA is a promising candidate to accelerate big-data applications, there is still a huge gap between the existing FPGA accelerators and the demanding requirement from the actual data center. As a result, today's data center designers usually have difficulties in designing efficient FPGA accelerators that can be scaled

to fully utilize the hardware features and deliver peak performance for the big-data applications. This dilemma is mainly due to three reasons. First, existing research on FPGA accelerators mostly concentrates on small-scale designs without considering the large-scale nature of the big-data applications. Second, even the same big-data applications but targeting different problems can have distinct characteristics, each requiring dedicated design considerations and vastly different hardware support. Finally, the hardware itself has also been continuously evolving, which means one specialized accelerator architecture can be obsolete and incapable of catching up with the hardware improvement.

There are various kinds of big-data applications in data centers [CO14, NMG15, WLZ15, CFH18b, CGH18, FMH19, GLR19, ZKP21, SCC22, SCG22] and it is impractical for one dissertation to cover all of them. Therefore, this dissertation focuses on the fundamental applications that are commonly found in today's data center. The first one is sorting, which is one of the most essential kernels in the relational database. The process of complete sorting contains abundant internal parallelism and could be an ideal application to be accelerated using customized hardware. For instance, using a customized comparator and multiplexer, one can compare the values of two records and swap them in one clock cycle. With multiple comparators and multiplexers, the relation of several pairs of records can be determined in parallel. In the past decade, many FPGA-based sorting accelerators have been proposed in the literature. Most of these sorting accelerators target on-chip sorting by limiting the problem size to be a few hundred kilobytes [KT11, CSP15, SKL16], or focus on designing building blocks such as sorting networks or merge units that can sort a couple of records in a parallel and pipelined fashion [CSP15, SKL16, ZMP16, MCK17, PBL18, SEC18, EK19]. While achieving competitive performance for small-scale

problems, these sorting accelerators cannot directly work in data centers, where the sequences to be sorted are usually much larger than what can be held on-chip. There are also a few works that consider using DRAM to store the intermediate and final results for the FPGA sorters [KT11, CO14, MK18, PBL20]. However, they only rely on the DRAM-based FPGAs for prototyping purpose and never investigate the performance impact of the associated hardware such as the off-chip DRAM bandwidth. Only two of the existing FPGA sorters support completely sorting sequences over one GB [JXA17, CMF20], but they only demonstrate the capability of working for certain problems and do not consider any potential bottlenecks that may limit the designs' scalability. In short, there are no existing scalable FPGA accelerated sorting solutions available to data center engineers.

In the first part of this dissertation, we propose a set of complete and high-performance sorting solutions that can be scaled to work with different problems while fully exploiting the hardware features. The proposed solutions are based on merge tree sort [KT11], which can be mapped onto FPGAs to exploit the inherent data-level and task-level parallelism of the sorting operations. Unlike prior works, our merge tree allows independently optimization of the merging throughput and the number of concurrently merged sequences. The structured architecture of the merge tree also enables comprehensive modeling of the performance and resource utilization. Thus, depending on the problem sizes and the available hardware, we can tune the proposed sorting designs to work efficiently with various memory hierarchies such as DDR DRAM, high-bandwidth memory (HBM) and solid-state drive (SSD). We start from the most common case where a conventional DRAM-based FPGA is used for sorting acceleration. In this situation, we show that a single merge tree can be optimally configured to work with the off-chip DRAM and the sorting perfor-

mance is primarily bounded by the available off-chip memory bandwidth. Later on, we investigate how to map the sorting acceleration onto the emerging HBM-based FPGAs, where the bottleneck is shifted from the off-chip memory bandwidth to the limited on-chip resources. To alleviate the resource bottleneck, a two-phase merge tree sorting solution with resource sharing between the two phases is proposed. Finally, to deal with the case that the problem size is larger than what can be held in the DRAM, we couple the merge tree with the first industrial in-storage computing device, Samsung SmartSSD. The proposed in-storage sorter also has two separately optimized phases and employs the unique re-programming feature of the FPGA to switch between the phases at runtime. The proposed DRAM, HBM and storage-oriented FPGA sorters all achieve state-of-art sorting performance and approach the hardware utilization limit. The first part of the dissertation is mainly based on the work from [SQA20, QOG21, QGF22].

The second big-data application explored in this dissertation is lossless compression. Lossless compression is the primary technique to save the storage space and the communication bandwidth in the cloud. According to [KDH15], roughly one quarter of the tax cycles are spent on compression related operations in data centers. Therefore, there is a growing need of accelerating lossless compression. There is always a trade-off between the compression quality and the compression speed. The better the compression quality achieved, the lower the compression speed is. Most of the existing FPGA-based compression accelerators are devoted to targeting high compression speed [AHS14, MJA13, FKB15]. These accelerators usually process the input data in a streaming fashion, where the compression speed is equal to the throughput of the compression engine. However, scaling current high-throughout compressors have encountered the bottleneck. First, the resource utilization grows

super-linearly with the compressor's throughput [FKB15]. Second, the critical path of the compressor also increases. As a result, further scaling the number of bytes the compressor processes per cycle will result in saturated throughput and lower area efficiency. On the other hand, there are also some hardware accelerators attempting to implement the key algorithm in high-quality lossless compression — Burrows-Wheeler Transform (BWT) [MCF05, CK13, PMM16, ZLW17]. The best one in [ZLW17] only supports BWT on a data window size of 4 KB and cannot be further scaled due to the limited on-chip resources and the complex routing issues, but practical high-quality compression requires executing BWT with a data window containing at least one hundred kilobyte symbols. Apparently, existing FPGA-based BWT implementations cannot catch up with the high-quality compression standards. In a word, existing FPGA-based compression implementations have poor scalability and cannot meet the increasing demand of today's big-data compression.

The second part of this dissertation discusses how to design both high-throughput and high-quality compressors on FPGAs, with the desire to meet the practical requirements of big-data compression in different scenarios. We first improve the scalability of the existing high-throughput compression accelerators by adopting a multi-way parallel architecture, where each way represents a well-optimized and fully-pipelined streaming compression engine. Compared to the existing high-throughput compression accelerators whose throughput cannot be further scaled, the scalability of the proposed design is significantly improved as the clock frequency is independent of the number of compression engines and the resource utilization grows linearly with the throughput. To maintain a comparable compression ratio to prior works, we also equip the compressor with a better data feeding method to reduce the loss of dictionary records as well as a chained hash table to increase the hash dictionary

history. We further study the impact of CPU-FPGA communication bandwidth on system-level compression throughput and illustrate that our design achieves 2.3× speedup over the previous record with an end-to-end compression speed of up to 12 GB/s. Moreover, we also discuss the possibility of accelerating high-quality lossless compression. Specifically, we explore how to enable an FPGA to perform Burrows-Wheeler Transform (BWT) on a data window containing hundreds of kilobyte symbols. Since the direct BWT implementation is extremely difficult to be supported on hardware, we find the anti-sequential suffix sorting algorithm could be promising alternative for FPGA implementations and propose several methods to further reduce the corresponding search time. Our BWT accelerator is the first FPGA-based implementation that supports up to 500 KB window size. The second part of the dissertation is mainly based on the work from [QDF18, QFC19].

The remainder of the dissertation is organized as follows. Chapter 2 introduces the necessary background and related work. Chapter 3 presents the sorting acceleration on the conventional DRAM-based FPGAs. Chapter 4 discusses how to scale merge tree sort on the emerging HBM-based FPGAs. Chapter 5 shows how merge tree sort can be properly deployed on existing in-storage computing devices. Chapter 6 exhibits a scalable high-throughput lossless compression accelerator design. Chapter 7 demonstrates the possibility of performing BWT with a window size of up to 500 KB on an FPGA. Finally, Chapter 8 concludes the whole dissertation.

CHAPTER 2

Background and Related Work

This chapters presents the background information and related work for this dissertation. We first introduce the sorting and compression algorithms commonly adopted in the data centers. Next, we review the mainstream FPGA platforms and identify the key features of each platform. Finally, we summarize the related work to this dissertation, including the previous studies on sorting, high-throughput lossless compression and BWT acceleration.

2.1 Algorithms

We present several algorithms for big-data sorting and compression in this section. First, we briefly introduce the merge sort algorithm, which is widely employed in the standard software stack. Next, we discuss the Deflate algorithm, which is the core of many lossless compression standards. Finally, we present the basic concept of the Burrows-Wheeler Transform and illustrate its usage in the high-quality compression.

2.1.1 Merge Sort

The merge sort algorithm [Knu98] is widely used as it has both the asymptotically optimal number of operations and predictable, sequential memory access patterns

(ideal for memory burst and coalescing). Moreover, due to its asymptotically optimal I/O complexity [AV88], merge sort is generally regarded as the preferred technique for sorting large amounts of data within a single computational node [SHG09].

Merge sort is a divide and conquer algorithm. It always divides the input array into two halves and calls itself again for the two halves until each of the sub arrays is sorted. Then it merges the sorted halves into a completely sorted array. Code 2.1 lists the pseudo code of the merge sort process, which recursively utilize one processing element to sequentially merge the arrays throughout the entire iterations. In the case where multiple processing elements are available, multiple arrays can be merged in parallel to reduce the number of iterations.

Code 2.1: Pseudo Code of Recursive Merge Sort

```
1  void merge_sort(arr[], left, right) {
2     if (left == right) arr[] is sorted;
3     if (left < right) {
4        mid = (left + right) / 2;
5        merge_sort(arr[], left, mid); // Sort the first half recursively
6        merge_sort(arr[], mid + 1, right); // Sort the second half
            recursively
7        merge(arr[], left, mid, right); // Merge the two sorted halves
8     }
9     exit;
10 }
```

2.1.2 Deflate

The Deflate algorithm is the core of many lossless compression standards such as ZLIB [GA22b] and GZIP [Deu96]. It mainly includes two stages: first, it performs the dictionary-based LZ77 [ZL77] compression; second, it performs the Huffman encoding [Huf52] to compress the output from the LZ77 phase at bit level. Figure 2.1

Figure 2.1: Example of the Deflate algorithm, which consists of the LZ77 compression part and the Huffman encoding part.

shows an example of the Deflate algorithm.

In the LZ77 stage [ZL77], we scan the incoming byte stream and compare the new input with the entries in a dictionary which is populated by past input data. After finding a common string of length L, this repeated section of the incoming string is replaced with a (L, D) pair, where D is the distance between the history and the incoming string. Then, the incoming string is recorded in the dictionary for future references. For example, in Figure 2.1, the incoming word "inflate" contains the same characters of "flate" as the existing word "Deflate" and their distance is 10 characters. As a result, we replace the second "flate" with a (5, 10) pair. The Deflate format limits the distance to 32K bytes and the length to 258 bytes, with a minimum length of 3.

Huffman encoding is known to produce the minimum length encoding given the

alphabet and the relative frequency of all characters. The Deflate algorithm has a dynamic Huffman encoding option, where it creates Huffman codes based on the relative frequency of the current input, resulting in the minimum length encoding for the given input's LZ77 result. However, it requires frequency tracking and must be done after the LZ77 phase. Dynamic Huffman encoding is typically used in higher levels of ZLIB and GZIP standards, where a high compression ratio is favored over throughput. Meanwhile, the Deflate algorithm also allows a static Huffman encoding option, where the Huffman code is generated by a golden frequency and is statically available. The static Huffman encoding option is usually favored in scenarios requiring high throughput, as it does not need to wait for the completion of the LZ77 phase and can start encoding as long as there are symbols coming.

Code 2.2: Pseudo Code of the Deflate Algorithm

```
1  while (curr_position < data_size) do
2      Hash table build
3          calculate hash value hv
4          prev_string <- Hash_Table[hv]
5          compare curr_string & prev_string
6          Hash_Table[hv] <- curr_string
7      String matching
8          candidate from the hash table v.s. current string
9      Match selection
10         find the longest match
11         Huffman encoding
12         move to next position
13 end while
```

Code 2.2 lists the pseudo code of the Deflate algorithm from [FKB15]. To increase the chance for the incoming stream to find a possible match from the past input, the history data are stored in a hash table. When a new byte comes, we calculate the corresponding hash value based on the content of the current 4-byte sequence, use

the hash value to fetch the corresponding 4-byte entry in the hash table, compare the current sequence and the history sequence, and then update the hash table with the new sequence. To improve the compression quality, the match found in this step is not immediately committed. Instead, it will be examined with the following bytes to see if a longer match exists. Once the longest match is found, we represent the match with the (L, D) pair and perform the static Huffman encoding.

2.1.3 Burrows-Wheeler Transform

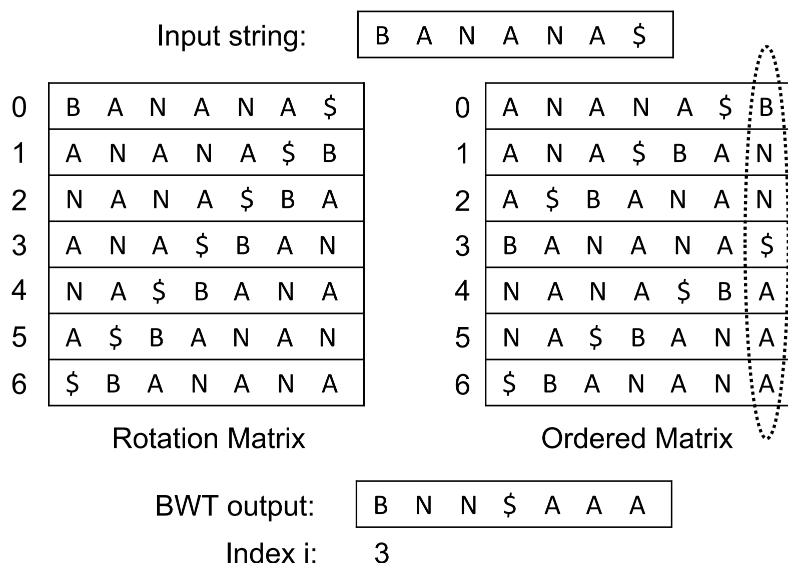

Figure 2.2: Example of the BWT on a string ”BANANA$”.

The Burrows-Wheeler Transform (BWT) [Bur94], invented by Burrows and Wheeler in 1994, has played an important role in high-quality lossless compression such as bzip2 [Sew19]. The basic algorithm of BWT is illustrated in Figure 2.2: given a string of length N, we generate all rotations of the string, sort the N rotations in lexicographical order and store them into the ordered matrix. The last column of the

ordered matrix will be the BWT output of the string; and an index i is also recorded, where i is the position of the original string in the ordered matrix.

Please note BWT itself does not compress data, but it rearranges the characters within the original string so that the same characters tend to be placed together, which can be easily compressed using other entropy encoding techniques. For instance, in the example of Figure 2.2, the BWT output of the string "BANANA$" is "BNN$AAA". These consecutive symbols of "N" or "A" can be replaced by a single symbol "N" or "A" and the corresponding repeated times so as to achieve compression.

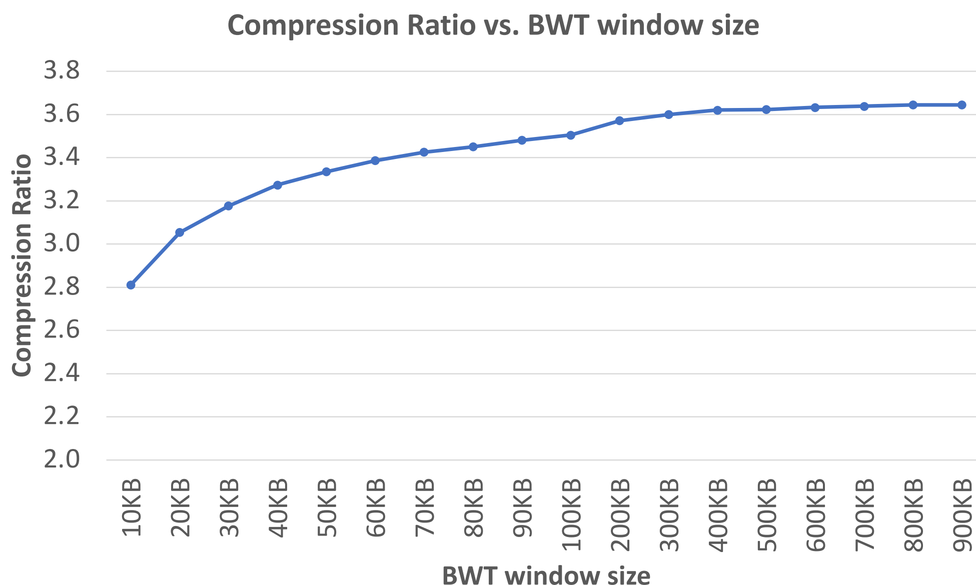

Figure 2.3: The average compression ratio of the Calgary Corpus benchmarks when using the bzip2 standard with various BWT window sizes.

The string length N is usually referred to as the BWT window size and it determines the compression quality of the compression algorithm. The larger the BWT

window size is, the better compression ratio will be achieved. Figure 2.3 shows the average compression ratio of the Calgary Corpus benchmarks when using the bzip2 compression standard with various BWT window sizes from 10 KB to 900 KB. In general, modern high-quality compression standards usually require a BWT window size of at least 100 KB to get high compression ratio. For instance, the bzip2 compression standard supports nine different compression levels with BWT window sizes ranging from 100 KB to 900 KB.

2.2 Hardware Platforms

In this section, we review the existing FPGA platforms, including the conventional DRAM-based FPGA boards, the emerging HBM-based FPGA boards, the computational storage integrating FPGAs and the tightly-coupled CPU-FPGA platforms.

2.2.1 DRAM-based FPGAs

The most common FPGA platforms in today's data centers are DRAM-based FPGAs, such as AWS F1 instance [Ama], Alibaba F1 and F3 instances [Ali20], Xilinx Alveo boards in Nimbix [Nim20] and Intel Stratix X [Int20a]. These FPGA boards usually consist of one to four FPGA dies and DDR DRAM channels, as shown in Figure 2.4. The communication between a DDR DRAM and the user logic of an FPGA die is through an AXI interface. The users may need to manually configure the parameters of the AXI interface such as the data width, the burst size and the number of outstanding bursts to fully utilize the off-chip memory bandwidth. Typically, one DDR DRAM channel delivers 14-18 GB/s peak performance for sequential reads or writes, depending on the built-in DRAM transfer rate [LFL21].

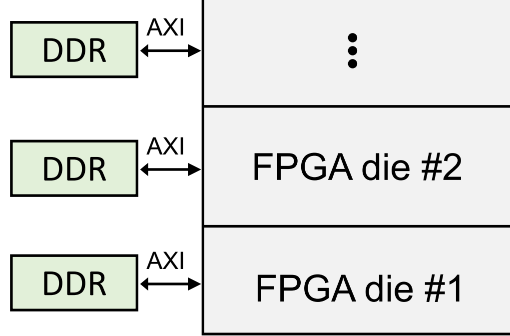

Figure 2.4: Architecture layout for modern DRAM-based FPGAs, there could be multiple FPGA dies and multiple DDR DRAM channels.

2.2.2 HBM-based FPGAs

HBM achieves higher bandwidth than DDR4 DRAMs by stacking multiple small DRAM dies together and is one of the most promising candidates enabling memory-centric designs [GAK15]. Recently, major FPGA vendors such as Intel and Xilinx have released HBM2-based FPGA boards: Xilinx's Alveo U50 [Xil21b], U280 [Xil21a] and Intel's Stratix 10 MX [Int20b]. Taking Xilinx U280 board as an example, the board is equipped with 2 HBM2 stacks and each stack contains 16 pseudo channels. The aggregate off-chip bandwidth a user can get from such HBM-based FPGAs is up to 460 GB/s [CCW20, CCQ21]. Figure 2.5 exhibits the connections between the user logic and the HBM channels. Each HBM channel can be accessed through a 256-bit wide AXI interface running at 450 MHz. The vendor tool will by default implement AXI rate converters (denoted as AXI Cvt in the figure) in the HBM Memory Subsystem (HMSS) to adapt the original 256-bit AXI interfaces to 512-bit AXI interfaces running at 225 MHz to the user logic. In the case of routing congestion which happens if the on-chip resources are over-utilized or the design

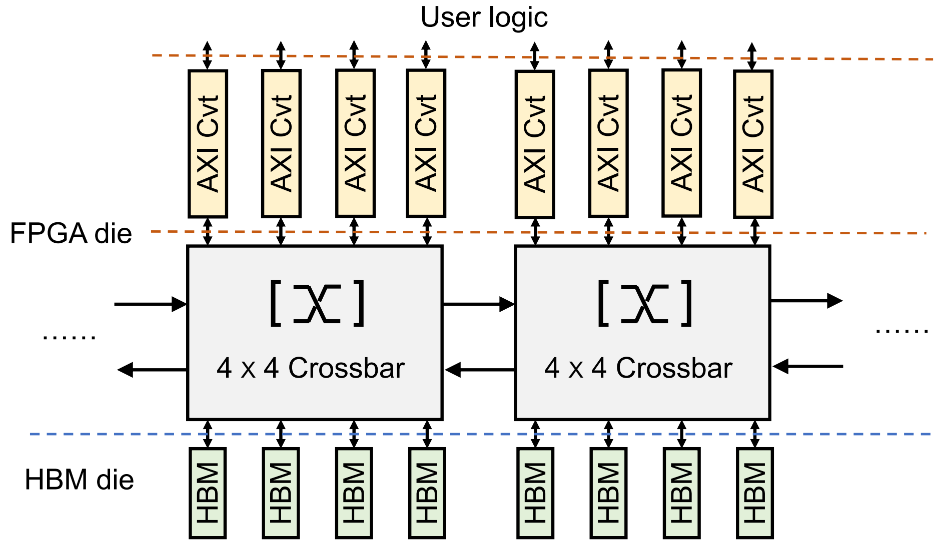

Figure 2.5: Connections from user logic to HBM channels on an FPGA. AXI Cvt is short for AXI Rate Converter, which is the AXI-based memory controller sub-module. Lateral connections between nearby crossbars enable AXI to access HBM channels that are not located in the same group.

is not well pipelined, both the HBM-side AXI frequency and the user-side design frequency will be reduced and the available HBM bandwidth will be degraded.

There is no global crossbar to allow 32 AXI interfaces to access the 32 HBM channels at the same time. Instead, the 32 HBM channels are physically bundled into 8 groups and each group contains 4 adjacent channels joined by a built-in 4×4 crossbar, as is shown in Figure 2.5. The crossbar provides full connectivity within the group. Meanwhile, each AXI interface at the user side can still access any HBM channels outside its group. The data will sequentially traverse through each of the lateral connections until it reaches the crossbar connecting to the target channel.

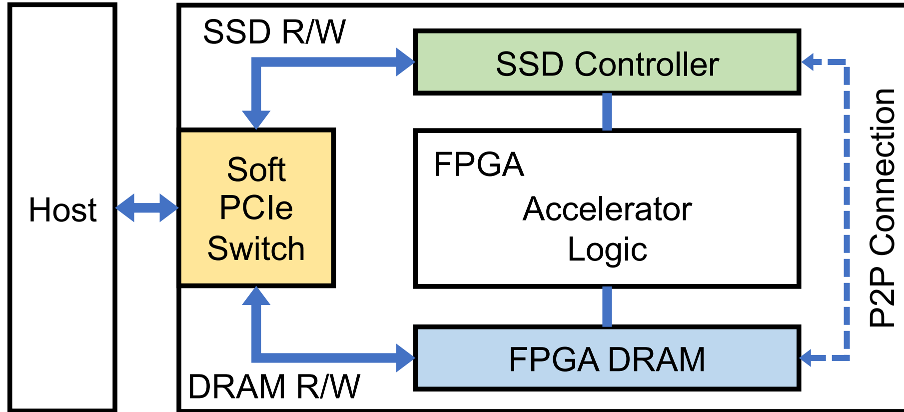

Figure 2.6: High-level diagram of Samsung SmartSSD.

2.2.3 In-Storage Computing Devices with FPGAs

FPGA is a good fit for in-storage computing, due to its high reconfigurability, massive parallelism and high energy efficiency [RHC19]. Samsung SmartSSD is the first industrial programmable computational storage, where a Xilinx KU15P FPGA is integrated with a 3.84 TB NAND flash into the same package in the U.2 format [Xil20a]. The flash itself has a sequential read and write bandwidth of 3.1 GB/s, respectively. It also reports a random read rate of up to 800K IOPS and a random write rate of up to 135K IOPS. With SmartSSD, many of the computational tasks can be offloaded from the host to the FPGA, which is next to the flash. By reducing the data transfer between the host and the flash, one could potentially save the host-drive bandwidth and boost the kernel performance [LZL20].

As seen in Figure 2.6, the datapath from the drive to the host and the FPGA is through PCIe Gen3.0×4 [JB12]. The FPGA inside SmartSSD is roughly 3× smaller

than a datacenter FPGA due to the cost and power consideration and contains near 500K LUTs and 1 million Flip-Flops. The FPGA is also equipped with a 4 GB DRAM, which is exposed to the FPGA kernels as its internal DRAM and to the host as a common memory area (CMA). The CMA is exposed to the host address space as a PCIe Base Address Register (BAR) and can be mapped into an application address space using buffer allocation. Once mapped, the host can initiate peer-to-peer (P2P) transfers between the SSD and the FPGA DRAM. Note that although the P2P transfer is initiated by the host, the actual data transfer will directly go through the PCIe link connecting the SSD controller and the FPGA DRAM without host involvement.

2.2.4 Tightly-Coupled CPU-FPGA Platforms

The aforementioned FPGA platforms all have their own device memory such as DRAMs or HBMs and rely on direct memory access (DMA) to access the data from a CPU. First, the FPGA needs a memory controller IP to read the data from the CPU's DRAM to its own DRAM through PCIe. And then the FPGA performs specified acceleration in its accelerator function units (AFUs). This DRAM-to-DRAM communication may introduce significant overhead when accelerating streaming applications [CCF16].

Another type of CPU-FPGA platforms features no device memory and the FPGA can directly access the CPU's memory through cache-coherent protocols. One known example is the Intel-Altera HARP platform with the QuickPath Interconnect protocol (QPI) [Int16b]. For instance, HARPv2 uses one QPI channel and two PCIe channels within a single package to connect the CPU cores and FPGA accelerators,

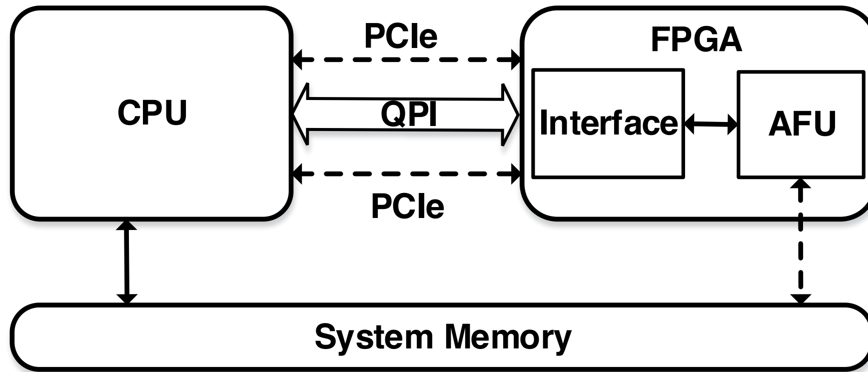

Figure 2.7: Intel-Altera HARP and HARPv2 CPU-FPGA architecture.

as shown in Figure 2.7. The CPU and FPGA can communicate using the shared memory to significantly increase the communication bandwidth. AFU can read/write data directly from/to system virtual memory through the core-cache interface (CCI). This makes FPGA have first-class access to the system memory and thus achieve a high bandwidth and low latency for CPU-FPGA communication. According to the study in [CCF16], HARP can provide 7.0 GB/s FPGA read bandwidth and 4.9 GB/s FPGA write bandwidth. Using the same benchmark method in [CCF16], we find that HARPv2 has further improvement and can provide more than 15 GB/s FPGA read and write bandwidth. Moreover, this shared system memory architecture eliminates explicit memory copies and increases the performance of fine-grained memory accesses. Such high communication bandwidth and low latency between the CPU and the FPGA provide opportunities to achieve a much higher end-to-end performance for streaming applications in practice.

2.3 Related Work

2.3.1 Sorting Accelerators

Many FPGA-based sorting accelerators have been proposed in the literature and Table 2.1 summarizes the features of the existing FPGA-based sorting designs. A significant portion of them focuses on designing building blocks such as streaming sorting networks or merge units. [ZMP16] presents a domain-specific language to automatically generate RTL designs of sorting networks given the high-level description of the required throughput and area. [SKL16, MCK17, PBL18, SEC18, EK19] propose various kinds of high-throughput merger units, which can merge two sorted sequences in a parallel and pipelined fashion. These building blocks are necessary components for designing efficient sorters on FPGAs, but they themselves cannot sort a complete problem. [ZMP12, CSP15] propose two bitonic sorters that could sort kilobyte-scale data by designing huge on-chip bitonic sorting networks, but they could not sort data whose size is larger than several megabytes. Our sorting solutions in this dissertation benefit from the advancement of these studies in that using efficient merge units may always reduce the entire resource consumption.

There are some FPGA-based sorting accelerators working on completely sorting with the external memory. [MNC09] presents a design that contains one FIFO-based merge unit and the corresponding control logic to sort a problem recursively. This method does not make good of the abundant parallelism of the FPGAs and can only merge two sequences in one pass. [KT11] presents the first merge tree-based sorter and adds small FIFOs among different tree levels to avoid back-pressure flow controls. By using the tree-like architecture, it can merge multiple sequences simultaneously,

Table 2.1: A summary of the existing FPGA-based sorter designs. The Complete column indicates if the work completely sorts the data and the Data size column shows the actual size of the data it sorts. The Analysis column lists whether the work does comprehensive analysis on either the sorting performance or the resource utilization. The efficiency is defined as the overall sorting performance in GB/s divided by the LUT utilization in the unit of 100K LUTs.

	Complete	Data size	Analysis	Efficiency
FCCM'17-2 [MCK17]	No	N/A	N/A	N/A
FCCM'18 [SEC18]	No	N/A	N/A	N/A
FPT'18-2 [PBL18]	No	N/A	N/A	N/A
MCSoc'19 [EK19]	No	N/A	N/A	N/A
TC'22 [PLB22]	No	N/A	N/A	N/A
DAC'12 [ZMP12]	Yes	KB	Yes	N/A
FPGA'15 [CSP15]	Yes	KB	Yes	N/A
FCCM'16 [SKL16]	Yes	KB	Yes	N/A
FPGA'16 [MRL16]	Yes	KB	No	0.72
FPGA'11 [KT11]	Yes	MB	No	0.22
MEMOCODE'08 [FKN08]	Yes	MB	No	0.87
FPGA'14 [CO14]	Yes	MB	No	N/A
FPL'20 [PBL20]	Yes	MB	Yes	0.5
FPGA'20 [CMF20]	Yes	GB	No	1.3
FPT'18-1 [MK18]	Yes	GB	Yes	N/A
FCCM'17-1 [JXA17]	Yes	TB	No	N/A

but its throughput is only 1/16 of the available bandwidth of a single DRAM channel and cannot be scaled. [CSP15] and [PBL20] give some analysis of the sorting performance on the FIFO-based merge unit implementation and the merge tree sort implementation, respectively. However, their analysis is incomplete and the problem sizes that their designs can sort are reported to be on the order of MB. [CMF20] presents a hardware-software co-design for sample sort, where the sampling part is done on the CPU side and the sorting part is done on the FPGA side. This design achieves good performance when sorting less than 2^{30} records. Sorting problems larger than this limit will incur large overhead due to the slow sampling step on the CPU side. In general, these works lack of comprehensive analysis of the complete sorting process and the readers cannot derive direct intuition of how they can further scale these designs for better sorting performance.

There are also a few studies considering the presence of the storage in the sorting process. [MK18] presents a many-leaf tree that can merge up to 1024 sequences while maintaining a throughput of 1 record per cycle. Such a tree can be a good fit for storage-based sorting because saturating the slow I/O does not require the tree throughput to be high-rate. However, this design fails to consider the practical limitation of the long access latency of the storage and can be futile in real world applications. [JXA17] builds an FPGA-accelerated flash sorter where a DRAM-based FPGA board is directly connected to an SSD through the FMC ports. This work focuses on designing a merge tree to saturate the slow I/O bandwidth, but neglects the optimization of other design choices. In fact, the bottlenecks of the storage-based merge sort are more than just the flash storage bandwidth and can result from a variety of issues, including the available FPGA resources, the device DRAM bandwidth and other design choices, as will be shown in this dissertation. These bottlenecks

are not obvious unless there is a systematic analysis that puts the whole sorting process together and examines all of the system parameters that may impact the overall performance. Such an analytical framework is of great importance for designing FPGA-based large-scale sorting accelerators but is still missing in the current works.

2.3.2 High-Throughput Lossless Compression Accelerators

A lot of efforts have been made to increase the throughput for high-throughput compression accelerators in the FPGA community and Table 2.2 summarizes the throughput and design frequency of the existing FPGA-based high-throughput lossless compression accelerator designs. The primary goal is to increase the number of bytes processed per cycle while maintaining a desirable clock frequency. Altera [AHS14] implements a Deflate accelerator using OpenCL that processes 16 bytes per cycle at a kernel frequency of 193 MHz. They achieve a 2.84 GB/s throughput and 2.17x compression ratio at the expense of using 47% of the logic and 70% of the RAM. IBM also presents a Deflate accelerator [MJA13]. which achieves a throughput of 16 bytes/cycle but improves the design frequency to 250 MHz. However, its scalability is limited due to certain architectural choices like a 256-port hash table. Microsoft [FKB15] improves the compression throughput to 5.6 GB/s by scaling up to 32 byte/cycle at a clock frequency of 175 MHz, which made a record of compression throughput on FPGAs. Further scaling these designs will be even harder as they already consume large area, and from [FKB15], the growth of FPGA area usage is much faster than the incremental bytes processed per cycle. The kernel's running frequency may also drop with the FPGA area usage increasing, which will degrade

the performance improvement.

Table 2.2: A summary of the existing FPGA-based high-throughput lossless compression accelerator designs.

	Frequency	Throughput
IBM'11 [SAB11]	125 MHz	4 GB/s
IBM'13 [MJA13]	250 MHz	4 GB/s
Altera'14 [AHS14]	193 MHz	2.84 GB/s
Microsoft'15 [FKB15]	175 MHz	5.6 GB/s

An alternative approach to improve the compression throughput is taking advantage of the task-level parallelisms. IBM proposes a multi-way parallel compression engine design based on the Deflate algorithm [SAB11]. They implemented a single compression engine that processes 8 bytes per cycle at a clock frequency of 125 MHz and achieves a compression ratio of 1.96. By applying 4 engines they can get a throughput of 4 GB/s, but the compression ratio will be further sacrificed based on our study in this dissertation. Moreover, there are no system interface and data feeding methods to support the multi-way parallel compression kernels, and thus no in-depth analysis or solid implementation when one scales the compression engine to achieve larger equivalent throughput, as will be done in this dissertation. Another Xpress9 compressor [KHB14] integrates 7 engines to support heavily multi-threaded environments. However, these parallel engines are used for independent compression tasks and are not intended for increasing the throughput of a single compression task.

2.3.3 BWT Implementations on FPGAs

Table 2.3: A summary of the existing BWT implementations on FPGAs.

	Max Window Size
ReConFig'13 [CK13]	4 KB
IPDPS'16 [PMM16]	128 B
FPT'17 [ZLW17]	4 KB

Direct BWT implementation is extremely difficult to be mapped onto the FPGAs and there are only three works found in the literature [CK13, PMM16, ZLW17], as listed in Table 2.3. They all focus on building efficient parallel sorting networks that can support the lexicographical sorting described in Figure 2.2. The best one in [ZLW17] builds a sorting network that can process up to 4KB window size in parallel in 4,049 cycles. However, those sorting networks can only do one iteration of the sorting. To achieve a complete BWT implementation, the string needs to be stored and fed into the sorting networks multiple times until no equal consecutive characters appear any more, making the total execution still time-consuming. Besides, the window size that the sorting networks can process is very small. In fact, it takes $O(N)$ stages for such a sorting network to process N characters, making the total number of the compare and swap elements $O(N^2)$. Considering the bzip2 standard requires performing BWT on a window size of 100s of KB to achieve good compression ratio, it is impractical to implement BWT with such sorter-based approach for the high-quality compression standards.

CHAPTER 3

Sorting Acceleration on DRAM-Based FPGAs

Sorting data has always been one of the most fundamental computational kernels in the data center and there is a growing interest in using FPGAs to accelerate the sorting task. While FPGAs have been shown to be very effective in sorting data sets that fit on chip, big-data sorting usually has a much larger problem scale and requires the external memory to hold the entire problem. As a result, the sorting performance could be explicitly related to the problem size, the performance of the on-chip sorting logic and the off-chip memory bandwidth. Unfortunately, there is no existing FPGA-based approach that entirely examines the impacts of these factors and proposes a complete yet efficient solution to support large-scale sorting. On the one hand, it is unclear how to leverage the existing building blocks such as high-throughput merge units to build a realistic sorting system on FPGAs that works smoothly with the off-chip memory and sorts large-scale problems. On the other hand, it is not obvious how to optimize the design that best utilizes the on-chip resources as well as the off-chip memory bandwidth to deliver the peak performance given a specific problem and a concrete FPGA board.

In this chapter, we target at the most common scenario of big-data sorting, where an off-chip DDR DRAM is employed to store the entire problem. We present Bonsai, a scalable merge tree-based FPGA sorter that can be optimally adapted to the on-

chip resources and the off-chip DRAM bandwidth. The proposed merge tree is built from multiple layers of multi-rate streaming merge units and the structured merge tree architecture allows for the independent optimization of the merge tree's throughput and the number of concurrently merged sequences. To achieve the best sorting performance, the proposed merge tree is further coupled with a set of comprehensive models that precisely captures the sorting performance and the resource utilization. We implement the complete DRAM sorting system on the Amazon AWS F1 cloud FPGA and demonstrate that Bonsai achieves state-of-the-art sorting performance as well as superior bandwidth efficiency compared to existing DRAM-based CPU and FPGA sorters.

3.1 Merge Tree Sort Architecture

In this section, we first present how the bitonic merge unit can be employed to build a streaming hardware merge unit that merges two sorted streams. Then we show the architecture of a merge tree and illustrate how a merge tree can be built on top of the hardware merge units with various rates.

3.1.1 Streaming Hardware Merge Unit

A hardware merger takes two sorted sequences of elements as its inputs and then merges them into one sorted sequence. Specifically, an E-rate hardware merge unit takes E input elements from its two E-element wide inputs and outputs E sorted elements every cycle. The compare-swap cell is the basic building block for hardware merge unit, which compares two elements' values and swaps them into the desired ordering. A compare-swap cell usually contains a comparator and a 2-input multi-

plexer, which are suitable to be implemented using the Look-Up Tables (LUTs) on the FPGAs [KT11]. The bitonic merge unit shown in Figure 3.1 is one of the most widely used hardware merge unit, where each vertical line that connects two dots is a compare-swap cell and the compare-swap cells in the same box are processed in the same cycle. An E-rate bitonic merger requires $\log E \cdot (\log E + 1)/2$ cycles to get the correct outputs. In Figure 3.1, c_{0-3} will be the outputs after 3 cycles. Since there are $\log E \cdot (\log E + 1)/2$ pipeline stages in an E-rate bitonic merge unit and each pipeline stage requires E compare-swap cells, the total number of compare-swap cells consumed by such an E-rate bitonic merge unit is $E \cdot \log E \cdot (\log E + 1)/2$.

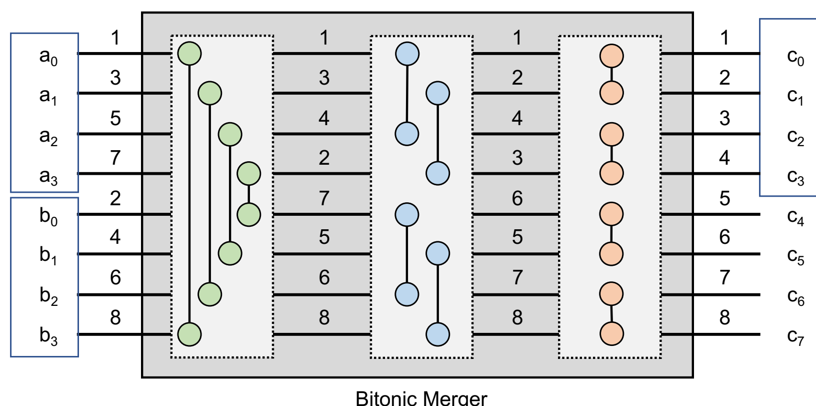

Figure 3.1: An example of 4-rate bitonic merge unit: each vertical line that connects two dots is a compare-swap element and the compare-swap elements in the same box are processed in the same cycle. c_{0-3} will be the outputs.

If the two sorted input sequences are longer than E, the E-rate merge unit has to be time multiplexed (i.e., executed multiple rounds) with extra control signals to ensure the outputs are in order. For example, in Figure 3.1, c_{4-7} represent the largest four elements of a_{0-3} and b_{0-3}. In the next cycle, c_{4-7} need to be sent back to the input side of the same bitonic merge unit and merged with either a_{4-7} or b_{4-7}

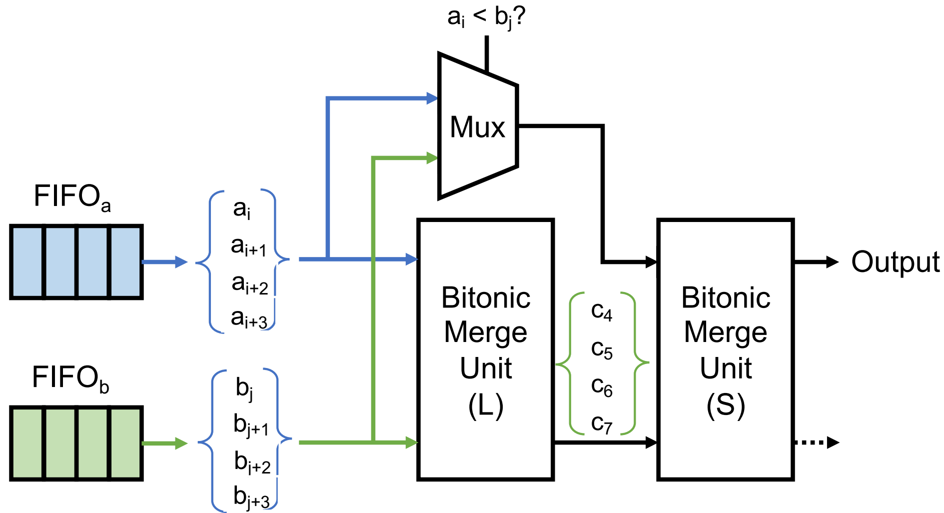

Figure 3.2: The topology of a 4-rate streaming merge unit.

to create the second 4-element sorted outputs. The feedback paths from the output side of the bitonic merge unit to its input side mean the merge unit has to wait for extra cycles before it can do the next merge operation. In other words, the initial interval (II) is equal to the number of pipeline stages in the bitonic merge unit.

To improve the merging performance with an II equal to 1, we adopt an E-rate merge unit from [SEC18] that outputs E elements every cycle using two bitonic merge units, as shown in Figure 3.2. The intuition is that the larger half outputs of the original bitonic merge unit can be first calculated through the bitonic merge unit (L) and are later fed into the bitonic merge unit (s) with delayed inputs from either sequence a or b. One advantage of such topology is that it is free of feedback paths and helps alleviate the design routing congestion. To enable the E-rate merge unit to have the streaming behavior, we use two E-element wide FIFOs to feed the inputs into the merge units and rely on the FIFO empty signal to determine if there are

more elements needed to be processed. Note that we may use the merge units from other works such as FLiMSj from [PLB22] to build the streaming hardware merge units as long as these newly proposed merge units reduce the resource consumption.

3.1.2 Merge Tree

Using a combination of various streaming hardware merge units with different rates, we can build a complete binary tree that consumes l unsorted input sequences concurrently at its leaves and outputs p sorted elements at its root every cycle. Figure 3.3 shows the architecture of a merge tree, which can be uniquely denoted by (p, l).

To implement a (p, l) merge tree, we put a p-rate streaming merge unit (denoted as p-M in Figure 3.3) at the root, two $p/2$-rate streaming merge units ($p/2$-M) as its children, then four $p/4$-rate streaming merge units as their children, etc., until the binary tree has $\log_2 \ell$ levels and can thus merge ℓ sequences. In general, the tree nodes at the k-th level are $p/2^k$-rate streaming merge units. Notably, l does not have to be equal to p and can be arbitrarily larger than p. In that case, for a given level k where $2^k > p$, we use 1-rate streaming merger units. For example, to scale the ($p = 16$, $l = 16$) merge tree from Figure 3.3 to a ($p = 16$, $l = 32$) merge tree, we can have another layer of 32 1-M merge units on top of the current leaf layer.

In order to feed the output of a $p/2$-rate streaming merge unit to the next-level p-rate streaming merge unit, we place couplers in between tree levels to match the streaming rate. The core of a coupler is a FIFO, but it has two distinct features. First, a p-rate coupler always concatenates the adjacent two $p/2$-element tuples into one p-element tuple before writing to its internal FIFO. Second, the coupler relies on the programmable full signal of its internal FIFO to control the inputs of the

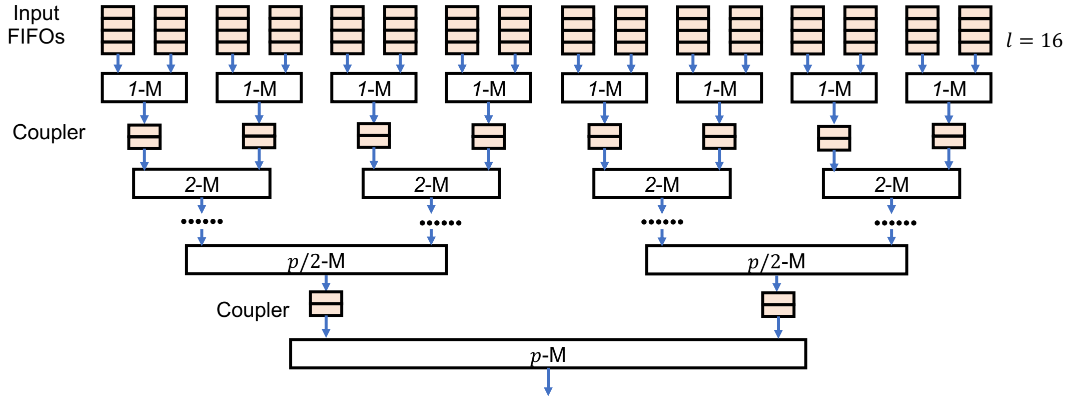

Figure 3.3: An example of the merge tree. 1-M and 2-M represent the 1-rate merge unit and the 2-rate merge unit. Different input sequences are stored in the different input FIFOs. The couplers are used to match the output rate of the lower-level merge units to the input rate of the merge units at the next level.

streaming merge unit at its previous level. The programmable full threshold is equal to the FIFO depth minus the pipeline depth of the streaming merge unit at the previous level. If the programmable full signal is asserted, that means the next-level streaming merge unit cannot process the inputs from the coupler's FIFO. Thus the previous-level streaming merge unit will stop feeding more data and only the data having been consumed by it is allowed to flow into the coupler's FIFO.

3.2 Modeling of Merge Tree Sort

In this section, we first characterize the DRAM-based FPGA sorting, including its data movement flow and the necessary parameters. Then, we introduce the performance and resource model of a merge tree that works with DRAM. Based on the

derived models, we also show the optimal merge tree configuration parameters given the specific hardware.

3.2.1 Characterization of DRAM-based FPGA Sorting

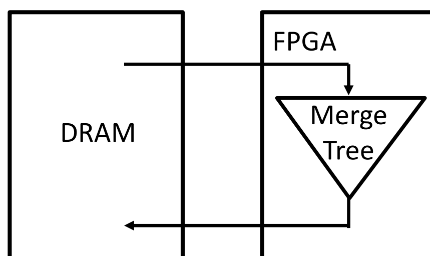

Figure 3.4: Data movement flow of DRAM-based merge tree sort.

Merge tree sort is run by recursively merging sequences through the merge tree. In one sorting pass, the input elements are streamed from the off-chip memory into the input FIFOs and then through the merge tree, and the output elements are streamed back into the off-chip memory, as shown in Figure 3.4. Assume there are N unsorted elements initially stored in the off-chip memory, these N elements need to be streamed into the merge tree for multiple passes and after each pass, the size of the partially sorted sequence grows exactly by l times. During the first pass, the merge tree reads these N sub sequences (each containing one element) from the off-chip memory, merges them into N/l sorted sub sequences, each containing l sorted elements, and writes them back to the off-chip memory. In the next pass, the l-element sorted sub sequences are fed into the merge tree again to get N/l^2 sorted sequences of length l^2. These steps are repeated until the last pass, where l sorted sequences of length N/l are processed by the merge tree to form a complete N-element sorted sequence. In general, the k-th pass will produce ℓ^k-element sorted

subsequences. Therefore, the total number of merge passes required to sort an N-element array is $\lceil \log_\ell N \rceil$.

As one can imagine, merging more sequences (i.e., increasing ℓ) reduces the total number of passes required, thereby reducing the sorting time. On the other hand, using a merge tree with higher throughput (i.e., increasing p) reduces the execution time of each pass. Thus, there is a natural trade-off between p and ℓ, as increasing either of them requires using limited on-chip resources.

Table 3.1: Parameters that impact the performance of Bonsai

	Symbol	Definition
Input Param.	N	Number of records in array
	r	Record width in bytes
Hardware Param.	β_{DRAM}	Bandwidth of the DRAM
	C_{BRAM}	On-chip memory capacity in bytes
	C_{LUT}	Number of on-chip logic units
Merge Tree Param.	f	Design frequency
	p	Merge tree throughput
	l	Number of leaves
	b	Size of read batches in size

In fact, the available hardware, the merge tree architecture and the data input all play important roles in the sorting performance. Table 3.1 lists the specific parameters considered by this thesis and we will examine their impacts in details in the following subsections.

3.2.2 Performance Model

As discussed in Section 3.2.1, the total number of merge passes required to sort an N-element array using a (p, l) merge tree is $\lceil \log_\ell N \rceil$. The amount of time required for each pass depends on the throughput of the merge tree (=pfr) and the available off-chip DRAM bandwidth, denoted β_{DRAM}. Since the merge tree streams the merged results directly back to the off-chip DRAM, the actual throughput is $\min\{pfr, \beta_{\mathrm{DRAM}}\}$. Thus, the time needed to complete merging N r-byte records in each pass is $Nr/\min\{pfr, \beta_{\mathrm{DRAM}}\}$. The sorting time is equal to the amount of time needed to complete all $\lceil \log_\ell N \rceil$ passes:

$$\text{Latency} = \frac{Nr \cdot \lceil \log_\ell N \rceil}{\min\{pfr, \beta_{\mathrm{DRAM}}\}}. \tag{3.1}$$

3.2.3 Resource Model

The merge tree's structured architecture allows us to develop precise models for logic and on-chip memory utilization given a (p, l) merge tree.

The merge tree is made up of mergers and couplers. Thus, we approximate the look-up table (LUT) utilization of a merge tree by adding up the LUT utilization of the mergers and couplers level by level. We use c_{2^n} and m_{2^n} to represent the number of LUTs used by a 2^n-rate coupler and 2^n-rate merger, respectively. Since there are 2^n mergers and $2^n + 1$ couplers at depth n of the tree, we derive the LUT utilization of a (p, l) tree as

$$\text{LUT}\,(p, \ell) = \sum_{n=0}^{\log \ell} 2^n (m_{\lceil p/2^n \rceil} + 2c_{\lceil p/2^n \rceil}), \tag{3.2}$$

On the other hand, we need to access the DRAM in batch mode to make sure the DRAM operating at its peak bandwidth. As each of the l input leaves to the merge tree corresponds to separate segments on DRAM, each leaf requires a separate input FIFO for storing the batched reads. In order for a (p, l) merge tree to be synthesizable on chip, we must ensure all l input buffers can fit in on-chip memory. Assuming the DRAM reads are performed in batched size b, we have

$$b \cdot \ell \leq C_{\text{BRAM}}. \tag{3.3}$$

3.2.4 Optimal Configurations

We now put the performance and resource models together to select the merge tree configuration that achieves the optimal performance for a specific hardware and problem. Formally, we need to find

$$\arg\min_{p,\ell} \left\{ \frac{N \lceil \log_\ell N \rceil}{\min\{\beta_{\text{DRAM}},\ pfr\}} \right\},$$

subject to

$$\begin{cases} \text{LUT}(p, \ell) & \leq C_{\text{LUT}} \\ b\ell & \leq C_{\text{BRAM}}. \end{cases}$$

In general, our model suggests that increasing p is more beneficial than increasing l up until the merge tree throughput reaches the DRAM bandwidth. For instance, when targeting an AWS EC2 F1 instance that contains a modern FPGA and four DDR4 DRAM channels with a total capacity of 64 GB, the optimized merge tree

configuration derived from our model is a (32, 128) tree. The throughput of the $p = 32$ merge tree for 32-bit records is exactly 32 GB/s. Thus this configuration matches the peak bandwidth of DRAM and then builds as many leaves (ℓ) as can be implemented on the FPGA. The reason why ℓ cannot be made larger than 256 in this case is the the input FIFOs use up the on-chip memory (Equation 3.3).

3.3 Implementation Details

In this section, we cover the necessary details of implementing a high-performance merge tree sorting system on the AWS EC2 F1 instance [Ama]. The whole system is shown in Figure 3.5. The host configures the merge tree kernel on the FPGA device and loads the data to be sorted on the FPGA DRAMs. The four FPGA DRAM channels can be accessed simultaneously through a fully connected AXI crossbar. To make full use of the external DRAM bandwidth, the communication between the sorting kernel and the DDR controller is always through 512-bit wide AXI-4 interfaces, regardless of the record width.

3.3.1 Data Loading Units

The merge tree reads and writes data in a streaming fashion. All reads from a specific leaf of the merge tree are from continuous memory addresses and performed in batches. In the early passes when the size of the sequence sent into a specific leaf is less than a full batch, we still perform a full-batch read for that leaf and divide the batch into multiple sequences before feeding them into the tree leaf. Thus, the merge tree architecture has amenable off-chip memory access patterns.

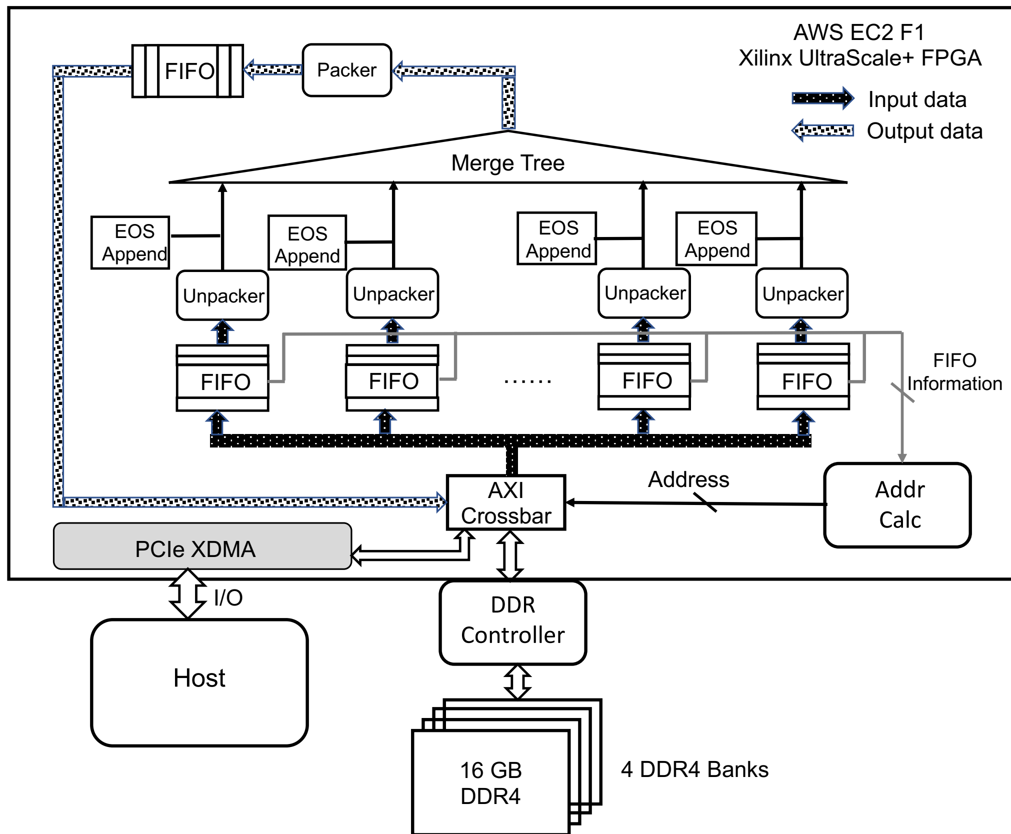

Figure 3.5: The merge tree system implemented on the AWS EC2 F1 instance. The host can configure the merge tree kernel and prepare the data to be sorted through the PCIe DMA channels.

The data loading unit performs the data reads and ensures off-chip memory is always operating at peak bandwidth. Reads from any specific leaf occur in 1-4 KB batches and are initiated by the data loading unit. Each leaf has an input buffer that is implemented as a FIFO, which is as wide as the bus of the DRAM controller (512 bits). Each FIFO can hold two full read batches and works as a ping pong buffer to hide the DRAM access latency.

One strict requirement for the merge tree to function correctly is that its leaf nodes always ready to provide the data to be merged, unless they reach the end of the sequences. In case one input buffer becomes empty, the merge tree will automatically stall until the data loading unit fetches more data into the buffer. To avoid the input buffers from being empty and enable the merge tree to work at full speed, the data loading unit checks in a round-robin fashion if any input buffer has enough free space to hold a new read batch. Whenever the data loading unit detects that an input buffer has sufficient free space, it performs a batched load into the buffer. In order to achieve this feature, the data loading unit maintains pointers to which addresses were last loaded into each input buffer.

Due to batched and sequential reads/writes as well as the dedicated scheduling of the off-chip memory accesses, the data loading unit allows the off-chip memory to operate at peak bandwidth and completely hides the off-chip memory access latency, which will be verified by our experiments in the next section.

3.3.2 Record Dispatching and Packing

The input data path from the DDR controller is always set to be 512-bit wide, while the record width usually varies depending on the specific problem. To support arbitrary record width, we place a module called unpacker at the output side of each input leaf FIFO. The core of the unpacker is a finite-state machine that extract one record from the 512-bit wide FIFO every cycle automatically once the record width is specified by the user. On the other side, the packer will concatenate the output from the merge tree root into 512-bit wide data before writing it into the output FIFO.

For each leaf node, we also keep track of the number of records that have been fed into the merge tree and append the end of sequence (EOS in Figure 3.5) signal with the ending record. The EOS signals are used to indicate that the merging operations of the sequences have been done for the current pass so that each streaming merge unit can be reset and prepared to receive the new sequences. In the initial pass where each sequence fed into the merge tree leaf contains only one record, we append the EOS signal with every record. In the second pass where each sequence fed into the merge tree leaf has l records, we always append the EOS signal with the lth record, so on so forth.

3.3.3 Presorter

Additionally, we may place a presorter between the AXI crossbar and the input FIFOs to perform an initial ordering of the records in the 512-bit wide AXI data. With such a presorter, the initial unsorted sequence size no longer has to start from one. For instance, when sorting 32-bit records and using a 16-element wide bitonic sorter as the presorter, we have the 512-bit wide data stored in each input FIFO to be a sorted sequence with 16 records. As a result, we will append the EOS signal with every 16th records in the first pass, which reduces the times for the merge tree to flush its intermediate states by 16×.

The presorter can be more complicated and take significant amount of resources such as the one in [PBL20]. Compared to the existing presorters, the proposed presorter has three major advantages. First, it is extremely resource-efficient, as it is only composed of a few compare-and-swap elements based pipeline stages and consumes no on-chip memory. Compared to the utilization of the entire merge tree

kernel, its resource utilization is already known and does not affect the resource models. Second, the presorter is well-embedded in the original datapath and does not require any extra control logic during the execution of different merge passes. This is because the size of the merged output sequences after the first pass is larger than the width of the presorter. These sorted output sequences will be fed as the inputs in the next passes and can directly go through the presorter without changing the order. Finally, the presorter can work as the extra pipelines between the user logic and the DDR DRAM controller and thus helps improve the design timing closure.

3.4 Experimental Evaluation

In this section, we perform experiments to validate the model's resource and performance predictions, demonstrate the performance of Bonsai and show how different parameters impact the sorting performance.

3.4.1 Experimental Setup

We perform all experiments on an Amazon AWS F1.2x large instance that uses a single Virtex UltraScale+ 16nm VU9P FPGA with a 64 GB DDR DRAM that has 4 channels, each with 8 GB/s concurrent read and write bandwidth and a capacity of 16 GB. We use Xilinx Vivado version 2018.3 for synthesis and implementation of our designs, which we developed in System Verilog. We also tune our designs so that they run at 250 MHz, which is required for the FPGA to fully utilize the off-chip DRAM bandwidth.

We mainly benchmark our sorter using 32-bit integers with a uniform distribution.

We also use `gensort` to generate 100-byte records (10-byte key, 90-byte value) from Jim Gray's sort benchmark [Gra]. We hash the 90-byte value to a 6-byte index, which allows us to concatenate the 10-byte key and 6-byte value into a 16-byte key-value pair that saves the memory. The sizes of the datasets are from 64 MB to 32 GB. We choose the maximum size to be 32 GB is because the total available DRAM capacity is 64 GB and we need half space for storing the unsorted data and the other half space for storing the merged results.

3.4.2 Model Validation

Figure 3.7 and Figure 3.6 list the achieved performance (represented as bars) and the predicted performance by our performance model (represented as $\cdot$) when we sort 64 MB - 16 GB data. All sorting time results are within 10% of those predicted by our performance model.

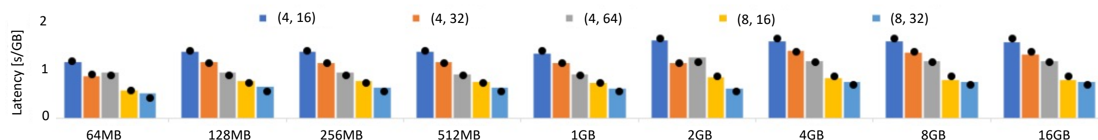

Figure 3.6: Sorting time per GB of various merge tree configurations.

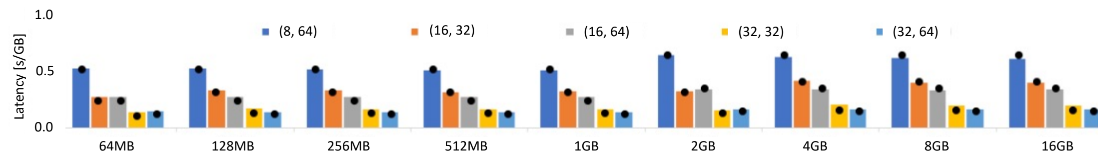

Figure 3.7: Sorting time per GB of various merge tree configurations.

We can also validate the impact of the tree throughput p and the tree leaf number l from the derived results. When the merge trees have the same throughput p, the

merge tree with the greater number of leaves ℓ gives better or equal performance, depending on the size of the dataset. On the other hand, when two merge trees have the same number of leaves ℓ, the merge tree with the higher throughput p always has better performance, as long as the DRAM bandwidth is not saturated. Once DRAM bandwidth is saturated, increasing throughput p does not decrease sorting time; however, increasing the number of leaves ℓ reduces the total number of merge passes, thus reducing the sorting time. Thus, optimal DRAM-based merge tree sorter configurations always have throughput p exactly high enough to saturate DRAM bandwidth, with as many leaves ℓ as on-chip resources permit.

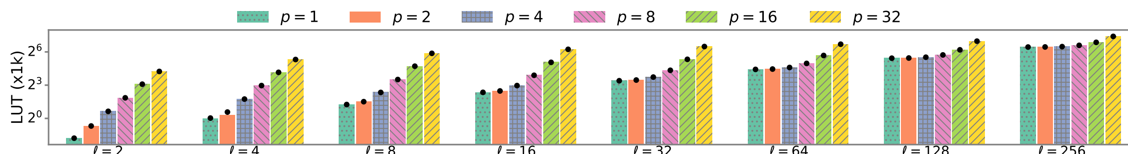

Figure 3.8: LUT utilization of various merge tree configurations.

We also show the LUT utilization reported by the synthesis tool (represented as bars) and the predicted LUT utilization (represented as $\cdot$) in Figure 3.8. The differences of various merge tree configurations are all within 5%, which proves that the proposed merge tree has a structured architecture and can be accurately modeled.

3.4.3 DRAM Sorting Results

The optimized DRAM sorter we implement on the Amazon AWS F1 instance is a (p=32, l=64) merge tree, with its resource utilization breakdown given in Table 3.2. The merge tree's throughput is set to 32 so that it saturates the off-chip DRAM bandwidth. The tree leaf number is limited to 64 because further increasing the tree

leaf number will results in routing failure on the existing FPGAs.

Table 3.2: Resource utilization breakdown of the best-performing DRAM sorter on AWS F1.

Component	LUT	Flip Flop	BRAM
Data loading unit	110,102	604,550	960
Merge tree	102,158	100,264	0
Presorter	75,412	64,092	0
Total	287,672	768,906	960
Available	862,128	1,761,817	1,600
Utilization	33.3%	43.6%	60%

When sorting 32 GB data, the proposed sorter has a latency of 5.5 seconds. That is, Bonsai achieves an overall sorting performance of 5.8 GB/s (32 GB divided by 5.5 seconds). Table 3.3 summarizes the performance comparison among the proposed merge tree sorter and the best DRAM-based CPU and FPGA sorters. Compared to a 32-thread Intel Xeon CPU based merge sort implementation that is based on the standard C++ library and widely accessible to general programmers, the proposed DRAM-based merge tree sorter has a speedup of 29×. Compared to the state-of-the-art DRAM-based CPU and FPGA sorters [CBB15, CMF20], it achieves 2.3× and 3.7× speedup.

3.4.4 Bandwidth Efficiency

As mentioned in the above subsection, the optimal DRAM-based merge tree configuration always has its throughput p saturate the off-chip DRAM bandwidth. In other

Table 3.3: Comparison of the sorting time when sorting 32 GB data using different DRAM-based solutions.

Solution	Sorting time
32-thread CPU merge sorter	159.6 seconds
Best CPU sorter	12.3 seconds
Best FPGA sorter	20.6 seconds
Bonsai	5.5 seconds

words, sorting with DRAMs is primarily bounded by the off-chip DRAM bandwidth. In fact, the off-chip memory accesses also account for most of the energy consumed by modern computer systems [STD94] and the efficiency of utilizing the off-chip memory is directly related to the energy consumption. In this subsection, we show the bandwidth-efficiency of the proposed sorters. Formally, we define the bandwidth-efficiency as the ratio of the overall performance of the sorter to the available bandwidth of off-chip memory. For example, our optimized DRAM sorter operates at an overall performance of 5.8 GB/s when working with 4 DRAM channels; since the DRAM bandwidth is 32 GB/s, the bandwidth-efficiency of our DRAM sorter is $5.8/32 = 0.18$. Compared to the existing DRAM-based sorters, our implementation has the highest bandwidth efficiency: specifically 2.7× better bandwidth-efficiency than any other sorter, when working with 8 GB/s DRAM. When working with 32 GB/s DRAM bandwidth using 4 DRAM banks, Bonsai gives 1.8× improvement in bandwidth-efficiency compared to all other sorters.

3.4.5 Scalability in Record Width

The proposed merge tree sorting solution works with any key-value record format of up to 512-bit width. Table 3.4 shows that the merge units for 32- and 128-bit records exhibit comparable resource utilization for equal-throughput elements, with 128-bit records offering better throughput per LUT. For example, a 128-bit record 4-rate merge unit has the same throughput as a 32-bit record 16-merger, but almost 30% less logic utilization.

Table 3.4: LUT utilization and throughput of building-block elements.

	32-bit record		128-bit record	
Element	Throughput	LUT	Throughput	LUT
1-rate merge unit	1 GB/s	60	4 GB/s	205
2-rate merge unit	2 GB/s	419	8 GB/s	1,622
4-rate merge unit	4 GB/s	995	16 GB/s	4,062
8-rate merge unit	8 GB/s	2,409	32 GB/s	9,572
16-rate merge unit	16 GB/s	5,556	64 GB/s	20,854
32-rate merge unit	32 GB/s	13,622	128 GB/s	56,559

This is because given the same throughput, a larger record width means less number of records to be processed in parallel within each merge unit. Specifically, the 128-bit record 4-rate merge unit has the same throughput as the 32-bit 16-rate merge unit, but the 128-bit 4-rate merge unit needs a much smaller number of compare-and-swap pipeline stages to output 4 records per cycle versus the 16 records per cycle that the 32-bit 16-rate merge unit must output. More formally, the logic complexity of the compare-and-swap unit grows linearly with record width,

while the number of compare-and-swap units within a merge unit grows superlinearly ($\Theta(k \log k)$) with the number of records. Thus, the same amount of wider records requires less resources to be sorted in the same amount of time compared to narrower records.

3.5 Summary

In this chapter, we present Bonsai, a complete merge tree sorting solution for DRAM-based FPGAs. The throughput and the tree leaf numbers of the merge tree can be independently configured to adapt to the available off-chip memory bandwidth and the on-chip resources. The optimal configuration of the merge tree can be derived through a set of comprehensive performance and resource models, due to the inherently structured architecture of the merge tree. When implemented on the Amazon AWS F1 cloud FPGA, the proposed merge tree solution achieves 2.3× and 3.7× speedup over the state-of-the-art DRAM-based CPU and FPGA sorters. The experiments also indicate that the merge tree sorting is primarily bounded by the available off-chip memory bandwidth when implemented on the DRAM-based FPGAs.

CHAPTER 4

Sorting Acceleration on HBM-Based FPGAs

Chapter 3 shows that the performance of sorting acceleration on conventional DRAM-based FPGAs is primarily bounded by the available off-chip memory bandwidth. However, the emergence of HBM-based FPGAs has brought the potential to further boost the performance of sorting acceleration by offering a much higher off-chip memory bandwidth. For example, the recent HBM-based datacenter FPGAs report a peak memory bandwidth of about 420 GB/s, which is roughly 6× of that on a similar DRAM-based datacenter FPGA such as the Amazon AWS F1 instance [LFL21, CCQ21]. A natural question is: if we simply port and scale the DRAM-based merge tree designs onto an HBM-based FPGA, can we get 6× higher merge tree throughput? Unfortunately, the answer is no due to two challenges.

On the one hand, with the tremendous off-chip bandwidth increase, the bottleneck of sorting acceleration would shift from the off-chip memory to the available on-chip resources. As will be analyzed in Section 4.1 in this chapter, for a merge tree-based sorter that can output p elements per cycle, the needed off-chip bandwidth increases linearly with p, while the required on-chip resources increase much faster at the rate of $\theta(plog^3(p))$. Unfortunately, for an HBM-based FPGA (e.g., Xilinx Alveo U280 FPGA [Xil21a]) that provides roughly 6× more bandwidth than a similar DRAM-based FPGA (e.g., AWS F1 FPGA), it only offers merely 1.2× more on-chip

resources. Indeed, we will show that a single merge tree accelerator can not be scaled to use more than 1/4 of the available HBM bandwidth.

On the other hand, it is nontrivial to fully utilize the HBM bandwidth in an accelerator design. Typically, the HBM is composed of 32 small channels. The accelerator has to rely on multiple memory controllers to access those HBM channels in parallel to maximize the memory bandwidth, at the expense of on-chip resources. Moreover, there could easily be contention between multiple channel accesses due to HBM's internal channel switching, which would degrade the effective bandwidth. Therefore, it requires delicate memory access control to improve the bandwidth utilization and reduce resource overhead. Finally, the HBM stacks are physically connected to a datacenter FPGA's bottom die only, making it difficult to spread the resource utilization across multiple FPGA dies to achieve desirable timing closure. As shown in Table 4.1, none of the published HBM-based accelerator designs [HDU21, SCG21, WHT21, DHZ22, CGC22] are able to fully utilize the entire bandwidth of the 32 HBM channels even with the help of the dedicated floorplanning tool [GCW21].

Table 4.1: The design frequency, the number of HBM Channels used and the effective HBM bandwidth utilized in prior HBM-based FPGA accelerators.

Work	Frequency	Channel Used	Bandwidth Used
iccad21	165 MHz	19	247 GB/s
sextans	189 MHz	28	289 GB/s
sssp	130 MHz	28	200 GB/s
spmv	237 MHz	18	258 GB/s

In this chapter, we demonstrate an alternative way called TopSort that better scales the merge tree up through a two-phase design. The key of this method is to split the complete sorting process into two separate merge phases with smaller sorters and reuse the resources between the two phases. In the first phase, we employ 16 parallel small merge tree kernels, each of which acts like a DRAM-based sorter and sorts a portion of the input sequence with two HBM channels (one for reading unsorted inputs and the other for writing sorted outputs). In this phase, the effective merge tree throughput is equal to the bandwidth of 16 HBM channels. In the second phase, we merge the sorted results from all HBM channels into one final sorted sequence. To reduce the resource consumption, this phase reuses 4 merge tree kernels from phase one to form a wider merge tree with a 4× higher throughput, since merge trees with different throughput share a similar operation pattern.

To improve the effective HBM bandwidth utilization, we profile the corresponding HBM channel characteristics and carefully optimize the merge tree's memory access pattern for each phase, including optimizing its data layout and burst access, as well as reducing the usage of HBM controllers. Besides, the novel merge tree reuse architecture allows us to easily improve the design frequency through coarse-grained floorplanning of each separate merge tree kernel.

When implemented on the Xilinx Alveo U280 FPGA, TopSort runs at 214 MHz and achieves an overall performance of 15.6 GB/s, which is 2.2× faster than the DRAM-based FPGA sorter presented in Section 3.

4.1 Scalability Analysis of A Single Merge Tree

In this section, we explain why the single merge tree employed on existing DRAM-based FPGAs cannot be directly scaled to efficiently work on HBM-based FPGAs.

The time for a single merge tree to sort N r-byte records has been given in Equation 3.1. Assume there are unlimited on-chip resources to build a merge tree with a tree throughput p saturating the off-chip memory bandwidth, we have the overall sorting performance (defined as the number of bytes of unsorted elements divided by the time it takes to get the completely sorted elements) in Equation 4.1.

$$\beta_{overall} = \frac{\beta_{memory}}{\lceil \log_{\ell} N \rceil} \tag{4.1}$$

Clearly, the on-chip resources are finite. Next, we analyze the on-chip resource requirement in scaling the merge tree-based sorter. The merge tree in Figure 3.3 can be viewed as two sub merge trees whose root rates are $p/2$ each, plus a merge unit whose rate is p. Since a merge unit is a variation of a parallel sorting network, such as bitonic or even-odd sorting network, and such a p-rate merge unit consumes $plog^2(p)$ number of comparison operators [MCK17, SEC18], the number of comparators, $L(p)$, required when scaling p can be summarized in Equation 4.2.

$$L(p) = 2L(p/2) + \theta(plog^2(p)) \tag{4.2}$$

Based on this equation, we can derive that the total number of logic elements (i.e., comparators) for the complete binary tree is $\theta(plog^3(p))$ [BHS80]. Considering that migrating from 4 DRAM channels to 32 HBM channels increases the available off-chip bandwidth by nearly 8×, it is not possible to directly scale a single merge tree's throughput to catch up with the increase of the HBM bandwidth.

In fact, we find that the root throughput of a single merge tree can only be scaled to match 4 HBM channels (for write operations) on the Xilinx Alveo U280 board. Since the read operations of the tree also occupies the same amount of HBM bandwidth, the bandwidth utilized in such a merge tree is at most equal to 8 HBM channels. In other words, such a merge tree only uses 25% of the HBM bandwidth in each pass.

4.2 Architecture of TopSort

In this section, we present the methodology of TopSort, a two-phase merge tree sorting solution optimized for HBM-based FPGAs. First, we demonstrate the idea of the two-phase sorting. Second, we show how we reuse the logic between two phases to alleviate the resource contention and improve the performance. Finally, we illustrate the architecture and micro-architectures of TopSort.

4.2.1 Two-Phase Merge Tree Sort

Since the required resources grow super-linearly when directly scaling a single merge tree, we choose to split the N unsorted records evenly into k parts and have k smaller merge trees work in parallel. Each merge tree sorts N/k records and is configured to have a throughput that saturates the bandwidth of a single HBM channel, which is similar to a DRAM-based merge tree sorter. This method has better scalability as its resource consumption grows linearly with the number of trees k. However, the k sorted sequences still need to be merged into the final sorted sequence. This can be done by streaming the k sequences into another merge tree for the final merge. We refer to the process during which k merge trees work in parallel as phase 1, and the

subsequent final merge as phase 2.

Since sorting GB-scale data only takes several seconds while reprogramming the FPGA itself has an overhead of several second, we cannot separate the two phases. In fact, the two phases have to contend for the on-chip resources and the configurations of the parallel merge trees of phase 1 and the final merge tree of phase 2 all play an important role in the overall sorting performance. For example, we may have 16 merge trees in the first phase to fully utilize all 32 HBM channels and achieve the best performance. But if the remaining resources only allow us to build a final merge tree whose throughput is equal to the bandwidth of a single channel, then the overall sorting performance will be limited by the final merge, no matter how much better the memory bandwidth utilization we achieve in phase 1.

4.2.2 Logic Reuse Between The Two Phases

To alleviate the resource contention between phases 1 & 2, we propose to reuse the merge units in phase 1 to build the final merge tree. This is based on the observation that a merge tree can be decomposed as sub merge trees plus some extra levels of merge units. For example, a merge tree whose throughput is p in Figure 3.3 consists of four merge trees of throughput $p/4$, plus two $p/2$-rate merge units and one p-rate merge unit, as shown in Figure 4.1. In other words, if we want to build a merge tree whose throughput is equal to 4 HBM channels' bandwidth in phase 2, we only need extra resources to build the 3 more merge units. The rest of the resources including the input buffers can be reused from four of the merge trees in phase 1.

Figure 4.2 illustrates the detailed dataflow of the merge tree reuse mechanism in the proposed two-phase approach, which starts with four $p/4$-rate merge trees. If

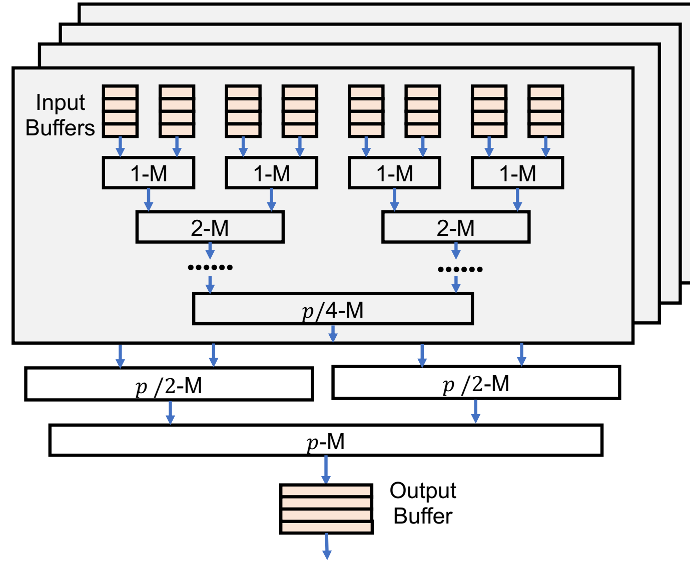

Figure 4.1: Merge tree decomposition: four $p/4$-rate merge trees can be combined to build a larger p-rate merge trees.

the current phase is phase 1, then the four output streams from the roots of the four trees will be directed by the four de-multiplexers into the output buffers on the left side before they are written into the corresponding HBM channels. If the current phase is phase 2, then the four output streams from the four $p/4$-rate merge trees will be switched to flow into the two $p/2$-rate merge units. Later on, the outputs of the two $p/2$-rate merge units are fed into the final p-rate merge unit, thus forming a p-rate merge tree. In this way, the p-rate merge tree in phase 2 is built by reusing the four $p/4$-rate merge trees in phase 1 plus two $p/2$-rate merge units and one p-rate merge unit. The memory write operations in phase 2 will be discussed in detail in Section 4.2.5.

With the merge tree reuse, conceptually, we are able to integrate 16 merge trees

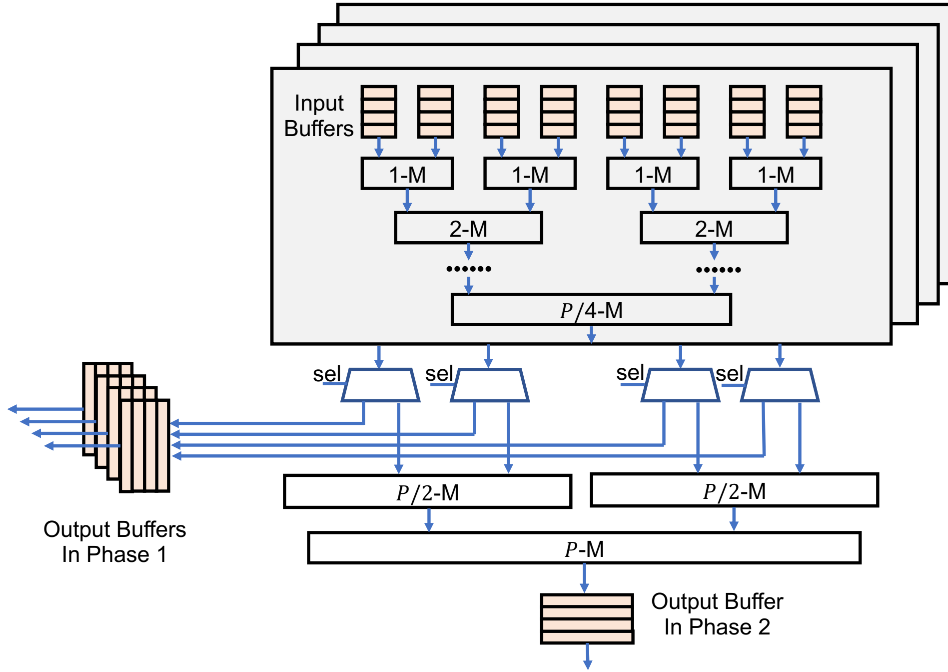

Figure 4.2: Detailed dataflow of the merge tree reuse in the proposed two-phase approach: the four demuxes (trapezoids in the figure) are controlled by the same sel signal, which indicates whether the current phase is phase one or phase two.

to saturate all 32 HBM channels' bandwidth in phase 1 and another merge tree whose throughput saturates 4 HBM channels' bandwidth in phase 2 onto one FPGA.

4.2.3 Architecture to Support Logic Reuse

The overall architecture of the two-phase merge tree sorter is shown in Figure 4.3. We use 16 merge tree kernels in phase 1. Given a total of N unsorted records, we split them into 16 sequences, each of which contains $N/16$ records and is stored in one HBM channel. Then each of the 16 trees will stream the unsorted sequence

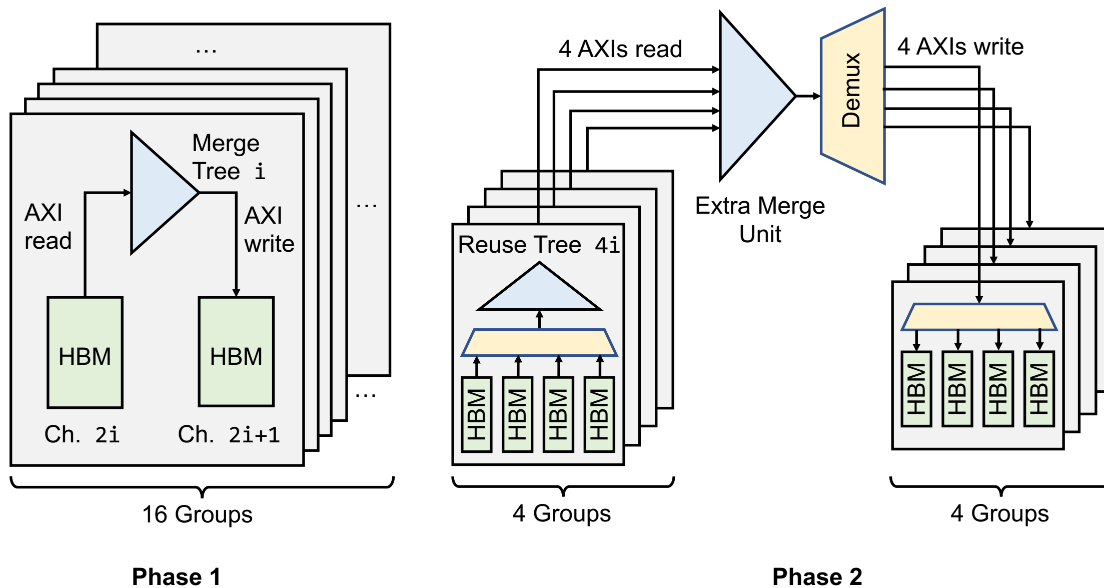

Figure 4.3: The overall architecture of the two-phase merge tree sort. The merge trees rely on AXI interfaces to access HBM channels. The role of demux is covered in Section 4.2.5.

from one HBM channel and stream the merged (sorted) sequence to another HBM channel. We design each tree so that it has 16 leaves and outputs 64 bytes per cycle at its root to saturate the bandwidth of a single HBM channel. The leaf number for each merge tree is chosen to be 16 is because the leaf buffers cannot exceed the available on-chip BRAM limit. Assuming each record is 64-bit, the merge tree will output 8 records every cycle.

In phase 2, we reuse 4 merge trees from phase 1 and add two levels of merge units to form a wider merge tree. This merge tree has 64 leaves and outputs 256-byte records per cycle. Since there are 16 sequences sitting in 16 HBM channels after phase 1, we split each sequence into 4 segmented sub sequences. Then each of the

64 sub sequences is fed into one of the 64 leaves for one pass to get the final sorted results. During phase 2, the input buffers and each merge unit of the four reused merge trees stay exactly the same as phase 1. We only need to change the memory addresses and the length of the sequences that the AXI interfaces read from each channel for each of the 64 leaves. The reason for choosing to use 16 merge trees with the specific configuration in phase 1 and reuse 4 of them in phase 2 will be covered in Chapter 4.3.

4.2.4 Tuning Sorted Sequence Size from Phase 1

One requirement for a merge tree to output p records per cycle is that, on average, it has p leaves out of its total l leaves providing records every cycle. Since there are 64 sub sequences need to be fed into the 64 leaves of the tree in phase 2, each of the 16 sequences in 16 HBM channels after phase 1 needs to be divided into 4 segmented sub sequences. If each sequence is totally sorted after phase 1, then the 4 sub sequences have the relation that records in sub sequence 0 are always smaller than records in sub sequence 1, and so on. When such 4 sub sequences are fed into 4 leaves, only 1 of the 4 leaves will feed the records to the merge tree at any cycle, e.g., initially, only the leaves feeding sub sequence 0 from each channel are active, then are the leaves feeding sub sequences 1, and so on so forth. For each of the reused merge trees that has 16 leaves, this means only 4 leaves are providing records into the tree per cycle. Figure 4.4 illustrates such behavior: leaves marked as yellow are actively feeding data. As a result, each merge tree is idle half of the time. Although each tree can output 8 records per cycle, the average throughput of the tree is 4 records per cycle, which is below our expectation.

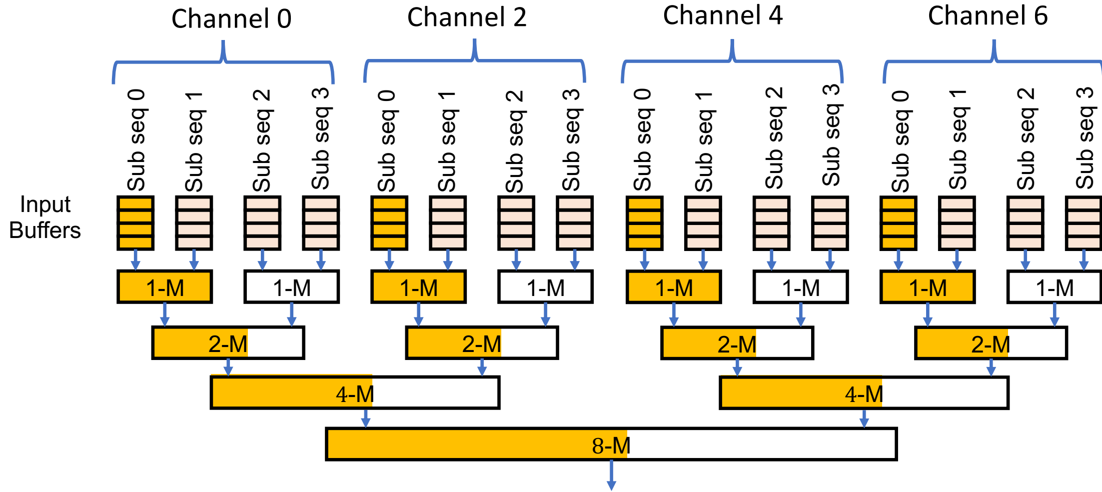

Figure 4.4: The active rate of one reused merge tree in phase 2, if the sequence in each channel is completely sorted. Initially, only the leaves feeding sub sequence 0 from each channel are active, then are the leaves feeding sub sequences 1, and so on. Leaves marked as yellow are actively feeding. Merge units partially marked mean they are idle half of the time.

To solve this issue, we need to tune the sorted sequence size from phase 1 so that at least 8 leaves of the merge tree are always actively feeding data in phase 2. This is done by directing each of the merge tree in phase 1 to sort either $N/32$ or $N/64$ elements at a time. For instance, instead of completely sorting the $N/16$-element sequence in each channel, we want 4 $N/64$-element sorted sub sequences from phase 1. This is done by controlling the number of records that go into the leaves in the last pass of phase 1. Since the sorted sequence size always grows by the number of leaves l, we deliver $N/1024$ records into each leaf. At the tree root side, we get $N/64$-element sorted sequence. We repeat this process 4 times for each merge tree in phase 1 to get 4 $N/64$-element sorted sub sequences.

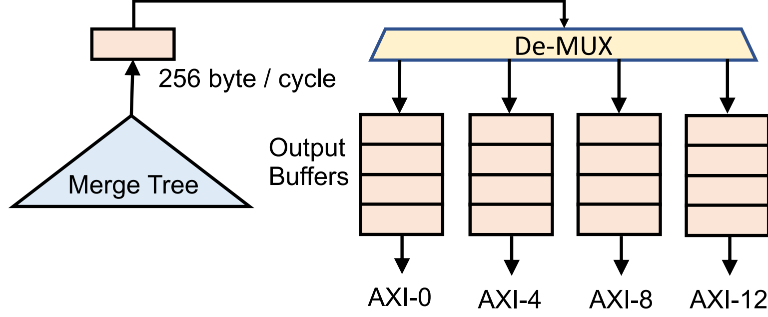

Figure 4.5: The memory write behavior of phase 2. Writes from 4 AXI interfaces are used to match the tree's throughput.

4.2.5 Memory Write Pattern in Phase 2

Since the 4× wider merge tree of phase 2 outputs 256-byte sorted records per cycle, there are 4 AXI interfaces writing to the HBM channels in parallel. The on-chip memory is not big enough to hold all the continuously sorted records before they are written back to entirely fill one HBM channel. Therefore, these continuously sorted records have to be split into batches and then written back to separate HBM channels through 4 AXI interfaces. Figure 4.5 depicts the details of such behavior and the choice of the specific AXI interfaces will be explained in Section 4.4. If we choose the batch size to be B bytes, e.g., we write the first B bytes sorted records into the first output buffer, then write the second B bytes sorted records into the second output buffer, and so on. The records in the four buffers will then be written to HBM channels via the four AXI interfaces. As a result, although the records are completely sorted, one needs to pick the B-byte batches channel by channel to form the final sorted results.

4.3 Analysis of TopSort

In this section, we perform a comprehensive analysis on the performance of TopSort and demonstrate why it delivers better overall performance than simply scaling a single merge tree.

4.3.1 Performance Model

We first list the associated parameters of TopSort in Table 4.2. If there are k parallel merge trees in phase 1 and each merge tree has a throughput of p_1 and leaf number of l_1, the average performance of phase 1 is given in Equation 4.3. Note that Chapter 3.2 shows the merge tree throughput only needs to saturate the off-chip memory bandwidth and further scaling the merge tree throughput does not improve the effective throughput, we use p_1fr to represent the actual throughput of each merge tree in phase 1 and make sure p_1 is selected so that p_1fr is not larger than the available off-chip memory bandwidth to each merge tree.

$$\beta_1 = k \cdot \frac{p_1 fr}{\lceil \log_{l_1}(N/k) \rceil} \tag{4.3}$$

In the second phase, the final merge tree merges the k sorted sequences into the final result. Since the merge tree in this phase is built on top of the k merge trees in phase 1, its leaf number is guaranteed to be large than k. As a result, it will require only one pass for this merge tree to merge the sorted sequences together. Besides, according to the scalability analysis in the previous section, the single merge tree in this phase only exploits a portion of the HBM bandwidth due to the on-chip resource constraints. Therefore, the performance of phase 2 is directly reflected by the tree

Table 4.2: Parameters associated with TopSort

	Symbol	Definition
Input Param.	N	Number of records in array
	r	Record width in bytes
Hardware Param.	β_{chan}	Bandwidth of a single HBM channel
	C_{BRAM}	On-chip memory capacity in bytes
	C_{LUT}	Number of on-chip logic units
Merge Tree Param.	f	Design frequency
	p_1	Throughput of the merge tree in phase 1
	l_1.	Leaf number of the merge tree in phase 1
	k	Number of the merge trees in phase 1
	β_1	Aggregate performance of phase 1
	p_2	Throughput of the merge tree in phase 2
	l_2.	Leaf number of the merge tree in phase 2
	β_2	Performance of phase 2
	β	Overall performance of the two-phase sort
	b	Size of read batches in bytes

throughput in phase 2, as shown in Equation 4.4:

$$\beta_2 = p_2 f r. \tag{4.4}$$

The overall performance of TopSort is thus calculated by dividing the problem size by the overall time spent in both phases in Equation 4.5.

$$\beta = \frac{Nr}{\frac{Nr}{\beta_1} + \frac{Nr}{\beta_2}} = \frac{fr}{\frac{\lceil \log_{l_1}(N/k) \rceil}{k \cdot p_1} + \frac{1}{p_2}} \tag{4.5}$$

4.3.2 Resource Model

We derive the resource utilization of TopSort in a similar way to Equation 3.2. Equation 4.6 lists the on-chip LUT utilization of the whole design:

$$\mathrm{LUT} = k \cdot \sum_{n=0}^{\log \ell_1} 2^n (m_{\lceil p_1/2^n \rceil} + 2c_{\lceil p_1/2^n \rceil}) + \sum_{n=0}^{\log \frac{p_2}{p_1}} 2^n (m_{\lceil p_2/2^n \rceil} + 2c_{\lceil p_2/2^n \rceil}). \tag{4.6}$$

The first item on the right hand side is the LUTs utilization of k merge trees, each of which has a configuration of (p_1, l_1). Since the final merge tree in phase 2 is built on top of the merge trees in phase 1, we only need to add $\log \frac{p_2}{p_1} + 1$ extra levels of mergers and couplers, which correspond to the second item on the right hand side.

On the other hand, since the merge tree in phase 2 can fully reuse the input FIFOs of phase 1's trees for its own leaves, the on-chip memory usage is constrained as follows:

$$k \cdot b \cdot \ell_1 \leq C_{\mathrm{BRAM}}. \tag{4.7}$$

4.3.3 Optimized configuration on Xilinx U280 Board

There are various combinations of k, (p_1, l_1), (p_2, l_2), each of which has a different performance and requires different amount of on-chip resources. To get the configuration that delivers the best performance, we could apply the constraints of the

available resources to the performance and resource utilization models derived above. Formally, we need to find

$$\arg\max_{p_1,\ell_1,p_2,\ell_2,k} \left\{ \frac{fr}{\frac{\lceil \log_{l_1}(N/k) \rceil}{k \cdot p_1} + \frac{1}{p_2}} \right\},$$

subject to

$$\begin{cases} \text{LUT} & \leq C_{\text{LUT}} \\ k \cdot b \cdot \ell_1 & \leq C_{\text{BRAM}}. \end{cases}$$

Figure 4.6 lists the performance and resource utilization of various configurations when sorting (32-bit key, 32-bit value) pairs on the Xilinx U280 board. Note that there are roughly 1 million LUTs available on the Xilinx U280 board and it is recommended that the maximum LUT usage is no more than 70%, otherwise the design may not be routable. The configuration we adopt is marked by the red circle in Figure 4.6, which sets k to be 16, (p_1, l_1) to be (8, 16), and (p_2, l_2) to be (32, 64). There is one more point next to our current configuration point, which consumes nearly the same amount of LUTs while achieving higher performance. That configuration sets k to be 2, (p_1, l_1) to be (32, 128), and (p_2, l_2) to be (64, 256). However, the design of this configuration is hard to be evenly distributed across the FPGA dies to achieve routable solutions, which is in contrast to our current configuration that allows for coarse-grained floorplanning (see Section 4.4.4).

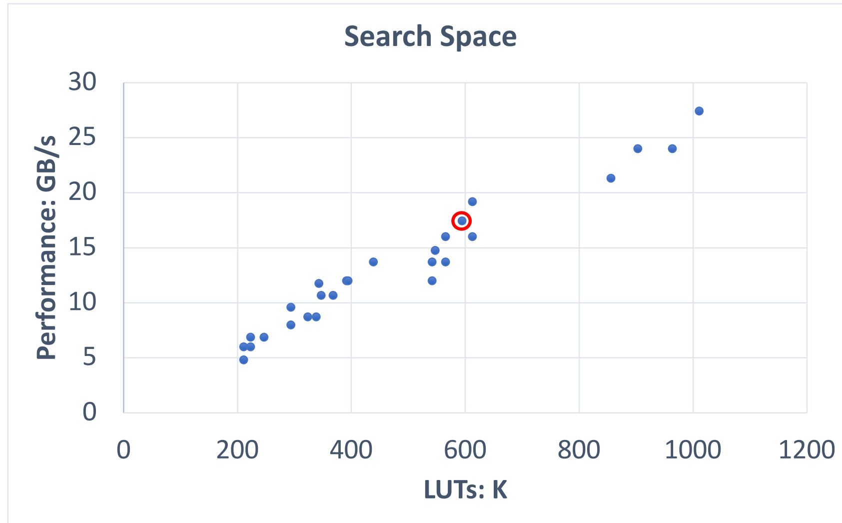

Figure 4.6: Search space of different configurations when sorting (32-bit key, 32-bit value) pairs on the Xilinx U280 board. Each dot represents a possible configuration of k, $(p_1,\ l_1)$, $(p_2,\ l_2)$. The X-axis indicates the LUT utilization and the Y-axis indicates the theoretical performance of each configuration. The dot marked by a red circle is the actual configuration we choose.

4.4 HBM-Specific Profiling and Optimization

The high memory bandwidth of HBM cannot be fully utilized without an in-depth understanding of HBM's characteristics. In this section, we profile the HBM's characteristics and introduce the design choices that are coupled with specific HBM characteristics. These HBM-related optimizations are necessary to achieve a full bandwidth usage.

4.4.1 Resource Overhead From AXI Interface

According to our measurement, a memory controller (shown as an AXI rate converter in Figure 2.5) for an HBM channel requires about 5K LUTs and about 6K Flip-Flops (FFs) on the Xilinx U280 FPGA. As a result, the naive choice of instantiating one AXI interface for each of the 32 HBM channels will result in about 320K LUTs, which is more than 40% of the available LUTs on the bottom die where the HBM resides. Such high resource overhead will squeeze the space for user logic and cause severe routing issues.

To address this issue, one must utilize the internal crossbar among HBM channels as much as possible, e.g., for each merge tree of phase 1, we only need one AXI module to read from and write to two adjacent HBM channels. This way halves the AXI area overhead and effectively reduces the routing congestion in the bottom die of the FPGA.

4.4.2 Data Layout Consideration

Although each AXI interface from the user side can access any of the 32 HBM channels, non-local data accesses could cause contention over the lateral connections between channel groups, thus leading to bandwidth decrease. To access an HBM channel outside the group, the data must occupy and traverse through each of the lateral connections until it reaches the destination. Two memory accesses requiring the same lateral connection will cause a conflict and one will be blocked, thus the performance of each memory accesses will be halved.

Therefore, we must carefully arrange for the data placement to minimize out-of-group channel accesses and avoid conflicts in lateral connections. Table 4.3 sum-

Table 4.3: Correspondence between each user side AXI and the HBM channels it accesses, i=0-3.

AXI NO.	HBM Channel NO.
AXI-$4i$	$8i$ - ($8i$+7)
AXI-$(4i+1)$	($8i$+2) - ($8i$+3)
AXI-$(4i+2)$	($8i$+4) - ($8i$+5)
AXI-$(4i+3)$	($8i$+6) - ($8i$+7)

marizes the target HBM channels for each AXI interface in our two-phase merge tree design. In the first phase, merge tree i uses AXI-i to only read/write a pair of nearby HBM channels $2i$ and $2i+1$, where i ranges from 0 to 15. In the second phase, the merge tree $4i$ uses AXI-$4i$ to read/write between channels $8i+2j$ and channels $8i+2j+1$, where i & j both range from 0 to 3.

Figure 4.7 illustrates the access pattern of the first four AXI interfaces in different colors, the same behavior of which is repeated for the rest of the 12 AXI interfaces. AXI-1, 2 and 3 will only access the two channels within its crossbar group. Although the AXI-$4i$ in the second phase needs to access channels outside of its local group, our data layout guarantees that different AXI interfaces will not cause conflicts in lateral connections.

4.4.3 Choice of Burst Size

Each of the AXI interfaces will always access the HBM in burst mode and we need to properly choose the burst size, as a large burst requires excessive on-chip buffers while a small burst may not fully utilize the bandwidth.

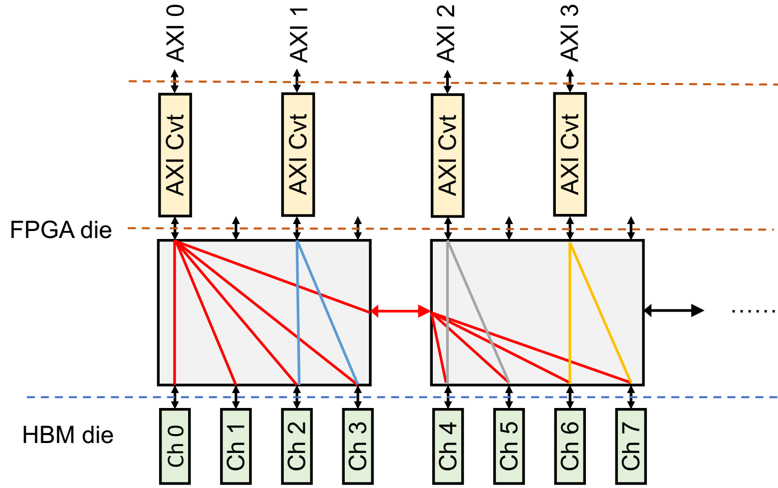

Figure 4.7: Usage of the first 4 AXI interface. The same behavior is repeated for rest of the 12 AXI interfaces.

In order to select proper burst sizes, we first profile the HBM performance based on the memory access pattern of each AXI interface. As shown in Fig. 4.7, In phase one, each AXI reads from one channel and writes to the adjacent channel; In phase two, AXI-0 reads from four even/odd channels in a round-robin way and writes to the other four nearby odd/even channels.

We define reading from m HBM channels and writing to their nearby m channels as pattern $m{\times}m$. The patterns of phase 1 and phase 2 are thus 1×1 and 4×4 by this definition. For completeness, we also perform tests for patterns 2×2 and 8×8. Then we measured the achieved bandwidth when using different AXI burst sizes as seen in Figure 4.8.

Based on our experiments, any AXI burst size from 512 B to 4 KB can achieve the peak HBM bandwidth for the 1×1 pattern, which is used by merge trees in phase

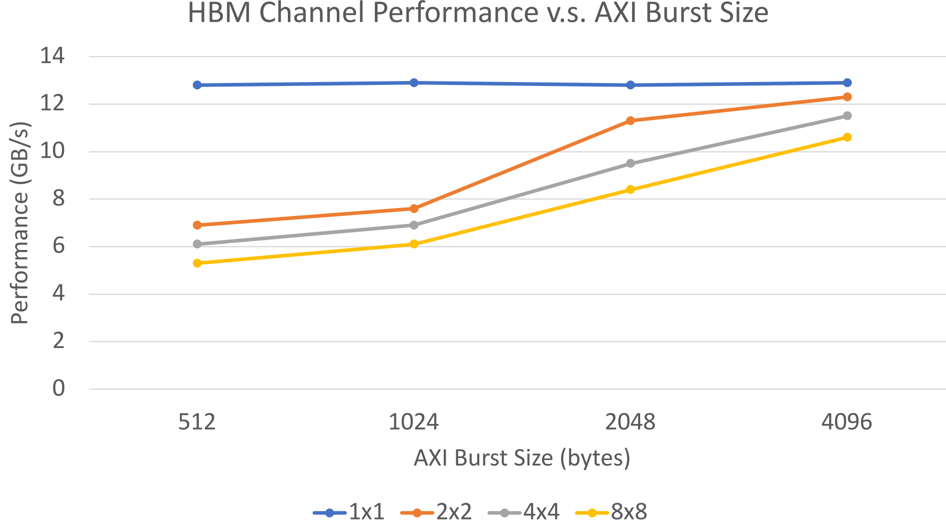

Figure 4.8: HBM channel performance of inter-crossbar & intra-crossbar behaviors when varying the AXI burst size.

1. However, for the 4×4 inter-crossbar memory access pattern used in phase 2, the AXI burst size needs to be maximized to 4 KB to achieve the highest bandwidth. Therefore, we set the burst size to be 1 KB for the 12 merge trees that are not reused in phase 1, and 4 KB for the 4 merge trees that are reused in phase 2.

Minimizing the burst sizes under the bandwidth constraints could significantly save resources because not all buffers are implemented in BRAM. For each merge tree, each leaf node requires a 512-bit wide input buffer to hold the read bursts from AXI interfaces. When implemented using BRAM, each buffer consumes at least 7.5 BRAMs on Xilinx FPGAs. Since we have 16 merge trees and each tree has 16 leaves, the FPGA does not have enough BRAMs to implement all the buffers. So, about 3/4 of the buffers are implemented using LUT-based shift registers. While BRAM-based buffers are insensitive to the burst size, the area of a LUT-based buffer is

directly proportional to the buffer depth. A single LUT can implement a 1-bit shift register with a depth of 32 and 512 LUTs can hold two AXI bursts of 1 KB. Reducing the bursts size from 4 KB to 1 KB directly reduces the LUTs consumption by 4×. Further reducing the burst size from 1 KB to 512 B does not save more LUTs because 2 bursts of 512 B still require 512 LUTs.

4.4.4 Floorplanning Optimization

Floorplanning is an important technique for an FPGA design to evenly spread across the FPGA dies and achieve desirable timing closure, but most of the existing HBM-based FPGA accelerators still lack of appropriate floorplanning due to their clumsy architecture. However, thanks to the parallel sorting in the first phase and merge tree reuse in the second phase, we are able to separate each of the merge tree kernels independently and apply a simple yet efficient resource model to better distribute these kernels across the three FPGA dies, as shown in Figure 4.9. Please note that the merge tree kernels in the first phase only expose simple AXI interfaces and each of the merge tree has the same size. This feature allows us to easily implement a coarse-grained floorplanning with the granularity of the separate merge tree kernel. In contrast, if we simply scale the single merge tree and want to make use of all of the FPGA dies, it would requires significantly more engineering efforts to manually split the large merge tree and fit it into the three dies.

Considering that the sorted output from the merge tree of phase 2 will be directly written to the HBM channels, a natural choice is to place the newly added merge units in phase 2 on the bottom die to minimize the signal crossing. After that, we will migrate as many merge tree kernels to the mid and top dies, to save more spaces

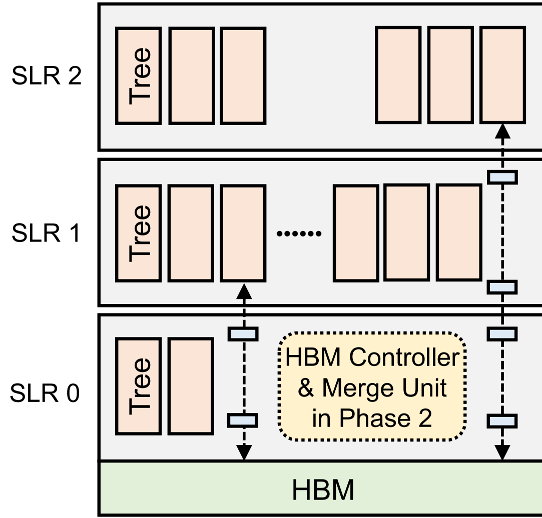

Figure 4.9: The floorplan of the two-phase merge sorter on the Xilinx U280 FPGA. Each Super Logic Region (SLR) is a single FPGA die slice in the Stacked Silicon Interconnect (SSI) FPGA device.

in the bottom die to alleviate the routing issues.

We derive a simple resource model to solve this floorplanning problem. Let S be the resource consumption of each merge tree, u_i be the number of merge trees and a_i be the available resources on the i-th die, w be the width of an AXI interface, and W be the available signals crossing two neighbor dies. Then the problem can be formulated as *maximizing* $(u_1 + u_2)$ under the following constrains:

$$\begin{cases} (u_1 + u_2) \cdot w \leq W & (1) \\ u_1 \cdot S \leq a_1 & (2) \\ u_2 \cdot S \leq a_2 & (3) \end{cases}$$

where constraint (1) limits the AXI connections from all merge trees in both die 1

and 2 to die 0 so that they do not exceed the available amount of signals crossing die 0 and 1, and constrains (2) & (3) make sure the merge trees placed on die 1 and 2 do not exceed each die's available resource.

4.5 Evaluation

In this section, we evaluate the performance, resource utilization and timing closure of TopSort implemented on the Xilinx U280 FPGA board. We also compare the two-phase merge tree sorter to the existing DRAM-based FPGA sorters and show how it can better utilize the tremendous off-chip memory bandwidth.

4.5.1 Experimental Setup

We perform all of the experiments on the Xilinx U280 FPGA board. The FPGA kernel is developed using System Verilog and is synthesized and implemented using Xilinx Vitis 2020.2. We use the pblock method [Xil20b] to place the design modules during the floorplanning. The input data are in a key-value pair format, each with a 32-bit key and a 32-bit value. The keys used in the experiments have a uniform distribution. To ensure the keys are fully randomized, we first generate N records whose keys are incremented from 1 to N, then we use the $random_shuffle()$ function from the python library to shuffle the records. The sorted results are validated by checking whether their keys are from 1 to N.

4.5.2 Merge Tree Configuration & Resource Utilization

We implement 16 merge trees in total, each with 16 leaves and outputs 8 elements (each 8 bytes) per cycle. In phase 1, all 16 merge trees are working in parallel. In phase 2, 4 of the merge trees are reused to form a wider merge tree that can sort 32 elements (each 8 bytes) per cycle.

The resources utilization of the dynamic region[1] is listed in Table 4.4, including the TopSort kernel and HBM controllers. The sorting kernel consumes 54.6% LUTs, 35.5% FFs and 56% BRAMs. DSPs are used to calculate the addresses of AXI read and write.

Table 4.4: Resources utilization and percentage of TopSort.

Components	LUTs	Flip-Flops	BRAMs	DSPs
Available	1,066,848	2,144,089	1,487	8,484
Kernel	582,244	750,505	834	84
Percentage	54.6%	35.5%	56.0%	1.0%
HBM Controllers	87,848	113,814	4	0
Percentage	8.2%	5.3%	0.2%	0.0%

Table 4.5 shows the resource breakdown of each type of the merge trees. For those trees that are not reused (e.g., merge tree 1), each of them takes less than 3% of the available resources. This is consistent with the observation in Section 4.1 that it takes significantly less resources for a single merge tree to saturate one HBM channel than multiple HBM channels. Meanwhile, a reused tree (e.g., merge tree 4) requires

[1] The FPGA is partitioned into two regions, with the dynamic region reserved for user logic and the static region used for infrastructure IPs.

more LUTs because its input buffers need to accommodate larger data bursts. As for the extra 3 merge units used in phase 2, they takes much more resources than a single merge tree, which validates the theoretical analysis that directly scaling the tree throughput requires the resources to grow superlinearly.

Table 4.5: Resources utilization of individual merge trees. Here we pick one merge tree from each type since they have slightly different resource consumption.

Components	LUTs	Flip-Flops	BRAMs	DSPs
Merge tree 1	28,788	39,905	37.5	2
Percentage	2.7%	1.9%	2.5%	0.02%
Merge tree 4	39,195	39,459	60	10
Percentage	3.7%	1.8%	4.0%	0.1%
Extra logic of phase 2	60,239	102,864	144	20
Percentage	5.6%	4.8%	9.7%	0.2%

4.5.3 Design Layout & Frequency

Figure 4.10 shows the final placements of all merge trees and the extra phase 2 logic. Specifically, number 0-15 label the 16 merge trees of phase 1. The orange part in the bottom die represents the extra logic used to form the wider merge tree of phase 2. Clearly, this layout matches the proposed floorplan in Section 4.4.4.

The floorplanning and pipelining effectively reduces local routing congestion, so that the proposed design could achieve 214 MHz in user clock and 414 MHz in the HBM control clock. Without the proper floorplanning, Vivado failed in routing even with the highest optimization level.

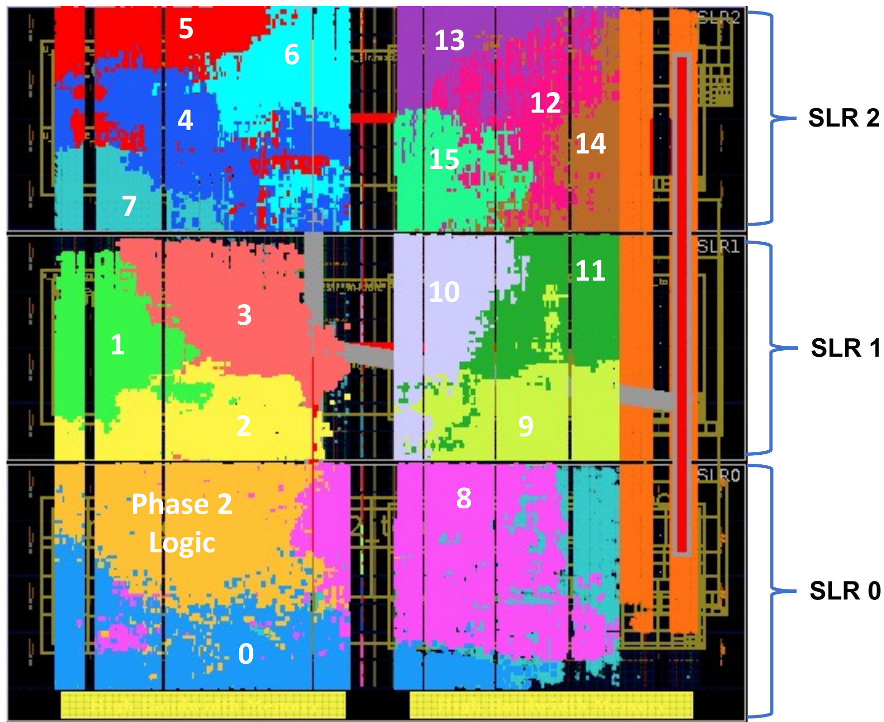

Figure 4.10: The actual layout of the design. Number 0-15 label the 16 merge trees of phase 1. The orange part in the bottom die represents the extra logic used to form the wider merge tree of phase 2.

4.5.4 Overall Sorting Performance

Figure 4.11 presents the performance of sorting data ranging from 32 MB to 4 GB in size. The overall sorting performance is calculated through dividing the data size by the total sorting execution time and the performance of each phase is calculated through dividing the data size by the time each phase takes. The maximal data size that we can handle is half of the HBM capacity, which is 4 GB in the Xilinx U280 FPGA. This is because half of the HBM channels are used for reading and

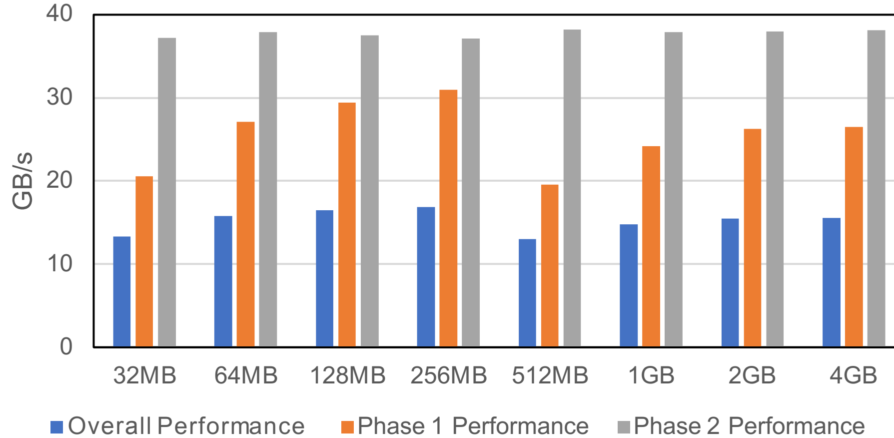

Figure 4.11: Sorting performance with different data sizes. The overall performance is calculated through dividing the data size by the total sorting execution time.

the other half for writing. For datasets whose sizes are larger than 4 GB, another layer of memory with larger capacity such as the host DRAM or FPGA DRAM must be presented: we may need to divide the entire dataset into multiple smaller pieces that can fit into the HBM to get them independently sorted. Then we may merge these sorted pieces of data through another pass. When sorting 4 GB of data, TopSort achieves an overall performance of 15.6 GB/s. Specifically, sorting 4GB data has a performance of 26.5 GB/s in the first phase and 38 GB/s in the second phase.

Figure 4.11 also shows a performance drop when the data size is increased from 256 MB to 512 MB. This is because sorting 512MB data takes 6 passes in phase 1 while sorting 256 MB data takes 5 passes. In fact, we choose to direct each of the merge trees in phase 1 to get two $N/32$-element sorted sub sequences, as illustrated in Section 4.2.4. Since the merge trees in phase 1 have 16 leaves, each of them can

work with at most $2 \times 16^5 = 2^{21}$ elements within 5 passes. There are 16 merge trees working in parallel and each element is 8-byte wide, giving a total data size to be $2^{21} \times 16 \times 8 = 256$ MB. Similarly, data with a size larger than 256 MB but no more than than 4 GB requires 6 passes in phase 1.

Besides, sorting 512 MB to 4 GB data requires the same number of passes in phase 1, but the sorting performance still varies, which is similar for sorting 32 MB to 256 MB data. This is because some input leaves of the merge trees may not be fully active in the last pass of phase 1, similar to what is illustrated in Figure 4.4. For instance, in the case of sorting 512 MB data, each merge tree in phase 1 needs to get two $512/16/2 = 16$ MB sorted sub sequences, or equivalently 2^{21}-element sorted sub sequences. However, we always get $16^5 = 2^{20}$-element sorted chunks after the first 5 passes. That means the 2^{21}-element sub sequence to be sorted contains two 2^{20}-element sorted chunks. During the last pass, each of the two sorted chunks is divided into 8 continuous portions and each portion is fed into one merge tree leaf. In our design, portion 1 of the first sorted chunk goes to leaf 1, portion 2 of the first sorted chunk goes to leaf 2, and portion 8 of the first sorted chunk goes to leaf 8. The correspondence of the 8 portions from the second sorted chunk to leaf 9-16 is the same. In this case, since the records from leaf 1 are always smaller than the records from leaf 2 and the records from leaf 2 are always smaller than the records from leaf 3, etc, only one of the leaf 1-8 is active all the time: e.g., the merge tree will always fetch the records from leaf 1 until it fully consumes portion 1, then it always fetches the records from leaf 2 until portion 2 is fully consumed, so on and so forth. The same situation happens for leaf 9-16. As a result, only 2 leaves out of the 16 leaves are active, which cannot sustain the merge tree to output 8 elements every cycle and leads to a lower performance. Similarly, if the data size is 1 GB, then each merge tree

in phase 1 gets the sorted sub sequences with a size of 32 MB, which contains four 2^{20}-element sorted chunks. Each of the sort chunks will be divided into 4 portions, which will feed 4 leaves. As a result, 4 leaves out of the 16 leaves are active in the last pass, which gives better performance than the case of sorting 512 MB data. Likewise, in the cases of sorting 2 GB and 4 GB data, 8 and 16 leaves out of the 16 leaves are active, since the tree outputs 8 elements per cycle, the merge trees in both cases are fully active. That's why sorting 2GB and 4GB data has quite similar overall performance, which is higher than that of sorting 512MB or 1GB data.

Based on the measurement, we are able to utilize at least 318 GB/s of HBM bandwidth in the first phase, where we run 6 passes and each pass reads and writes to HBM channels simultaneously. Thus the average HBM bandwidth utilized is at least $26.5 \times 6 \times 2 = 318$ GB/s. In fact, the used bandwidth is even higher because we have not account for the idle time of the merge units, which needs to be reset after merging two sorted sequences. In the initial pass, the length of the sorted sequences is 1, so the merge units have to be reset frequently.

4.5.5 Sorting Performance with Different Burst Sizes

Figure 4.12 compares the performance of two burst size choices. Both tests set the burst size as 1 KB in phase 1. In the second phase, we set the burst sizes to be 1 KB and 4 KB. The results show that reading in 4KB bursts in phase 2 gives better performance, which matches the profiling results in Section 4.4.3 that the designers need to use 4KB AXI bursts to maximize the HBM channels' bandwidth when they use one AXI to perform intra-channel accesses.

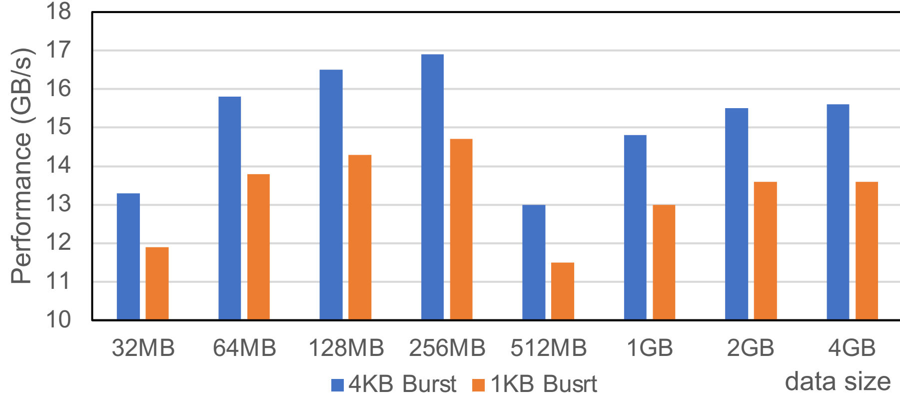

Figure 4.12: Overall sorting performance with the same AXI burst size in phase one but different AXI burst size in phase two.

4.5.6 Comparison with DRAM-based FPGA Sorters

Table 4.6 summarize the performance comparison of TopSort and the existing FPGA-based DRAM sorters. The best-performing DRAM-based FPGA sorter is the proposed sorter in Section 3, which can fully utilize 4 DDR4 DRAM channels on the AWS F1 datacenter FPGA. When sorting 4 GB data, it achieves an overall sorting performance of 7.1 GB/s. Comparing to it, the proposed HBM sorter achieves 2.2× speedup. Besides, we also list the absolute merge tree throughput, which is the number of elements (p) that the merge tree can output per cycle multiplied by the design frequency and the element's width in bytes, to show the utilization of the off-chip memory bandwidth of different merge tree designs. The proposed HBM sorter has an entire tree throughput of 219 GB/s in phase 1 and saturates the 420 GB/s HBM bandwidth (HBM is half-duplex so half of its bandwidth is for reading data and the other half is for writing data). In phase 2, it still has 1.7× and 14× higher merge

Table 4.6: Comparison with existing FPGA based DRAM sorters.

	Off-Chip Memory	Merge Tree Th-put	Overall Perf.
[CMF20]	4 DDR4 channels	N/A	4.3 GB/s
[PBL20]	1 DDR4 channel	4 GB/s	0.2 GB/s [2]
Bonsai	4 DDR4 channels	32 GB/s	5.8 GB/s
TopSort	32 HBM channels	219 GB/s (phase 1) 55 GB/s (phase 2)	15.6 GB/s

tree throughput than that of the proposed DRAM sorter and [PBL20], respectively.

Table 4.7: Comparison of resource utilization for various FPGA based in-memory sorters. The resource efficiency of the sorter is defined as the overall sorting performance in GB/s divided by the LUT utilization in the unit of 100K LUTs.

	LUTs	Flip-Flops	BRAMs	Resource Efficiency
FPGA'20 [CMF20]	328,366	391,900	760	1.3
FPL'20 [PBL20]	44,000	61,000	64	0.5
Bonsai	287,672	768,906	960	2.0
TopSort	582,244	750,505	834	2.7

Table 4.7 compares the resource utilization of TopSort with the existing FPGA in-memory sorters. Since the performance and resource utilization are different among various sorters, we introduce a more unified metric called the resource efficiency,

[2] [PBL20] reported 49× speedup over the C++ std::sort() implementation on an ARM Cortex A53 core when sorting 256 KB data. We re-implement C++ std::sort() on the same CPU and multiply the performance when sorting 32 MB data by 49× to get the listed value.

which is defined as dividing the overall sorting performance of a sorter in GB/s by its LUT utilization in the unit of 100K LUTs. For example, TopSort has an overall sorting performance of 15.6 GB/s and utilizes roughly 5.8×100K LUTs, thus its resource efficiency is 15.6/5.8 = 2.7 GB/s/100K LUTs. Table 4.7 shows that TopSort has the best resource efficiency, due to the two major design choices. First, TopSort chooses to use multiple smaller merge trees to independently work with the HBM channels in the first phase, which provides a linear performance improvement while avoiding the super-linear growth of resource utilization shown in Section 4.1. Second, the logic reuse in the second phase further saves the overall resource utilization. Since the on-chip resource utilization of a design is directly related to its power consumption, the resource efficiency can also serve as a metric to indicate the sorter's power efficiency. As a result, we believe that TopSort also delivers better power efficiency than the existing FPGA sorters.

4.5.7 Comparison with Scaled Single Merge Tree

We also compare TopSort with a single giant merge tree, which is scaled to output 32 elements per cycle and has a throughput equal to writing 4 HBM channels, i.e., the same as the merge tree throughput of phase two in the proposed sorter. Unfortunately, we are not able to route this single merge tree, as it requires significantly more engineering efforts of manual placement optimization compared to the two-phase merge tree.

Nonetheless, we can still estimate its performance, assuming it achieved the same design frequency as TopSort. The best estimation is that the single merge tree has the same number of leaves as the sum of the 16 merge tree leaves in the existing

phase one, which is 256. This means sorting 4 GB data still requires 4 passes. Since the throughput of the merge tree is the same as the merge tree throughput in phase 2, which is 38 GB/s, the overall sorting performance of the single merge tree is thus at most $38/4 = 9.5$ GB/s, much less than the 15.6 GB/s achieved in the proposed two-phase merge tree sorter.

4.5.8 Sensitivity to the Input Data

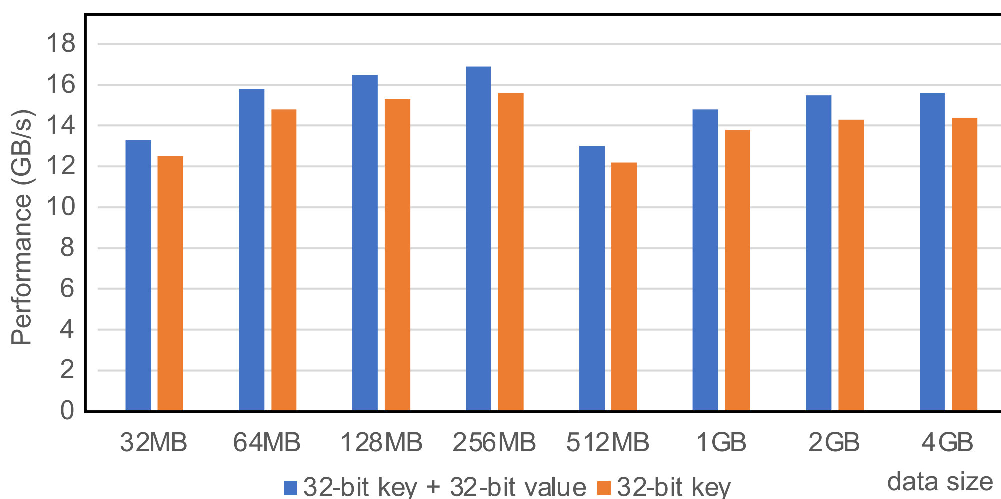

Figure 4.13: Overall sorting performance when sorting (32-bit key, 32-bit value) pairs and 32-bit keys. The performance variation is due to the difference in the design frequencies.

The methodology and architecture of TopSort can be adopted to sort different datasets with different kinds of keys or (key, value) pairs. Figure 4.13 shows the overall sorting performance when sorting two kinds of datasets with uniformly random distributions. One kind of them is (32-bit key, 32-bit value) pairs, as mentioned above. The other kind is pure 32-bit keys. In the case of sorting 32-bit keys, we still

use 16 merge trees in phase 1, each having 16 leaves and output 16 elements per cycle at its root. At the leaf layer of each merge tree, we have 8 2-rate merge units. After the first phase, each merge tree will generate 2 $N/32$-element sorted sub sequence. In phase 2, we still reuse 4 merge trees from phase 1 to form a wider tree that has 64 leaves and outputs 64 elements per cycle. We can see that both cases report similar overall sorting performance, this is because they have the same number of passes when sorting the same amount of data, despite their record width is different.

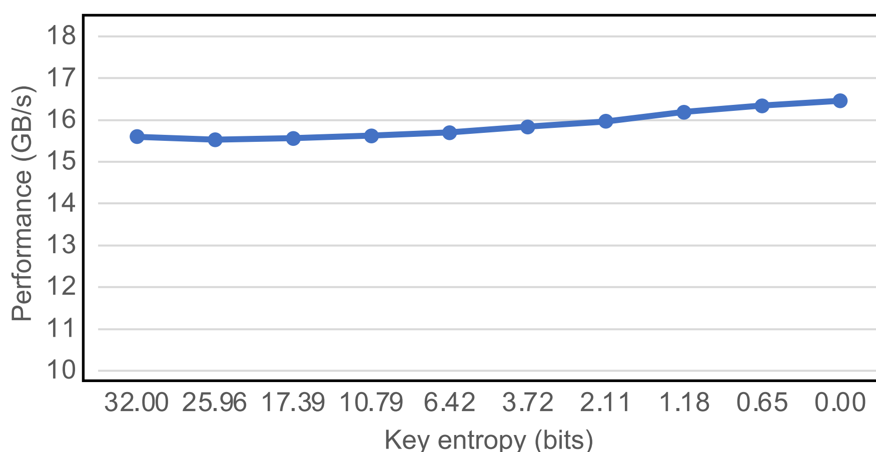

Figure 4.14: Overall sorting performance when sorting 4 GB data of (32-bit key, 32-bit value) pairs with different key distributions.

We also evaluate the performance of TopSort using different key distributions. In order to generate various key distributions with different skew, we adopt the benchmark from [TS92], which uses the Shannon entropy to measure the key distribution. The idea is that skewed keys can be generated by performing *bitwise AND* operations on the adjacent keys which originally have a uniformly random distribution. For instance, an entropy of 32 bits for 32-bit keys means the keys are uniformly random distributed. On the other hand, an entropy of 0 bit indicates all the keys

are the same.

Figure 4.14 shows the overall performance of TopSort when sorting vastly different skewed data. We observe that TopSort is insensitive to the data skew. Specifically, TopSort achieves the highest performance of 16.4 GB/s when sorting data that share the same key. This is because we design each merge unit so that it always fetches records in a round-robin fashion from both input ports if the keys from the two input ports are the same. In the case of sorting data that have the same key, any E-rate merge unit at any level of the merge tree is guaranteed to first consume E records from its left-side input port, then consume E records from its right-side input port, etc. In this way, the merge tree is always fully active during the sorting process.

4.5.9 Comparison with GPU Sorter

Another type of computing platforms that is equipped with HBMs is the modern GPU. Here we compare the sorting performance of TopSort with that of the latest NVIDIA A100 GPU [NVI20] using CUDA Thrust::sort [NVI22]. Table 4.8 summarizes the comparison details, including the technology, the available HBM bandwidth, the sorting performance, the power consumption and the power efficiency. We can see that the NVIDIA A100 GPU is 2.6× faster than TopSort, but it also has 3.4× higher available HBM bandwidth as well as consumes 4.5× more power. In terms of the power efficiency, TopSort is 2× power efficient than the NVIDIA A100 GPU. This indicates that the datacenter engineers may have the choice to use different sorters depending on the requirements of the application scenarios. On the one hand, in the case where the power efficiency is prioritized over the sorting performance, TopSort should be adopted to better save the energy. On the other hand, if the sorting

Table 4.8: Comparison with the latest NVIDIA A100 GPU sorter. The unit of Perf./Watt is GB/s/W.

	Technology	HBM BW.	Performance	Power	Perf./Watt
NVIDIA A100	TSMC 7nm	1555 GB/s	40.5 GB/s	278 W	0.15
TopSort	TSMC 16nm	460 GB/s	15.6 GB/s	51 W	0.30

performance is the biggest consideration, a high-end GPU such as the NVIDIA A100 may be more suitable.

4.6 Summary

In this chapter, we present TopSort, a high-performance two-phase sorting solution specialized for HBM-based FPGAs. We show that the sorting performance on HBM-based FPGAs is bounded by the limited on-chip resources. To achieve better performance and alleviate the resource bottleneck, we propose to perform parallel sorting with smaller merge trees in the first phase and reuse the logic from the first phase to form the final giant tree in the second phase. Several HBM-related characteristics have been profiled and the proposed sorting design is coupled with HBM-specific optimizations to reduce the resource overhead, improve the bandwidth utilization and improve the timing closure.

CHAPTER 5

Sorting Acceleration on In-Storage Computing Devices

If the data to be sorted is too large to be held in the DRAM or HBM, then a secondary storage device such as a hard disk or a flash is needed to store the intermediate data and final results. The most common storage-based sorting algorithm is external merge sort [Knu98], which also consists of two phases: the sorting phase and the merging phase. In the sorting phase, data are first sorted into intermediate chunks that can fit into the main memory such as DRAM or HBM. In the following merging phase, those sorted chunks are merged and written into the external storage through one or multiple merging passes to be the final result. Unlike the HBM-based sorting case from Chapter 4 where the two phases are strategically employed to maximize the usage of the off-chip memory bandwidth, the two phases in storage-based sorting is rather straightforward as the main memory now acts as a buffer to hold the intermediate data and avoid the slow access to the storage as much as possible.

The emerging in-storage computing devices bring new opportunities to accelerate external merge sort. Instead of first copying the data to the host side, the co-processor placed right next to the storage could process the data in the drive directly, thus it can perform the sorting tasks more efficiently. Specifically, the newly released

Samsung SmartSSD is a suitable candidate for in-storage sorting acceleration. Samsung SmartSSD is a programmable computational storage platform that integrates an FPGA into the same package as the flash. With Samsung SmartSSD, one can accelerate the sorting phase using customized FPGA accelerators while avoiding the expensive data transfer between the host and the flash in the merging phase. However, it is still an open question of how to achieve optimized performance when accelerating sorting on in-storage computing devices such as Samsung SmartSSD, especially with the consideration of the low bandwidth of the storage and limited near-storage computing resources.

In this chapter, we propose FANS, an FPGA-accelerated in-storage sorting solution that achieves remarkable overall sorting performance when targeting Samsung SmartSSD. First, through a comprehensive analysis of the sorting system based on the memory hierarchy of SmartSSD, we identify that in addition to the flash bandwidth, there are multiple key factors that determine the overall performance, including the merge sort kernel configuration, the FPGA DRAM specification and the sorted chunk size after the sorting phase. Based on our analysis, we provide an optimized sorting solution for Samsung SmartSSD. Notably, the proposed solution has a distinct sort phase and merge phase, and each phase is configured separately to maximize the entire sorting performance. Different phases can be activated at runtime by reprogramming the FPGA. We also provide valuable architectural insights based on our analysis and experiments to help vendors further improve their in-storage computing devices. The proposed in-storage sorting accelerator with optimized design configuration achieves more than 3× speedup over the standalone FPGA-based flash storage.

5.1 System Architecture of FANS

In this section, we present the necessary architecture support for in-storage sorting acceleration. First we show the internal architecture of the sorting kernel that works with the FPGA DRAM. Then we illustrate how we improve the end-to-end system performance that takes into consideration the performance of the merge tree sorting kernel and the flash access bandwidth.

We specifically split the entire merge sort process into two phases: the sorting phase and the merging phase. In the sorting phase, the data is sorted into DRAM-scale subsequences and the merging phase assembles the subsequences into the final output. Note that in the merging phase, the size of the merged output is larger than the FPGA DRAM capacity and we have to write the partially merged sequences directly back onto the flash when the DRAM capacity has been fully occupied.

5.1.1 Sorting Kernel

Since the sorting kernel implemented on the FPGA still primarily interacts with its DRAM, we use the same merge tree architecture as the DRAM-based sorter from Chapter 3 as the fundamental kernel. The detailed micro-architecture of the merge tree kernel is shown in Figure 5.1. The FPGA DRAM is still accessed through a 512-bit wide AXI interface and we need to split the 512-bit data into individual records before sending them cycle by cycle into the merge tree. The role of the pre-sorter and the round-robin arbiter is the same as those described in Chapter 3.3. We also use the same definition of (p, l) to denote the merge tree's root throughput and tree leaf number.

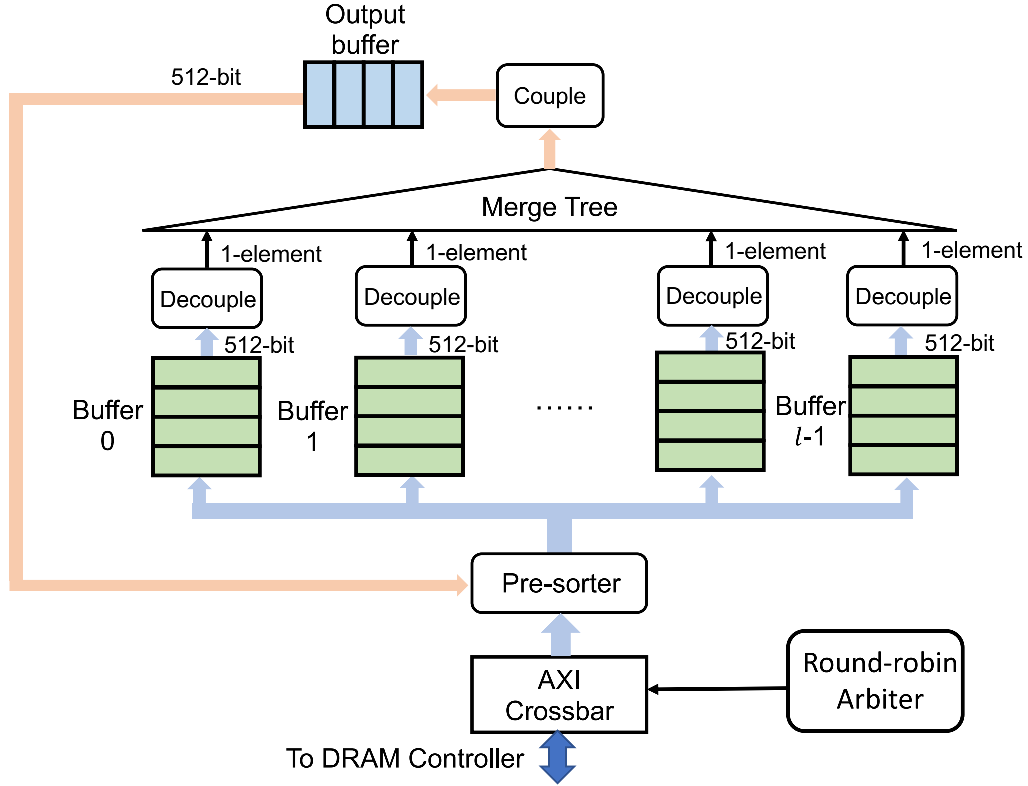

Figure 5.1: The internal architecture of the sorting kernel.

5.1.2 Sorting Phase Design

Initially, the original data are stored entirely in the flash and we fetch them into the FPGA DRAM in batches. We use the merge tree sorting kernel described in the previous sub-section to sort the data in one or more passes. Assume the configuration of the sorting kernel is (p_1, l_1) where p_1 refers to the merge tree throughput and l_1 is the tree leaf number. After each pass, the partially sorted chunks will grow by a factor of l_1 in size correspondingly. Note that since the bandwidth of the FPGA DRAM is higher than the flash and the FPGA DRAM is closer to the kernel, a

natural design choice is to sort the data for multiple passes in the FPGA DRAM before we write them back to the flash.

Since the end-to-end execution time of the sorting phase includes both the sorting time and flash accessing time, we apply the double buffering scheme [CWY17] to improve the overall performance. As shown in Figure 5.2, the FPGA DRAM is split into two sets and each set contains two buffers of equal size. The merge tree kernel works with a single set each time: in one pass, it reads the data from the read buffer 0 and write to the write buffer 0; in the next pass, it reads the partially sorted data from the write buffer 0 and write to the read buffer 0, etc. At the same time, the write buffer 1 sends the sorted results of the previous batch back to the flash and the read buffer 1 fetches the next batch of data to be sorted from the flash. Please note that although the P2P transfer can stream data directly between the FPGA DRAM and the flash, it still relies on the host to issue the transfer commands. As a result, we use three threads on the host side to overlap the data transfer and FPGA kernel execution.

5.1.3 Merging Phase Design

The merging phase requires the modification of the merge tree kernel to support additional host-FPGA communication feature, which will be explained below. As a result, we use another merge tree kernel and assume the tree configuration is (p_2, l_2) in this case. Since the size of the output data in this phase will exceed the capacity of the FPGA DRAM, we explicitly reserve l_2 input buffers in the FPGA DRAM, each for a sorted chunk of data that is stored in the flash. There is also an output buffer to store the merged output data, and the size of the output buffer is equal

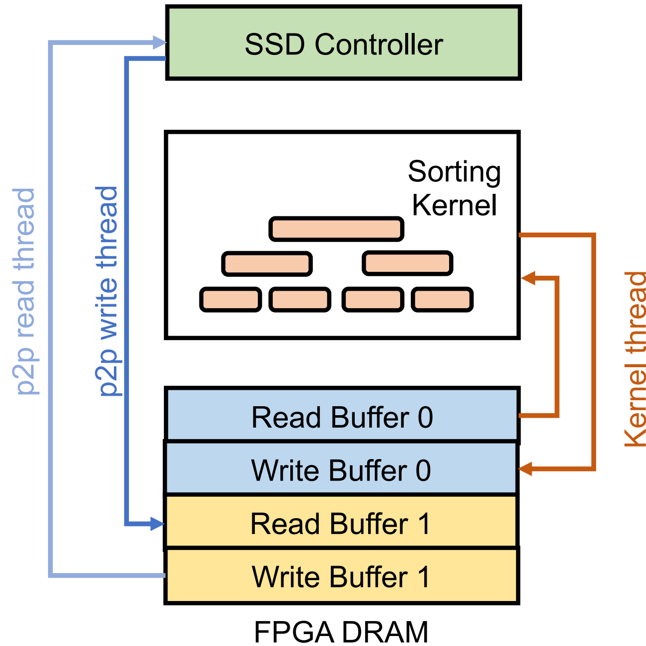

Figure 5.2: The double buffering scheme with the host-side multi-threading in the sorting phase.

to the sum of the input buffers. A detailed example is shown in Figure 5.3. At the beginning of the merging phase, the first batch of the sorted chunk 0 is fetched into the buffer 0 in the FPGA DRAM and the first batch of the sorted chunk 1 is also fetched into the buffer 1 in the FPGA DRAM, etc. After all of the buffers are filled with data, the merge tree kernel starts to merge these data into a larger sorted chunk and write it back to the FPGA DRAM. If the data in a buffer have been consumed by the kernel, then we will fetch another batch of data from its corresponding sorted chunk in the flash to the same buffer again. For example, in Figure 5.3 we will fetch the second batch from the sorted chunk 0 to the buffer 0 after the first batch is fully

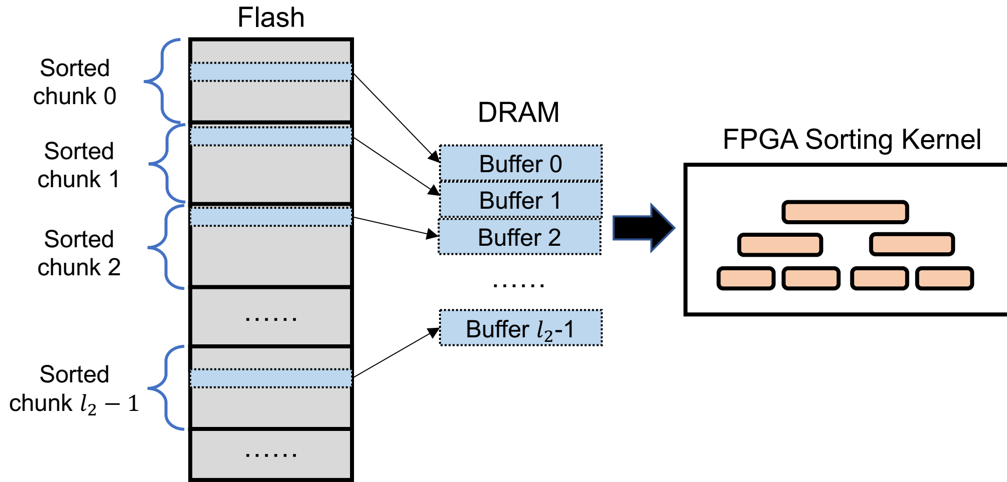

Figure 5.3: Illustration of using the DRAM as buffers for the flash in the merging phase. Here we only show the input buffers and omit the output buffer for simplicity.

fed into the merge tree.

Depending on the relative order of different data batches, some buffers will be emptied earlier than others. To make the merging result correct, we have to suspend the merge tree kernel, fetch the next batches for those empty buffers and then resume the merge tree kernel again. To support this feature, the merge tree kernel needs some minor modifications. First, we keep track of the amount of data that have been sent into each leaf node. Whenever a leaf node has consumed the same amount of data as the FPGA DRAM buffer size, it means the corresponding buffer in the FPGA DRAM is already empty. In this case, the rest of the on-chip leaf buffers stop feeding data into their corresponding leaf nodes. Second, the FPGA needs to message the host which buffer gets empty so that the host can issue the P2P transfer commands to fetch the data into the corresponding buffer from the flash. Thanks to

the fact that the bitstream on the FPGA stays the same, we could resume the merge tree kernel again without changing its intermediate states.

To hide the latency of the flash storage access and improve the end-to-end performance, we utilize the same double buffering approach as the one in the sorting phase. That is, we have two sets of input buffers on the DRAM and each set contains l_2 buffers. There are also two output buffers and each output buffer has a size equal to the total size of the l_2 input buffers. We denote the first set of input buffers as $readBufA_0$ to $readBufA_{l_2-1}$ and the second set of input buffers as $readBufB_0$ to $readBufB_{l_2-1}$. We also denote the two output buffers as $writeBufA$ and $writeBufB$. Then the general process of the double buffering approach is shown in Figure 5.4.

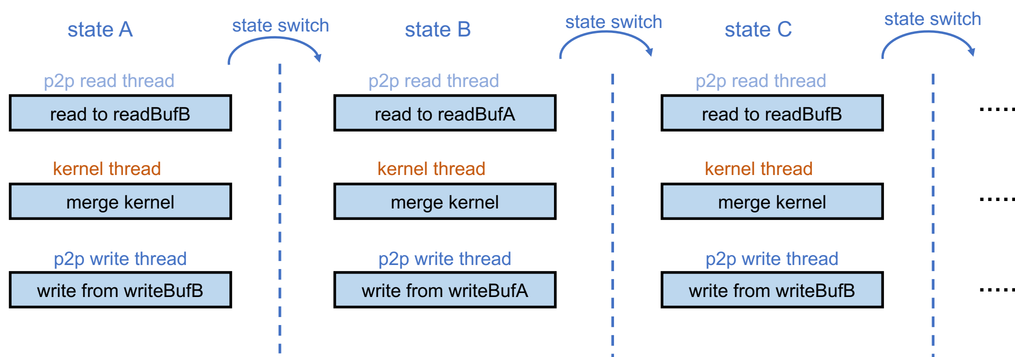

Figure 5.4: Illustration of the double buffering scheme in the merging phase. There are three threads working in parallel seen from the host side. At any time of the process, the kernel thread is invoking the FPGA merging kernel to read from a set of read buffers and to write to the write buffer, while the p2p read thread and the p2p write thread are working with the other set of read buffers and the write buffer.

Let us assume that in the current state the merge kernel is merging the inputs

from $readBufAs$ and writing the merged results to $writeBufA$, which corresponds to the state A in Figure 5.4. At the same time, the p2p read thread keeps fetching inputs from the SSD to the l_2 read buffers for the $readBufB$ set in a round-robin fashion. Whenever the merge tree kernel consumes all of the inputs in one of the read buffers, e.g., it consumes all the records in $readBufA_1$ from its leaf 1, it will continue to fetch the inputs from $readBufB_1$. The state switch happens when we find that $writeBufA$ is full, then we switch to the state B, where we direct the merge kernel to writing its outputs to $writeBufB$ while the p2p write thread is writing the merged results from $writeBufA$ to the SSD. We repeat the process until the entire dataset is merged.

5.2 Performance Modeling

In this section, we combine the external memory model and extend the performance model of the DRAM-based sorter from Chapter 3.2 to analyze the overall performance of FANS. In addition to the parameters summarized in Chapter 3.2, we list a few more parameters that we will take into consideration in Table 5.1, in addition to the common parameters used for the DRAM sorter in Table 3.1.

We show that the bottlenecks of the two phases may come from the FPGA DRAM bandwidth, the flash bandwidth and even the choice of intermediate sorted chunk size M after the sorting phase. Based on our analysis, we give an optimized configuration for the current Samsung SmartSSD device. Our analysis also serves as an in-depth case study that could help the vendors improve their near-storage computing products.

Table 5.1: Extra parameters considered in FANS.

Symbol	Definition
n	Number of elements the pre-sorter can sort
M	Size of sorted chunk after the sorting phase
β_{read}	Read bandwidth of the flash
β_{write}	Write bandwidth of the flash
(p_1, l_1)	Merge tree configuration of the sorting phase
(p_2, l_2)	Merge tree configuration of the merging phase

5.2.1 Modeling of The Sorting Phase

In the sorting phase, the initial input data produced by the pre-sorter is in a sorted chunk of size n and then we feed them into the merge tree of (p_1, l_1) through multiple passes. We use M to denote the size of the sorted output after the sorting phase. Thus the number of passes is $\lceil \log_{l_1}(M/n) \rceil$. Using the Equation 3.1 in Section 3.2, the total time for sorting a batch of M data can be expressed as:

$$t_M = \frac{Mr \cdot \lceil \log_{l_1}(M/n) \rceil}{\min\{pfr, \beta_{\text{DRAM}}\}}. \tag{5.1}$$

Assume the available FPGA resources allow us to properly scale the merge tree to saturate the DRAM bandwidth, then the kernel performance β_{kernel} is :

$$\beta_{\text{kernel}} = \frac{Mr}{t_M} = \frac{\beta_{\text{DRAM}}}{\lceil \log_{l_1}(M/n) \rceil} \tag{5.2}$$

In the sorting phase, since we overlap the flash read and write with the kernel execution, the actual performance of sorting the data is

$$\beta_{\text{sort}} = \min\{\beta_{\text{read}}, \beta_{\text{kernel}}, \beta_{\text{write}}\}$$

Therefore, the total time for sorting the entire data of size N would be:

$$t_1 = \frac{N}{M} \cdot \frac{M}{\beta_{\text{sort}}} = \frac{N}{\min\{\beta_{\text{read}}, \beta_{\text{kernel}}, \beta_{\text{write}}\}} \tag{5.3}$$

5.2.2 Modeling of The Merging Phase

In this phase, the data in the DRAM will go through the merge tree by one pass and the throughput of the merge tree kernel itself is $\min\{pfr, \beta_{\text{DRAM}}\}$. The end-to-end merging performance is also dependent on the flash read and write bandwidth, and since we use double buffering to hide the flash access latency, the actual merging performance will be

$$\beta_{\text{merge}} = \min\{\beta_{\text{read}}, \min\{pfr, \beta_{\text{DRAM}}\}, \beta_{\text{write}}\}$$

Since the read and write bandwidth of the flash are smaller than those of the FPGA DRAM and one can easily implement a merge tree that saturates the flash access bandwidth, the merging performance can be simplified as:

$$\beta_{\text{merge}} = \min\{\beta_{\text{read}}, \beta_{\text{write}}\} \tag{5.4}$$

We use (p_2, l_2) to denote the configuration of the merge tree in the merging phase. The merging phase starts with partially sorted chunk of size M, so we need $\lceil \log_{l_2}(N/M) \rceil$ more passes to get the entire N size data to be sorted. As a result, we derive the total time of the merging phase as follows:

$$t_2 = \lceil \log_{l_2}(N/M) \rceil \cdot \frac{N}{\min\{\beta_{\text{read}}, \beta_{\text{write}}\}} \tag{5.5}$$

5.2.3 Optimized Configuration for Samsung SmartSSD

The above analysis on the two phases applies for any external merge sort implementations when using FPGA as the hardware accelerator. Formally, we need to find

$$\arg\max_{p_1,\ell_1,p_2,\ell_2,M} \left\{ \frac{N}{\min\{\beta_{\text{read}}, \beta_{\text{kernel}}, \beta_{\text{write}}\}} + \lceil \log_{l_2}(N/M) \rceil \cdot \frac{N}{\min\{\beta_{\text{read}}, \beta_{\text{write}}\}} \right\},$$

subject to

$$\begin{cases} LUT(p_1, l_1) & \leq C_{\text{LUT}} \\ b \cdot \ell_1 & \leq C_{\text{BRAM}} \\ LUT(p_2, l_2) & \leq C_{\text{LUT}} \\ b \cdot \ell_2 & \leq C_{\text{BRAM}}. \end{cases}$$

For Samsung SmartSSD we can get an optimized sorting configuration based on its physical characteristics. Inside SmartSSD, the SSD controller is connected to the FPGA through the PCIe Gen3.0×4 links, which has a theoretical 4 GB/s full-duplex bandwidth. Our profiling shows that the actual bandwidths for continuous read and write are both around 3 GB/s. Meanwhile, the FPGA is equipped with a single DRAM, whose theoretical bandwidth is 16 GB/s and the actual bandwidth is around 14 GB/s in half-duplex.

For the sorting phase, we notice that the kernel performance is lower than the flash read or write bandwidth. There are three reasons for the phenomenon. First, there is only one DRAM bank connecting to the FPGA kernel and the effective DRAM read

and write bandwidth β_{DRAM} is roughly 7 GB/s. Second, sorting random data into DRAM-scale chunks (e.g. hundreds megabytes) usually takes more than 2 passes, as constructing a merge tree with a larger number of leaves will require more FPGA resources than available. Finally, due to the limit of the SmartSSD system, the data from the flash must be first written to the FPGA DRAM and then be read into the sorting kernel, which will further reduce the effective DRAM bandwidth for the sorting kernel. As a result, the total time for the sorting phase in Samsung SmartSSD is

$$t_1 = \frac{N \cdot \lceil \log_{l_1}(M/n) \rceil}{\beta_{\text{DRAM}} \cdot (1-\gamma)}, \tag{5.6}$$

where γ represents the DRAM degradation factor and indicates that P2P transfers also consume DRAM bandwidth.

The analysis of the merging phase for Samsung SmartSSD stays the same and we use β_{read} to denote the minimum number of the flash read and write bandwidth. Then we derive the total time of both phases as

$$t_{\text{total}} = \frac{N \cdot \lceil \log_{l_1}(M/n) \rceil}{\beta_{\text{DRAM}} \cdot (1-\gamma)} + \lceil \log_{l_2}(N/M) \rceil \cdot \frac{N}{\beta_{\text{read}}}, \tag{5.7}$$

Equation 5.7 shows that, for the current Samsung SmartSSD device, both the communication bandwidth of the slow storage device and the merge tree configuration on the FPGA could be the bottlenecks to the overall performance. This modeling provides more comprehensive analysis compared to the previous belief that only the flash bandwidth could be the weak point [JXA17]: ensuring that the merge tree's throughput saturates the DRAM's bandwidth is not enough, one must take the tree leaf number and the intermediate sorted chunk size after the sorting phase into consideration to best overlap the merge tree kernel execution and the flash access.

To get good sorting performance, we need to select both the appropriate merge tree configuration (p, l) and the intermediate sorted chunk size M. We summarize the intuition as below:

- Firstly, since the FPGA DRAM and the flash bandwidth is relatively small, we only need to choose the minimal p_1 and p_2 required to saturate the FPGA DRAM bandwidth and the flash bandwidth. For example, when each data element takes 16 bytes and assume the merge tree kernel is running at 250 MHz, a merge tree with $p_1 = 2$ for the sorting phase and another merge tree with $p_2 = 1$ for the merging phase is optimal.

- Secondly, with the tree throughput p determined, we try to build the merge trees with the maximum number of leaves l such that the on-chip resources allow, as increasing l_1 and l_2 could always be beneficial.

- Finally, the choice of M should also be appropriate. Although we get the optimal value of M from solving Equation 5.7, we provide the intuition as below. In the sorting phase, we fetch all of the data from the storage to the FPGA kernel only once and the performance is bounded by the kernel execution. If reducing M can reduce the number of passes that the sorting kernel spends on each unsorted data batch, the sorting phase's performance will be improved. On the other hand, the merging phase always runs at a speed that saturates the flash access bandwidth. But, if increasing M could reduce the number of passes that the data travels between the flash and the on-chip merge tree, the merging phase's performance could also be improved. Considering that the number of passes (the logarithm item in the Equation 5.7) needs to be rounded into discrete integers, M is chosen as the minimum

number that will not increase the number of merging passes.

5.2.4 Architectural Insights

The proposed performance model is not only a guide to determine the optimal design choices for the current Samsung SmartSSD devices. In fact, the in-storage device vendors can also benefit from our study and foresee the potential performance gain from the architectural advancement.

- The drive's bandwidth plays an important role in the end-to-end execution of the merging phase. While the current solid-state drives use the four-lane PCIe Gen3 links as the primary interconnection to the host and the FPGA, the technology scaling that incorporates the next generation's PCIe links will boost the merging performance.
- While the capacity of the FPGA DRAM does not impact the performance, its bandwidth is the primary bottleneck in the sorting phase. We anticipate that the high-bandwidth memory (HBM), which features smaller capacity but much higher bandwidth will be a better candidate to perform sorting tasks in in-storage computing devices.
- The requirement that SSD data must first be transferred to the FPGA DRAM further reduces the performance in the sorting phase. In contrast, allowing the PCIe data to be directly sent to the FPGA kernel as in [RHC19] could remove the DRAM degradation factor γ in Equation 5.7.

5.3 Evaluation

In this section, we evaluate the overall performance of FANS implemented on Samsung SmartSSD. First, we describe the experimental setup. Then, we show the detailed merge tree configuration, the performance breakdown of the sorting execution and the impact of P2P transfers on the effective FPGA DRAM bandwidth. We also compare FANS with the existing standalone FPGA-based flash storage and illustrates why FANS achieves better performance. Lastly, we quantitatively evaluate the benefits that arise from the in-storage characteristics of Samsung SmartSSD.

5.3.1 Experimental Setup

We perform the experiments on Samsung SmartSSD with the Xilinx OpenCL runtime. We prepare the bitstreams of the two phases in advance and reprogram the FPGA at runtime to switch between different bitstreams. The host-side multithreading is implemented through POSIX Threads (pthreads). The sorting kernel is developed using System Verilog and is synthesized and implemented using Xilinx Vitis 2019.2. The kernel is tuned to run at a minimum frequency of 230MHz.

The benchmark used in our experiments is generated from the public Terasort benchmark [Gra], where each record is 100 bytes with a 10-byte key and a 90-byte value. To save the storage and the PCIe bandwidth, we use the same method in [JXA17] that converts the 90-byte value into the 6-byte index. That is, the record we actually sort is 16 bytes with a 10-byte key and a 6-byte index. We generate 2^{33} records, which is 128 GB in size and place them into SmartSSD in advance.

Table 5.2: Detailed Configuration for Samsung SmartSSD.

Component	LUT	Flip Flop	BRAM
Available	326679	668532	647
$(p_1 = 2, l_1 = 64)$	160543	243145	127
Utilization	49%	36%	20%
$(p_2 = 1, l_2 = 64)$	152646	246865	127
Utilization	47%	37%	20%
Sorted chunk size M		256MB	

5.3.2 Design Configuration

We implement the merge trees as Section 5.2 suggest: for the sorting phase, we set the tree throughput p_1 to 2; for the merging phase, we set the tree throughput p_2 to 1. We also use a pre-sorter that is able to sort 4 elements every cycle in the sorting phase. Then we maximize the corresponding leave number l_1 and l_2 that on-chip FPGA resources allow and make sure the design does not have a routing failure. Furthermore, to relieve the routing congestion, we use LUTRAMs instead of BRAMs to implement most of the leaf node buffers. Finally, we find the maximum l_1 and l_2 to be 64, respectively. After fixing l_1 and l_2, the intermediate sorted chunk size M is also determined to be 256MB. A detailed resource utilization of the merge tree kernels is listed in Table 5.2.

5.3.3 Performance Breakdown

The performance of each phase of FANS when sorting 128GB data is shown in Table 5.3. In the sorting phase, the time spent is around 85 seconds, meaning the average performance is 1.5GB/s. The performance is much less than the roughly 3GB/s flash read and write bandwidth and it matches the analysis that the merge sort kernel but not the slow I/O is the bottleneck in this phase.

In the merging phase, it takes 2 passes to merge the 256MB sorted subsequences into the final sorted results. The actual measurement of the performance for each pass is around 1.3GB/s, which is slightly less than the flash read and write speed. This is because halting the merge tree kernel and notifying the host to issue the flash access commands have overhead. In fact, the merge tree kernel is suspended whenever its DRAM buffers become empty. This issue can be resolved in future in-storage computing devices where the FPGA may have direct control over the PCIe link and no longer rely the host to issue the flash access commands.

The minimum time it takes to reprogram the FPGA is roughly 4 seconds for the merging phase, which is highly dependent on the size of the bitstream. Although it takes several seconds, it only accounts for 2% of the total execution time so this reprogramming overhead is completely acceptable in the in-storage sorting case. Please note that while we configure the merge tree to have different tree throughput and tree leaf number between the two phases, the reprogramming overhead is unavoidable for the current SmartSSD, even if the two phases have the same tree configuration. This is because the kernels in the sorting phase and the merging phase have different behaviors, as explained in Section 5.1. In the sorting phase, the merge tree is guaranteed to consume the records from all of its leaf buffers before notifying the

Table 5.3: Performance breakdown for each phase.

Phase	Time (s)
Sorting phase	85
Merging phase	100
Reprogram	4
Total	189

host that the execution is done. In contrast, in the merging phase, some of the leaf buffers may be fully consumed earlier so the merge tree needs to be switched to fetch the inputs from the other set of the read buffers.

5.3.4 Impact of the DRAM

Since the direct P2P transfers between the flash and the FPGA DRAM reduce the effective DRAM bandwidth available to the merge sort kernel and thus harm the performance of the sorting phase. we anticipate that if the FPGA can access the flash data directly from the PCIe transaction packets instead of from its DRAM, we can remove the DRAM degradation factor γ in Equation 5.6. To validate the claim, we do another experiment that loads 256MB data onto the FPGA in advance and we measure the pure kernel performance of the merge tree kernel in the absence of P2P transfers.

It turns out the pure merge sort kernel without the simultaneous access of the FPGA DRAM from the P2P transfers has a performance of 1.8 GB/s, which is 20% faster than the actual performance we observe in the sorting phase. This indicates that if the on-chip merge tree kernel is able to directly stream data into the flash via

PCIe links, we can get another 20% performance speedup in the sorting phase.

5.3.5 Comparison with Related Work

[JXA17] builds an FPGA-accelerated flash storage system where a Xilinx Vertex 7 FPGA is directly coupled with a custom flash expansion card via two FMC ports. Using the system, [JXA17] is able to sort 10^{10} 16-byte key-index pairs or 150GB of data in 700 seconds. In contrast, FANS sorts 128GB of data in 190 seconds. Since the amount of time it takes to sort a dataset is linear to the size as long as the number of merging passes does not change, we can use the overall performance, or the total size of data divided by the entire sorting time, to make a fair comparison. In terms of the overall performance, FANS achieves 3.2× speedup over the sorting system in [JXA17].

Table 5.4: Comparison between FANS and [JXA17]

	Item	[JXA17]	FANS
Hardware Spec.	DRAM BW. (GB/s)	16	16
	DRAM Capacity (GB)	1	4
	Flash Read BW. (GB/s)	2.4	3
	Flash Write BW. (GB/s)	2	3
Design Config.	No. of Leaf Mergers	16	64
	Frequency (MHz)	125	250
	Sorted chunk size M (MB)	512	256
Performance	Sorting Performance (GB/s)	0.21	0.68 (3.2×)

Table 5.4 summarizes the difference of the hardware and the design choices be-

tween [JXA17] and FANS. We notice that Samsung SmartSSD has a higher flash access bandwidth, which enables the merge tree kernel to access the flash at a faster speed of 1.5×. The different design choices further broaden the performance gap: on the one hand, we are able to use a merge tree configuration with a 4× larger number of leaves l to reduce the number of passes it takes to sort the entire data. On the other hand, we select a smaller intermediate sorted chunk size M and uses the pre-sorter to decrease the number of passes that the kernel takes to access the FPGA DRAM, thus improving the performance in the sorting phase.

5.3.6 Benefits from In-Storage Acceleration

The feature of integrating the FPGA into the flash package and allowing P2P transfers without interfering the host gives another level of system gain. To quantitatively compare the gain, we perform a check-experiment using SmartSSD but relying on the host to access the flash storage as well as the FPGA: in this case the FPGA is utilized as a conventional accelerator and does not have the direct access to the flash.

As shown in Table 5.5, the involvement of the host reduces the effective bandwidth between the FPGA and the flash from 3GB/s to 1.5GB/s. Using the analysis in Section 5.2, we anticipate that the performance bottleneck in the sorting phase will be shifting from the merge tree kernel to the effective FPGA-flash bandwidth and the performance of the merging phase is going to be directly reduced by 2×.

Table 5.5: Effective communication bandwidth between the FPGA and the flash without P2P transfers.

Direction	Performance (GB/s)
flash to FPGA	1.5
FPGA to flash	1.3

5.4 Summary

In this chapter, we demonstrate that how the DRAM-based merge tree sorter can be modified to support sorting large-scale data, where a secondary storage with slow I/O bandwidth involved. We target the sorting acceleration on the first industrial in-storage computing device, Samsung SmartSSD and reveal the performance bottlenecks of the in-storage sorting process through the analytical analysis as well as experimental evaluation: in contrast to the prior belief that the slow flash access is the only limitation to the sorting performance, we find that flash access bandwidth, the FPGA DRAM bandwidth, the configuration of merge tree kernel and the intermediate sorted chunk size all impact the overall performance. The proposed in-storage sorting accelerator also utilizes the unique reconfigurability of the FPGA to optimize the different phases with bitstreams and shows negligible reprogramming overhead.

CHAPTER 6

Acceleration of High-Throughput Lossless Compression

Data compression techniques have been widely used in the data center to reduce the data storage and movement overhead. Owing to their inherent support for massive parallelism, FPGAs are well suited to accelerate the high-throughput lossless compression algorithms. However, big data compression with parallel requests intrinsically poses two challenges to the overall system throughput.

First, unlike the sorting application, high-throughput lossless compression is a streaming application where the FPGA reads its input data from the processor, performs the compression and then writes the output back to the CPU at line rate without storing any intermediate data in its DRAM. This CPU-FPGA communication can introduce significant overhead and degrade the end-to-end throughput for the CPU-FPGA platforms. According to [HWY16, CFH18a], the FPGA-accelerated compression system with limited CPU-FPGA communication bandwidth may achieve marginal improvement compared to a multi-core CPU implementation for big data workloads that usually perform compression on different data partitions concurrently.

Second, scaling current FPGA-based high-throughput compressors also encoun-

ters the bottleneck, even given higher CPU-FPGA communication bandwidth. On the one hand, the resource utilization usually grows super-linearly with the compressor's throughput. On the other hand, the critical path of the design also increases when one directly scales the number of bytes the compressor processes per cycle. As a result, further scaling the compressor's throughput may result in lower clock frequency, saturated throughput and lower area efficiency.

In this chapter, we present a multi-way parallel compression accelerator design where each way represents a well-optimized and fully-pipelined compression engine. It improves the overall design scalability as the clock frequency of the design is not affected by the number of compression engines. It also improves the performance-per-area efficiency since the resource utilization scales almost linearly with the throughput. A naive multi-way parallel design comes at a cost of degraded compression ratio due to the fact that compression opportunities in one engine may disappear as the matching records reside in another engine. To maintain a satisfactory compression ratio, we provide novel optimizations within each Deflate accelerator engine, including 1) a better data feeding method to reduce the loss of dictionary records, 2) a hash chain to expand the hash dictionary history, 3) a double-bank memory and dedicated multiplexers to increase the chance of history matching. By parallelizing up to four compression engines, we can compress up to 64 bytes of data per cycle with a fixed clock frequency of 200 MHz, which means a single FPGA-based compression accelerator can achieve a peak compression throughput of 12.8 GB/s.

We also explore the impact of CPU-FPGA communication bandwidth on system-level compression throughput. We wrap our FPGA accelerator with the CPU software invocation and abstract it as a software library on modern Intel-Altera HARP and HARPv2 platforms [Int15, Int16a]. With the significant increase of the CPU-

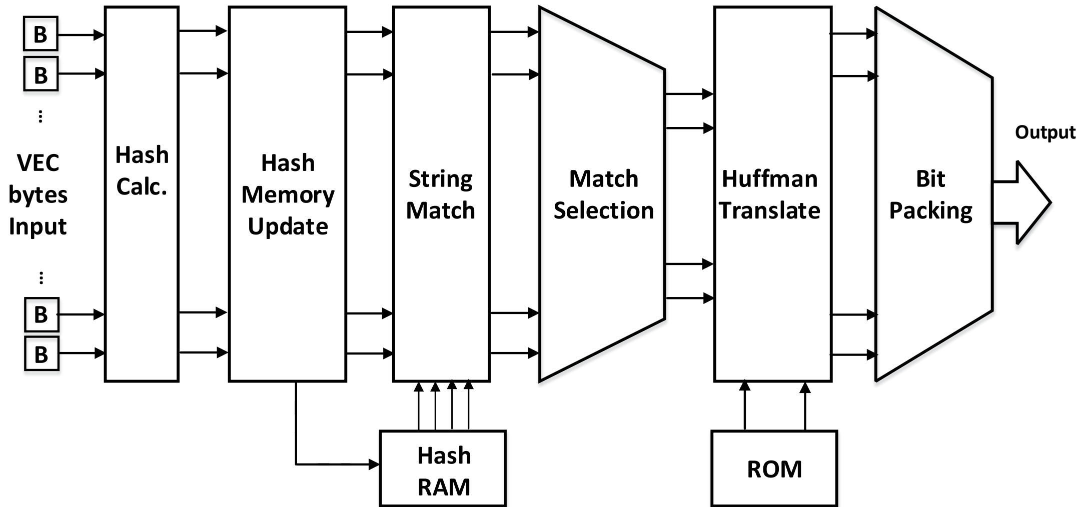

Figure 6.1: The architecture of a single-engine fully-pipelined Deflate compressor.

FPGA communication bandwidth, we see an average practical end-to-end compression throughput of 3.9 GB/s on HARP and 10.0 GB/s (up to more than 12 GB/s for large input files) on HARPv2. This shows that the proposed FPGA-based high-throughput lossless compressor is rather powerful in real-world applications.

6.1 Single-Engine Fully-Pipelined Compressor

In this section, we present the general architecture of the hardware based single Deflate engine, which is adopted in most of the existing works [AHS14, FKB15, LCH20], as well as ours. We also identify the key components which incur excessive resource utilization, degrade the design frequency and impact the compression quality.

Figure 6.1 shows the six major steps inside the single Deflate engine. Unlike the original algorithm that only processes one byte per time, the hardware-based Deflate

engine can process multiple bytes per cycle in a pipelined fashion. We denote the number of bytes that are streamed into the engine every cycle as *"VEC"* and the *VEC* bytes we are currently processing as the current window.

Stage 1: Hash calculation. In each cycle, we first extract all the sub strings of length *VEC*, starting from every byte in the current window, and index each sub string to its corresponding history memory through hashing. To support this, VEC-1 more bytes from the next window are always required for extracting sub strings starting from later bytes of the current window. For the example of VEC=4 in Figure 6.2, it extracts four sub strings (each with length 4) for the current window. These sub strings are then hashed and later matched with history data in parallel.

We use a multi-banked memory to store the history strings. The number of banks and entries in each bank can be set individually. Let us denote the history memory with a 2D array: mem[BANKS][OFFSETS], where BANKS is the total number of banks and OFFSETS is the number of entries in each bank. In this work, we choose a BANKS of 32 (the reason will be explained in Section 6.2.3) and OFFSET of 256. So the hash function should create a 13-bit hash value that can work with a single hash table of $32 \times 256 = 8K$ entries. Let us also denote each input string as a character array of length VEC: str[VEC]. For each substring, we calculate the hash value with the first few bytes of the string, and calculate the potential bank number and bank address offset from the hash value in the following way (here << is the left shift operator, % is the modulo operator and xor is the exclusive OR operator).

$$\begin{aligned} hash = (string[0] \ll 5) xor (string[1] \ll 4)(string[2] \ll 3) \\ xor (string[3] \ll 2) xor (string[4] \ll 1) xor (string[5]) \end{aligned} \tag{6.1}$$

$$banknumber = hash\%BANKS \tag{6.2}$$

$$bankoffest = hash/BANKS \tag{6.3}$$

Stage 2: Hash memory update. The hardware compares these VEC input sub strings to the records in the VEC hash history memory banks, and replaces the old records with these input strings. The hash history memory is a dual-ported RAM that enables one read and one write per clock cycle. To help reduce bank access conflict, [FKB15] suggested the hash memory runs at twice the clock rate of the rest of the system so each bank can handle two read requests in one system clock cycle. Figure 6.3 shows an example of bank conflict. Assuming strings "AABA," "ABAA," and "AABC" are all mapped to bank 2, in this case only the first two strings are allowed to access the bank. The switch part that connects each sub string to its corresponding memory bank will become the critical path when we scale VEC to a higher number. It will limit the clock frequency of the design, since each time it needs to reorder VEC inputs while each input is a string of VEC bytes. The situation

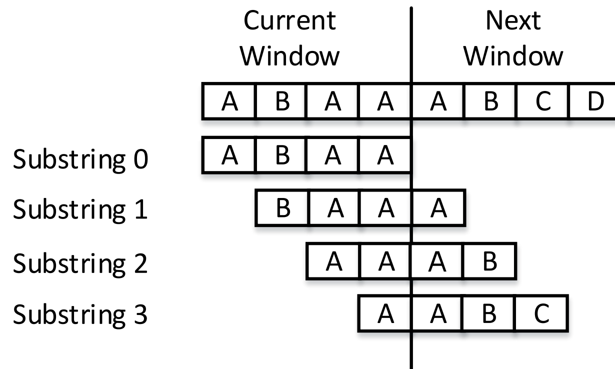

Figure 6.2: Current and next window of input string to be compressed.

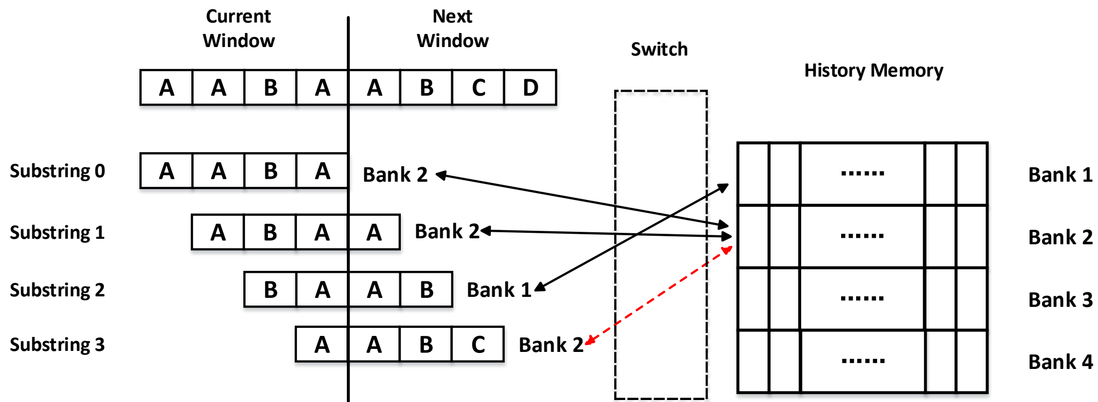

Figure 6.3: Example of hash conflicts in a VEC=4 design.

will be even worse when using a doubled-rate clock frequency since the switch also needs to be operated at the same clock frequency.

Stage 3: String match. The VEC records from hash history memory are matched with their corresponding input strings in parallel, and any match with match length smaller than 3 is rejected. Another switch is required to remap the VEC records to their counterparts, and it faces the same timing challenges as the one in Stage 2. According to the LZ77 algorithm [ZL77], an (L,D) pair is emitted for each input position. If there is a match, then L represents match length, and D represents the distance between the history record and the current input string. If the match is rejected, then L is the byte value and D is 0. A special case that needs to be addressed is: if match length L is larger than D, L needs to be truncated to D to avoid overlapping.

The switch that maps the read results of previous strings from the memory banks for the inputs can be very complex. A straightforward way is using a k-to-1 128 bit-width input multiplexer to select the record string from the k output ports of the

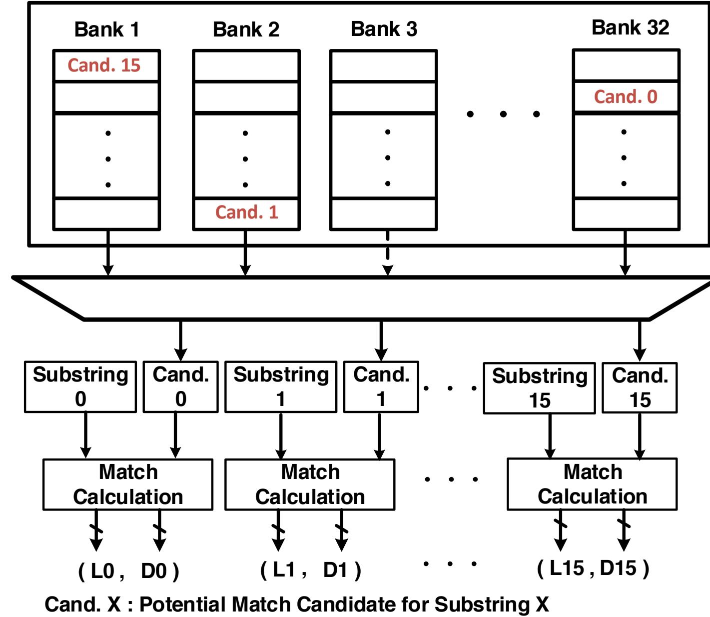

Figure 6.4: Memory bank mapping to input strings. Here we perform string matching after selecting the right candidate strings from the 128 bit-width multiplexers.

memory banks (assuming the hash table has k memory banks), and then compare it with the input string, as shown in Figure 6.4. This way ensures that for those strings which cannot access the memory (e.g., bank conflict, their mapped banks have been accessed) are still able to measure if they are matched with the previous strings. In the example of Figure 6.4, there are 16 input sub strings needed to compare with the candidates from the 32 banks. Correspondingly, 16 32-to-1 128 bit-width multiplexers are required, which consume the huge amount of resources.

Stage 4: Match selection. The matches in the current window could stretch to the next window and need to be pruned to avoid match overlapping between windows. This is implemented in two phases. First, in each cycle VEC matches

from each byte in the current window will be compared to select a best match that extends farthest. We define this match as "the tail" of the current window. When processing the next window, all the matches already covered by this tail must be skipped since we cannot change the match that is already committed in the last cycle. Similarly, we need to skip all the matches that are covered by the previous tail. We define "the tail reach" as the position reached by the tail: tail reach = tail index + match length, and we call the tail reach from a previous window "the head" of the current window. Then the bytes in-between the current best match and the best match extended from the previous window will be examined one by one to resolve the conflict between adjacent matches, this is referred to as lazy evaluation. This phase is a loop behavior and can be implemented as a series of VEC pipeline stages.

Figure 6.5 shows some match selection examples. The first case shows the condition of no overlap between the head, which actually is the tail reach from the previous window, and the tail in the current window. The second case shows that the potential tail in the current window overlaps with the head from the previous window. In that case, we need to trim the potential tail. The last three cases are the special scenarios where the trimmed tail match length is less than a minimum match length of 3 and needs to be considered as literal bytes. These cases drive us to decrease the minimum match length to 3 since that will identify a tail match length of 3 as a real match and improves the compression ratio. After committing the tail, we need to remove all the matches already covered by the head and the tail. For the matches starting between the head and the tail we need to (1) trim them so they do not cover the tail; (2) use lazy evaluation to resolve the conflict between adjacent matches: (i) if position i is a no-match, commit it and proceed to position i+1; (ii) if

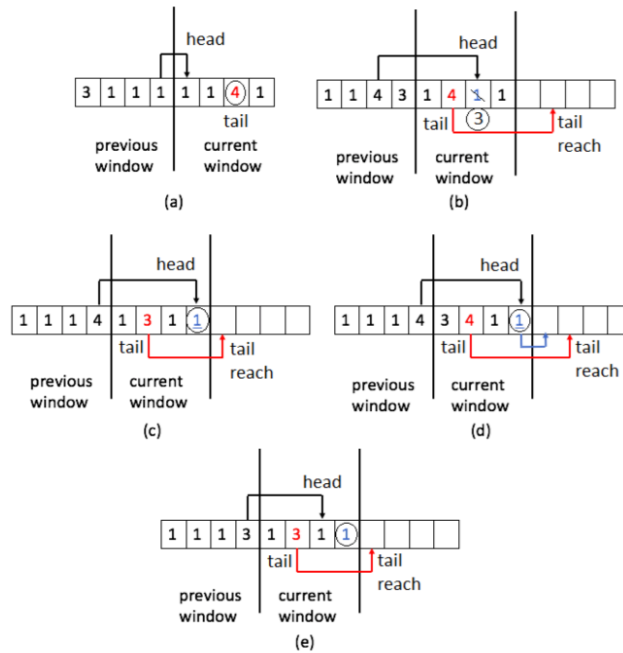

Figure 6.5: Match selection examples: The numbers in the box represent the match length from each position. Black arrows are head matches (tail matches extending from the previous window); red arrows are the potential tail matches, and the position with a circle is the actual tail match: (a) the head does not overlap with the current tail; (b) if the head is covering the start of the tail match, the tail must be trimmed to start from the head pointer's position; (c) after performing the trimming in (b) the length is 1, so use a no-match at the head pointer's location; (d) after performing the trimming in (b) the length is 2, which is not allowed by the Deflate algorithm, so a no-match length 1 is used instead, and tail reach is changed; (e) similar to (d). (d) and (e) are the only cases in which a trimmed match will have length 2 and needs special handling.

position i is a match, and position i+1 is not a match, commit it and proceed to next position after the match from position i; (iii) if position i is a match, and position i+1 is a match, and the match length L(i) is smaller than L(i+1), then the latter is a better match, and we have to commit position i as a no-match (literal).

Stage 5: Huffman translation. The following stage is Huffman encoding. The method of counting symbol frequency and using dynamic Huffman encoding no longer works because the Huffman packing pipeline must run simultaneously with LZ77 stages in a pipeline fashion. Therefore, static Huffman encoding is used in this design. Static Huffman encoding is nothing but a dictionary lookup, which is implemented as a ROM (ROM shown in Figure 6.1). The VEC (L,D) pairs can be looked up within VEC ROMs in parallel. After each (L,D) gets translated, we get a four-code tuple (Lcode, Dcode, Lextra, Dextra). *Lcode* and *Dcode* are the codes for literal and distance; *Lextra* and *Dextra* are the extra bits to encode literal and distances.

Stage 6: Bit packing. The last step involves packing the binary codes of different lengths together and aligning them to byte boundaries. This is because the length of the four-code tuple output ranges from 8 bits to 26 bits, while the data we finally stored are in byte format. Packing can be easily achieved by a series of shift-OR operations.

As shown in Figure 6.6, we first use VEC local packers to pack each (length, distance) pair in parallel. For each data window, the output ranges between 12 bits to 8*(VEC-1) + 27 bits. 12 bits corresponds to the case there is only one match starting from the first byte of the current window and the match length is equal to VEC. 8*(VEC-1) + 27 bits indicates that the VEC bytes in the windows are all

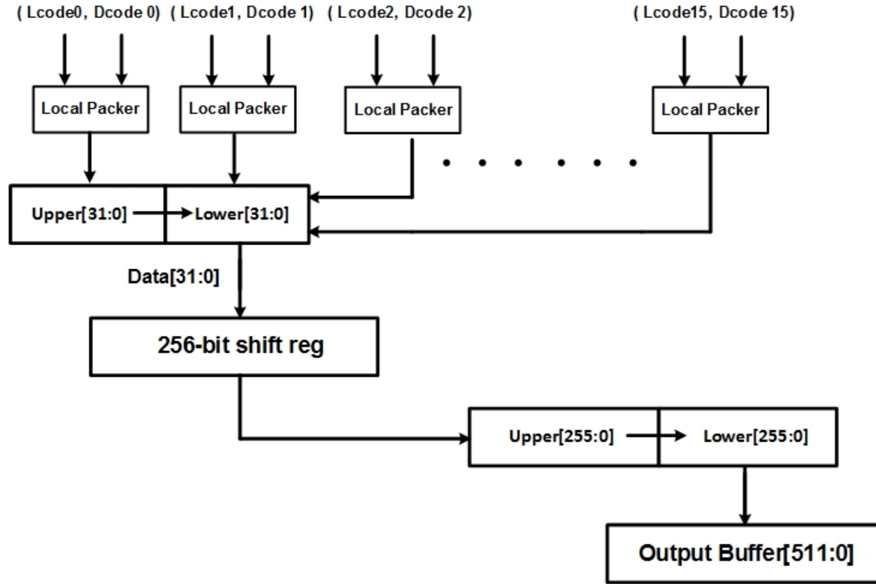

Figure 6.6: Illustration of the parallel bit packing.

no-match (literals). The upper bound is not a regular number and more than 256 bits for VEC=16, so we use a 64-bit barrel shifter to align the sequential output of local packer first and then use a 256-bit shift register to store the output data for each data window. The 256-bit shift register sends the output bits to another 512-bit barrel shifter to be aligned. When the 512-bit barrel shifter is half-fulled, it will send its lower 256 bits to the final 512-bit wide output buffer. We set the width of the output buffer to 512 bits to make full use of the communication bandwidth.

6.2 Optimization Associated with Scaling Engines

For a single-engine compressor, increasing the data window size VEC directly improve the compression throughput. However, as the data window size VEC increases from

16 to 32 bytes (per cycle), the resource usage on FPGAs is increased super-linearly by roughly 2.6×. This is mainly due to the overhead from the multiplexers shown in the previous section. On the other hand, the length of VEC represents the sub string lengths to be compared, and therefore the maximum match length. It seems the better maximum match length, the better the compression ratio. However, the study in [FKB15] shows no compression ratio improvement when increasing VEC from 24 to 32. This is because in most of the standard benchmarks there is no match whose length is larger than 24, or such a match cannot be detected in the proposed algorithm.

To further improve the compression throughput, we decide not to directly scale VEC, but exploit a multi-way parallel design, where each accelerator engine can compress a relatively small amount of data concurrently. For each individual compression engine, we stick to using a data window size of 16. In this way, the resource utilization scales linearly with the aggregate compression throughput and the critical path will not be further increased.

To compensate for the potential compression ratio loss in the proposed multi-way design and improve the overall efficiency, in this section, we will present several optimizations. First, we propose a better data feeding method to multiple engines to reduce the loss of dictionary records and thus improve the compression ratio. Second, we propose single engine optimizations, including hash chains to improve the hash dictionary length and compression ratio, double bank design and switch optimization to improve clock frequency and resource utilization.

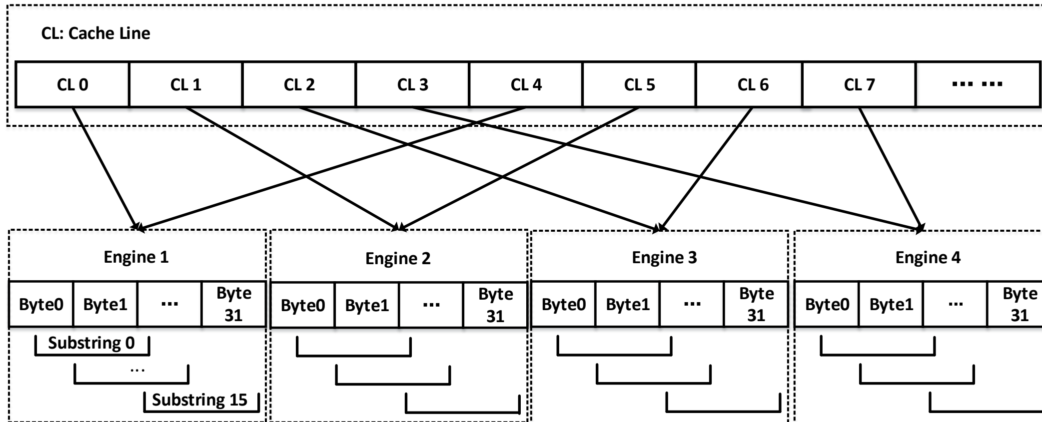

Figure 6.7: Cyclic data feeding.

6.2.1 Multi-Engine Data Feeding Method

For a multi-way parallel FPGA Deflate accelerator design, we need to divide the input file into multiple segments to feed each Deflate engine. There are two ways to fetch data for the parallel accelerator. The first is cyclic data feeding, shown in Figure 6.7. Each time it fetches multiple small consecutive blocks (e.g., VEC bytes of data or 64-byte cachelines) and feeds them to the four parallel engines. The second is block data feeding, shown in Figure 6.8; it is also employed by multi-core software compression solution such as pigz [GA22a]. It segments the entire input file into multiple large consecutive parts. Each time it fetches one (or multiple) blocks from one of the parts and feeds them into the corresponding compression engine.

Due to the fact that similar pieces of data are usually located in nearby regions in a file, the compression ratio (measured by input file size divided by output file size) of the block data feeding is much higher than that of cyclic data feeding. In fact, the block feeding method suits the Deflate algorithm perfectly to compress large

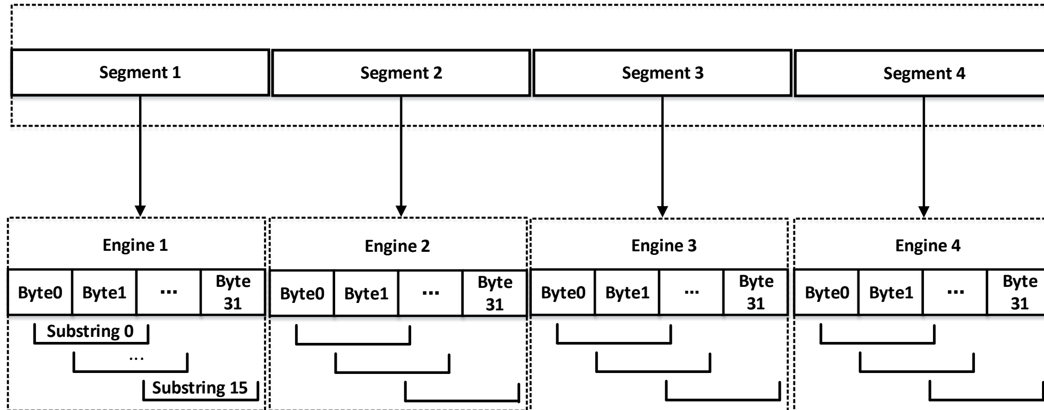

Figure 6.8: Block data feeding.

files since the Deflate format limits history distance to 32K bytes. For files much larger than 32K bytes, strings in the latter part will not be compared to the previous part with a distance longer than 32K bytes. Therefore, it makes minor impacts on compression ratio to segment large files for block data feeding, because only the compression of data strings in block boundaries might be affected. The experimental results that support this can be found in Table 6.2. Compared to cyclic data feeding, the degradation on compression ratio due to block segmentation is negligible. On the other hand, cyclic data feeding offers better streaming capability. However, as will be shown in Figure 6.15, block data feeding can work as well for streaming applications by processing a collection of streaming data each time. Considering these, we will use the block data feeding as our default data feeding method.

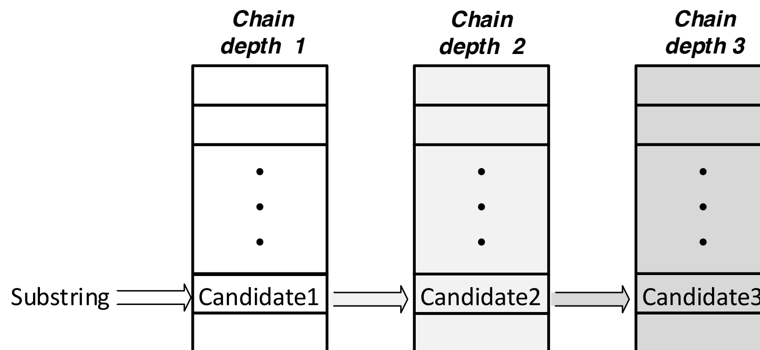

Figure 6.9: Hash memory chain for each hash history.

6.2.2 Hash Chain

Even the drop of the compression ratio may be negligible for the block data feeding method, we may still want to compensate for the potential compression ratio loss in a single engine. Therefore, we increase the history dictionary size to find a better match. To achieve this, a hash memory chain, as shown in Figure 6.9, is implemented in the hash memory update stage. It is like a shift register, but in more of a register file format. Every cycle different depths of the chained memory all return a candidate string as the read output, and the current depth's read output is the input written into its next depth's memory. The sub string in the current window will be stored into the memory in the first depth of the chain. In the example, candidates 1, 2 and 3 are the output of each memory at different depths of the memory chain, and all of them will be compared to the sub string to find the best match. After that, the current sub string will be stored into chain depth 1, candidate 1 will be stored into chain depth 2, and so on.

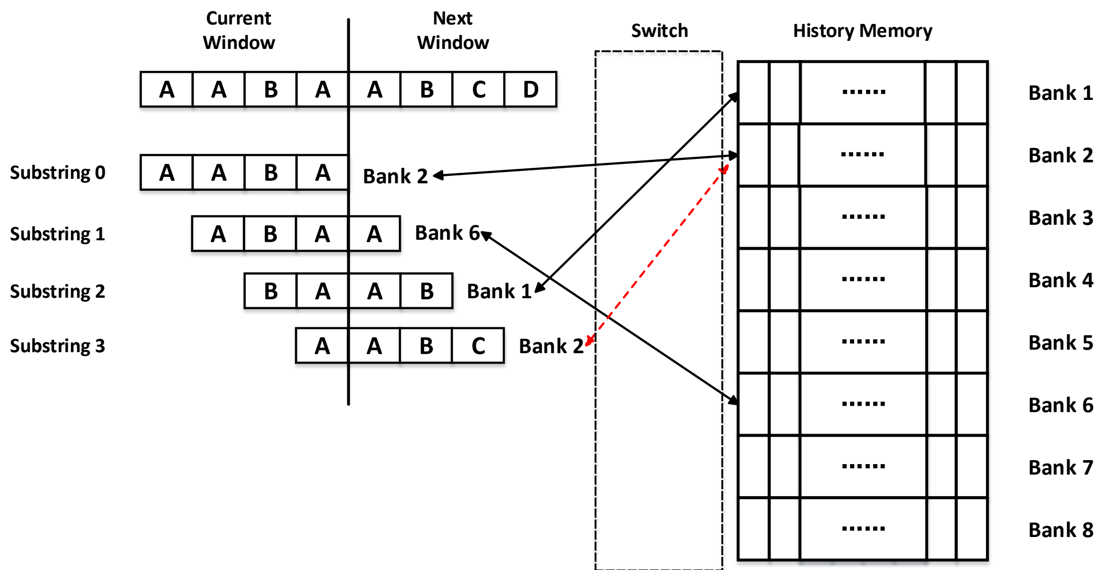

Figure 6.10: Reduce hash conflicts by doubling the hash banks.

6.2.3 Increase Bank Number

To reduce the compression ratio drop caused by bank conflicts, the example in 6.3 uses a second clock whose frequency is twice the global clock to read and update the 16 (VEC) hash memory banks (dictionaries). The doubled clock frequency of the memory part will become the bottleneck of the whole design and significantly limits the system performance, especially when we integrate the idea of hash memory chain. We propose an alternative design to use a single clock while doubling the number of banks to 32 (2*VEC), as shown in Figure 6.10. As a result, we increase the length of the hash dictionary. This approach avoids the doubled clock frequency bottleneck and enables us to integrate more depth of hash chain.

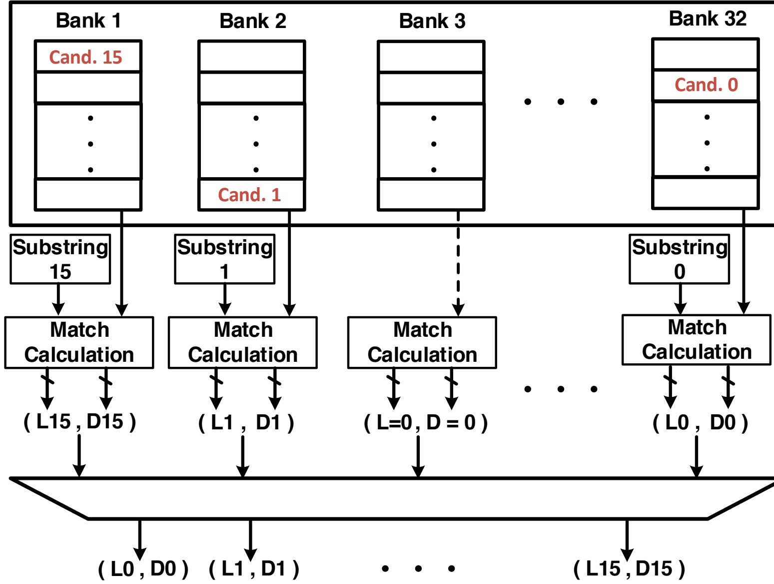

Figure 6.11: Optimized memory bank mapping to reduce resource utilization, here we use 5-to-1 multiplexers after string matching. Cand. X is short for the potential match candidate for Substring X.

6.2.4 Switch Optimization

Since the straightforward implementation of the switches in Figure 6.4 incur significant resource overhead, we propose to leave those strings that cannot access the hash memory as mismatched literals and just compare output port data of memory with the data delayed by 1 cycle from its input port, as shown in Figure 6.11. If the bank has not been accessed, (e.g., Banks 3), then the match calculation unit will generate a pair of (L=0, D=0) to indicate the case. The compared result is simply a 5-bit length data, which is much shorter than the original 128-bit data string. So we can use 5 bit-width multiplexers to select the corresponding hash match results for each

input string. This approach avoids the overhead of multiplexers so as to improve the timing and reduce the resource consumption. The compression ratio only degrades by 3% since we have already mapped the 16 input strings to 32 memory banks to avoid the hash conflicts as much as possible. Notably, this way may also eliminate the potential timing bottleneck when we want to further scale up VEC. For example, when we scale to process 32 bytes per cycle, we only need to use 32 6-to-1 multiplexers, which introduces much less overhead and does no harm to the clock frequency of the design compared with the original 32 256-to-1 multiplexers that are used to select the 256-bit strings.

6.3 Evaluation

In this section, we evaluate the performance of the proposed multi-way Deflate accelerator on the Intel-Altera HARPv2 platform. We first present the CPU-FPGA communication flow of the proposed multi-way accelerator architecture on the tightly coupled HARPv2 CPU-FPGA platform. Next, we compare the performance of the proposed design and previous designs. Finally, we report the end-to-end performance of the proposed accelerator.

6.3.1 Experimental Setup

We implement our FPGA Deflate accelerators on the HARPv2 [Int16a] platform, which integrates a 14-core Broadwell EP CPU and an Altera Arria 10 GX1150 FPGA. We use the Calgary Corpus [Cor01] datasets as the benchmarks to measure the average compression throughput and compression ratio. To explore how CPU-FPGA communication bandwidth may limit end-to-end throughput and avoid

FPGA hardware generation gap, we also test the designs on the HARP platform as a comparison, which integrates an Intel Xeon E5-26xx v2 processor and an Altera Stratix V GX FPGA. Compared to HARPv2, HARP still has a very limited communication bandwidth and it uses the same Stratix V FPGA as previous studies [FKB15, AHS14].

6.3.2 System Integration

A system-level overview of our multi-way accelerator design on the HARPv2 platform is shown in Figure 6.12. The FPGA is connected to the processor through one QPI channel and two PCIe channels. The accelerator can directly read/write from/to system virtual memory through the CCI-P interface. The arbiters are used to ensure each Deflate core have each access to the communication interface.

Figure 6.13 presents the flow of compression execution. At initialization, the CPU first allocates memory workspace which is shared by the CPU and the FPGA. This includes device status memory (DSM) workspace to keep track of the status of the FPGA accelerator, a source buffer for storing the source data, and a destination buffer to store the processed output data. After loading the source file into the source buffer, the CPU segments the source file equally into N blocks (four in our implementation) and then writes the segmented base addresses and block sizes to the FPGA. The initial setting being done, the CPU enters a wait state and polls the status bit in DSM until the FPGA sends an ending signal. On the other side, the FPGA accelerator begins execution after loading the addresses and source data block sizes. Each engine is allowed an equal share of time to send read requests for source data from the shared memory using a round-robin arbiter. Since read responses are

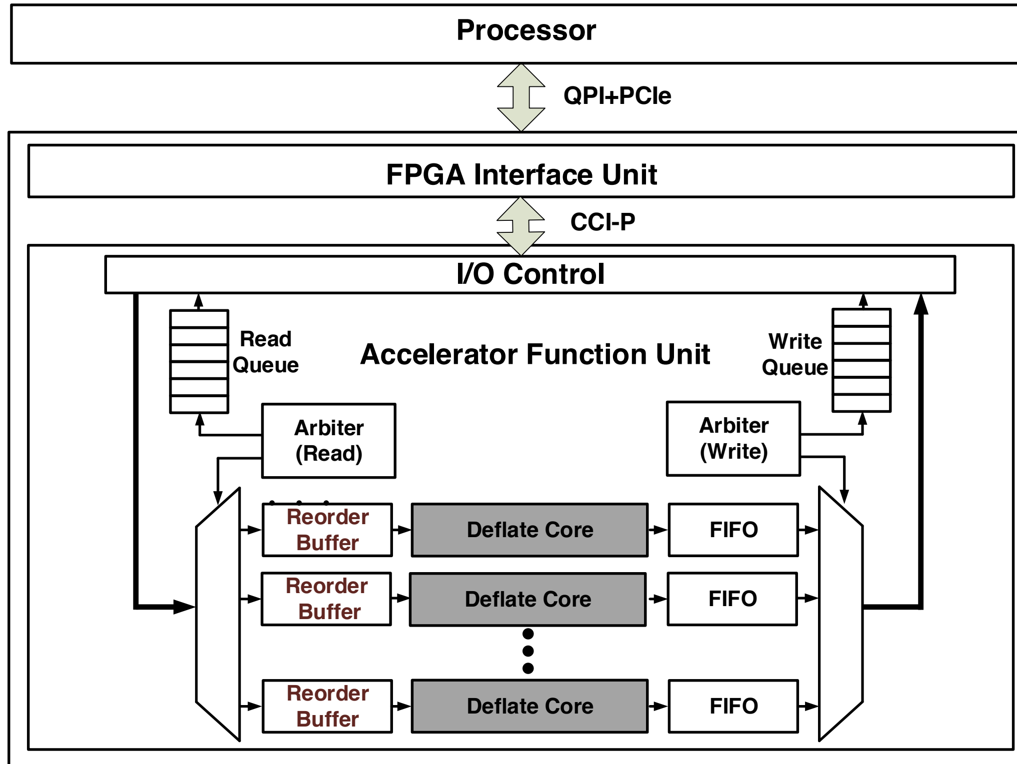

Figure 6.12: Overview of the multi-way parallel and fully-pipelined Deflate accelerator implementation.

out of order in the HARPv2 platform, re-order buffers are used to restore source data order before compressing. The processed results are kept temporarily in first-in-first-out (FIFO) buffers before writing back to the destination buffer in the shared memory. Another arbiter is used to control write access among the Deflate engines. After all the source data are processed, the FPGA will signal the CPU that all work has been done.

In HARPv2, the data transfer size for each read/write request can be 1,2 or 4

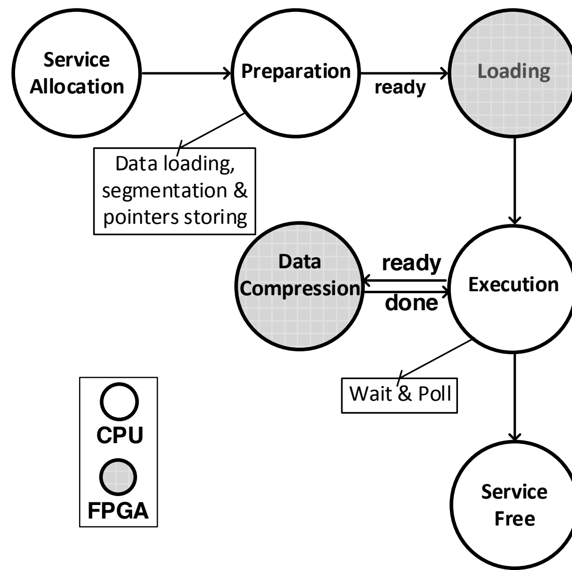

Figure 6.13: Flow chart of CPU-FPGA system execution.

consecutive cache lines (64 bytes per cache line); and for maximum CPU-FPGA data bandwidth, our design uses 4 cache lines per request. The three physical links (one QPI and two PCIes) are configured to one virtual channel in our accelerator design, and the FIU will optimize the communication bandwidth by steering requests among the three physical links.

6.3.3 Comparison of Deflate Accelerators

We first compare our accelerator design to previous studies in Table 6.1, where the efficiency is defined as the compressor's throughput in MB/s per kilo ALMs resources it consumes. By using the multi-way parallel and fully-pipelined accelerator architecture where each way features a single clock, 32 banks and a memory chain depth of 3 designs that can process 16 bytes per cycle, we can compress 64 bytes/cycle

Table 6.1: Comparison of FPGA Deflate accelerators.

Design	Frequency	Throughput	Compression ratio	Area(ALMs)	Efficiency
[MJA13]	250 MHz	4 GB/s	2.17	110,000	36
[AHS14]	193 MHz	2.84 GB/s	2.17	123,000	23
[FKB15]	175 MHz	5.6 GB/s	2.09	108,350	52
Our design on HARP	200 MHz	9.6 GB/s	2.05	134,664	71.3
Our design on HARPv2	200 MHz	12.8 GB/s	2.03	162,828	78.6

(4-way parallel) at a clock frequency of 200 MHz. This is more scalable than merely scaling up the data window size of a single Deflate engine, since the area increases non-linearly with the data window size, and the maximum frequency the engine can achieve drops as a result of routing problems. Our FPGA Deflate accelerator achieves a record compression throughput of 12.8 GB/s, which is 2.2x faster than the prior record in [FKB15]. Our design on the inferior Stratix V FPGA of the HARP platform still achieves a compression throughput of 9.6 GB/s, which is at least 1.7x faster and 1.4x more efficient in performance per area over previous designs. In addition, our design is also much more resource-efficient in terms of compressed MB/s per kilo ALMs, which is 1.5x efficient than the prior record in [FKB15]. Please also note we measure our designs at a fixed clock frequency of 200 MHz, since the platform provides the fixed clock. In fact, due to our optimizations, our FPGA kernel can work at an even higher clock frequency up to 250 MHz.

The compression ratio drops by 4.7% compared to Microsoft's work [FKB15], but it is still acceptable to most applications. This drop is due to the fact we divide the input file into four parts to feed into four different compression engines. If the data in one part should be matched better with the data in another part, then in our case the compression ratio drops.

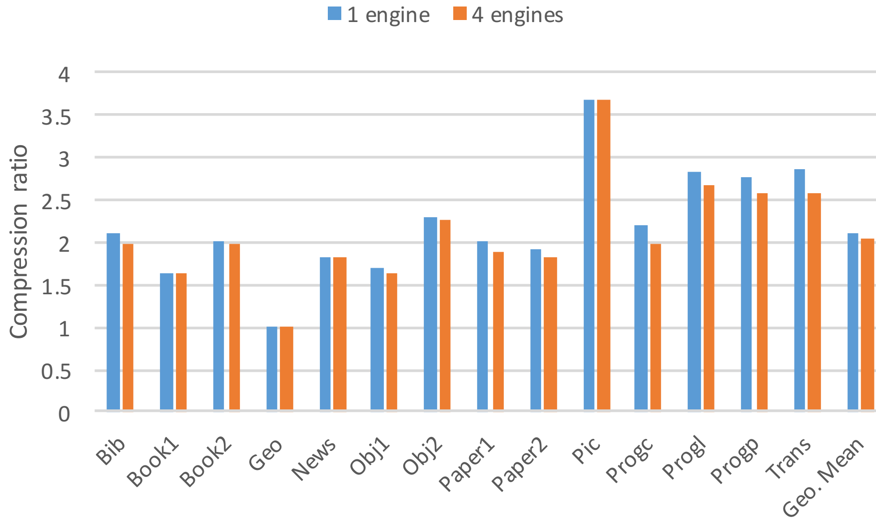

Figure 6.14: Compression ratio when scaling from 1 engine to 4 engines.

6.3.4 Scaling Effect

Figure 6.14 shows the corresponding compression ratio of Calgary Corpus benchmarks as we change the parallel engine number from 1 to 4. The single engine (VEC=16) achieves an average compression ratio of 2.11. When we increase the throughput to 12.8 GB/s with 4 engines, the average compression ratio drops by 3.7%. Note that the compression ratio of those large files (e.g., book1, book2, news and pic) only degrades by less than 1%. This result proves the analysis we present that the multi-way parallel Deflate compressor perfectly suits the applications where large files need to be compressed.

Table 6.2 also lists the area usage and compression ratio for scaling up engines, and the total area usage increases roughly linear. Considering that the area when we directly change the data window size of a single compression engine from 16 to

Table 6.2: Scaling effect on the performance, compression ratio and resources.

Parallel Engine No	Throughput	Compression ratio	Area (ALMs)
1	3.2 GB/s	2.11	38,297
2	6.4 GB/s	2.07	78,604
3	9.6 GB/s	2.05	118,891
4	12.8 GB/s	2.03	162,828

32 bytes will increase the resource utilization by 2.6x, it is also difficult to keep the frequency to a high value. This becomes the bottleneck for scaling in some cases where the system needs to run at a fixed clock frequency. For example, on HARP the system needs to run at 200 MHz. Exploiting the parallel engines can avoid this scaling problem as each engine is designed and placed separately.

6.3.5 Design Tradeoff Evaluation

We first compare the impact of different hash chain depths on a single Deflate engine in Table 6.3. Increasing one depth only augments 3,500 more ALMs, thanks to the multiplexer optimization, and there is more than a 9% improvement on the compression ratio, increasing depth from 1 to 3. Note that further increasing memory chain depth will not benefit the system performance much, and the compression ratio gain becomes marginal (less than 1.5% improvement measured). In fact, the dictionary strings read from each depth need to be compared with the same input sub string simultaneously, so matching comparison of the input string and the output dictionary string from the deepest memory bank will become the critical path. As a result, it is reasonable choice to set the chain depth to be 3.

Table 6.3: Impact of changing hash chain depth

Depth	Compression ratio	Area(ALMs)
1	1.92	31,360
2	2.04	34,761
3	2.10	38,297

Table 6.4: Double Clocks v.s. Double Banks

Design	Frequency	Throughput	Compression ratio	Area(ALMs)	Efficiency
Double clock design	150 MHz	9.6 GB/s	2.10	115,436	83
Double bank design	200 MHz	12.8 GB/s	2.03	162,828	78.6

We also evaluate the design choices in reducing hash conflicts by implementing two designs. The first design is what we have used in the multi-way accelerator, which uses a single clock and doubles the number of hash memory banks. The second one is a double clock design where the hash memory uses a clock frequency that is twice the global clock but the same number of bank numbers is equal to VEC (i.e., 16 banks). Table 6.4 summarizes the two designs. For the double clock design, since BRAM blocks are placed as arrays and the interconnect between two adjacent arrays will consume more time, the memory chain which occupies more than one single array can only run at a frequency of 340 MHz under the fastest configuration, thus limiting the system performance. Since HARPv2 provides another clock group of 150/300 MHz, we take 150 MHz as the system clock for the design and achieve the equivalent throughput of 9.6 GB/s, which is only 75% of the performance of the single clock design with double banks.

However, the area efficiency of the double clock design is slightly better than

the double bank design since it reduces the use of memory banks and corresponding multiplexers. Another interesting benefit of the double clock design is that it gives a slightly better compression ratio. This is because raising clock frequency while reducing memory banks enables similar strings to have more chances at being mapped to the same bank.

6.3.6 End-to-End Compression Throughput

We measure the end-to-end compression throughput by inserting a counter on the FPGA side. The counter counts the total clock cycles from when FPGA sends the first request to read the memory to the time it sets the data valid bit in the memory—which indicates the FPGA work is done and all of the data has been written back to the memory. Note that on HARP and HARPv2, the CPU and FPGA share the same DRAM so that there is no additional memory copy overhead on conventional PCIe-based platforms. Denote the total clock cycles as t_{cycle} and the total amount of data fed to the compression engine as A_{data} bytes, then the end-to-end compression throughput can be measured as A_{data} /t_{cycle}.

Table 6.5: End-to-end compression throughput on HARP and HARPv2

Platform	FPGA Throughput	End-to-end Throughput
HARP	9.6 GB/s	3.9 GB/s
HARPv2	12.8 GB/s	10.0 GB/s

The end-to-end overall compression throughput results are shown in Table 6.5. To show the overhead of CPU-FPGA communication, we first test our design on HARP, where the QPI bandwidth is limited (7 GB/s for read and 4.9 GB/s for

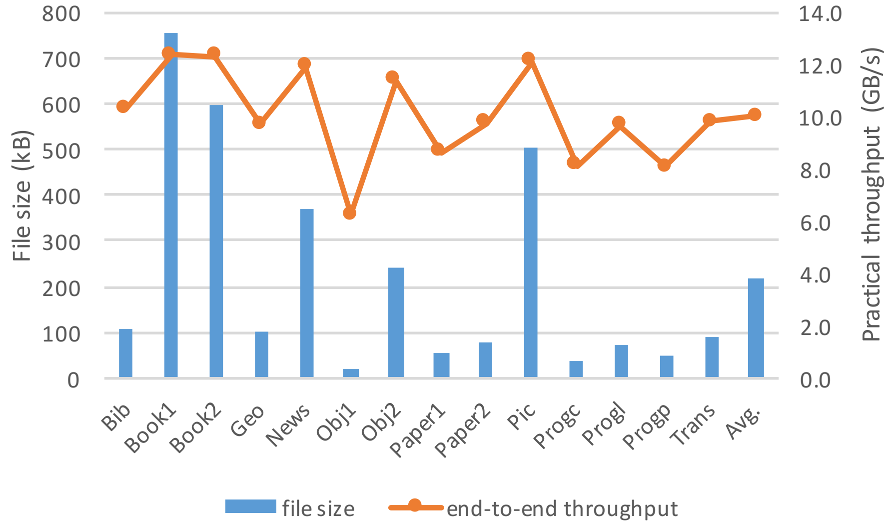

Figure 6.15: Compression throughput for different files of Calgary Corpus datasets.

write). We integrate 3-way parallel engines on HARP since the Stratix V FPGA module on HARP provides less resources and cannot accommodate more engines. The FPGA kernel throughput is 9.6 GB/s, while we can only achieve an average of 3.9 GB/s practical throughput based on the Calgary Corpus benchmarks.

On the other hand, even considering the read and write latency, we can see that HARPv2 can achieve an average end-to-end compression throughput of 10.0 GB/s since it has over 15 GB/s CPU-FPGA communication bandwidth. We also list each benchmark's size and its corresponding practical throughput in Figure 6.15. If the file size is larger than 500 kB (e.g., book1, book2 and pic), then the overhead of communication latency will be negligible and the end-to-end throughput can be up to 12 GB/s.

6.4 Summary

This chapter presents an FPGA Deflate accelerator that can be easily scaled to achieve a record compression throughput of 12.8 GB/s while maintaining comparative compression ratio to existing high-throughput Deflate accelerators. Compared to directly scaling the number of bytes processed per cycle, the proposed accelerator focuses on leveraging the task-level parallelism and utilizes multiple Deflate engines to compress the same task. We further equip this multi-way parallel compression accelerator with optimized methods for efficiently feeding data into the parallel compression engines, improving the resource utilization, augmenting compression ratio, and improving the clock frequency of the single fully-pipelined compression engine. The proposed accelerator is at least 2.2× faster and 1.5× more area-efficient than prior works and can fully leverage the host-FPGA communication bandwidth to deliver efficient end-to-end speedup.

CHAPTER 7

FPGA-Based BWT Acceleration

The Burrows-Wheeler Transform (BWT) has played an important role in high-quality lossless compression algorithms. To achieve a good compression ratio, the BWT window size needs to be several hundreds of kilobytes, which requires a large amount of on-chip computing and memory resources. For instance, the direct implementation of the rotation and sorting based BWT algorithm in Chapter 2.1 given a window size N consumes $O(N^2)$ memory space. Unlike the high-throughput Deflate compression kernel, implementing BWT with huge window size on hardware inherently poses great challenges and it is unknown to the community whether there exists such a hardware solution that can completely work with this problem. First, the on-chip sorting network that compares any two strings in the rotation matrix cannot be large enough to compare strings of hundreds of kilobytes. It is also unclear how to reuse the small-size sorting network to completely compare the two long strings. Second, the on-chip memory requirement is well beyond the available on-chip memory resources. As a result, existing FPGA designs can merely perform the BWT string comparison on a window size of 4 KB and no complete BWT implementation is reported [ZLW17].

In this paper, we present the first FPGA-based hardware design of the BWT algorithm that can perform BWT with a window size of hundreds of kilobytes. The

implementation is based on the uncommonly used antisequential suffix sorting algorithm [BB05], which is originally proposed to reduce the worst-case complexity issue (highly repetitive symbols) of the ternary-split quicksort method [BS97, Lar99, Sew00]. In most cases (less repetitive symbols), it is slower than the ternary-split quicksort method on CPUs due to the high cache miss rate. However, we find FPGAs are good candidates to implement such an algorithm since there are a large amount of distributed on-chip memories with low access latency. We implement two acceleration architectures—direct suffix list design and two-level suffix list design with several optimization techniques to reduce the searching time during the process. Our designs can support up to 500KB window size on modern FPGAs and achieve an average speedup of 2x over the BWT kernel in the high-quality compression software [Sew19] for standard large Corpus benchmarks.

7.1 Algorithm Selection

It is very expensive to directly building the rotation matrix in Chapter 2.1 for an n-byte string, as the required space will be $O(n^2)$. Apparently, such rotation and sorting based BWT algorithm is not a scalable approach to support BWT window sizes of 100s KB on current FPGAs. In this section, we compare two BWT algorithms that could be possibly mapped onto FPGAs and discuss if the implementation can be scaled to meet the high-quality compression standard.

7.1.1 Suffix Sorting

The suffix sorting approach can avoid the excessive memory consumption and it works as follows: each rotation can be denoted using an index called $suffix$, which

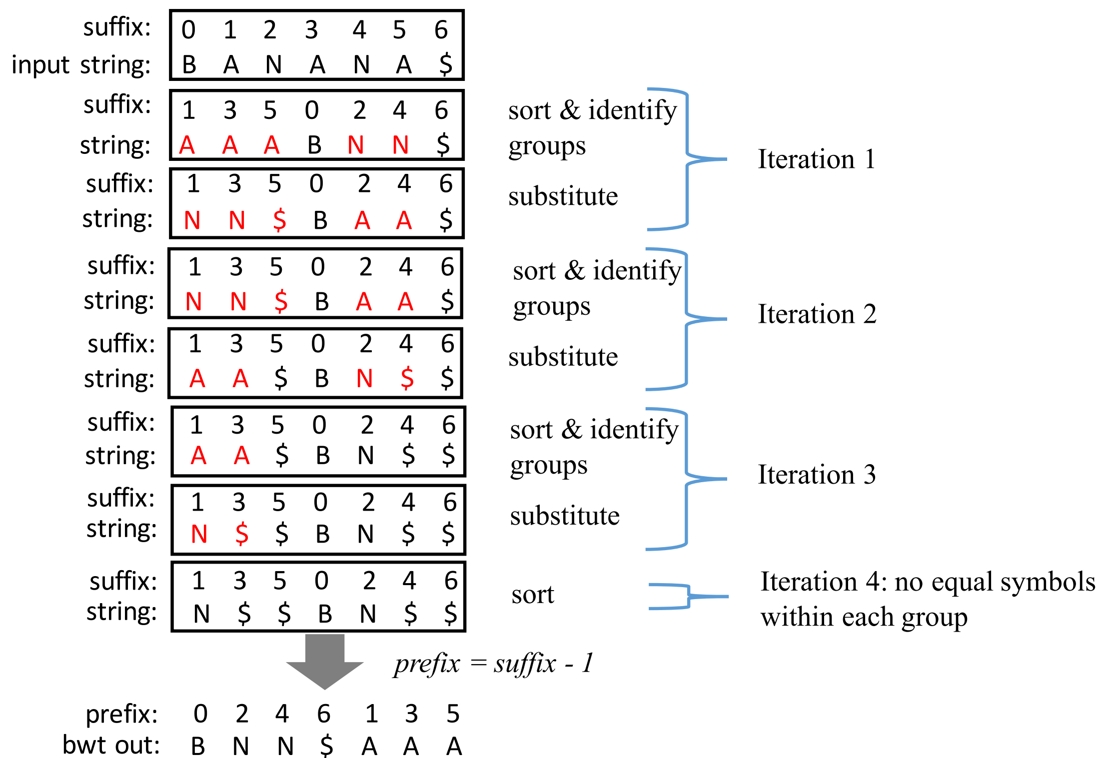

Figure 7.1: Example of the BWT using suffix sort.

is the position of its first byte in the original string, and then the corresponding last byte for that rotation is at position $suffix - 1$. BWT then can be performed by rearranging the suffix based on the relative order of each rotation. As shown in Figure 7.1, the string (b, a, n, a, n, a, \$) can be denoted using $(suffix, symbol)$ pair: ((0, b), (1, a), (2, n), (3, a), (4, n), (5, a), (6, \$)). In the first iteration, we sort the symbols lexicographically without modifying the index assigned to them and identify groups of equal symbols. For those pairs which have different symbols, their relative order is fixed now. Then we determine the order of pairs that have the same symbol within each group. To do so, we substitute those equal symbol with the symbols

next to them at position $suffix + 1$; note that the $ symbol is a sentinel symbol and larger than any other symbols. In the second iteration, we perform a similar sort and substitute for each group of equal symbol identified from the first iteration. These iterations are repeated until there are no equal symbols in a single group (i.e., no equal consecutive symbols) any more. Then we can get the BWT output indexes using $suffix - 1$. In this method, the iteration times are determined by the longest length of common symbols.

Existing hardware implementations [MCF05, CK13, PMM16, ZLW17] all choose the suffix sorting as their core algorithm and focus on accelerating the sorting part based on the parallel sorting networks. However, such a sorting network has poor scalability in terms of on-chip resources. Given a BWT window size N, it takes $O(N)$ stages for such a sorting network to process N characters, making the total total number of the compare and swap elements $O(N^2)$. Considering the high-quality compression standard requires performing BWT on a window size of 100s of KB, no existing FPGAs can afford the such high on-chip resource usage, let alone the associated routing issues. Besides, the sorting network can only do one iteration of the sorting. To achieve a complete BWT, the string needs to be stored and fed into the sorting networks multiple times until no equal consecutive characters appear any more. It is easy to infer that the entire process is still time-consuming especially in the case where there are many repetitive symbols in the input file that leads to many iterations.

7.1.2 Antisequential Suffix Sorting

The antisequential suffix sorting algorithm was originally proposed in [BB05], aiming to reduce the worst case complexity of the ternary-split quicksort approach [Sew00]. It is 4× slower than the ternary-split quicksort approach on average and thus is usually neglected by the hardware researchers in the past. It works as below: let's denote the sequence of $x_i x_{i+1} ... x_j$ as x_i^j. Given a string $x_1 x_2 .. x_N$ and suffix i, we define the suffix string of suffix i as $s_i = x_i^N$ for $1 \leq i \leq N$. Since the last byte of each suffix string is x_N, which is the sentinel symbol \$ and is larger than any input symbol, each suffix string is unique. In other words, sorting each rotation is equivalent to sorting its suffix string.

Now assume input symbols come in a reverse order from x_N to x_1, we can form a sorted list of previously processed suffix strings. When a new symbol x_i comes in, we only need to insert its suffix i into the right position of previously sorted suffix list, based on the relative order of its suffix string $s_i = x_i s_{i+1}$. Consider s_j such that $j > i$, suffix j is already in the sort list. We define a bucket S_i:

$$S_i = \{s_j : j > i, x_j = x_i, s_{j+1} > s_{i+1}\} \tag{7.1}$$

If S_i is not empty, then its smallest element is the right suffix j in front of whom the input suffix i should be inserted. It can be easily found since the position of suffix $i+1$ in the list is already known and we can scan down the list from suffix $i+1$ to check if such suffix j whose corresponding $x_{j-1} = x_i$ exists. If it exists, we insert the suffix i before $j-1$. An example illustrating such process is shown in Figure 7.2.

If no such suffix j that $x_{j-1} = x_i$ found, that means either the input symbol x_i never appeared before (equivalently bucket S_i is empty) or its suffix string s_i is the

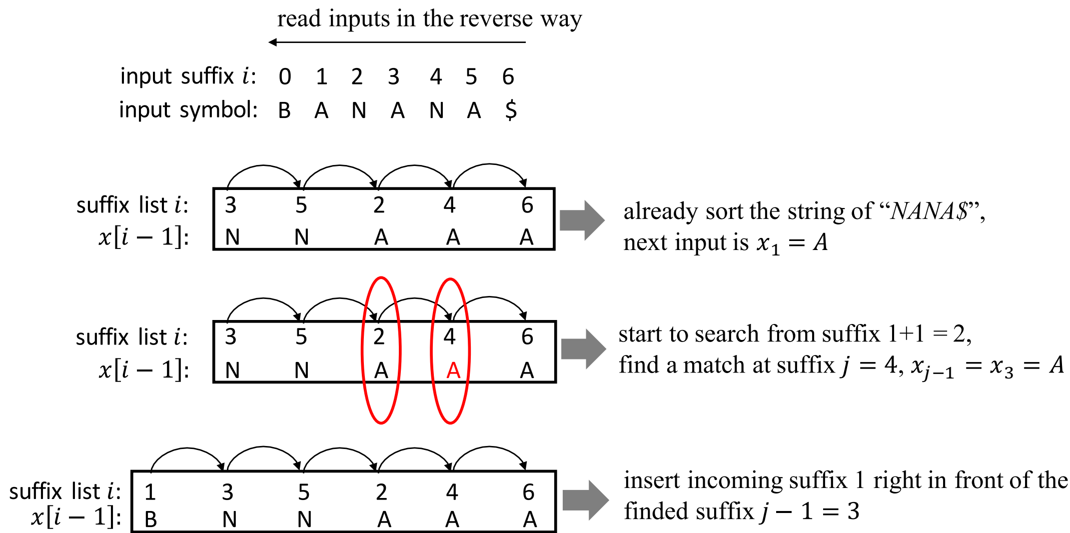

Figure 7.2: Example of the BWT using antisequential suffix sort.

largest suffix string that starts with symbol x_i. To deal with the two special cases, a dictionary table is established. The table records if each type of symbol appeared before and if appeared, the largest suffix whose suffix string starts with that symbol. In the first case, we find the largest symbol b that is lexicographically smaller than x_i and insert suffix i right after the corresponding largest suffix for symbol b. In the second case, since x_i already appeared, we insert suffix i right after the previously largest suffix starting with x_i. In both cases we also need to update the dictionary table with the latest suffix i for symbol x_i.

The antisequential suffix sorting algorithm not only greatly reduces the required on-chip memory amount, but also avoids the exhaustive string comparison. Given a BWT window size N, it takes $O(N)$ on-chip memory space, reuses the same control logic and consumes at most $O(N^2)$ time. As a result, it makes the implementation of BWT with large window sizes (i.e., 100s KB) possible on an FPGA.

7.2 Antisequential Suffix Sorting in Hardware

In this section, we discuss the implementation of antisequential suffix sorting in hardware. First, we present a straightforward design of the antisequential suffix sorting algorithm and discuss its design trade-offs. Then we introduce a two-level suffix list approach to accelerate the antisequential suffix sorting algorithm by reducing the suffix list search time.

7.2.1 Direct Suffix List Design

The direct implementation of the antisequential suffix sorting algorithm is quite straightforward, which contains dictionary table, a suffix list and the corresponding control logic. Figure 7.3 shows the architecture of the direct suffix list design. At each cycle one symbol will be fed into the design. The input symbol will be first compared with the entries in the dictionary table to find if the same symbol appeared before, and the finding results will be sent to the controller. If no such symbol appeared before, the controller will insert the input suffix after the previously largest suffix and update the dictionary table. If the input symbol has appeared, the controller will start searching the suffix list until it finds a match or reaches the tail of the list. After that the input suffix will be inserted into the right position and the dictionary table will get updated correspondingly.

Although the overall procedure is quite sequential, we can exploit some inherent parallelism and design optimizations to increase the performance of the design as below. In this design, we consider three major optimizations: (1) referring to the dictionary table should be as fast as possible; (2) the organization of suffix list and its searching operation should be efficient; (3) the tasks of dictionary reference, flow

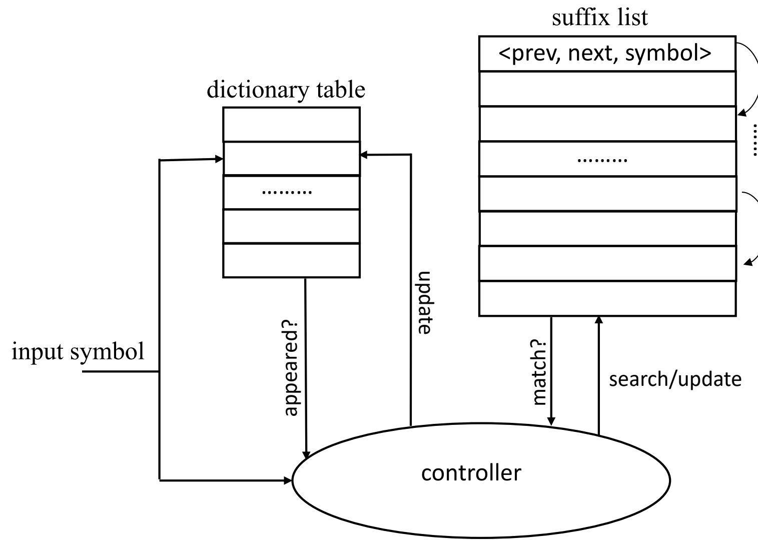

Figure 7.3: High-level diagram for direct suffix list design.

control and suffix list searching should be overlapped.

Dictionary table: The dictionary table can be implemented as a simple hash ram of 128 entries (since regular text files have 128 ASCII codes) and be accessed through the 8-bit input symbol value. This configuration works well when the input symbol already exists in the table. However, when the input symbol never appeared, we need to search through the whole table to find the largest symbol. For example, consider the case where the first two symbols coming in are "A" and "z", for input "z", it takes 57 cycles to find the largest existing symbol (57 is the difference of the two symbols' ascii value). To speedup this process, the dictionary table is optimized as a ram of 32 entries and each entry contains 4 consecutive symbols' index information in parallel. Reading 1 entry can get the largest appeared symbol index in the 4-symbol entry in 1 cycle through encoding and a priority multiplexer. Thus, in the previous example the total cycle to find the largest existing symbol is reduced to $\lceil 57/4 \rceil = 15$.

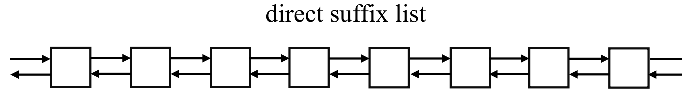

Figure 7.4: Illustration of the direct suffix list.

Suffix: The suffix list is also implemented as a ram whose depth is rounded to the BWT window size, an illustration of which is shown in Figure 7.4, The content of the suffix list entry is $< prev, next, symbol >$. For a window size of 100kB design, each entry takes 42 bits, since the *prev* and *next* pointers take 17 bits each and another 8 bits are used to store the corresponding symbol x_{i-1}. The BRAM is configured as a dual-port ram and each cycle one read and one write operations are allowed to perform. There is no read output register so read latency is only 1 cycle. To access the next element of current node, we only need to read the content of current node and use its 17-bit *next* part as the next read address. In general, it takes 1 cycle to search the next element and 2-3 cycles to insert a new element depending on if the node to be inserted is a head or not.

Tasks overlap: Since it takes only 1 cycle to update the dictionary table and 3 cycles to insert a new entry into the suffix list, these two parts can be overlapped after searching the list. We also overlap the dictionary table references between even and odd input symbols. The only sequential part left is the suffix list searching.

In summary, we overlap the tasks to make sure there is no bubble in the process and reduce the cycles of dictionary reference. The performance of the direct suffix list approach is thus determined by the total search depth of the window, which depends on input file characteristics, and the maximum frequency the design can run.

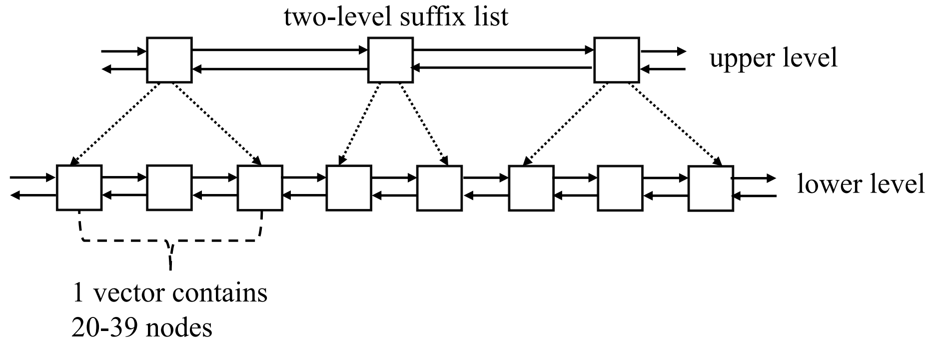

Figure 7.5: Accelerating the search time by using two levels of list.

7.2.2 Two-Level Suffix List Design

In the direct suffix list design, the performance is determined by the total search depth of the suffix list, which takes up to 90% of the total execution time on an FPGA. To reduce this searching procedure, we adopt a two-level suffix list design where the lower level is the original suffix list and the upper level is a sparse version of the lower list, as shown in Figure 7.5. Each upper-level node collects the information of what symbols are in its K lower-level nodes; here we call K as the coarse-grained searching range. Whenever we want to search the suffix list to see if there is a match, we first look at the up-level nodes. If there is no matched symbol information recorded in the up-level node, we will skip to the next up-level node until we find a matched one or till the end. Then we perform fine-grain searches in the corresponding lower-level list cycle by cycle.

Figure 7.6 shows the overall flow of the two-level suffix list design. Whenever a new symbol comes in, we first refer to the dictionary table to see if it appeared before,

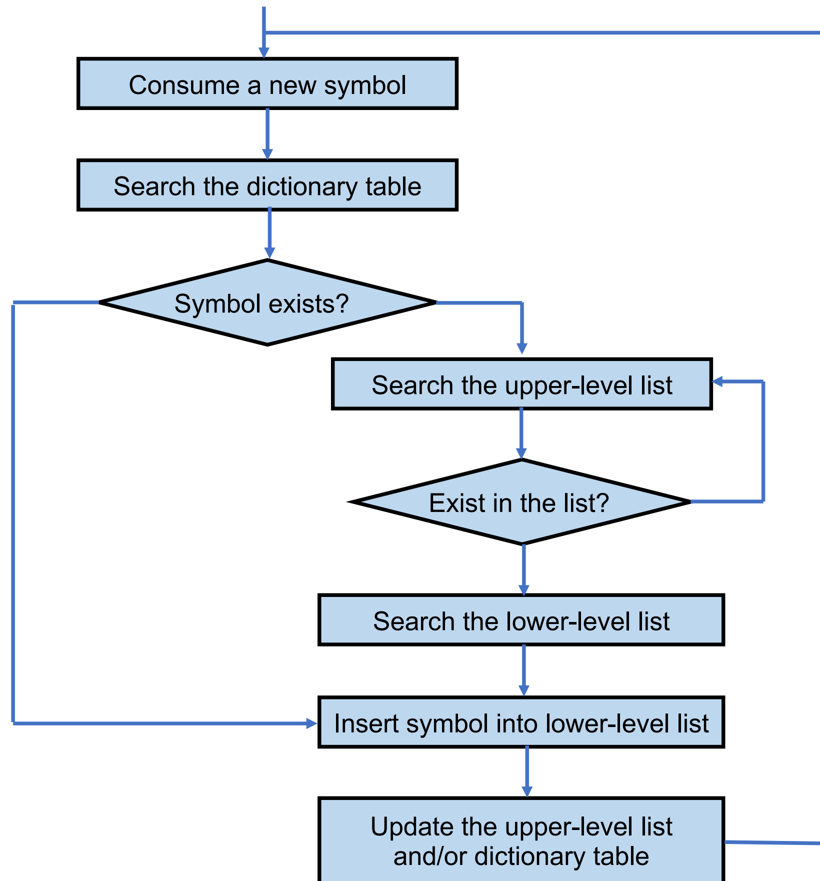

Figure 7.6: The overall flow of the two-level suffix list design.

which is the same as what we do in the direct suffix list design case in Section 7.2.1. If the symbol does exist, that means we need to search the list to insert the symbol into the appropriate position in the lower-level list. But instead of directly searching the lower-level list, we keep searching the upper-level list first. If we find that the upper-level list node indicates there is a match in its corresponding range of the lower-level list, then we start to search the lower-level list. Afterward, we insert the symbol into the right position of the lower-level list. Finally, we need to update the

corresponding node in the upper-level list before we consume the next input symbol.

Lower-level suffix list: the content of the lower-level suffix list entry now becomes $< prev, next, symbol, upper, bound >$. The *prev* and *next* pointers are the same as those in the direct suffix list design; the *up* pointer is the address of its corresponding node in the upper-level list; the 1-bit *bound* indicates if the entry is at the boundary of its coarse-grained searching range. When insert a new node after the boundary node, the *bound* bit of the new node is copied from the original boundary node and the *bound* bit of the old boundary node is cleared.

Upper-level list: the organization of each upper-level node is $< prev, next, lower, flag, count, last >$. The *prev* and *next* pointers records the positions of its previous and next upper-level nodes; the *lower* pointer indicates the smallest suffix of its lower-level nodes; the $flag$ is 128 bits and each bit indicates if its corresponding symbol exists in the lower-level coarse-grained searching range; the 6-bit *count* records the size of the coarse-grained searching range and the 1-bit *last* signal indicates if it is the last node in the upper-level. For a window size of 100kB design with the coarse-grained searching range set to 20, the number of upper-level list entries is $100k/20 = 5k$. We choose the size of the coarse-grained searching range set to be 20 as it corresponds to a reasonable size for the upper-level list as well as a moderate searching time during the fined-grained search in the lower-level list, as indicated in [BB05].

Updating the upper-level list: in general, to insert a new entry into the lower-level list, we only need to update the $flag$ bits and add *count* by 1 in its corresponding upper-level node. If the new entry is inserted before the original smallest suffix, we also update the *lower* pointer with the new suffix. To ensure the updating flags part

not be the critical path, we allocate 1 extra clock cycle for it.

A special case is when the size of coarse-grained searching range becomes larger. We maintain the size of the coarse-grained searching range between 20 and 39. If a new entry is inserted and the size of the coarse-grained searching range is about to become 40, we will split the coarse-grained searching range into two size-20 coarse-grained searching ranges. Since the $flag$ information needs to be recorded precisely for each coarse-grained searching range, we need to refer to all the lower-level nodes in the original coarse-grained searching range to get the updated $flag$ information.

7.3 Evaluation

In this section, we evaluate the performance of the proposed FPGA-based BWT implementations.

7.3.1 Experimental Setup

To better understand the performance difference, we perform experiments using both software and hardware. On the software side, we implement the basic antisequential suffix sorting algorithm and the BWT kernel extracted from the bzip2 software on a Intel Xeon E5-2689 CPU with 256KB L2 cache and 35MB L3 cache. On the hardware side, we implement both the direct suffix sorting and the two-level suffix sorting designs and measure the kernel execution time on a Xilinx Virtex UltraScale+ VCU1525 board. The benchmarks we use are selected from the standard Calgory Corpus and Large Corpus benchmarks whose file size is larger than 500 kB.

7.3.2 Resource Utilization

Our hardware implementation uses less than 1% of LUTs slices and Flip-Flops, 0 DSPs, as the antisequential suffix sorting algorithm is not computation-intensive. The BRAM usage of both the direct suffix list design and the two-level suffix list design is reported in Table 7.1: it increases with the BWT window size. The two-level suffix list design for 500KB window size consumes up to 35.5% BRAM resources, while the BRAM usage for the direct suffix list design is slightly less, as it only has one suffix list.

Table 7.1: Resources utilization

Block Size	BRAM Utilization	Frequency (MHz)
two-level(100KB)	141 (6.53%)	222
two-level(300KB)	462 (21.4%)	187
two-level(500KB)	768 (35.5%)	156
direct (100KB)	120.5 (5.58%)	231
direct (300KB)	385 (17.82%)	192
direct (500KB)	642 (29.7%)	157

For window sizes larger than 500KB, the clock frequency of the designs will further drop and there is no more speedup from the hardware implementation. This is because the critical path sits between the operation of reading the suffix list and send its result to the controller to decide which element to read for the next time.

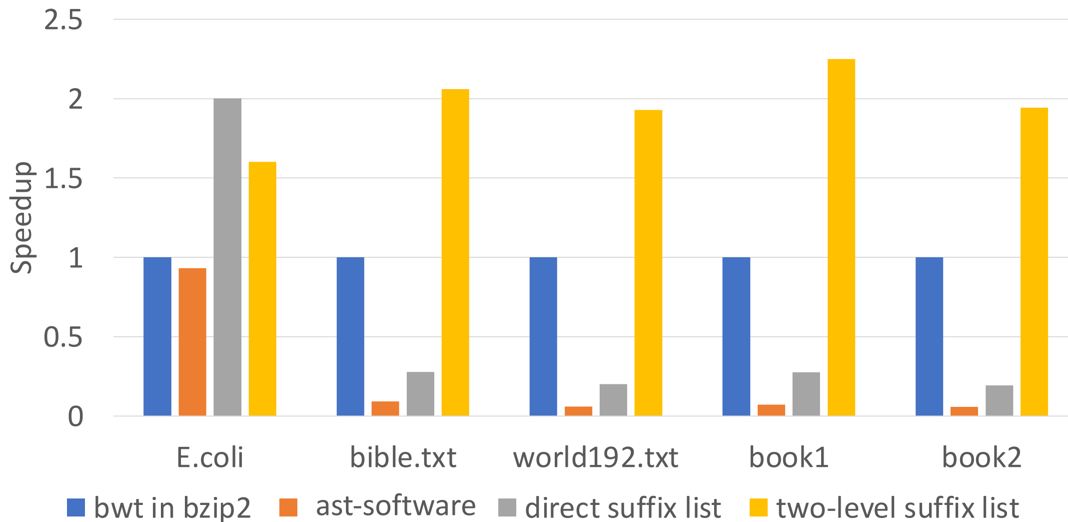

Figure 7.7: Performance comparison for various implementations.

7.3.3 Performance Evaluation

A performance comparison of various designs with a window size of 300KB is shown in Figure 7.7 (ast-software is short for the software antisequential suffix sorting implementation on the CPU). Although the software implementation of the antisequential suffix sorting algorithm is the slowest in the four cases, our FPGA implementations of the two-level suffix list achieves an average speedup of 2x over the fastest software bzip2 BWT kernel on the CPU. A similar speedup of 2.3× and 1.6× is achieved for 100 KB and 500 KB window sizes, respectively, with the major difference sitting in the design frequency.

As shown in Figure 7.7, in general, the direct suffix list design is slower than the CPU bzip2 BWT kernel. However, an interesting observation is for benchmark E.coli, the direct suffix list method works the best. This is because E.coli is a genomic

sequence file whose input symbols have only 4 types and in most cases only a few searches are needed to get a match. For the rest four benchmarks whose input symbols are more diverse, the two-level suffix list design is the best and outperforms the direct suffix list design by around 10x. This gives us the options to select between the two designs depending on the input file characteristics: for input files with a small alphabet size we can choose the direct suffix list design while for input files with more types of symbols we need to use the two-level suffix list to achieve possible speedup.

7.3.4 Comparison of the Compression Quality

Table 7.2 summarizes the comparison result of the compression speed and the compression ratio of the BWT-based compression design and Deflate-based compression design. To measure the BWT-based compression speed, we use the result from the Bzip-2 software implementation and replace the BWT part with the hardware accelerated result to derive the value listed in the table. We can see that the high-quality BWT-based compression design is 1.8× better than the high-throughput Deflate-based compression design in terms of the compression ratio, but this is at the cost of roughly 1075× lower compression speed.

Table 7.2: Comparison of the compression speed and the compression ratio of the high-quality BWT-based compression design and the high-throughput Deflate-based compression design in Chapter 6.

Design	Compression speed	Compression ratio
Deflate-based design	10 GB/s	2.03
BWT-based design	9.3 MB/s	3.61

In today's datacenter, the datacenter engineers may adopt different compression designs for different application scenarios, e.g., in the case where the stored files are frequently accessed, the datacenter engineers may choose the high-throughput compression design to save the time used for the compression; whereas in the case where the stored files are barely touched in the foreseeable future, the engineers may choose the high-quality compression design to further save the storage space.

7.4 Summary

In this chapter, we design and implement the first complete BWT solution on FPGAs with a window size up to 500KB, which makes it possible to accelerate high quality compression on FPGAs. Although the original antisequential suffix sorting algorithm slows down in CPUs, we find it is a good fit for FPGA implementation since FPGAs feature a rich set of distributed on-chip BRAMs. We start from a naive direct suffix list design where the searching is done by sequentially going over the entire list. Then we further accelerate the searching process by adopting a two-level list where the upper-level list acts as the first-hand bookkeeping of the low-level direct suffix list. Experiments show that our FPGA implementations achieve around 2x speedup over the CPU BWT implementation of the high-quality compression software. We also believe our effort in selecting the appropriate algorithm for hardware acceleration may inspire the researchers to dig more into the hardware-friendly algorithms to further improve the performance of this slow operation.

CHAPTER 8

Conclusion

This dissertation is dedicated to enabling high-performance customized accelerator designs for big-data applications. We focus on using FPGAs to accelerate the two fundamental applications: sorting and compression, with the aim of providing scalable and high-performance sorting and compression accelerators that can meet the ever-increasing requirements of today's data center. For the sorting application, we find that the merge tree sort can be well mapped and delicately tuned on FPGAs to work efficiently with different problem sizes ranging from MB to TB as well as vastly distinct memory hierarchies such as DDR DRAM, high-memory bandwidth and solid-state drive. The beauty of the merge tree sorting on hardware is to allow us to independently configure the tree throughput to saturate the off-chip memory bandwidth and scale the tree leaf number to concurrently merge as many sequences as possible. Besides, the structured architecture of the merge trees enables comprehensive modeling of the sorting performance and resource utilization. Moreover, the re-programmable feature of modern FPGAs permits us to modify the accelerator's architecture at runtime to further improve the performance. With all these advantages, we deliver to the data center engineers a set of complete and high-performance sorting solutions on FPGAs that can be scaled to work efficiently with different problems while fully exploiting the hardware features.

For the compression application, we identify the opportunities to accelerate both high-throughput and high-quality compression on FPGAs. On the one hand, we find that directly scaling the existing high-throughput compression accelerators no longer delivers benefits. Instead, we propose a multi-way parallel compressor architecture which employs multiple smaller compression engines to compress the same file. This approach significantly improves the scalability of the high-throughput compressor since the resource utilization grows linearly with the throughput while the critical path stays the same. We further heavily optimize the single smaller compression engine to address the potential compression quality loss issue of the multi-way compression scheme. On the other hand, we implement the first FPGA-based BWT design that supports up to 500 KB window size. This design is based on the uncommon anti-sequential suffix sorting algorithm and we manage to map it onto the hardware using a dedicated linked list. We further accelerate the search process of the list by adopting a two-level list combining both coarse-grain and fine-grain search. With these two distinct types of accelerators, we enrich the capability of modern FPGAs to accelerate big-data compression for different scenarios.

We believe our methodologies and designs have equipped modern FPGAs with adequate efficiency to accelerate sorting and compression in the data center. Besides, this dissertation also acts as a case study to motivate the designers to optimize their customized accelerators holistically from both the algorithm and the architecture perspectives for future big-data applications. Moreover, many of the strategies presented in this dissertation could also be adopted in other types of customized accelerators such as ASICs, if the workload is stable and the cost is affordable.

REFERENCES

[AHS14] Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. "Gzip on a Chip: High Performance Lossless Data Compression on FPGAs Using OpenCL." In *IWOCL*, 2014.

[Ali20] Alibaba. "Alibaba compute optimized instance families with FPGAs." https://www.alibabacloud.com/help/doc-detail/108504.htm, 2020.

[Ama] Amazon. "Amazon EC2 F1 Instances." https://aws.amazon.com/ec2/instance-types/f1/.

[AV88] Allok Aggarwal and Jeffrey S. Vitter. "The input/output complexity of sorting and related problems." In *Research Report, RR-0725 INRIA*, 1988.

[Bar21] Jeff Barr. "AQUA (Advanced Query Accelerator) – A Speed Boost for Your Amazon Redshift Queries." https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/, 2021.

[BB05] Dror Baron and Yoram Bresler. "Antisequential Suffix Sorting For BWT-Based Data Compression." *IEEE Transactions on Computers*, **54**(4):385–397, April 2005.

[BHS80] Jon Louis Bentley, Dorothea Haken, and James B Saxe. "A general method for solving divide-and-conquer recurrences." *ACM SIGACT News*, **12**(3):36–44, 1980.

[BS97] Jon L. Bentley and Robert Sedgewick. "Fast Algorithms for Sorting and Searching Strings." In *Proceeding of Data Compression Conference*, pp. 360–369. ACM, 1997.

[Bur94] M. Burrows et al. "A block-sorting lossless data compression algorithm." *SRC Research Report*, 1994.

[CBB15] Minsik Cho, Daniel Brand, and Rajesh Bordawekar. "PARADIS: An Efficient Parallel Algorithm for In-Place Radix Sort." In *Very Large Data Bases (VLDB)*, 2015.

[CCF16] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms." In *DAC*, 2016.

[CCQ21] Young Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. "HBM Connect: high-performance HLS interconnect for FPGA HBM." In *Proceedings of the 29th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2021.

[CCW20] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. "When hls meets fpga hbm: Benchmarking and bandwidth optimization." *arXiv preprint arXiv:2010.06075*, 2020.

[CFH18a] Jason Cong, Zhenman Fang, Muhuan Huang, Libo Wang, and Di Wu. "CPU-FPGA Co-Scheduling for Big Data Applications." *IEEE Design Test*, **35**(1):16–22, 2018.

[CFH18b] Jason Cong, Zhenman Fang, Muhuan Huang, Libo Wang, and Di Wu. "CPU-FPGA Coscheduling for Big Data Applications." *IEEE Design & Test*, pp. 16–22, 2018.

[CGC22] Yuze Chi, Licheng Guo, and Jason Cong. "Accelerating SSSP for Power-Law Graphs." In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022.

[CGG12] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. "Charm: A composable heterogeneous accelerator-rich microprocessor." In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pp. 379–384, 2012.

[CGH18] Jason Cong, Licheng Guo, Po-Tsang Huang, Peng Wei, and Tianhe Yu. "SMEM++: A Pipelined and Time-Multiplexed SMEM Seeding Accelerator for Genome Sequencing." In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 210–2104, 2018.

[CK13] Umer I. Cheema and Ashfaq A. Khokhar. "A high performance architecture for computing burrows-wheeler transform on FPGAs." In *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1–6, 2013.

[CMF20] Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. "FPGA-Accelerated Samplesort for Large Data Sets." In *Proceedings of the 28th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2020.

[CO14] Jared Casper and Kunle Olukotun. "Hardware Acceleration of Database Operations." In *Proceedings of the 22th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2014.

[Cor01] Canterbury Corpus. "The Calgary Corpus." http://corpus.canterbury.ac.nz/descriptions/#calgary, 2001.

[CSP15] Ren Chen, Sruja Siriyal, and Viktor Prasanna. "Energy and Memory Efficient Mapping of Bitonic Sorting on FPGA." In *Proceedings of the 23th ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA)*, 2015.

[CSR10] Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. "Customizable domain-specific computing." *IEEE Design & Test of Computers*, **28**(2):6–15, 2010.

[CWY17] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. "Bandwidth Optimization Through On-Chip Memory Restructuring for HLS." In *Proceedings of the 54th Design Automation Conference (DAC)*, 2017.

[Deu96] P. Deutsch. "Gzip file format specification version 4.3." https://tools.ietf.org/html/rfc1952, 1996.

[DGY74] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions." *IEEE Journal of Solid-State Circuits*, **9**:256–268, 1974.

[DHZ22] Yixiao Du, Yuwei Hu, Zhongchun Zhou, and Zhiru Zhang. "High-Performance Sparse Linear Algebra on HBM-Equipped FPGAs Using HLS: A Case Study on SpMV." In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2022.

[DTH20] William J Dally, Yatish Turakhia, and Song Han. "Domain-specific hardware accelerators." *Communications of the ACM*, **63**(7):48–57, 2020.

[EK19] Elsayed A. Elsayed and Kenji Kise. "Towards an Efficient Hardware Architecture for Odd-even Based Merge Sorter." In *Proceedings of the 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2019.

[FKB15] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. "A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs." In *FCCM*, pp. 161–164, 2015.

[FKN08] Kermin Fleming, Myron King, Man Cheuk Ng, Asif Khan, and Muralidaran Vijayaraghavan. "High-throughput Pipelined Mergesort." In *2008 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pp. 155–158, 2008.

[FMH19] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. "In-memory Database Acceleration on FPGAs: a survey." *The VLDB Journal*, pp. 1–27, 2019.

[GA22a] Jean loup Gailly and Mark Adler. "Parallel Gzip Compression Library." https://zlib.net/pigz/, 2022.

[GA22b] Jean loup Gailly and Mark Adler. "Zlib Compression Library." http://www.zlib.net/, 2022.

[GAK15] Mingyu Gao, Grant Ayers, and Christos Kozyrakis. "Practical Near-Data Processing for In-Memory Analytics Frameworks." In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pp. 113–124, 2015.

[GCW21] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. "Autobridge: Coupling coarse-grained floorplanning and pipelining for high-frequency hls design on multi-die FPGAs." In *Proceedings of the 29th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2021.

[GLR19] Licheng Guo, Jason Lau, Zhenyuan Ruan, Peng Wei, and Jason Cong. "Hardware Acceleration of Long Read Pairwise Overlapping in Genome Sequencing: A Race Between FPGA and GPU." In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 127–135, 2019.

[GMZ22] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. "RapidStream: Parallel Physical Implementation of FPGA HLS Designs." In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, p. 1–12, 2022.

[Gra] Jim Gray. "Sort Benchmark Home Page." http://sortbenchmark.org/.

[HDU21] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. "GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs." In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9. IEEE, 2021.

[Huf52] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes." *Proceedings of the IRE*, **40**(9):1098–1101, Sept 1952.

[HWY16] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. "Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale." In *SoCC*, 2016.

[Int15] Intel. "Xeon+FPGA Platform for the Data Center." https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf, 2015.

[Int16a] Intel. "Accelerating Datacenter Workloads." http://fpl2016.org/slides/Gupta%20--%20Accelerating%20Datacenter%20Workloads.pdf, 2016.

[Int16b] Intel. "Intel QuickPath Interconnect." https://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html, 2016.

[Int20a] Intel. "Intel Stratix 10 FPGAs." https://www.intel.ca/content/www/ca/en/products/programmable/fpga/stratix-10.html, 2020.

[Int20b] Intel. "Intel Stratix 10 MX FPGA Overview." https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10/mx.html, 2020.

[JB12] Mike Jackson and Ravi Budruk. *PCIe Express Technology.* MindShare Inc., 1st edition, 2012.

[JXA17] Sang-Woo Jun, Shuotao Xu, and Arvind. "Terabyte Sort on FPGA-Accelerated Flash Storage." In *Proceedings of the 25th IEEE international symposium on Field-programmable Custom Cumputing Machines (FCCM)*, 2017.

[KDH15] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. "Profiling a Warehouse-Scale Computer." In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, p. 158–169, 2015.

[KHB14] Joo-Young Kim, Scott Hauck, and Doug Burger. "A Scalable Multi-Engine Xpress9 Compressor with Asynchronous Data Transfer." In *FCCM*, 2014.

[Knu98] Donald E. Knuth. *Art of Computer Programming: Sorting and Searching.* Addison-Wesley Professional, 2nd edition, 1998.

[KT11] Dirk Koch and Jim Torresen. "FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting." In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays (FPGA)*, 2011.

[Lar99] N. J. Larsson et al. "Fast Suffix Sorting." *technical report LU-CS-TR*, 1999.

[LCH20] Morgan Ledwon, Bruce F. Cockburn, and Jie Han. "High-Throughput FPGA-Based Hardware Accelerators for Deflate Compression and Decompression Using High-Level Synthesis." *IEEE Access*, **8**:62207–62217, 2020.

[LFL21] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. "Demystifying the Memory System of Modern Datacenter FPGAs for Software Programmers through Microbenchmarking." In *2021 International Symposium on Field-Programmable Gate Arrays*, FPGA '21, p. 105–115, 2021.

[LZL20] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. "SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD." In *IEEE Computer architecture letters*, 2020.

[MCF05] J. Martinez, R. Cumplido, and C. Feregrino. "An FPGA-based parallel sorting architecture for the Burrows Wheeler transform." In *2005 International Conference on Reconfigurable Computing and FPGAs (ReConFig'05)*, pp. 7 pp.–17, 2005.

[MCK17] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. "High-Performance Hardware Merge Sorter." In *Proceedings of the 25th IEEE international symposium on Field-programmable Custom Cumputing Machines (FCCM)*, 2017.

[MJA13] A. Martin, D. Jamsek, and K. Agarwal. "Fpga-based application acceleration: Case study with gzip compression/decompression streaming engine." In *ICCAD Special Session 7C*, 2013.

[MK18] Kristiyan Manev and Dirk Koch. "Large Utility Sorting on FPGAs." In *Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT)*, 2018.

[MNC09] Rui Marcelino, Horácio C. Neto, and João M. P. Cardoso. "Unbalanced FIFO Sorting for FPGA-based Systems." In *International Conference on Electronics, Circuits, and Systems (ICECS)*, 2009.

[MRL16] Janarbek Matai, Dustin Richmond, Dajung Lee, Zac Blair, Qiongzhi Wu, Amin Abazari, and Ryan Kastner. "Resolve: Generation of High-Performance Sorting Architectures from High-Level Synthesis." In *Proceedings of the 24th ACM/SIGDA international symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[Nim20] Nimbix. "Xilinx Alveo Accelerator Cards." https://www.nimbix.net/alveo, 2020.

[NMG15] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, and Houman Homayoun. "Accelerating Big Data Analytics Using FPGAs." In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 164–164, 2015.

[NVI20] NVIDIA. "NVIDIA A100 Tensor Core GPU Architecture." https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, 2020.

[NVI22] NVIDIA. "CUDA Thrust Library." https://docs.nvidia.com/cuda/thrust/index.html, 2022.

[PBL18] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. "FLiMS: Fast Lightweight Merge Sorter." In *Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT)*, 2018.

[PBL20] Philippos Papaphilippou, Chris Brooks, and Wayne Luk. "An Adaptable High-Throughput FPGA Merge Sorter for Accelerating Database Analytics." In *Proceedings of the 30th International Conference on Field-Programmable Logic and Applications (FPL)*, 2020.

[PCC14] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, Doug Burger, Jim Larus, Gopi Prashanth Gopal, and Simon Pope. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services." In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, pp. 13–24, 2014.

[PLB22] Philippos Papaphilippou, Wayne Luk, and Chris Brooks. "FLiMS: a Fast Lightweight 2-way Merger for Sorting." *IEEE Transactions on Computers*, pp. 1–1, 2022.

[PMM16] Juan Andrés Pérez-Celis, José Martínez-Carranza, Alicia Morales-Reyes, Claudia Feregrino-Uribe, and René Cumplido. "An FPGA Architecture to Accelerate the Burrows Wheeler Transform by Using a Linear Sorter." In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 156–161, 2016.

[QDF18] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. "High-Throughput Lossless Compression on Tightly Coupled CPU-FPGA Platforms." In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 37–44, 2018.

[QFC19] Weikang Qiao, Zhenman Fang, Mau-Chung Frank Chang, and Jason Cong. "An FPGA-Based BWT Accelerator for Bzip2 Data Compression." In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 96–99, 2019.

[QGF22] Weikang Qiao, Licheng Guo, Zhenman Fang, Mau-Chung Frank Chang, and Jason Cong. "TopSort: A High-Performance Two-Phase Sorting Ac-

celerator Optimized on HBM-based FPGAs." In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2022.

[QOG21] Weikang Qiao, Jihun Oh, Licheng Guo, Mau-Chung Frank Chang, and Jason Cong. "FANS: FPGA-Accelerated Near-Storage Sorting." In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 106–114, 2021.

[RHC19] Zhenyuan Ruan, Tong He, and Jason Cong. "INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive." In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[RSW13] Brandon Reagen, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. "Quantifying acceleration: Power/performance trade-offs of application kernels in hardware." In *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 395–400. IEEE, 2013.

[SAB11] Bharat Sukhwani, Bulent Abali, Bernard Brezzo, and Sameh Asaad. "High-Throughput, Lossless Data Compresion on FPGAs." In *FCCM*, 2011.

[SCC22] Atefeh Sohrabizadeh, Yuze Chi, and Jason Cong. "StreamGCN: Accelerating Graph Convolutional Networks with Streaming Processing." In *2022 IEEE Custom Integrated Circuits Conference (CICC)*, pp. 1–8, 2022.

[SCG21] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. "Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication." *arXiv preprint arXiv:2111.12555*, 2021.

[SCG22] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. "Serpens: A High Bandwidth Memory Based Accelerator for General-Purpose Sparse Matrix-Vector Multiplication." In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, p. 211–216, 2022.

[SEC18] Makoto Saitoh, Elsayed A. Elsayed, Thiem Van Chu, Susumu Mashimo, and Kenji Kise. "A High-Performance and Cost-Effective Hardware Merge Sorter without Feedback Datapath." In *Proceedings of the 26th IEEE international symposium on Field-programmable custom cumputing machines (FCCM)*, 2018.

[Sew00] J. Seward. "On the Performance of BWT Sorting Algorithms." In *Proceeding of Data Compression Conference*, DCC '00, pp. 173–182. IEEE, 2000.

[Sew19] Julian Seward. "Bzip2 Compression Library." http://www.bzip.org, 2019. [Online; accessed 13-Jan-2019].

[SHG09] Nadathur Satish, Mark Harris, and Machael Garland. "Designing Efficient Sorting Algorithms for Manycore GPUs." In *International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009.

[SKL16] Wei Song, Dirk Koch, Mikel Lujan, and Jim Garside. "Parallel Hardware Merge Sorter." In *Proceedings of the 24th IEEE international symposium on Field-programmable Custom Cumputing Machines (FCCM)*, 2016.

[SQA20] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. "Bonsai: High-Performance Adaptive Merge Tree Sorting." In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 282–294, 2020.

[STD94] Ching-Long Su, Chi-Ying Tsui, and Alvin M. Despain. "Saving Power in the Control Path of Embedded Processors." In *Design & Test of Computers*, 1994.

[TS92] K. Thearling and S. Smith. "An improved supercomputer sorting benchmark." In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, 1992.

[WGC21] Jie Wang, Licheng Guo, and Jason Cong. "AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA." In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, p. 93–104, 2021.

[WHT21] Johannes Wirth, Jaco A. Hofmann, Lasse Thostrup, Carsten Binnig, and Andreas Koch. "Scalable and Flexible High-Performance In-Network Processing of Hash Joins in Distributed Databases." In *Proceedings of the 2021 International Conference on Field-Programmable Technology (FPT)*, pp. 1–9, 2021.

[WLZ15] Chao Wang, Xi Li, and Xuehai Zhou. "SODA: Software defined FPGA based accelerators for big data." In *2015 Design, Automation and Test in Europe Conference & Exhibition (DATE)*, pp. 884–887, 2015.

[Xil20a] Xilinx. "SmartSSD Computational Storage Drive: Installation and User Guide." https://www.xilinx.com/support/documentation/boards_and_kits/accelerator-cards/1_0/ug1382-smartssd-csd.pdf, 2020.

[Xil20b] Xilinx. "Vivado Design Suite Tutorial." https://www.xilinx.com/support/documents/sw_manuals/xilinx2020_2/ug938-vivado-design-analysis-closure-tutorial.pdf, 2020.

[Xil21a] Xilinx. "Alveo U280 Data Center Accelerator Card Data Sheet." https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds963-u280.pdf, 2021.

[Xil21b] Xilinx. "Alveo U50 Data Center Accelerator Card Data Sheet." https://www.xilinx.com/content/dam/xilinx/support/documents/data_sheets/ds965-u50.pdf, 2021.

[ZKP21] Bingyi Zhang, Rajgopal Kannan, and Viktor Prasanna. "BoostGCN: A Framework for Optimizing GCN Inference on FPGA." In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 29–39, 2021.

[ZL77] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression." *IEEE Transactions on Information Theory*, **23**(3):337–343, May 1977.

[ZLW17] Baofu Zhao, Yubin Li, Yu Wang, and Huazhong Yang. "Streaming Sorting Network Based BWT Acceleration on FPGA for Lossless Compression." In *Proceedings of the 2017 International Conference on Field-Programmable Technology (FPT)*, pp. 247–250, 2017.

[ZMP12] Marcela Zuluaga, Peter Milder, and Markus Püschel. "Computer generation of streaming sorting networks." In *DAC Design Automation Conference 2012*, pp. 1241–1249, 2012.

[ZMP16] Marcela Zuluaga, Peter A. Milder, and Markus Püschel. "Streaming Sorting Networks." *ACM Transactions on Design Automation of Electronic Systems*, **21**(4):55, 2016.