TECHNISCHE UNIVERSITÄT DRESDEN

Database System Acceleration on FPGAs

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Mehdi Moghaddamfar, M.Sc.

Gutachter:	**Prof. Dr.-Ing. Wolfgang Lehner** Technische Universität Dresden Fakultät Informatik Institut für Systemarchitektur Professur für Datenbanken 01062 Dresden
Fachreferent:	**Prof. Dr. Akash Kumar** Technische Universität Dresden Fakultät Informatik Institut für Technische Informatik Professur für Prozessordesign 01062 Dresden
Tag der Verteidigung:	26. April 2023

Dresden, im Februar 2023

Doctoral Committee:

- Prof. Dr. Christof Fetzer (Head of the Committee)
  Technische Universität Dresden, Germany
- Prof. Dr.-Ing. Wolfgang Lehner (Co-Advisor, Reviewer)
  Technische Universität Dresden, Germany
- Prof. Dr. Akash Kumar (Co-Advisor, Subject Expert)
  Technische Universität Dresden, Germany
- Prof. Dr.-Ing. Jürgen Teich (External Reviewer)
  Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
- Prof. Dr.-Ing. Diana Goehringer (Committee Member)
  Technische Universität Dresden, Germany

Additional Members of the Thesis Advisory Board:

- Dr. Norman May
  SAP SE, Germany
- Dr. Christian Färber
  Intel Corporation, Germany

ABSTRACT

Relational database systems provide various services and applications with an efficient means for storing, processing, and retrieving their data. The performance of these systems has a direct impact on the quality of service of the applications that rely on them. Therefore, it is crucial that database systems are able to adapt and grow in tandem with the demands of these applications, ensuring that their performance scales accordingly. In the past, Moore's law and algorithmic advancements have been sufficient to meet these demands. However, with the slowdown of Moore's law, researchers have begun exploring alternative methods, such as application-specific technologies, to satisfy the more challenging performance requirements. One such technology is field-programmable gate arrays (FPGAs), which provide ideal platforms for developing and running custom architectures for accelerating database systems.

The goal of this thesis is to develop a domain-specific architecture that can enhance the performance of in-memory database systems when executing analytical queries. Our research is guided by a combination of academic and industrial requirements that seek to strike a balance between generality and performance. The former ensures that our platform can be used to process a diverse range of workloads, while the latter makes it an attractive solution for high-performance use cases.

Throughout this thesis, we present the development of a system-on-chip for database system acceleration that meets our requirements. The resulting architecture, called CbMSMK, is capable of processing the projection, sort, aggregation, and equi-join database operators and can also run some complex TPC-H queries. CbMSMK employs a shared sort-merge pipeline for executing all these operators, which results in an efficient use of FPGA resources. This approach enables the instantiation of multiple acceleration cores on the FPGA, allowing it to serve multiple clients simultaneously. CbMSMK can process both arbitrarily deep and wide tables efficiently. The former is achieved through the use of the sort-merge algorithm which utilizes the FPGA RAM for buffering intermediate sort results. The latter is achieved through the use of KeRRaS, a novel variant of the forward radix sort algorithm introduced in this thesis. KeRRaS allows CbMSMK to process a table a few columns at a time, incrementally generating the final result through multiple iterations. Given that acceleration is a key objective of our work, CbMSMK benefits from many performance optimizations. For instance, multi-way merging is employed to reduce the number of merge passes required for the execution of the sort-merge algorithm, thus improving the performance of all our pipeline-breaking operators. Another example is our in-depth analysis of early aggregation, which led to the development of a novel cache-based algorithm that significantly enhances aggregation performance. Our experiments demonstrate that CbMSMK performs on average 5 times faster than the state-of-the-art CPU-based database management system MonetDB.

CONTENTS

I Database Systems & FPGAs 13

1 Introduction 15

1.1 Databases & the Importance of Performance 16

1.2 Accelerators & FPGAs 16

1.3 Requirements 18

1.4 Outline & Summary of Contributions 19

2 Background on Database Systems 21

2.1 Databases 22
2.1.1 Storage Model 22
2.1.2 Storage Medium 23

2.2 Database Operators 23
2.2.1 Projection 24
2.2.2 Filter 24
2.2.3 Sort 25
2.2.4 Aggregation 25
2.2.5 Join 25
2.2.6 Operator Classification 26

2.3 Database Queries 26

2.4 Impact of Acceleration 27

3 Background on FPGAs 29

3.1 FPGA 30
3.1.1 Logic Element 30
3.1.2 Block RAM (BRAM) 32
3.1.3 Digital Signal Processor (DSP) 33
3.1.4 IO Element 33
3.1.5 Programmable Interconnect 33

3.2 FPGA Design Flow 33
3.2.1 Specifications 34
3.2.2 RTL Description 34
3.2.3 Verification 36
3.2.4 Synthesis, Mapping, Placement, and Routing 36

3.2.5 Timing Analysis . . . 36
3.2.6 Bitstream Generation and FPGA Programming . . . 37
3.3 Implementation Quality Metrics . . . 37
3.4 FPGA Cards . . . 37
3.5 Benefits of Using FPGAs . . . 38
3.6 Challenges of Using FPGAs . . . 39
4 Related Work **41**
4.1 Summary of Related Work . . . 42
4.2 Platform Type . . . 43
4.2.1 Accelerator Card . . . 44
4.2.2 Coprocessor . . . 45
4.2.3 Smart Storage . . . 45
4.2.4 Network Processor . . . 46
4.3 Implementation . . . 46
4.3.1 Loop-based implementation . . . 47
4.3.2 Sort-based Implementation . . . 48
4.3.3 Hash-based Implementation . . . 49
4.3.4 Mixed Implementation . . . 50
4.4 A Note on Quantitative Performance Comparisons . . . 51
II Cache-Based Morphing Sort-Merge with KeRRaS (CbMSMK) **53**
5 Objectives and Architecture Overview **55**
5.1 From Requirements to Objectives . . . 56
5.2 Architecture Overview . . . 57
5.3 Outline of Part II . . . 58
6 Comparative Analysis of OpenCL and RTL for Sort-Merge Primitives on FPGAs **61**
6.1 Programming FPGAs . . . 62
6.2 Related Work . . . 63
6.3 Architecture . . . 63
6.3.1 Global Architecture . . . 64
6.3.2 Sorter Architecture . . . 64
6.3.3 Merger Architecture . . . 67
6.3.4 Scalability and Resource Adaptability . . . 67
6.4 Experiments . . . 68
6.4.1 OpenCL Sort-Merge Implementation . . . 68

6.4.2 RTL Sorters . . . 69
6.4.3 RTL Mergers . . . 70
6.4.4 Hybrid OpenCL-RTL Sort-Merge Implementation . . . 71

6.5 Summary & Discussion . . . 72

7 Resource-Efficient Acceleration of Pipeline-Breaking Database Operators on FPGAs 73

7.1 The Case for Resource Efficiency . . . 74

7.2 Related Work . . . 75

7.3 Architecture . . . 76
7.3.1 Sorters . . . 77
7.3.2 Sort-Network . . . 79
7.3.3 X:Y Mergers . . . 80
7.3.4 Merge-Network . . . 81
7.3.5 Join Materialiser (JoinMat) . . . 81

7.4 Experiments . . . 81
7.4.1 Experimental Setup . . . 82
7.4.2 Implementation Description & Tuning . . . 82
7.4.3 Sort Benchmarks . . . 84
7.4.4 Aggregation Benchmarks . . . 85
7.4.5 Join Benchmarks . . . 86

7.5 Summary . . . 87

8 KeRRaS: Column-Oriented Wide Table Processing on FPGAs 89

8.1 The Scope of Database System Accelerators . . . 90

8.2 Related Work . . . 91

8.3 Key-Reduce Radix Sort (KeRRaS) . . . 92
8.3.1 Time Complexity . . . 95
8.3.2 Space Complexity (Memory Utilization) . . . 95
8.3.3 Discussion and Optimizations . . . 95

8.4 Architecture . . . 96
8.4.1 MSM . . . 97
8.4.2 MSMK: Extending MSM with KeRRaS . . . 97
8.4.3 Payload, Aggregation and Join Processing . . . 101
8.4.4 Limitations . . . 102

8.5 Experiments . . . 102
8.5.1 Experimental Setup . . . 102
8.5.2 Datasets . . . 103
8.5.3 MSMK vs. MSM . . . 104
8.5.4 Payload-Less Benchmarks . . . 104
8.5.5 Payload-Based Benchmarks . . . 106
8.5.6 Flexibility . . . 107

8.6 Summary . . . 108

9 A Study of Early Aggregation in Database Query Processing on FPGAs 109
9.1 Early Aggregation . . . 110
9.2 Background & Related Work . . . 111
9.2.1 Sort-Based Early Aggregation . . . 112
9.2.2 Cache-Based Early Aggregation . . . 112
9.3 Simulations . . . 114
9.3.1 Datasets . . . 114
9.3.2 Metrics . . . 115
9.3.3 Sort-Based Versus Cache-Based Early Aggregation . . . 115
9.3.4 Comparison of Set-Associative Caches . . . 116
9.3.5 Comparison of Cache Structures . . . 117
9.3.6 Comparison of Replacement Policies . . . 119
9.3.7 Cache Selection Methodology . . . 120
9.4 Cache System Architecture . . . 121
9.4.1 Window Aggregator . . . 122
9.4.2 Compressor & Hasher . . . 123
9.4.3 Collision Detector . . . 123
9.4.4 Collision Resolver . . . 124
9.4.5 Cache . . . 124
9.5 Experiments . . . 125
9.5.1 Experimental Setup . . . 126
9.5.2 Resource Utilization and Parameter Tuning . . . 127
9.5.3 Datasets . . . 127
9.5.4 Benchmarks on Synthetic Data . . . 128
9.5.5 Benchmarks on Real Data . . . 129
9.6 Summary . . . 129
10 The Full Picture 131
10.1 System Architecture . . . 132
10.2 Benchmarks . . . 133
10.3 Meeting the Objectives . . . 135
III Conclusion 137
11 Summary and Outlook on Future Research 139
11.1 Summary . . . 140
11.2 Future Work . . . 141
Bibliography 145
List of Figures 165
List of Tables 169

AKNOWLEDGMENTS

I would like to express my gratitude to my academic supervisors, Wolfgang Lehner and Akash Kumar, for their insightful guidance, critiques, and support throughout my thesis. I am also thankful to my industry supervisors, Norman May and Christian Färber, whose technical counsel and recommendations have been instrumental in shaping my research. My appreciation also extends to Jürgen Teich for his thorough examination of my work. Additionally, I would like to recognize the support provided by all members of my family during my doctoral studies. Lastly, I would like to acknowledge my use of the AI tool ChatGPT for its paraphrasing abilities, which helped to enhance the conciseness and readability of this manuscript.

Mehdi Moghaddamfar
April 28, 2023

Part I

Database Systems & FPGAs

1 INTRODUCTION

1.1 Databases & the Importance of Performance

1.2 Accelerators & FPGAs

1.3 Requirements

1.4 Outline & Summary of Contributions

1.1 DATABASES & THE IMPORTANCE OF PERFORMANCE

Databases are pillars of the modern society. Almost every online activity including searching the web, accessing social media, checking the weather, and banking uses databases to serve up information. Moreover, data gathered in most research and investigations such as genome sequencing, particle studies, and political polls are all stored in databases [GUW09]. Nowadays, databases commonly hold terabytes if not petabytes of data and are stored in a distributed manner [TSJ+10, CER18, MK19]. A (relational) *database* is a collection of tables.

A *database management system* (DBMS) or more colloquially a "database system" is a tool used to create, update, and analyze a database efficiently [GUW09]. In essence, it can execute queries on a database. Database systems allow us to maintain and utilize large volumes of data, far faster than a human ever could. In fact, performance is one of the key reasons behind the invention of database systems [Dat03, RG03].

With large volumes of data acquired and processed every day and with fast data-driven decision-making being a competitive advantage for most businesses, it is crucial to consider and improve the performance of database systems as parts of business intelligence solutions [CDN11]. Database system performance also plays an important role in the behaviour of the business' customers. For instance, the users of a website expect a certain latency between the click of a button and when they receive a response from the website. An experiment run by Bing showed that inserting a delay of 2000 ms when responding to a search query drops user satisfaction by 3.8% and revenue/user by 4.3% [SB09]. A separate study by Google saw a decrease in the number of searches by users submitted to longer search delays, weeks after the end of a similar experiment [Bru09]. The results were so negative that Google ended the experiment prematurely [HP17]. Furthermore, the large number of active users on many data-based applications (e.g., social media, maps, etc. [Wik22f]) requires their database systems to process multiple queries at the same time and quickly. In a nutshell, high and improving database system performance is of critical importance for both businesses and their customers.

1.2 ACCELERATORS & FPGAS

In 1965, Gordon E. Moore posited that the number of components per integrated circuit (IC) would roughly double every year for the next 10 years [Moo98]. A decade later, the law was revised. The now called *Moore's Law* observes that the maximum number of transistors in a dense integrated circuit doubles every two years [HP17]. The increase in the number of transistors is typically accompanied by a decrease in their (feature) size, allowing higher clock frequencies to be achieved. The two factors allowed ICs to accommodate **more and faster** circuits. The law can be observed across generations of Intel CPUs, shown in Figure 1.1. A slowdown of Moore's law, in the frequency domain, can be observed around the years $2000 - 2005$. Before the slowdown, a database system could run faster by simply updating the server's CPU once every few years. After the slowdown, new performance-improving techniques needed to be employed.

Nowadays, most hardware-based acceleration strategies involve the following devices [AAA+22]:

- **Multicore CPUs:** IC resources (transistors, SRAM, etc.) are used to implement many cores, large caches, and single instruction multiple data (SIMD) execution units.

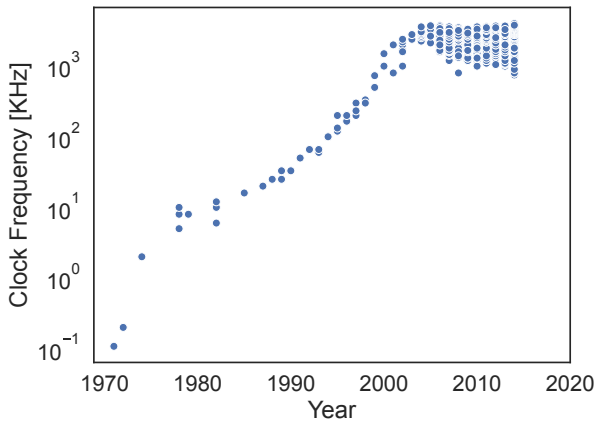


Figure 1.1: The frequency of Intel CPUs as a function of their release year. The data in this graph is sourced from the Stanford CPU DB platform [DKM+12, Gro16].

- **Graphics Processing Units (GPUs):** IC resources are used to implement as many cores (with deep pipelines) as possible, with relatively small caches, and often high-bandwidth memories.
- **Field-Programmable Gate Arrays (FPGAs):** IC resources are used to implement reconfigurable logic, memory, and interconnect modules that can be programmed to "mimic a circuit", executing a specific functionality.

Starting from multicore CPUs down towards FPGAs, these devices gain in performance (and power efficiency) by sacrificing generality and making design and programming more difficult. We cite notable database system acceleration research for all 3 categories of devices in Chapter 4. Note that as the most repetitive and expensive task of a database system is query execution [HAMS08], the terms "database system acceleration", "database query acceleration", or simply "database acceleration" are often used interchangeably.

In this thesis, we focus on FPGA-based database system acceleration. Indeed, now more than ever, FPGA technology is providing us with efficient and practical platforms for query processing:

- **Large amount of resources:** Thanks to advances in FPGA technology, modern FPGAs contain enough logic and memory resources to implement entire system-on-chips (SoCs) [Tri15, GK19, Bro96]. Database query acceleration requires these resources to process the various operators a query may contain, and to serve multiple clients simultaneously.
- **Advanced memory and communication systems:** FPGA cards are providing increasingly larger and faster memories [Cor21b, Cor19b, WZTD19, DSB19, KHD+20]. FPGA-CPU bandwidth has also been improving, allowing faster transfers of data between CPU RAM and FPGA memory [FMH+20]. Memory-intensive database system workloads can greatly benefit from these improvements and innovations.
- **Improved programming methodologies:** FPGA programming requires deep knowledge of digital design and hardware description languages (HDLs) such as Verilog [IEE18] and VHDL [IEE19]. This has been a major obstacle in their adoption by software developers. High-level synthesis (HLS) tools designed by major FPGA vendors (e.g., OpenCL by Intel) allow developers to implement their FPGA circuits

using more familiar languages such as C/C++ [Int18a, Xil21]. Modern HDLs such as Chisel [BVR+12] provide advanced circuit generation capabilities enabling high compile-time flexibility for customizing an accelerator to the needs of a workload and for the FPGA platform it will run on.

- **Availability in the cloud:** FPGAs are now both present [AWS17, Azu22, Clo21, SFJ+19] and being used [KS16, MA18, KMK+19] in the cloud. This combined with the popularity of databases in the cloud [AG12, Sak14, Wik22b] presents a perfect acceleration opportunity. For instance, Amazon Web Services (AWS) uses FPGAs to boost the performance of some analytical queries [Bar21].

- **Energy efficiency:** FPGAs typically achieve higher energy efficiency compared to general-purpose hardware [NMG+15, CSPJ03].

In summary, thanks to their architecture, widespread availability, and improved programmability FPGAs present great opportunities to get around the slowdown of Moore's law. FPGAs and their acceleration capabilities are discussed in length in Chapter 3.

1.3 REQUIREMENTS

Database queries are typically run by executing a set of operators in a specific order. Database system acceleration often boils down to reducing the execution time of those operators. There are a few dozen operators supported by most database systems. Unfortunately, given the application-specific nature of FPGA designs it is impractical, if not impossible, to develop an FPGA-based database system for accelerating all those operators, efficiently and within a 3-year PhD thesis period.

To help narrow down our research, we turned to our "stakeholders". Indeed, our research is the product of a collaboration between SAP and Intel. SAP's high-performance analytic appliance (SAP HANA) is a database system that loads its data into main memory for processing, instead of keeping it on disk [FML+12, FCP+11, SE22]. The goal of the joint SAP-Intel effort is to accelerate in-memory database systems such as SAP HANA. This leads us to the following requirements, and eventual goals:

1. SAP HANA is a database solution, used by many SAP customers working in various domains. To reach out to a majority of those customers, we want our platform to support as many database operators as possible, and be extensible enough to support more operators in the future.

2. The primary goal of our research is acceleration. We must focus on accelerating the database operators that are both popular (i.e., frequently used in queries) and computationally expensive to execute.

3. SAP customers might have different types of FPGAs, with various amounts of resources. We must ensure our acceleration architecture is flexible enough that it can be deployed on various FPGA hardware with minimum effort.

4. As our goal is to accelerate in-memory database systems, we must design an acceleration platform suitable for in-memory processing. Chapters 4 and 5 will expand more on this point.

5. Modern database systems are *multi-client*, i.e., they can process multiple queries at the same time. This may seem like a hard-to-satisfy requirement but is a blessing in disguise. Indeed, we may accelerate a database system by making both single queries run faster and multiple queries run in parallel.

While there is a large body of work on database system acceleration on FPGAs [FMH+20], most of it focuses on the acceleration of one or two operators, on a limited range of datasets, and for single-client setups. Our unique set of requirements leads us to contributions and innovations beyond those proposed in past research.

1.4 OUTLINE & SUMMARY OF CONTRIBUTIONS

This thesis is divided into 3 parts. Part I describes past work and motivates our research. In Chapter 2, we describe databases, queries, and operators in detail. We then classify popular core database operators into 2 categories: streaming and pipeline-breaking. The more complex pipeline-breaking operators become the focus of our acceleration efforts. A database systems expert may skip this chapter.

Chapter 3 is a primer on FPGAs. We start by describing the components of an FPGA, how to design and program for FPGAs, and the structure of a typical FPGA card. We then explain through an example the advantages of using FPGAs; before, finally outlining some of the major challenges in doing so. An FPGA developer may skip this chapter.

Chapter 4 surveys related work on database system acceleration on FPGAs. Given the large body of work in the domain, we concentrate on past work on processing complex pipeline-breaking database operators. We identify and highlight the most important characteristics differentiating current accelerators. They help orient us in our own research described next.

Part II of the thesis contains our major contributions. In Chapter 5, we convert our accelerator requirements defined in Section 1.3 into concrete goals and objectives. We then present the outline of an acceleration platform conforming to those goals, leaving the following chapters to fill in the specifics. The content of these chapters was published at major research venues.

A major decision when designing FPGA accelerators for database systems, traditionally built by software developers, is the choice of the programming methodology. We present a comparative study of HLS and HDLs for FPGA programming in Chapter 6. More specifically, we compare the performance, resource utilization, and required development effort of OpenCL and VHDL implementations of important algorithms in database query processing. We come to the conclusion that in most cases OpenCL is less efficient than VHDL in regard to both performance and resource utilization, while requiring the same amount of development time. Although a hybrid HLS-HDL implementation could offer the best of both words in some cases, a full HDL implementation is a better fit for performance-critical applications. In the same chapter, we also present a novel heapsort algorithm that eliminates data hazards (or dependencies) while traversing the heap, resulting in excellent performance characteristics. The algorithm is ultimately used in our architecture in Chapter 9.

In Chapter 7, we lay the foundation for our resource-efficient, high-performance sort-based query acceleration platform called morphing sort-merge (MSM). MSM achieves resource efficiency by reusing its dedicated FPGA resources to support 3 major database

operators through runtime configurability. There, we experiment with treap-based sorting instead of heapsort. Treaps are randomized data structures that can both sort and partially aggregate database rows. We propose novel treap implementation techniques, namely feedback, prefetching, and parallel IO to achieve high performance. MSM achieves efficient RAM bandwidth utilization and throughput speedups of up to $28\times$ compared to MonetDB, a multi-client, multi-threaded, state-of-the-art database system.

In Chapter 8, we transform our row-oriented database accelerator, MSM, into a column-oriented accelerator which we name MSMK. In a nutshell, column-oriented databases store tables one column at a time, whereas row-oriented databases store entire rows one after another. The column-oriented MSMK offers 3 key advantages. First, the accelerator now also supports projections, a streaming operator used in most database queries. Second, the platform can now process tables that are wider than its data path width. Indeed, only the columns needed in the query will be fetched and processed by the acceleration pipeline. Finally, queries involving more columns than the data path can accommodate can be processed using KeRRaS, our novel algorithm capable of handling arbitrarily wide tables in multiple iterations. Our experiments show that MSMK behaves similarly to MSM on narrow tables, and scales well as the number of columns increases.

Chapter 9 focuses on early aggregation, a technique used to improve the performance and memory utilization of the aggregation operator. We start by comparing existing early aggregation algorithms using accurate simulations. Our findings lead us to set-associative caches with a low inter-reference recency set (LIRS) replacement policy that exhibit both great performance and modest implementation complexity. We then present a novel scalable architecture for implementing set-associative caches. Integrating caches into MSMK, we name the updated platform CbMSMK. Benchmarks of CbMSMK demonstrate speedups of up to $3\times$ for end-to-end aggregation compared to MSMK.

Chapter 10 contains a detailed summary of our entire database acceleration platform. We also present benchmarks comparing the performance of our platform against MonetDB for both single-operator and full TPC-H query execution. We observe that our accelerator performs either as good as, or much better than the state-of-the-art in software. More importantly, it is capable of handling complex queries, without the need for the CPU to step in and finalize execution. Indeed, we manage to tackle one of the greatest problems with accelerators: data movement. Thanks to the capabilities of our acceleration platform, the FPGA can now own the data, keep it, and use it while only transferring the results of a query to the CPU.

Part III concludes our work. Chapter 11 begins with a summary of the thesis and ends with a set of possible directions for future research in our domain.

2

BACKGROUND ON DATABASE SYSTEMS

2.1 Databases

2.2 Database Operators

2.3 Database Queries

2.4 Impact of Acceleration

In this chapter, we briefly explore the world of relational databases. We start by defining databases and present two common ways their data can be stored in memory or on disk. We then explore core database operators, the building blocks of many database queries. We further classify these operators into two major categories according to their complexity. Next, queries and their transformation into query execution plans are defined. We end the chapter with a discussion on Amdahl's law and its implications on our research.

2.1 DATABASES

The father of (relational) databases, Edgar F. Codd, defines a *relation* as a set of n-tuples. Each n-tuple has its first element drawn from a set S_1, second element from a set S_2, and so on. The sets $S_1, ..., S_n$ are arbitrary and do not have to be distinct [Cod69].

Tables are perhaps a more modern interpretation of relations. A *table* is a finite (ordered) stream of rows that follow the same schema, i.e., have the same number and type of columns [GUW09]. A *cell* is the intersection of a row and a column. A simple database table representing a few students enrolled in a test is shown in Figure 2.1. It consists of 4 columns and 5 rows. The names of the columns (also called *attributes*) are shown on the top of the table. Each column has its own data type. For instance, the *name* column is of type string whereas the *score* column holds decimal values. The table is sorted on the *sid* (student ID) column, but this does not have to be the case.

sid	name	age	score
0	John	21	12.4
1	Liam	32	19.2
2	Elena	25	8.0
3	Ray	28	12.4
4	Elisabeth	24	17

Table 2.1: An example of a table in a relational database.

It shall be noted that, by definition, a table may have duplicate rows whereas a relation cannot have duplicate tuples. Most database systems allow duplicate rows if not explicitly restricted; thus, the table model seems more accurate. A (relational) *database* is simply a collection of tables.

2.1.1 Storage Model

There are two prominent methods for storing the tables of a database:

- ***Row storage***: *Row-oriented* database systems (a.k.a. row-stores) store the rows of a table one after another in the memory or on disk.
- ***Column(ar) storage***: *Column-oriented* database systems (a.k.a. column-stores) store the columns of a table separately, each in their own contiguous buffer in the memory or on disk.

In a comparison between the two storage models, column storage has 2 major advantages. First, it allows a database system to access only the columns involved in a query, effectively increasing its memory bandwidth efficiency. Given the memory-bound nature of many analytical database workloads, this often results in much higher performance. Second, since the data in a column are more likely to be "close together" in value, column storage may result in higher compression factors further decreasing memory bandwidth requirements. Row-stores, on the other hand, may perform better on update queries, because inserting, updating, or deleting a row of a column-store database requires a separate memory access to each column (as they are stored separately). The row-store may only require a single memory access to update a row [AMH08, SAB+05].

To conclude, column-stores are optimal for processing **analytical queries** used in business intelligence, data warehouses, and decision support applications; which happen to be the target of a lot of modern database systems and accelerators. Indeed, most high-performance database systems (e.g., MonetDB [IGN+12], HyPer [FfI22], Hyrise [Ins22], and Amazon Redshift [Inc22]) are column-stores, with some others (e.g., SAP HANA [SE17], Oracle DB [Cor18b], Microsoft SQL Server [Mic22]) supporting both row and column storage.

2.1.2 Storage Medium

Database systems can be further categorized based on the medium they use to store the database. *In-memory database systems* store the database in physical (main) memory. In *disk-oriented database systems*, the database resides on disk. In both cases, the database system may have copies of the data in other mediums (e.g., backup on disk, cache in main memory), but the main copy resides in the specified medium [GS92, Pet19]. More specifically, in-memory database systems load the working set of tables/columns into memory while disk-oriented database systems chunk rows or columns into pages and use a page buffer to load the accessed pages into memory.

In-memory database systems are typically faster than disk-oriented database systems. This is thanks to the higher performance of the main memory compared to the disk and the fact that database query execution can be both simpler and more optimal (e.g., require fewer CPU instructions) on data already in the main memory [Wik22c]. Notable in-memory database systems include SAP HANA [PRE21], HyPer [FfI22], Hyrise [Ins22], and MonetDB [Tea22a]. The latter uses both the swap space and memory-mapped files to deal with data exceeding the available physical memory [Tea19], making it a hybrid solution. Prominent disk-oriented database systems include Amazon Redshift [GAT+15], Impala [KBB+15], and PostgreSQL [Tea22b].

2.2 DATABASE OPERATORS

Database queries are composed using a collection of operators. We take a bottom-up approach by first describing the latter, before moving onto the former in Section 2.3. In simple terms, a database *operator* is a function of one or more tables, and produces a new table itself [RG03]. The most popular database operators are **projection, selection, sort, aggregation**, and **equi-join**. We describe these operators using tables R and S shown in Figures 2.1(a) and 2.1(b) as input.

k_1	k_2	v_1	v_2
21	34	4	3
45	11	3	1
21	34	1	5
45	11	5	6

(a) Table R

k_1	k_2	v_1	v_2
88	95	7	1
47	71	2	3
21	34	9	5
88	12	5	1

(b) Table S

k_1	k_2
21	34
45	11
21	34
45	11

(c) Project$_{k_1,k_2}(R)$

k_1	k_2	v_1	v_2
88	95	7	1
47	71	2	3

(d) Filter$_{k_2>40\&v_1<10}(S)$

k_1	k_2	v_1	v_2
21	34	9	5
47	71	2	3
88	12	5	1
88	95	7	1

(e) Sort$_{k_1,k_2}(S)$

k_1	k_2	v_1	v_2
21	34	4	3
21	34	1	5
45	11	3	1
45	11	5	6

(f) Grouping R based on k_1 and k_2

k_1	k_2	sum(v_1)	max(v_2)
21	34	5	5
45	11	8	6

(g) Aggr$_{k_1,k_2,sum(v_1),max(v_2)}(R)$

$R.k_1$	$R.k_2$	$S.k_1$	$S.k_2$	$R.v_1$	$R.v_2$	$S.v_1$	$S.v_2$
21	34	21	34	4	3	9	5
21	34	21	34	1	5	9	5

(h) Join$_{k_1,k_2}(R,S)$

Figure 2.1: Tables R and S and examples of database operators applied to them: (a) Table R, (b) Table S, (c) Projection operator, (d) Filter/Selection operator, (e) Sort operator, (f) Grouping step used in the aggregation operator, (g) Aggregation operator, (h) Join operator.

2.2.1 Projection

The *projection operator* applied to a table T produces a new table with a select set of columns from T [RG03]. We denote this operation Project$_C(T)$ where C is the sequence of "surviving" columns. The result of Project$_{k_1,k_2}(R)$ is shown in Figure 2.1(c).

2.2.2 Filter

The *filter operator* applied to a table T produces a new table composed of the rows of T that satisfy a certain predicate P. We denote this operation Filter$_P(T)$. The filter operator is also commonly referred to as the *selection operator* [RG03]. The result of Filter$_{k_2>40\&v_1<10}(S)$ is shown in Figure 2.1(d).

Before we describe the remaining operators, let us define some more table terminology. For many operators, the columns of a table must be partitioned into *key columns* and *value columns*. The value columns are together called *the payload*. The partitioning is done

based on the parameters of the operator itself and does not affect how the table is stored. Note that contrary to the traditional definition of key columns, here, we do not require them to be unique for each row.

2.2.3 Sort

The *sort operator* sorts a table based on one or more of its columns, called the key columns [GUW09]. We denote this operation $\text{Sort}_C(T)$ where C is a sequence of key columns. The result of $\text{Sort}_{k_1,k_2}(S)$ is shown in Figure 2.1(e). In our example k_1 and k_2 are the key columns. Concatenated together, $\overline{k_1 k_2}$ defines a sort key for each row. For instance, assuming k_1 and k_2 are integer columns with values ranging from 0 to 999, the composite key for the first row of table S is 088095. The non-key columns v_1 and v_2 constitute the value columns or the payload. Note that the database sort operator does not require stable sorting.

2.2.4 Aggregation

The *aggregation operator* starts by grouping the rows of a table T based on one or more of its columns, called the grouping key columns or key columns for short. Essentially, rows with the same values in their key columns belong to the same group. The value columns (i.e., the non-key columns) of the rows in the same group are then accumulated using a function [GUW09, CR07]. We denote this operation $\text{Aggr}_{C,V}(T)$ where C is a set of key columns and V a set of functions performed on the value columns. The two steps of the operation executing $\text{Aggr}_{k_1,k_2,sum(v_1),max(v_2)}(R)$ are shown in Figure 2.1. First, rows with the same value in both of their k_1 and k_2 columns are grouped together, as shown in Figure 2.1(f). Next, rows in the same group are accumulated using the SUM() function for column v_1 and the MAX() function for column v_2. The final result of the aggregation operator is shown in Figure 2.1(g).

Functions used in aggregation are either algebraic or holistic [GCB+97]. For *algebraic aggregation functions*, a fixed-size result can represent a sub-aggregate. Therefore, they can typically be computed incrementally, i.e., a new row belonging to a group can immediately be accumulated into that group without requiring additional storage space for the group. SUM(), MIN(), MAX(), and AVG() are all algebraic aggregation functions. However, in the case of *holistic aggregation functions*, there is no constant bound on the size of the storage needed to summarize a sub-aggregate. MEDIAN() is a holistic aggregation function. If an aggregation operator involves only algebraic functions, the grouping and accumulation steps can be (and typically are) combined in order to reach higher performance.

2.2.5 Join

The *equi-join operator* pairs rows from two tables T_1 and T_2 that agree in a common set of columns, called the key columns [GUW09]. We denote this operation $\text{Join}_C(T_1, T_2)$ where C is a set of columns present in both T_1 and T_2. The result of $\text{Join}_{k_1,k_2}(R, S)$ is shown in Figure 2.1(h). The first row of the result is made by pairing row 1 of table R and row 3 of table S as they both have the same key ($k_1 = 21, k_2 = 34$). The second row of the result is similarly obtained by pairing row 3 of table R and row 3 of table S. Each row of the result contains both sets of key and payload columns of each of the pairing rows. We use the

notation $T.x$ to refer to column x in table T. Given that both sets of key columns in the result are equal (in **equi**-joins), a projection operator typically follows the join (implicitly) in order to get rid of the redundancy. The rows involved in a pairing are said to *match* and sometimes called the *matching rows*.

We define an $M : N$ *equi-join* with $M, N \in \mathbb{N}$ as an equi-join where the size of the cross product of the matching rows from both input tables is upper bounded by $M \times N$, with at most M rows from the first table and N rows from the second table participating in the cross product. We use the term join to refer to equi-joins in the remainder of this thesis.

We refer the interested reader to [GUW09, Dat03, RG03] for more information on database operators and their formal definitions based on relational algebra. These references also explain the material in the remainder of this chapter in greater detail and with more examples.

2.2.6 Operator Classification

Database operators can be classified based on their complexity and execution model as follows [Gra93]:

- ***Streaming operators*:** The operators belonging to this category can be executed on a continuous stream of rows on the fly, i.e., rows do not need to be buffered or materialized from when they are first read until the output is produced. Streaming operators can be easily pipelined: load a row, apply operator 1, apply operator 2, ..., apply operator n, store result. Examples include the projection and filter operators.
- ***Pipeline-breaking operators*:** The operators belonging to this category need to read at least one of their input tables completely before being able to produce an output. The input tables(s) that need to be consumed completely "break the pipeline". These operators often perform multiple passes over their input (e.g., for sorting) or the insertion and probing of their input in special data structures such as hash tables or binary search trees. Examples include the sort, aggregation, and join operators.

2.3 DATABASE QUERIES

Database systems allow users to query and modify the data using a query language. A *query* is a question about the data. *Structured Query Language (SQL)* is the most popular query language for relational databases [GUW09].

A simple SQL query is shown in Figure 2.2. Let us briefly describe this query and how it relates to the database operators defined in Section 2.2. Line 2 of the query indicates that the inputs to this query are tables R and S (defined in Figures 2.1(a) and 2.1(b)). We use the notation $T.x$ to refer to column x in table T or a table derived from T. Line 3 defines a filter predicate on table R. We name the table resulting from this operation $F :=$ Filter$_{v_1<5}(R)$. Line 4 defines equality predicates for joining tables S and F. We call the result $J :=$ Join$_{k_1,k_2}(S, F)$. Line 1 requires the result to contain 3 columns $R.k_1, R.k_2$, and $SUM(R.v_2)$. We therefore need to remove all the other columns ($S.k_1, S.k_2, R.v_1, S.v_1$, and $S.v_2$) obtained after the join. The result is named $P :=$ Project$_{R.k_1,R.k_2,R.v_2}(J)$. Lines 5 and 1 together define an aggregation operator using k_1 and k_2 as grouping key columns,

```
1  SELECT R.k_1, R.k_2, SUM(R.v_2)
2  FROM R, S
3  WHERE R.v_1 < 5 AND
4        R.k_1 = S.k_1 AND R.k_2 = S.k_2
5  GROUP BY R.k_1, R.k_2
```

Figure 2.2: Example SQL Query on tables R and S from Figure 2.1.

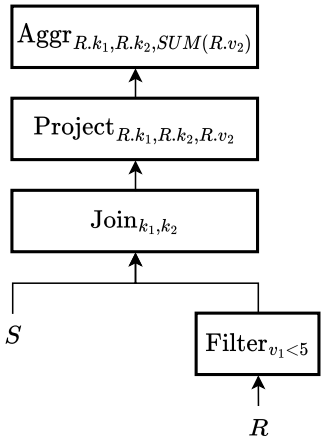


Figure 2.3: Logical query plan of the SQL query in Figure 2.2.

and $R.v_2$ as a single value column accumulated using the SUM() function. We call the resulting table $A := \text{Aggr}_{R.k_1,R.k_2,SUM(R.v_2)}(P)$. Table A is the final result of the query.

When a query is submitted to a database system, it is parsed and then transformed into a *logical query plan* which is an execution tree of database operators. The logical query plan of our running example is shown in Figure 2.3. The nodes of the tree are database operators. The leaves are the input tables. The logical query plan typically goes through a series of optimization steps transforming it (e.g., by removing redundant joins) into a more efficient execution tree [RG03].

The optimized logical query plan is then converted into a *physical query plan* by assigning to each one or combination of its nodes one or more algorithm(s) that could be used to evaluate the corresponding database operator(s). Indeed, there are multiple ways to evaluate a given logical query plan and this step selects an optimal implementation [Gra93, GUW09]. We describe alternative methods to implement pipeline-breaking database operators later in Chapter 4.

Once the physical query plan is generated, the *execution engine* executes it in order from the leaves up to the root [GUW09]. In push-based query processing, each operator generates a stream of rows with a common schema that is fed into the parent operator. The rows produced by the root operator constitute the result of the entire query.

2.4 IMPACT OF ACCELERATION

In general, database acceleration targets the execution engine of a database system. Indeed, for large databases that benefit the most from acceleration, the cost of executing

a physical query plan dwarfs that of other processing stages such as parsing and plan generation. Therefore, most accelerators propose a set of new and improved operator implementations [FMH+20, OLG+05] with the aim of reducing the cost (time, memory usage, etc.) of executing the physical query plan.

Based on their goals and requirements, database system architects often focus their efforts on the acceleration of a specific set of operators. Given the performance-oriented nature of our research, we assign higher priorities to operators whose acceleration has the highest impact on a query engine's overall performance. To this end, we use Amdahl's law [Amd67, Red11]: "The overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used". In database-friendly terms, higher query execution speedup is achieved by accelerating the most time-consuming operators in the query. We must therefore prioritize **popular** operators that make up a **large part of the execution time** of queries. Pipeline-breaking operators fit the bill. Indeed, pipeline-breaking operators are used in most analytical queries. Moreover, compared to streaming operators, they have a more complicated and compute- and memory-intensive execution flow resulting in higher execution times. In fact, they typically constitute most of the execution time of a query [DBRU20]. Based on these facts, we orient our research towards the acceleration of the 3 important pipeline-breaking operators discussed in this chapter, namely **sort**, **aggregation**, and **join**. Even though our acceleration platform also supports projections and static filtering, they remain secondary to our work.

3

BACKGROUND ON FPGAS

3.1 FPGA

3.2 FPGA Design Flow

3.3 Implementation Quality Metrics

3.4 FPGA Cards

3.5 Benefits of Using FPGAs

3.6 Challenges of Using FPGAs

In this chapter we present an overview of FPGA technology. We start by describing the structure of FPGAs and how they can be programmed to run digital circuits. We then present the typical FPGA design flow with a brief summary of FPGA compilation/synthesis. FPGA cards are described next. We finish this chapter with a discussion on both the advantages and the challenges of using FPGAs.

3.1 FPGA

Field-programmable gate arrays (FPGAs) are computer chips with reconfigurable resources allowing them to run various circuits after production. At its core, an FPGA is a grid of logic resources (logic elements), block RAMs (BRAMs), digital signal processors (DSPs), and input/output (IO) elements connected through a programmable interconnect [Int22]. Figure 3.1 depicts the simplified structure of an FPGA.

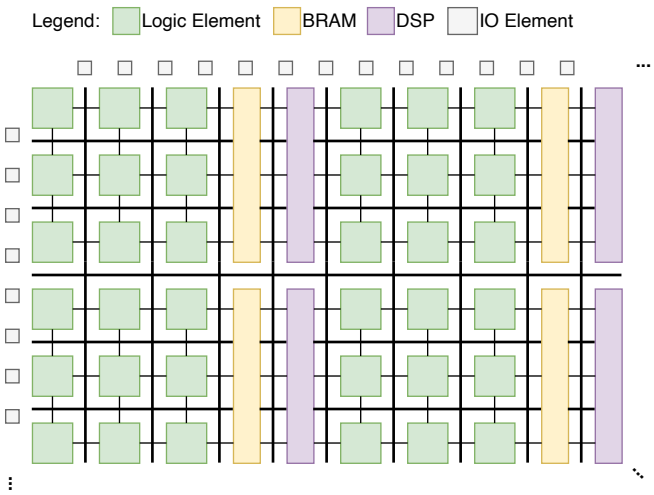


Figure 3.1: Simplified Structure of an FPGA.

3.1.1 Logic Element

Logic resources are composed of lookup tables (LUTs), flip-flops, and some basic logic circuits such as multiplexers. *LUTs* are small SRAM-based memories whose contents can be (re-)configured [TW13]. They are arguably the most important building blocks of FPGAs, allowing them to mimic logic gates in order to run (stateless) combinational circuits. A *flip-flop* is a bistable circuit used to store a single bit of data. A collection of flip-flops is often called a *register* [HH21]. They enable FPGAs to run (state-full) sequential circuits as well.

Logic resources are typically organized as logic elements shown in Figure 3.2 [Cor22d]. On the left-hand side, a 2-input LUT accepts two address bits as input and returns the contents of the memory at the specified address as its output. The numbers inside the LUT are in binary format. The output of the lookup table is connected to a flip-flop. A multiplexer (Mux) then chooses either the output of the LUT, or the (delayed) output of the flip-flop as the output of the logic element. The contents of the LUT and the configuration of the flip-flop (i.e., when to store its input as its output) and the multiplexer (i.e.,

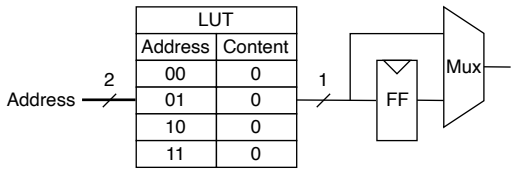


Figure 3.2: Simplified structure of a logic element.

which input to forward as its output) are all programmable, allowing the logic cell to run a simple logic circuit.

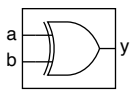
Let us illustrate the inner workings of a logic element using an example. Consider a single-bit half adder with input bits a and b, and output bit y. The truth table for the adder is shown in Figure 3.3(a). A *truth table* defines the output of a function, for every combination of its inputs. For instance, the second row of the truth table indicates that adding $a = 0$ and $b = 1$ results in 1.

The adder can be represented using a XOR gate shown in Figure 3.3(b) [Wak21]. An application-specific integrated circuit (ASIC) would typically implement the XOR gate using complementary metal-oxide-semiconductor (CMOS) transistors. Once the transistors are printed onto the IC substrate, the ASIC will only be able to perform single-bit addition.

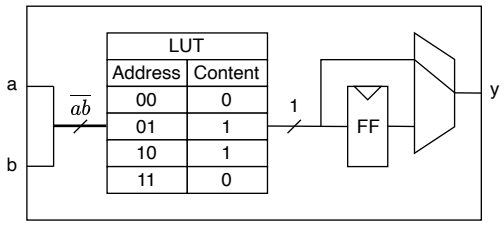
The FPGA-based implementation is shown in Figure 3.3(c). The LUT content is programmed with values $[0, 1, 1, 0]$. Note that the address column is shown for the sake of readability and does not need to be programmed into the LUT. The input address bits of the truth table are connected to input ports a and b. The output of the LUT is connected through the multiplexer to the output port y, effectively bypassing the flip-flop. Indeed, given the combinational nature of our single-bit adder, the flip-flop is not needed. Let us use test vector $a = 0, b = 1$ to explain how the LUT works. In this scenario, the address input to the LUT is $\overline{ab} = 01$. Note that we use the over-line notation to indicate concatenation, rather than Boolean negation. The LUT retrieves and outputs the content at address 01, which is 1. The value 1 is then forwarded all the way to the output of the logic element. The acute reader may have noticed that the contents of the LUT are similar to those in the truth table. Producing the truth table of a circuit is a common way to figure out the configuration bits of a LUT equivalent to that circuit. Note that contrary to an ASIC implementation, the contents of the LUT (and the configuration of the Mux) can be changed at a later time allowing the logic element and, by extension, the FPGA to execute a new functionality.

a	b	y
0	0	0
0	1	1
1	0	1
1	1	0

(a) Truth table



(b) Gate representation



(c) Logic element implementation

Figure 3.3: Alternative representations and the FPGA implementation of a single-bit half adder $y = a + b$: (a) Truth table, (b) XOR gate representation which can be implemented on ASICs using CMOS transistors, (c) FPGA implementation using a logic element.

In order to keep up with application demands, modern FPGAs have more sophisticated logic resources to handle complex circuits. Intel® Stratix® 10 FPGAs used in our benchmarks in this thesis have adaptive logic module (ALMs) each containing two **adaptive** LUTs (ALUTs) and 4 registers. An ALM can be used in different modes including one where it runs 2 separate 4-input circuits (like having two 4-input LUTs) and another where it runs one large 8-input circuit [Cor22a]. This amount of flexibility presents both performance and resource optimization opportunities for FPGA tools. Our Stratix® 10 FPGA, code-named $1SX280HN2F43E2VG$ from the SX 2800 product line, has $933,120$ ALMs that together with a programmable interconnect (allowing arbitrary connections between the ALMs) enable it to support relatively large circuits used in data center applications.

3.1.2 Block RAM (BRAM)

BRAMs are dense blocks of SRAM typically used for storing chunks of data or configuration bits. Somewhat similar to CPU caches, they provide high-throughput, low-latency access to a small amount of on-chip data. Unlike CPU caches, FPGA BRAMs are fully configurable. Indeed, FPGAs have many small columns of BRAM that can be accessed independently or combined to make bigger blocks of memory. As an example, the Stratix® 10 FPGA used in our benchmarks has $11,721$ BRAMs, called *M20Ks* [Cor18a]. Each M20K can be configured in depth $\times$ width as a 512×40, 1024×20, or 2048×10-bit memory [Cor22b]. They can of course also be combined to produce deeper and/or wider memories. The BRAMs on our FPGA can collectively store around 30 MB of data.

A BRAM is accessed using read and write ports. Given an address, a *read port* can provide the contents of the BRAM at that address within a fixed (typically) 1 clock cycle latency. Given an address and a value (called *data*), a *write port* will write the value to the address within a fixed (typically) 1 clock cycle latency. The write port also has a single-bit (Boolean) enable input indicating when its address and data inputs are valid, otherwise it would be continuously writing the (potentially garbage) values at those inputs to the BRAM. BRAMs on modern FPGAs can run at up to hundreds of megahertz with all of their ports beings used in parallel every clock cycle [TW13].

The BRAMs (i.e., M20Ks) in our Stratix® 10 FPGA support 3 modes of operation [Cor22c]:

- ***Simple dual-port***: The BRAM exposes 1 read port and 1 write port.
- ***True dual-port***: The BRAM exposes 2 read-write ports. A *read-write* port can be used to either read or write data, but not both at the same time.
- ***Simple quad-port***: The BRAM exposes 2 read ports and 2 write ports.

For each operating mode, all the ports can be accessed simultaneously. For instance, a BRAM used in quad-port mode allows 2 separate circuits to both read from and write to it at the same time, every single clock cycle. Note that when BRAMs are combined, the resulting larger memory will also support the same operating modes. We commonly use the expression "X BRAM" (e.g., X = simple dual-port) to refer to a BRAM used in mode X.

Given the reconfigurable nature of FPGAs, the line between memory and logic resources is blurry. For instance, LUTs in logic elements can be used as fast distributed memories [TW13]. BRAMs can also be used for computing logic functions of many variables [WIA14]. Therefore, when describing an implementation, designers typically mention the underlying technology used for memories (e.g., **BRAM-based** on-chip memory). If not, "on-chip memory" is often assumed to use BRAM technology.

3.1.3 Digital Signal Processor (DSP)

Although FPGAs' logic resources can run all types of digital circuits, there are complex yet frequently used circuits that warrant an efficient gate-based implementation. These are often arithmetic units used in digital signal processing applications such as fast Fourier transform, convolution, and filtering. Our Stratix® 10 FPGA contains $5,760$ variable-precision DSPs capable of performing fixed- and floating-point arithmetics [Cor18a, Cor21c].

3.1.4 IO Element

IO elements connect the FPGA to the outside world. They are responsible for providing a physical link between the circuits running on the FPGA and its pins. Those pins are used to connect the FPGA to a host server (e.g., through PCIe), DRAM, network, etc.

3.1.5 Programmable Interconnect

The programmable interconnect provides configurable routing connections between the FPGA resources. It allows the FPGA to run complicated circuits spanning multiple logic elements, BRAMs, and DSPs. The programmable interconnect consists of a mesh of wire tracks laid out vertically and horizontally, and connected through switch boxes. FPGA's resources are then connected to these routing tracks via connection boxes [FMM12]. This architecture is shown in Figure 3.4. A programmable connection box can connect the inputs/output of a logic element to any of its adjacent routing tracks. The red dots in our figure indicate a connection. A programmable switch box is a crossbar switch, capable of connecting arbitrary sets of wires that cross it. A red line in the figure is a connection between two routing tracks.

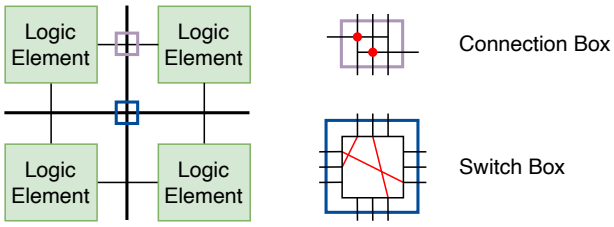


Figure 3.4: Simplified structure of an FPGA's programmable interconnect. Connection boxes provide configurable routing between the logic elements and the programmable interconnect. Switch boxes define the topology of the interconnect.

3.2 FPGA DESIGN FLOW

Application design for FPGAs is a semi-automated, multi-step process. It consists of a front-end where the designer develops and functionally verifies a circuit, and a back-end transforming the circuit into a bitstream that can be programmed onto an FPGA. The designer is typically involved in all steps of the process and may need to iterate over one or a series of steps in order to obtain the desired result. The automated steps of the design process are performed by computer-aided design (CAD) tools mainly provided by the FPGA vendors [CCP06]. A typical FPGA design flow is shown in Figure 3.5.

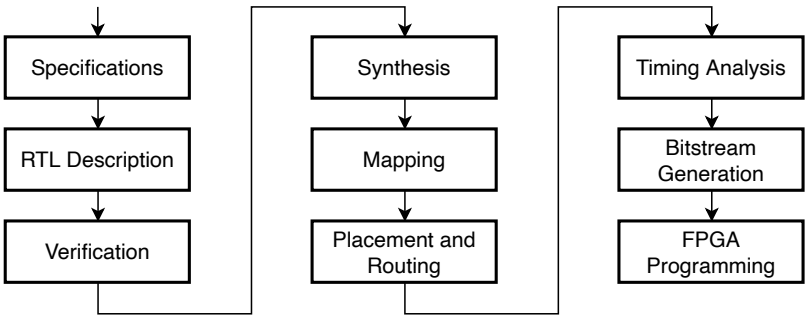


Figure 3.5: The steps in the design process for a digital system on an FPGA.

3.2.1 Specifications

Like most designs, the process starts with a list of well-defined specifications (or "specs" for short):

- ***Functional specifications*:** They define the behavior of the system.
- ***Performance specifications*:** They define the target performance of the system. For instance, a specification may require the design to process 1 database row per clock cycle.
- ***Timing specifications*:** They define the expected frequency of each part of the design.
- ***Area specifications*:** They define the amount of space the design is allowed to occupy on the target FPGA. This often translates into the amount of resources the FPGA can allocate to the design.
- ***IO specifications*:** They define a mapping between the IO pins of the FPGA and the inputs and outputs of the circuits running on the FPGA.

Unfortunately, FPGA design is generally not an agile process. A minor change in the spec might require months of going through the remaining steps of the design. Therefore, it is crucial for the specifications to be both fully- and well-defined before moving onto the next step.

3.2.2 RTL Description

Based on the specifications, the design's (micro)architecture can be developed. Basically, if the specs are the "what?", the architecture is the "how?". It defines a circuit (often split into a set of smaller inter-connected circuits) that performs the required functionality while respecting the constraints defined in the specs.

FPGA CAD tools expect a register-transfer level (RTL) description of the design as input. *RTL* is a design abstraction which defines a circuit as a set of registers and how digital signals are transformed between those registers [Vah10]. There are two common methods to obtain the RTL description of an architecture: using hardware description languages

(HDLs) or high-level synthesis (HLS). Both methods are described later in this Section. We provide a comprehensive comparison between the two in Chapter 6.

Along with the RTL description, designers must also produce design constraints based on the specifications. *Timing constraints* define the expected frequency of the design. *IO constraints* specify the connections between the pins of an FPGA and the inputs and outputs of the circuit running on that FPGA. Design constraints are well-defined machine-readable versions of some of the specifications. FPGA CAD tools optimize their efforts trying to satisfy these constraints.

RTL from HDLs

HDLs define an architecture using a hierarchy of modules directly implemented at the RTL abstraction. A *module* is a circuit with a well-defined interface (i.e., inputs and outputs) and a structural or behavioral description:

- ***Structural description***: It provides a description of the module in terms of its constituent modules, and the connections between them.
- ***Behavioral description***: The module is described by defining its behavior using the RTL abstraction.

In many cases, modules are defined using a combination of the two approaches. Indeed, even simple modules typically contain some primitive modules (e.g., adders, comparators) that the CAD tools can recognize and automatically implement. However, modules lower down the hierarchy tend to be mostly behaviorally defined; and those near the top structurally defined.

Examples of traditional HDLs include Verilog, SystemVerilog [IEE18] and VHDL [IEE19]. They are supported by all major CAD tools for both synthesis and functional verification.

Modern HDLs such as Chisel [BVR+12] introduce years of advances in software engineering (e.g., object-oriented programming, design patterns, etc.) into HDLs. They provide the designer with advanced circuit generation capabilities and higher flexibility, all while reducing the design time. These features make it easy to customize an architecture for a specific FPGA and workload. Chisel code is written in the feature-rich Scala programming language [Com22], and then compiled into Verilog before being fed into FPGA CAD tools. However, Chisel remains an RTL language as it defines specific hardware constructs (e.g., modules, registers, wires) in Scala that are equivalent to the ones used in other HDLs. As such, it provides equivalent performance, resource utilization, and achievable frequency compared to traditional HDLs [ANS+14].

RTL from HLS

HLS tools allow designers to implement an architecture using familiar software programming languages such as C/C++ [Int18a, Xil21] or even Java [Miy16]. HLS tools enable software developers to develop for FPGAs. They also accelerate hardware development by allowing designers to use high-level languages instead of an RTL description to both implement and verify their architectures. HLS tools typically compile the language of the tool into a traditional RTL language such as Verilog or VHDL. The latter is then fed into FPGA synthesis tools for further processing [DZS13]. We provide a lot more details on HLS tools, and in particular Intel® FPGA SDK for OpenCL, in the context of database system acceleration in Chapter 6.

3.2.3 Verification

Once a design is running on an FPGA, debugging it involves creating new circuits responsible for monitoring and probing its internal connections. This can become an extremely lengthy and tedious task. Hardware designers are therefore highly incentivized to detect and correct bugs and issues as early in the design process as possible. **Verification** is the process of comparing the behavior of the RTL description of a design, against a high-level model of the design often derived directly from its specifications [Wak21]. The latter is called the *golden reference model*.

The process is carried out by a simulator. Given the RTL description of a circuit, a golden reference model, and a set of test vectors, the simulator drives the inputs of the circuit and the golden reference model using the test vectors and checks their outputs against each other [Syn21]. Simulators can be very accurate. For instance, if need be, they can take into account the electrical properties of an FPGA's logic and memory resources.

If the simulator detects a bug, the designer must review and revise the RTL description, moving a step back in the design flow. Otherwise, the synthesis process can begin.

3.2.4 Synthesis, Mapping, Placement, and Routing

Synthesis transforms the RTL description of a circuit into Boolean functions, i.e., generic gate-level components. *Mapping* maps those functions onto the resources (logic elements, BRAMs, DSPs, etc.) of the target FPGA technology, resulting in a technology-specific circuit. *Placement* consists of assigning the components of the technology-specific circuit to the actual physical resources of the FPGA. *Routing* determines the routes to connect various components [Chu08].

After synthesis and mapping, the resource and area utilizations of the design can be roughly estimated. After placement and routing, their exact values will be known. If the design is larger than expected, the designer might need to go all the way back to the RTL description step to change the architecture or implementation of the design. If this does not work, a change in the specifications might be necessary. However sometimes, simply fine-tuning the tools might solve the problem [HH21]. Design constraints defined during the RTL description step are used here to guide the tools in their search for an optimal solution. For instance, a very high target frequency will instruct the tools to maximize their efforts.

The 4 steps described in this section are often collectively called "FPGA synthesis", "synthesis", or "compilation". Synthesizing large circuits on modern high-performance FPGAs can take tens of hours.

3.2.5 Timing Analysis

Timing analysis studies the physical layout of the circuit inside the FPGA in order to determine its timing properties. The *timing properties* of a circuit include signal delays through its paths and the maximum frequency at which it can run. They are compared against the design's timing constraints derived from its specifications. In case one or more constraints are violated (e.g., the design is too slow), the designer might need to move all the way back to the RTL description step in order to pipeline or even redesign parts of the architecture [HH21].

3.2.6 Bitstream Generation and FPGA Programming

Bitstream generation produces a bitstream according to the physical layout of the design on the FPGA. A *bitstream* is a programming file that specifies the configuration of the individual FPGA resources, in order to run the desired circuit. For instance, it defines the contents of the LUTs, the configuration of the interconnect switch boxes, and the initial contents of the BRAMs. *FPGA programming* consists of downloading the bitstream onto the FPGA. For large FPGAs, this may take seconds to perform [Chu08]. *FPGA image* is another common name for a bitstream.

3.3 IMPLEMENTATION QUALITY METRICS

In software design, the quality of an implementation is often measured by its **performance** and **memory utilization**. These metrics remain valid for FPGA implementations. In addition, FPGAs bring two more dimensions to quality evaluation:

- ***Maximum achievable frequency*:** It indicates the maximum frequency (in MHz) the design can run at. The frequency of the implementation is typically set to this value in order to reach maximum performance.
- ***Resource utilization*:** This is a measure of the amount of various FPGA resources used by the implementation. Resource utilization reports vary depending on the FPGA technology. Intel® Stratix® 10 FPGAs provide logic utilization, BRAM utilization, and DSP utilization values in their reports. The units of measurement are the number of ALMs, M20Ks, and variable-precision DSPs used by the design, respectively [Cor17].

3.4 FPGA CARDS

FPGAs are computer chips that require a set of peripherals for receiving, processing, and transmitting data. These peripherals are often placed together with the FPGA on an FPGA card, board, or development kit. Figure 3.6 shows an example FPGA card. The FPGA is placed in the center. It is connected to 2 DDR slots, hosting dual in-line memory modules (DIMMs). The FPGA can access the DRAMs either independently (as channels 0 and 1) or together using memory interleaving techniques [Wik22d]. We use the term *FPGA RAM* to refer to DRAM placed on the FPGA card and directly connected to the FPGA. The two DDR-T slots enable support for non-volatile (persistent) memory [Cor21a]. The peripheral component interconnect express (PCIe) and ultra path interconnect (UPI) are used to connect the FPGA to other FPGAs or CPU systems [BAS04, Mul19]. The quad small form-factor pluggable (QSFP) cages are typically used for connecting the FPGA to a network. These peripherals are not present on all FPGA cards. The one we use in our benchmarks has a PCIe Gen3 connector to communicate with a host server, and 4 DDR4 slots each supporting 8 GB of DRAM [Cor19a]. There are various ways to integrate an FPGA card into a server. The server is then called the *host*. This is discussed in length in Chapter 4.

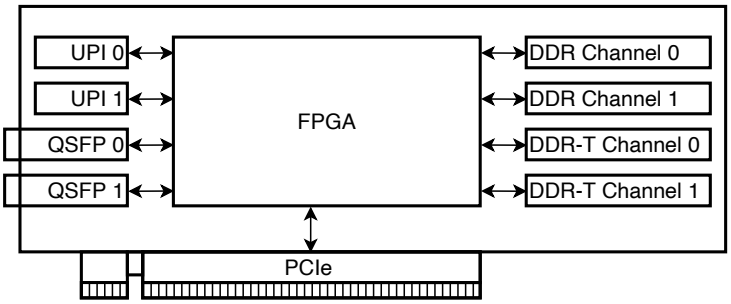


Figure 3.6: Example FPGA card, based on the Intel® Stratix® 10 DX FPGA development kit [Inc19].

3.5 BENEFITS OF USING FPGAS

The unique design of FPGAs provides them with features missing in traditional CPUs. They offer designers unique opportunities to accelerate their applications. In this section, we enumerate a few of these features. We use the example of hash table insertion and probing to illustrate these features. A *hash table* is a data structure that stores `(key,value)` pairs in its entries called *buckets*. The hash value of a `key` determines the bucket to which it belongs. Hash-tables are commonly used in database systems [GUW09] and database system accelerators [FMH+20]. In our example, we require the hash table to be stored in on-chip memory in order to achieve fast insertion and probing performance.

Let us start with the benefits of using BRAMs, a key component of FPGAs:

- **Dedicated memory:** BRAMs used in an application are entirely dedicated to that application. In contrast, a CPU cache is shared among applications and runs the risk of being (partially) flushed at every context switch. In our running example, BRAMs can be used to store the hash table for fast and exclusive access. Note that dedicated memories are only beneficial when they are relatively frequently accessed by the application. In other cases, a shared memory system like the CPU cache may be a better option.

- **Configurable memory:** The width and depth of BRAMs can be configured to exactly match the needs of an application. This not only avoids wasting precious on-chip memory but also enables more meaningful and efficient accesses avoiding alignment-related issues. In our running example, we can build a BRAM-based memory with the exact dimensions of our hash tables. A single access to this memory will then return exactly one bucket of the hash table.

- **Predictable low-latency memory:** Most BRAM configurations can be accessed within a fixed single clock cycle latency. This allows circuit designers to orchestrate the execution flow of an algorithm in a precise, cycle-accurate manner. It also reduces stalls caused by data dependencies. In our running example, during an insertion, the hash table can report within a single clock cycle whether the target bucket is full or empty so the algorithm can quickly decide what to do next.

- **Parallel access:** BRAMs often support simultaneous accesses to their contents. For instance, M20Ks in simple quad-port mode can perform 2 reads and writes at the same time (see Section 3.1.2). In our running example, this enables parallel inserts and probes; effectively increasing the throughput of the hash table operations.

A circuit is "on" when electrical current flows through it. An FPGA-ran circuit is no exception to this rule. As long as the FPGA is connected to power, all of its circuits will run in parallel. This, together with the sheer amount of configurable resources on the FPGA provide:

- **Fine-grained parallelism:** This enables designers to achieve higher performance through the inherent parallelism of small sub-circuits on the FPGA. In our running example, the circuit that performs the insertions and the one that does the probings can run at the same time, increasing the throughput of the hash table operations.
- **Coarse-grained parallelism:** This allows designers to achieve higher performance through duplication. *Duplication* involves instantiating multiple copies of the same circuit, to increase the overall performance of the design. In our running example, we can have multiple insertion and probing circuits, each working on a partition of the hash table. The distributed hash table can then achieve much higher peak operation throughput.
- **Fine-grained pipelining:** Pipelining is the process of dividing a circuit into multiple sequential stages, separated by memory elements such as registers. Each stage can work on a different instance of the problem the circuit is made to solve. This typically results in an increase in the maximum achievable frequency. In our running example, we can for instance divide insertions into 3 stages: (1) calculate the hash of the key, (2) read from the hash table, (3) write to the hash table.
- **Coarse-grained pipelining:** This is the process of pipelining large (sub-)circuits of the design. The resulting pipeline typically coincides with a natural division of a data processing pipeline into smaller tasks. As an example, consider a data processing pipeline that requires the collected data to be merged, filtered, timestamped, and inserted into a hash table. Each of these tasks, implemented as a circuit on the FPGA, becomes a stage in the pipeline. Data is passed from one stage to the next without being buffered in on-chip or off-chip memory. These stages work on different parts of the data in parallel. In a nutshell, coarse-grained pipelining reduces memory usage and improves performance.

3.6 CHALLENGES OF USING FPGAS

Despite the many benefits offered by FPGAs, using them requires developers to overcome a few challenges. FPGA design is an involved process. It demands a deep understanding of digital design and architecture, and experience with an arsenal of CAD tools such as simulators and synthesizers. Moreover, the large design space provided by FPGAs might be hard to navigate for new and experienced developers alike. The abundance of implementation opportunities requires tremendous effort to reach an optimal solution.

Another challenge in FPGA development is the lack of agile methodologies. Consider an FPGA-based data processing pipeline. Imagine that a post-development change in the specifications requires the addition of a new stage to this pipeline. This seemingly simple task raises the following concerns unique to FPGAs:

- **Performance matching:** The performance of the new stage must match that of the existing ones. Otherwise, it may slow down the whole pipeline.

- **Resource utilization:** Adding the new stage may require more resources than what the target FPGA can offer.

To avoid these issues, designers must plan out their architecture as much as possible in advance.

A final challenge in FPGA development is the large number of quality metrics that must be considered by the designers. Indeed, in addition to the traditional ones such as performance and memory utilization, FPGA developers must also consider the achievable frequency, resource utilization, and sometime even the power utilization of their implementation to make sure they conform to the specifications. This adds a lot more complexity to the design process.

4

RELATED WORK

4.1 Summary of Related Work

4.2 Platform Type

4.3 Implementation

4.4 A Note on Quantitative Performance Comparisons

The rise of big data, cloud computing, and the ever so growing demand for higher performance in data processing have led to a steady stream of research in database system acceleration. In this chapter, we present and discuss some of the key academic and industrial research and contributions in this domain. The goal is to identify distinct properties of current accelerators that will ultimately help guide us in our own research in Part II of this thesis.

4.1 SUMMARY OF RELATED WORK

Multicore CPUs, GPUs, and FPGAs constitute the three most popular categories of devices used in high-performance data processing platforms. They help classify current research in database system acceleration. Platforms using multicore CPUs benefit from core-level parallelism, modern memory and cache systems, application specific instructions, and architectural extensions (e.g., SIMD) [CR07, AKN12, KSC+09, BBC+12]. GPU-based platforms take advantage of the high bandwidth memory and the tremendous amount of compute power and parallelism provided by GPUs to accelerate database workloads [BHS+14]. FPGA-based platforms exploit the wide array of configurable resources (e.g., logic elements, BRAMs) present on FPGAs to design custom circuits optimal for query processing [UIO15, HANT15, WFS+19, ABW+16].

Given the large body of work present in each category and the nature of our research, this chapter focuses on database query acceleration on FPGAs alone. More specifically, we concentrate on past research related to the acceleration of the three high-impact pipeline-breaking database operators (i.e., sort, aggregation, join) central to our work (see Chapter 2). We refer the interested reader to [FMH+20] for an excellent survey on other database operators and database acceleration in general.

Table 4.1 presents and characterizes some of the most prominent past research related to our work. For each entry in the table, we present the following attributes:

- ***Platform type***: This attribute describes how the FPGA (card) is integrated into the data processing system. Platform types are presented in Section 4.2.
- ***Operator support***: It is a list of pipeline-breaking operators supported by the platform. Pipeline-breaking operators were presented in Section 2.2.
- ***Language***: This attribute specifies the (HLS or RTL) programming language used to describe the FPGA design. FPGA programming languages were described in Section 3.2.2.
- ***Implementation***: This attribute describes how the FPGA platform implements the database operators. In Section 4.3, we present and discuss various methods used to implement database operators on FPGAs.

In most of the platforms presented in our table, a host (i.e., CPU-based server) has the responsibility to schedule, configure, and launch the FPGA. The FPGA then executes the operators. However, in some cases operator execution is shared between the FPGA and its host. For instance, some authors propose sorting small chunks of data on the FPGA, later to be merged [STM+13, STM+15, SMT+14] or joined [CP16] by the CPU in order to execute the sort or join operators, respectively. In [DZT13], the FPGA performs aggregation but only on data pre-sorted by the CPU. In another platform, the FPGA performs hash-based aggregation but relies on the CPU to deal with hash collisions, should any

occur [WIA14, WTA13]. To deal with these intricacies, the operator support column of each entry in Table 4.1 indicates the capabilities of the entire corresponding acceleration stack (FPGA and CPU) rather than the FPGA alone.

It is also worth pointing out that some of the platforms in the table only support the sliding-window versions of pipeline-breaking operators. Given a stream of rows, a *sliding-window operator implementation* continuously executes the operator on a finite sliding-window over the rows [NSJ13, THSW15]. Sliding-window operators are commonly referred to as "streaming operators" [JMS+08, BDD+10], a term we shall **not** use in order to avoid any confusion with streaming operators defined in Section 2.2.6. Despite our own work targeting (non-sliding-window) pipeline-breaking database operators, we study sliding-window operators as they provide a great deal of insight in fast query processing.

The last two entries in the table refer to industry-developed platforms. IBM Netezza and Swarm64 are other notable products of industry research, omitted from our table due to their lack of support for pipeline-breaking operators. IBM Netezza supports data decompression, projection, and filtering [Fra14]. Swarm64 supports compression, decompression, and filtering [Int18b, Xil19].

In what follows, we present a brief description of various platform types and operator implementations. They are two important characteristics of FPGA-based database accelerators, unexplored in previous chapters.

Reference	Platform Type	Operator Support			Language	Implementation
		Sort	Aggr.	Join		
[UIO15]	Coprocessor			✓	Verilog	Reconfiguration: sort-merge join and hash join
[WPC+16]	Accelerator Card	✓	✓	✓	OpenCL	Reconfiguration: sort-merge, sort-based aggregation, and hash join
[HSM+13]	Accelerator Card			✓	n.d.	Hash join
[STM+13, STM+15]	Accelerator Card	✓			n.d.	Sort-merge
[HANT15]	Accelerator Card			✓	n.d.	Hash join
[WIA14, WTA13]	Smart Storage		✓		Verilog/VHDL	Hash-based aggregation
[NSJ13]	Network Processor			✓	n.d.	Sliding-window nested-loop join
[YKO+14]	Smart Storage		✓		VHDL	Hash-based aggregation
[MTA09b, MTA10]	Network Processor		✓		VHDL	CAM-based aggregation
[WFS+19]	Smart Storage		✓		n.d.	Hash-based aggregation
[CO14]	Accelerator Card	✓		✓	n.d.	Sort-merge and sort-merge join
[MTA09a]	Coprocessor		✓		VHDL	Sliding-window sort-based aggregation
[ABW+16]	Accelerator Card		✓		n.d.	Hash-based aggregation
[GS21]	Network Processor		✓		MaxJ HLD	Sliding-window hash-based aggregation
[DZT13]	Coprocessor		✓		n.d.	Streaming aggregation on pre-sorted data
[SJT+12]	Network Processor			✓	n.d.	Sliding-window nested-loop join
[SMT+14]	Accelerator Card	✓		✓	n.d.	Sort-merge and hash join
[CP16]	Coprocessor			✓	n.d.	Sort-merge join
Xilinx Vitis GQE [Xil22b, Xil22c]	Accelerator Card		✓	✓	Vitis HLS	Hash-based aggregation and hash join
Amazon AQUA [Bar21]	Smart Storage		✓		n.d.	n.d.

Table 4.1: Past academic and industrial work on database system acceleration on FPGAs, with a focus on pipeline-breaking database operators. Cells containing n.d. indicate that the corresponding attribute has not been disclosed.

4.2 PLATFORM TYPE

Platform type defines the position and connections of the FPGA card in a data processing pipeline. There are 4 common types of FPGA acceleration platforms used with standalone database systems. They are shown in Figure 4.1.

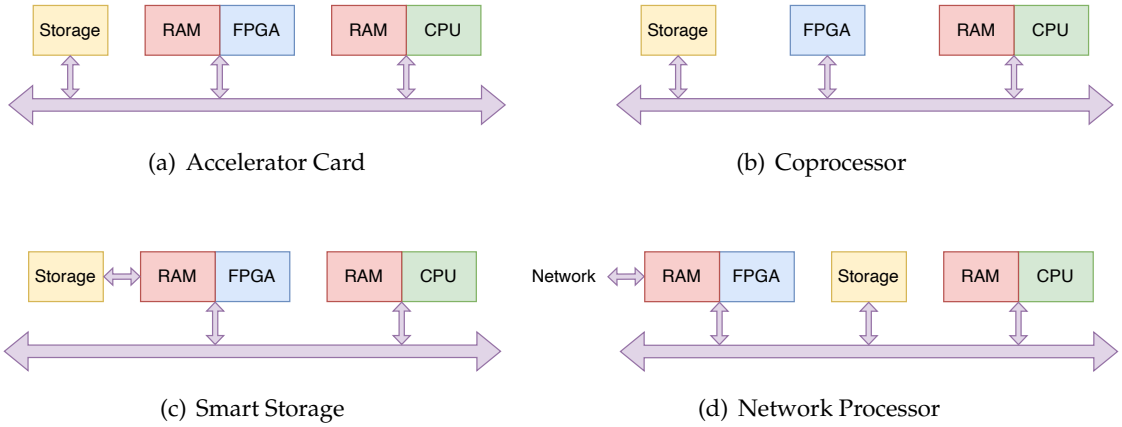


Figure 4.1: Various types of FPGA acceleration platforms used for database query processing. The CPU and its RAM together form the host (see Section 3.4). The FPGA and its RAM are placed on an FPGA card. Storage refers to permanent storage, such as hard disk drive (HDD) or solid-state drive (SSD), commonly referred to as the *disk* in database system literature.

4.2.1 Accelerator Card

In *accelerator card* platforms shown in Figure 4.1(a), the FPGA card is connected via PCIe or UPI to the host server. To execute an operator, input data is copied from the host RAM into the FPGA RAM. The FPGA then execute the operator, before copying the results back to the host memory. Note that the copying of data is not necessary anymore. Indeed, *shared virtual memory* featured in modern FPGAs gives them direct access into shared regions of the host memory [VMB19, Cor14, Sch22].

Accelerator card platforms are easy to setup as they can be simply plugged into most motherboards' PCIe slots. This has made them very popular in past research. The authors of [PHL18, WPC+16] use a DE5Net acceleration card with an Intel® Stratix® V FPGA and 4GB of RAM. The card is connected to the host via an 8-lane PCIe Gen 2 edge connector with a peak bi-directional bandwidth of 4 GB/s [Inc18]. They transfer input data over PCIe to the FPGA RAM before executing a query. The results of the query must then be transferred back to the host. In [HSM+13], an Intel® Stratix® IV GX530 FPGA card supporting an 8-lane PCIe Gen 2 connector is used for accelerating the join operator [Cor10]. The design streams 4KB database pages (containing rows) from host memory directly into the FPGA. The FPGA then decomposes the rows into 2 parts: (1) key columns relevant to the join operator are kept on the FPGA and advance through the processing pipeline, (2) value columns are stored on the FPGA RAM, later to be united with the results of the join. The authors of [HANT15] use the Convey-MX platform composed of 2 Intel® Xeon® E5-2643 processors and 4 Xilinx Virtex-6 760 FPGAs for join processing. The main memory, although entirely accessible by both the FPGAs and the CPUs, is divided into two regions connected through PCIe. The FPGAs (resp. CPUs) have fast DDR3 access to the hardware (resp. software) region of the memory. Cross accesses are slower as they must go through PCIe. Although the FPGAs can be setup as coprocessor sharing both memory regions with the CPUs, they are seemingly used by the authors as accelerator cards with their own dedicated memory. In [ABW+16], a more recent version of this setup called Convey HC-2ex is used to accelerate the aggregation operator [Con12].

4.2.2 Coprocessor

In *coprocessor* platforms shown in Figure 4.1(b), the FPGA is seen as "another" processing unit working on the same copy of data as the CPU. The CPU RAM is therefore shared with the FPGA.

The authors of [UIO15] execute joins on an accelerator card with two FPGAs: (1) a Stratix IV SE360 FPGA manages communication between the host and the FPGA Card, handling memory transactions for (2) a Stratix IV GX530 FPGA that performs the actual join. In the Xilinx XUPV2P platform used in [MTA09a], both the CPU and the FPGA are on a single chip connected to main memory through a shared bus. The CPU system consists of two PowerPC 405 cores directly wired to the FPGA fabric [Xil11b]. Much like DSPs discussed in Section 3.1.3, these are efficiently-implemented *hard IP cores* permanently instantiated on the FPGA. The authors propose multiple FPGA architectures. In their most efficient architecture, the CPU (much like a typical host system) simply configures and launches the FPGA to read, process, and write back data from and to the shared RAM. Another interesting type of coprocessor platform is based on Intel's Xeon+FPGA [OSC+11] architecture. It is used in [OSKA17] to perform arithmetic, filtering, and skyline operators. The setup consists of a two-socket server with an Intel Xeon E5-2680 CPU in one socket and an Intel Stratix V FPGA in the other. The CPU has fast (25 GB/s) DDR access to the shared main memory. The FPGA can access the memory through a slower (6 GB/s) Intel QuickPath Interconnect (QPI) link connecting the two sockets. The CPU uses a task queue to configure and launch the FPGA. IBM's Power8 CAPI provides a coprocessor type architecture similar to Intel's Xeon+FPGA platforms [SBJS15].

Note that many accelerator card platforms can also be used in coprocessor mode, making it hard to distinguish between the two. We define a coprocessor platform as one where the FPGA does not use (or simply lacks) dedicated memory.

4.2.3 Smart Storage

Smart storage platforms, shown in Figure 4.1(c), are typically used for accelerating disk-oriented database systems (see Section 2.1.2). Here, the FPGA is placed as a bump-in-the-wire device between the disk (or any other storage medium) and the host, providing two types of functionalities [FMH+20]:

- **Operator acceleration:** Some database operations can be offloaded to the FPGA for efficient near-storage processing, freeing the CPU.
- **Bandwidth amplification:** Compression/decompression can be applied by the FPGA to the data moving between the disk and the CPU, effectively increasing the bandwidth of the storage system.

This architecture allows the disk and the FPGA to act together as a "smart storage" system, capable of both storing and (to some extent) processing data.

The authors of [WIA14] use a Virtex 5 XC5VLX110T FPGA, connected via Gigabit Ethernet to the host and via SATA II to a 256 GB SSD, for executing the projection, filtering, and aggregation operators while transferring data from the SSD to the host. In [YKO+14], Stratix IV GX FPGAs are placed between an array of flash storage devices and the host. The FPGA can perform aggregations on data extracted from flash storage while it is being transferred via a 4-lane PCIe Gen 2 connector to the host. The platform used in [WFS+19]

has the host connected to a PCIe enclosure [Epr18] containing two Stratix V GX FPGAs and two Intel DC P3700 SSDs with a capacity of 800 GB each. The FPGA platform can execute the filter and aggregation operators while copying data from the SSDs to the host.

Most commercial database accelerators are smart storage platforms. IBM Netezza performs decompression, projection, and filtering on data extracted from storage [Fra14, FMH+20]. Amazon AQUA acts as a layer between permanent storage and Amazon Redshift compute clusters, performing aggregation and filtering during data retrieval [Bar21].

4.2.4 Network Processor

Network processor platforms, shown in Figure 4.1(d), have a direct connection to a network where they receive input data. After processing the input, the results are transmitted to the host. FPGA-host communication may go through network, PCIe, or other mediums. Network processor platforms are typically used for executing sliding-window operators on a stream of rows or events.

The authors of [NSJ13] transmit a stream of events over Gigabit Ethernet to the Virtex-5 XC5VLX50T FPGA onboard a Xilinx ML505 board [Xil11a]. The FPGA can perform projections, selections, and sliding-window joins on the stream of events. The result is also sent via Ethernet to a host computer. The ML505 board is also used in a similar configuration in [MTA10] for executing the projection, selection, and aggregation operators on stock data received from the network. The only difference is that the FPGA sends the results of the computation, not via Ethernet, but rather a serial line to the host. The authors of [GS21] perform Ethernet-to-Ethernet sliding-window aggregation on a Maxeler N-series ISCA (MAX4AB24B) FPGA card. The card provides its Intel Stratix V (5SGXAB) FPGA with 10 Gigabit Ethernet connectivity for both receiving input data and transmitting aggregation results.

4.3 IMPLEMENTATION

Database operators can be implemented in many ways. Here, we focus on FPGA-based implementations. They can achieve great performance by benefiting from the wide array of resources provided by FPGAs, yet must face a few challenges unique to their utilization.

In Section 4.3.1, we present simple but descriptive methods to implement pipeline-breaking database operators. They help us better understand these operators before introducing more efficient and popular implementation techniques in Sections 4.3.2 and 4.3.3. Finally, methods for combining different implementations on FPGAs are discussed in Section 4.3.4.

k	v
43	4
26	3
43	1
26	5

(a) Table R

k	v
88	7
64	2
43	9
88	5

(b) Table S

k	v
26	5
26	3
43	4
43	1

(c) $\text{Sort}_k(R)$

k	v
43	9
64	2
88	5
88	7

(d) $\text{Sort}_k(S)$

index	bucket
0	
1	(26, 3), (26, 5)
2	
3	(43, 4), (43, 1)
4	

(e) Hash Table of R

k	$\text{sum}(v)$
26	8
43	5

(f) $\text{Aggr}_{k,sum(v)}(R)$

$R.k$	$S.k$	$R.v$	$S.v$
43	43	4	9
43	43	1	9

(g) $\text{Join}_k(R, S)$

Figure 4.2: Examples of sort-based and hash-based aggregation and join operator execution: (a)Table R, (b)Table S, (c)Sorted table R, (d)Sorted table S, (e)5-entry hash table using hash function $h(x) = x\%5$ and containing all rows of R, (f)Aggregation of R, (g)Join of R and S.

4.3.1 Loop-based implementation

A straightforward, yet inefficient way to implement pipeline-breaking database operators is via nested loops. The sort operator can be implemented using the insertion sort algorithm with the outer loop selecting an unsorted row, and the inner loop inserting it into its correct position in the already sorted portion of the table [CLRS09]. For aggregations, the outer loop selects an unprocessed row, and the inner loop searches for its group (if any) in the list of processed rows. For a join of tables R and S, the outer loop selects an unprocessed row from table R while the inner loop searches for a matching row in S. This algorithm is called *nested-loop join* [GUW09]. Given the quadratic time complexity of these algorithms, they are seldom used in high-performance database systems. However, thanks to their simplicity, they are employed in a few FPGA platforms for executing sliding-window operators [NSJ13, SJT+12]. Nested-loop joins are also used as a fallback option to implement arbitrary join conditions (not simply **equality**-based joins) or Cartesian products, when more efficient algorithms fail.

4.3.2 Sort-based Implementation

Sorting is an important operation in database query processing. It is not only a database operator itself, but also a preprocessing step offering an efficient way to execute many other operators, as explained later in this section [DGN23, Gra06]. *Sort-merge* is the most popular FPGA-based sorting algorithm. In sort-merge, a streaming *sort phase* transforms a table into small runs of sorted rows. The size of the sorted runs depends on the algorithm and the amount of FPGA resources allocated to it. The runs are small enough that they can be sorted very efficiently on the FPGA itself, i.e., without transferring data or intermediate structures back and forth from the FPGA RAM. The sorted runs are typically stored on the FPGA RAM. A subsequent *merge phase* merges the sorted runs until, eventually after multiple passes, the sorted output is produced. Multi-way merging is a common method to accelerate this step. Indeed, a n_w-way merger requires $\lceil \log_{n_w}(n_r) \rceil$ passes to merge n_r sorted runs. Therefore, increasing n_w has an exponentially positive effect on merge performance. Sort-merge is inspired by external sorting described in [Knu73, HS76]. We study and explore sort-merge extensively and from different angles in Chapters 6, 7, and 8.

Sorting can be used to simplify the processing of complex pipeline-breaking operators. Indeed, aggregation and join become single-pass algorithms on sorted tables [GUW09]. An example is shown in Figure 4.2. Consider the $\text{Aggr}_{k,sum(v)}(R)$ operator. We start by sorting table R according to key column k. In the sorted table, shown in Figure 4.2(c), all rows belonging to the same group are placed next to each other. All that remains is to traverse the sorted table to accumulate neighboring rows with the same key column. This can be done in a streaming fashion, where we keep the last-seen row as state, bring in the next one to compare and (potentially) accumulate, before setting it as the new last-seen row. Basically, sorting does all the heavy lifting. The result of the aggregation operator is shown in Figure 4.2(f).

Now consider the $\text{Join}_k(R, S)$ operator. Again, we start by sorting both input tables R and S according to the join key k. The results of the sorts are shown in Figures 4.2(c) and 4.2(d), respectively. We then traverse both tables at the same time, looking for matching rows which we combine to produce the join result. For $M : N$ joins with both M and $N > 1$, it might be necessary to backtrack while traversing the sorted input tables in order to get all the join results. The result of the join is shown in Figure 4.2(g). This algorithm is called *sort-merge join*.

The authors of [STM+13, STM+15, SMT+14] propose sorting using the tournament tree sort algorithm [Knu73]. They instantiate two tournament tree sorters, each producing sorted runs of minimum 16K rows. The outputs of the two sorters are immediately merged, producing larger sorted runs of at least 32K rows, before being sent to the host. Using two sorters, they can process 1 row every 7 clock cycles instead of every 15 clock cycles, allowing them to saturate their PCIe bandwidth. For sorting large tables, they require the host to run a (potentially) multi-pass merge phase on the sorted runs produced by the FPGA.

An implementation of sort-based aggregation is proposed in [DZT13]. Here, the authors assume that the data is either sorted to begin with or sorted at runtime by the host, before being transferred to the FPGA. The FPGA then streams in the sorted rows, aggregating them using the technique described earlier in the section.

The authors of [CO14] use sort-merge for executing the sort and the join database operators. Their high-performance merge tree implementation is capable of merging multiple (narrow) rows at the same time. The very first merge pass uses a small sorting network to produce the initial sorted runs which are directly (i.e., without going through RAM) fed

to the merge tree [Bat68, MTA12]. Multi-way merging is used to reduce the number of merge passes necessary to process large tables. They propose a 3-FPGA setup for processing joins. 2 of the FPGAs sort an input table each, streaming the result to a third FPGA which materializes the result of the join. Sort-based joins are also attempted in [CP16]. The FPGA produces sorted runs using folded bitonic sorting networks. The sorted runs are then transferred to the host for both merging and to perform the join operator.

4.3.3 Hash-based Implementation

An alternative way to implement complex pipeline-breaking database operators is based on hashing [GUW09]. The process is exemplified in Figure 4.2. Consider the $\text{Aggr}_{k,sum(v)}(R)$ operator. We start by instantiating an empty hash table. We then insert every row of R into the hash table: A row is inserted into the entry specified by the hash of its key. Here, we simply use a modulo ($\%5$) hash function. For instance, row `(26,3)` is inserted into the hash bucket at index $26\%5 = 1$. The resulting table is shown in Figure 4.2(e). At this point, all rows belonging to the same group are in the same hash bucket. We complete the aggregation process by accumulating the rows belonging to the same group in every non-empty bucket. In case *hash collisions* occur, i.e., rows with different keys are mapped to the same bucket, care must be taken not to accumulate rows belonging to different groups. An almost universally applied optimization is to accumulate the rows belonging to the same group, while inserting them into the hash table. This often results in lower memory consumption and higher performance of the aggregation operator. The result of the aggregation operator is shown in Figure 4.2(f).

Now consider the $\text{Join}_k(R, S)$ operator. Like the aggregation operator, we start by instantiating and populating a hash table with the rows of table R. This process is called the *build phase*, and table R the *build relation*. In the subsequent *probe phase*, for each row of S, we perform a lookup inside the hash table trying to find matching rows from R. The matching rows are paired and output as part of the join result. Table S is called the *probe relation*. The result of the join operator is shown in Figure 4.2(g). This algorithm is commonly referred to as *hash join*.

Smart storage platforms using hash-based aggregation are introduced in [WIA14, WTA13, WFS+19]. The authors propose implementations using either the BRAMs or the DRAM of the FPGA for storing the hash table. The hash table can hold a single row per bucket. In case of hash collisions, they forward the colliding row to the host, later to be aggregated in software by the CPU. Another smart storage platform described in [YKO+14] uses a register array to store the hash table. They use the key of a row as its index into the hash table, i.e., the identity function is used for hashing. This works well with keys limited to a relatively small range of values. The authors of [ABW+16] use content-addressable memories (CAMs) as a cache for the hash table, which is itself stored in the FPGA RAM. The goal of the cache is both to improve memory access performance and to provide synchronization mechanisms (e.g., locking) between multiple inserting rows trying to access the same bucket in the memory.

The authors of [HSM+13, SMT+14] accelerate the join operator using hash-based techniques. They start by streaming the rows of the build relation from the host onto the FPGA. The latter extracts the join keys from the rows before storing them on the FPGA RAM. Every key along with a pointer to its corresponding row in memory is then inserted into a hash table on the FPGA. The hash table is implemented using two data structures. A bit vector indicates whether a given bucket of the hash table is valid (i.e., non-empty), and if so returns a pointer to the head of a linked list in the address table. The address table stores the buckets as linked lists of rows. Both structures are stored

on FPGA BRAMs for fast access. After the build phase, the probe relation is streamed into the FPGA with every row triggering a hash table lookup. Matching rows are paired and streamed out of the FPGA as the join result. A pairing requires the payload of the row from the build phase to be fetched from the main memory. Hash-based joins are also attempted in [HANT15]. This time, the hash table is built and stored in the FPGA RAM. They use multithreading to achieve high performance despite long memory access latency. While a task such as hash table insertion or probing is waiting for memory, it goes into idle state allowing other tasks to execute and make progress. They hide memory latency by launching thousands of tasks (or threads) at the same time. Atomic memory accesses are used as a synchronization mechanism among the threads.

4.3.4 Mixed Implementation

Database queries are composed of many types of operators that a database system must support. The same is true for accelerators. In order to offer significant performance improvements, the accelerator must support as many high-impact (pipeline-breaking) database operators as possible. This is a requirement discussed in length in Section 2.4. A common challenge faced by the designers is to fit all the circuits used for executing these operators on the FPGA. They have come up with a few solutions described in this section.

Reconfiguration or reprogramming is a property unique to FPGAs. It can be leveraged to run different operators or even different implementations of the same operator on a single FPGA. In [UIO15], an FPGA image running hash joins and another running sort-merge joins are prepared and ready to be programmed onto the FPGA. They choose an image depending on the characteristics of the workload. The image is then programmed onto the FPGA in order to run the operator. A more sophisticated approach proposed in [WPC+16] combines data processing primitives (e.g., map, filter, scatter, gather, sort, etc.) to execute database operators. The authors introduce a query plan generator and optimizer that determines the best combination of primitives and how they should be partitioned into (potentially multiple) FPGA images for executing a particular query. This means that for every new query, multiple images may need to be synthesized. If more than one image is involved, FPGA reconfiguration is used to switch between them during query execution. The authors of [MTA09b, MTA10] introduce a query-to-hardware compiler which given a query plan, generates the RTL description of a circuit implementing the plan. The code is then synthesized before being executed on the FPGA. Note that synthesizing can take hours to complete, meaning that ad-hoc queries are not supported.

Partial reconfiguration is another feature of FPGAs allowing designers to reprogram only a part of an FPGA while the other parts continue running [VF18, KTB+12, FdSJ19]. It is used in [DZT13] for processing the filter and aggregation operators. They instantiate query processing data paths composed of partially reconfigurable slots. Each slot can be programmed at run time to execute a simple filter operator (e.g., $<, >, =, AND, OR$), or two of them can be combined to perform aggregation. For aggregation, they assume the data to be pre-sorted. While reprogramming a data path, others may continue running.

In Part II of this thesis, we propose resource sharing as a means to support multiple operators on a single FPGA image. It consists of using an efficient common denominator algorithm such as sorting for accelerating all pipeline-breaking database operators. Indeed, after the input is sorted, aggregations and joins become simple streaming operators requiring few FPGA resources to implement. We take this idea to the extremes by ingraining elements of the aggregation and join operators across the entire sort pipeline.

4.4 A NOTE ON QUANTITATIVE PERFORMANCE COMPARISONS

In our presentation of related work in this chapter, we have not shown any performance numbers for comparing existing accelerators. In fact, it is almost impossible to compare these accelerators based on benchmarks published in past research. There are a few reasons behind this. First, there is an extreme variety of FPGA platforms used in these benchmarks. The platforms are different in the number and size of FPGAs, the FPGA family, the implementation frequency, the number and capacity of the RAMs connected to the FPGA, the bandwidth between the FPGA and its RAM, the bandwidth of the PCIe bus, the network speed, etc. Second, there is no standardized dataset used by these benchmarks. This is in part due to the wide range of capabilities offered by various platforms. Standardized TPC-H benchmarks commonly used for benchmarking database systems are often too complex to execute on an accelerator [Cou22]. Finally, there is a lack of platform-independent performance metrics such as [number of clock cycles / processed row] in current benchmarks.

In order to provide a fair comparison between these accelerators, they must all be benchmarked on the same FPGA platform. This leads to a new set of challenges, the most important being the lack of open source. A full re-implementation may not match the original code as a lot of implementation and optimization details are missing in research papers. Even if the original code is available, it might require a lot of effort to optimize for a given FPGA platform.

We believe benchmarking against state-of-the-art software database systems to be the best approach for evaluating accelerators. This also aligns with the ultimate goal of accelerators, that is to improve the performance of their host. Nonetheless, we refer the curious reader to [FMH+20] for some performance numbers on current FPGA-based database system accelerators.

Part II

Cache-Based Morphing Sort-Merge with KeRRaS (CbMSMK)

5

OBJECTIVES AND ARCHITECTURE OVERVIEW

5.1 From Requirements to Objectives

5.2 Architecture Overview

5.3 Outline of Part II

In Chapter 1, we defined a set of requirements for accelerating in-memory database systems. In this chapter we use our knowledge of database systems (Chapter 2), FPGA development (Chapter 3), and related work (Chapter 4) in order to transform those requirements into specific objectives and design guidelines. This allows us to sketch out a high-level architecture for our FPGA accelerator, leaving it to the chapters that follow to fill out the specifics.

5.1 FROM REQUIREMENTS TO OBJECTIVES

The accelerator requirements defined back in Chapter 1 are based on the context of our research and the type of database system and workload we want to accelerate. They are summarized below:

- ***Reqmt 1*:** The accelerator must be able to process a reasonable number of database operators while also being extensible (through modularity) to support new operators in the future. The operators must be able to process a diverse range of tables that have different depths, widths (or number of columns), and data types.
- ***Reqmt 2*:** Priority must be given to database operators whose acceleration has the greatest positive impact on the execution time of database queries. The accelerated operators should run reasonably faster than state-of-the-art database systems running on traditional (CPU-based) computers.
- ***Reqmt 3*:** The architecture must be flexible enough to allow deployment on different FPGA platforms.
- ***Reqmt 4*:** The target platform type must be suitable for accelerating in-memory database systems.
- ***Reqmt 5*:** The accelerator must be able to serve multiple clients in parallel.

Reqmt 1 and *Reqmt 2* focus on the choice of the operators to support. In Section 2.4, we argued that accelerating pipeline-breaking database operators is our best bet to offer great performance while covering various analytical workloads. We **target the sort, aggregation, and join operators** which are computationally expensive and frequently used in most analytical queries.

In order to comply with *Reqmt 1*, we would like our accelerator to support all three of our target operators at the same time. However, as explained earlier in Section 4.3.4, FPGAs' limited resources make this a real challenge. Fortunately, a common denominator for processing all these operators is sorting. Indeed, a high-performance sort-merge implementation can be used for accelerating the sort, join, and aggregation operators (see Section 4.3.2). Moreover, as we shall see in Chapters 7 and 8, sort-based database processing allows us to process arbitrarily deep and wide tables. We therefore use **sort-based implementations** of our operators allowing them to benefit from a shared sort-merge instance. Note that we still require the FPGA to execute these operators in their entirety. We do not want the accelerator to rely on host pre- or post-processing for operator execution, as it closes doors to holistic query execution on the FPGA. Finally, to enable and ease extensions to our platforms, we promote modular designs with standardized interfaces. This allows new operators to be easily integrated into the sort-merge pipelines. Furthermore, it simplifies the process of updating existing modules, such as ALUs, in order to include

more features, such as support for additional data types. The choice of our implementation strategy will be discussed and justified in much greater detail in Chapter 7 where we consider and compare many alternatives.

Reqmt 3 and *Reqmt 5* may be satisfied thanks to the decisions we made earlier. Indeed, sort-merge is a highly flexible algorithm, where both the sort and the merge phase can use a number of different implementations with various area/performance characteristics. Thanks to this, and the fact that our operators can share a single sort-merge instance, we design resource-efficient accelerator "cores" capable of processing these operators at various area/performance points. This offers us the flexibility we need to potentially deploy on different FPGA platforms. It also allows us to provide multi-client support by instantiating multiple cores on the FPGA. Although this idea may seem a bit suppositional, we shall see in Chapter 7 that it works well in practice.

Now all that remains is *Reqmt 4*, ensuring that the target FPGA platform is optimal for in-memory database system acceleration. Platform types are described in Section 4.2. In-memory database systems store their data in the main memory of the host (see Section 2.1.2). This eliminates both smart storage and network processor platforms as potential candidates. Modern coprocessor and accelerator card platforms give the FPGA access to the host CPU RAM. Accelerator cards additionally provide the FPGA with fast multi-channel dedicated RAM. The DDR link between the FPGA and its RAM has higher bandwidth than the PCIe or QPI link between the FPGA and the CPU RAM. Moreover, a dedicated FPGA RAM reduces the load on the CPU RAM which can be used for other tasks. In a nutshell, an accelerator card platform provides a superset of the features of a coprocessor platform with the addition of higher memory bandwidth; much appreciated for accelerating memory-intensive database workloads. Therefore, we target our design towards **accelerator card platforms**.

To summarize, our objective is to design a resource-efficient sort-based database query processor that can accelerate the sort, join, and aggregation operators. The architecture should be flexible and modular, and support parallelism through multiple clients. The design must run on accelerator card platforms.

5.2 ARCHITECTURE OVERVIEW

In Section 5.1, we hypothesized that a database system accelerator based on the sort-merge algorithm is best suited for efficiently processing our target pipeline-breaking database operators. As a result, the architecture of our accelerator is centered around the sort-merge algorithm. Sort-merge consists of a sort phase producing sorted runs and a merge phase for merging them. The details of the algorithm are described in Section 4.3.2. The input table, sorted runs, intermediate merge results, and the final sorted table are all placed on the FPGA RAM. Therefore, splitting the sort-merge implementation into a sort pipeline and a merge pipeline seems natural. The architecture is shown in Figure 5.1.

The *Sort-Network* executes the sort phase of the sort-merge algorithm. The *Load* module is configured to read a table from FPGA RAM and stream its rows into the pipeline. The pipeline may consist of a series of processing steps (e.g., projection, filtering, caching, ...) but ends with the *Sort* module producing the sorted runs and the *Store* module which stores them back on the FPGA RAM.

The *Merge-Network* executes the merge phase of the sort-merge algorithm. The *Load* module provides the *Merge* module access to a set of sorted runs, which are merged by the latter before continuing their way down the pipeline. The remainder of the pipeline may

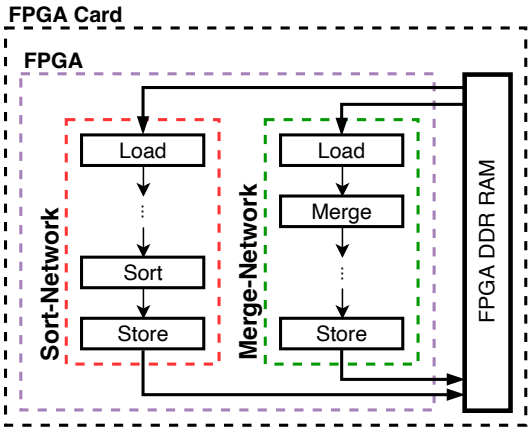


Figure 5.1: Overview of the architecture of our accelerator. The *Sort-* and the *Merge-Network* implemented on the FPGA execute the sort and the merge phases of the sort-merge algorithm, respectively. They are both pipelined, and read and write their data from/to the FPGA RAM.

consist of several processing steps for executing database operators such as aggregation, join, filtering, projection, etc. The rows are ultimately stored back on the FPGA RAM through the *Store* module.

In itself, the architecture offers 2 methods to improve sort-merge performance. First, each of the networks has a pipelined design, allowing it to process up to one row per clock cycle. We call this *line-rate* processing. Second, the two networks can work in parallel, allowing the FPGA to sort two tables at the same time: After the *Sort-Network* is done producing the sorted runs of the first table, it moves onto the second table while the *Merge-Network* starts merging those sorted runs.

In order to achieve modularity and extensibility, we require all modules in the pipeline to have a standard handshaking *ready/valid interface* [ARM20]. Moreover, they must be *bypassable*, i.e., if a module is not needed for a particular task, data should still be able to move through (or around) it **without being modified**.

Finally, a resource-efficient implementation of the networks on a large FPGA should allow us to instantiate multiple instances of each of them, enabling the accelerator to execute multiple operators at the same time. This is an efficient way to support multiple clients.

Throughout the following chapters, we will present alternative and improving versions of this architecture. The sort-merge algorithm will, however, remain the centerpiece of our accelerator.

5.3 OUTLINE OF PART II

Database system designers, who typically have a software background, are naturally drawn to HLS methodologies for FPGA acceleration. In Chapter 6, we compare traditional RTL programming with HLS in the context of database system acceleration. We propose a simple sort-merge architecture and implement it using both VHDL and OpenCL. The implementations are then compared in terms of performance, resource

utilization, and development effort. This helps us choose a programming methodology for designing our own accelerator. Chapter 6 is based on our publication in DaMoN'20 [MFLM20].

In Chapter 7, we propose the first version of our database system accelerator. It is based on the sort-merge algorithm, can execute the sort, aggregation, and join operators, and is able run up to 12 queries in parallel. MSM is the name of our row-oriented database system accelerator. It is on average $5\times$ faster than the state-or-the-art in software. Chapter 7 is based on our publication in DaMoN'21 [MFL+21].

MSM is capable of processing arbitrarily deep tables, as long as the width of the table is smaller or equal to the width of its data path. In Chapter 8, we propose an algorithm allowing MSM to process arbitrarily wide tables. In order to efficiently implement the algorithm, we convert MSM into a column-oriented database accelerator. The new platform, called MSMK, has performance equivalent to that of MSM. Chapter 8 is based on our publication in FCCM'22 [MFM+22].

In Chapter 9, we focus on the aggregation operator alone. We propose caching for early aggregation, resulting in a hybrid hash- and sort-based aggregation algorithm. The new implementation, called CbMSMK, achieves up to 3 times higher aggregation performance compared to MSMK. Chapter 9 is based on our publication in FPGA'23 [MMF+23].

Chapter 10 is the concluding counterpart to the current introductory chapter. There, we present a brief summary of our final architecture along with some TPC-H benchmarks showcasing its capabilities. We finally revisit the requirements and objectives defined in this chapter to examine the extent to which they are satisfied.

6

COMPARATIVE ANALYSIS OF OPENCL AND RTL FOR SORT-MERGE PRIMITIVES ON FPGAS

6.1 Programming FPGAs

6.2 Related Work

6.3 Architecture

6.4 Experiments

6.5 Summary & Discussion

Writing an RTL description for programming FPGAs requires deep knowledge of digital design and HDLs. Intel® OpenCL™ is a mature HLS platform allowing designers to implement their architecture using the relatively high-level C programming language. In this chapter, we conduct a comparative analysis of OpenCL- and RTL-based implementations of sort-merge primitives, including a novel heapsort algorithm. We quantitatively compare their performance, FPGA resource utilization, and development effort. Our results show that while requiring comparable development effort, RTL implementations of critical primitives used in the sort-merge algorithm achieve $4\times$ better performance while using half as much the FPGA resources.

Note that parts of the material presented in this chapter have been previously published in DaMoN'20 [MFLM20].

6.1 PROGRAMMING FPGAS

The difficulty to program FPGAs is an important obstacle on their way to ubiquity. Indeed, programming FPGAs requires knowledge of digital design and HDLs for describing an architecture at the RTL abstraction. One solution is to use HLS tools such as OpenCL-based programming platforms. OpenCL™ is a standard for parallel programming of accelerators such as multi-core CPUs, GPUs, and FPGAs [Inc20]. OpenCL programs are composed of multiple functions (i.e., kernels or modules) in C, running in parallel and communicating via channels or DDR RAM. The Intel® FPGA SDK for OpenCL (or simply OpenCL in the remainder of this chapter) is a platform for synthesizing, executing, and diagnosing OpenCL programs on Intel® FPGAs. Similar tools are available from other vendors. Two other notable C-based HLS tools are Vitis HLS from Xilinx [Xil22a] and Catapult from Siemens [Sie21]. Intel itself is developing a SYCL-based successor to OpenCL using oneAPI [Int23]. These tools offer functionality and capabilities that are comparable to those from OpenCL, which will be the focus of this chapter.

OpenCL promises several advantages compared to RTL [Int13] resulting in a gentler learning curve (specially for software developers), reduced development effort, and portability across OpenCL-enabled FPGAs. However, programming FPGAs using software programming languages like C, even on highly parallel platforms such as OpenCL, limits their capabilities. Indeed, one of the greatest advantages of FPGAs compared to CPUs and GPUs is that they can run any hardware architecture; therefore, provide a tailored execution model based on the problem at hand (see Chapter 3). RTL-based languages offer the designer a great deal of control over FPGA resources to define a suitable architecture. In contrast, the architecture generated by OpenCL is limited by its expressiveness. OpenCL was initially developed for multi-core CPU and GPU acceleration and with their fixed architecture in mind [The08]. Therefore, despite the FPGA-specific features added to the platform [Int19], it might not always be able to generate a suitable architecture. In fact, given the large design space provided by FPGAs, an exhaustive exploration by OpenCL (or HLS tools in general) proves impossible in practice.

In this chapter, in addition to testing some of the claims on the advantages of employing OpenCL instead of RTL, we set out to quantify the disadvantages entailed by its usage. More specifically, we compare the performance, resource utilization, and required development effort of OpenCL and RTL implementations of the sort-merge algorithm. We chose this algorithm because in addition to being one of the most common and critical functions in many data and query processing pipelines [STM+13, CO14, TW13, UIO15, WPC+16]; its main primitives, i.e., sorting and merging, represent respectively compute- and memory-intensive workloads very well. Therefore, our OpenCL vs. RTL benchmarks

for these primitives would be good indicators of performance for other database processing tasks in OpenCL compared to RTL. Furthermore, since sort-merge will serve as the foundational building block of our database system accelerator (as outlined in Section 5.1), these benchmarks will help us in selecting the most optimal implementation for our platform.

In Section 6.3, we propose a high-performance scalable sort-merge algorithm capable of sorting large datasets. We also introduce a novel heapsort architecture that eliminates data hazards while sorting small runs of data. In Section 6.4, we benchmark the performance and resource utilization of our OpenCL-based implementation of sort-merge. We then replace the sorter and merger modules with their equivalent in RTL and present benchmarks comparing the two. We finally compare the initial OpenCL-based implementation to the now hybrid OpenCL-RTL implementation. In Section 6.5, we discuss our results and present some recommendations on developing high-performance FPGA solutions with OpenCL and RTL.

6.2 RELATED WORK

There is a variety of prior work discussing sorting on FPGAs [STM+13, CO14, TW13, UIO15, WPC+16]. They all focus on a particular hardware platform and programming language to implement a sorting algorithm. There has also been research comparing the performance of other (than OpenCL) HLS tools (e.g., Vivado® HLS, LegUp, Bluespec) to that of RTL for many algorithms [CDL11, ANA10]. Comparisons between these HLS tools often show that they result in varying degrees of performance depending on their maturity, supported features, target applications and input language [ANS+14, NSP+16, HWFH08].

To the best of our knowledge, we are the first to compare the performance, resource utilization, and development effort of OpenCL and RTL implementations of a generic sort-merge algorithm. In what follows, we introduce related work comparing OpenCL and RTL implementations of other algorithms.

The authors of [ANS+14] compare the OpenCL and RTL implementations of bitonic sorting networks and linear sorters. For bitonic sorting, RTL achieves $28\times$ higher performance while using 50% of the resources used by OpenCL. For linear sorting, RTL achieves $2\times$ higher performance while using 10% of the resources used by OpenCL. In the domain of image processing, the authors of [HCGL15] observe that typical OpenCL kernels for edge detection and feature extraction achieve similar performance to their VHDL counterparts but at the cost of 2 to 3 times additional resources. For convolutional neural networks, the authors of [MSC+16] observe $1.9\times$ performance improvements achieved by using RTL compared to OpenCL. Finally, in the domain of high-energy physics for floating-point pipelines with low execution hazards, the author of [Fae17] reports a $1.3\times$ performance boost from OpenCL to RTL.

6.3 ARCHITECTURE

In this section, we outline our sort-merge architecture and describe its constituent modules and the flow of data among them. We then explain in more detail the design of two of our key modules: sorters and mergers. We conclude this section by explaining how our architecture enables resource adaptability and scalability in both input data size (i.e., input cardinality) and input data width (i.e., key size).

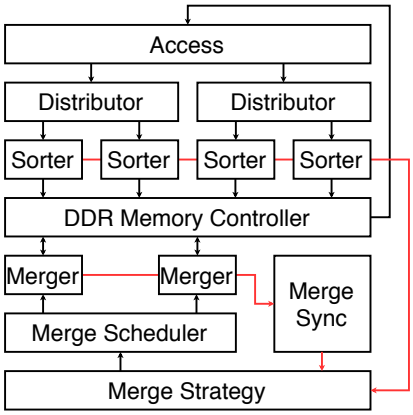


Figure 6.1: Sort-merge architecture used for comparing RTL (VHDL) and HLS (OpenCL) methodologies in the context of database system acceleration.

6.3.1 Global Architecture

Figure 6.1 shows our sort-merge architecture with 4 *Sorter* and 2 *Merger* modules. We assume the input data to be initially placed on the FPGA RAM. The *Access* module reads the data and passes it down in a round-robin fashion to the *Distributors*. The *Distributors* in turn pass the data to the *Sorters* which sort the data in BRAM and write the sorted runs to the FPGA RAM. Note that depending on the number of *Sorters*, we can have a network of *Distributors* structured like a binary tree.

Once a round of sorting finishes, i.e., every *Sorter* has written a sorted run to the FPGA RAM, the *Sorters* notify the *Merge Strategy* module to start merging the currently available sorted runs. They then start the next round of sorting. In the meantime, after being notified by the *Sorters*, the *Merge Strategy* calculates which addresses need to be merged and sends them, as jobs, to the *Merge Scheduler*. The latter distributes the jobs to the *Mergers* as they become available. Finally, when a stage of merging finishes, the *Merge Synchronizer* notifies the *Merge Strategy* so it can launch new jobs for the next stage.

The arrows in Figure 6.1 represent *FIFO buffers* that can be implemented in OpenCL as channels or pipes. They pass data among modules without the need for a shared memory space. A FIFO buffer is commonly referred to as a *queue* for short.

To add context relative to the sort-merge architecture introduced in Chapter 5, note that in Figure 6.1 the modules placed above the *DDR Memory Controller* make up the *Sort-Network*, and the ones below the controller the *Merge-Network*. The *DDR Memory Controller* provides access to the FPGA RAM.

6.3.2 Sorter Architecture

When designing an on-chip sorting algorithm on FPGAs, one considers performance, resource efficiency, and scalability in input cardinality as well as key size. Our additional challenge was to make sure OpenCL is expressive enough for efficiently describing the architecture of the algorithm.

With these objectives in mind, we immediately dismissed sorting networks and linear sorting because of the amount of FPGA resources they require, especially for large input

cardinalities. In fact, a bitonic sorting network requires $O(n \log^2(n))$ comparators and a linear sorter $O(n)$ comparators in order to sort n numbers. Among traditional sorting algorithms that require $O(1)$ comparators, we only considered those with a worst-case complexity in $O(n \log(n))$. This of course also excludes quicksort because its worst-case complexity can reach $O(n^2)$. Between mergesort and heapsort, we opted for heapsort because of its memory (BRAM) efficiency, given that an efficient mergesort implementation requires an intermediate result buffer equal in size to the input data. In what follows, we propose our heapsort algorithm. It is scalable, pipeline efficient (no execution hazards or stalls), as well as memory efficient (no need for memory duplication or any special type of on-chip memory). Moreover, it is a traditional sorting algorithm designed for execution by CPUs. Therefore, OpenCL is expressive enough to describe it as efficiently as RTL. Our heapsort algorithm uses a min-heap in order to output data in ascending order.

The heapsort algorithm can be divided into two phases: In phase 1, at each iteration, the algorithm accepts a new input and inserts it into the heap. With n the total size of the heap, for every new input it takes up to $\log(n)$ reads, $\log(n)$ writes and $\log(n)$ comparisons for the heap property to be valid again. Indeed, the ith input must be compared with the ancestors of the ith position in the heap until one that is smaller, say at position k, is found. Then all ancestors connecting position i to k should be shifted down for the new element to become the direct child of position k. In a pipelined implementation, this requires a read, a comparison with the new element, and ultimately a write to the lower position for each ancestor. Finally, a single write is required to insert the new element in its proper position. Because the paths of the reads, writes and comparisons in the heap are deterministic (starting from the parent of position i, from child to parent, up to the root), this results in a fully efficient stall-free execution pipeline. For the same reason, we can guarantee full pipeline efficiency even with multi-cycle comparators needed to compare larger data types, resulting in key width scalability. Moreover, given that at every clock cycle a single read and write is required, a simple dual-port BRAM available on almost all FPGAs is sufficient. Overall, this phase of the algorithm requires at most $\log(n) + p$ clock cycles to insert a new element where p is the pipeline depth accounting for the fixed cost of initially filling the execution pipeline.

In phase 2 of the algorithm, at each iteration, the root (being the smallest element in the heap) is removed and output. Next, the current last element, say at position i, must be compared with both children lc and rc of the root for the smallest of the three to become the new root. If it happens to be element i, the iteration is finished. If, however, the smallest is one of the children, say lc, then i needs to compete with the children of lc for its old position. This process must continue until the element at position i becomes the smallest in comparison with its children, or when it becomes a leaf again. In a pipelined implementation, an iteration requires up to $2\log(n)$ reads to access the siblings, $2\log(n)$ comparisons to find the smallest of the three elements, and $\log(n)$ writes for the smallest of the three elements to then be written at the available position, at each level of the heap.

The second phase of heapsort as presented here has two main inefficiencies. In what follows, we shall discuss these inefficiencies and propose optimizations to overcome them.

Dual-Memory Optimization

The first observed inefficiency is that the second phase requires two BRAM reads and one write at every level of the heap. Simple dual-port BRAMs (i.e., BRAMs used in simple dual-port mode described in Section 3.1.2) readily available on all FPGAs require stalling the pipeline until all 3 operations are done. Thus however, the execution pipeline needs twice the number of clock cycles to handle an iteration of phase 2 of the algorithm,

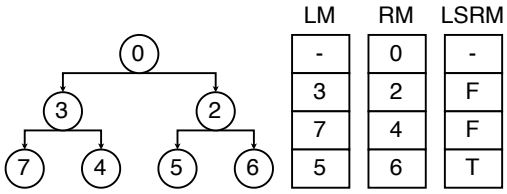


Figure 6.2: An example of the memory model used by our heapsort algorithm. Left memory (LM) and right memory (RM) are used to store the left and right siblings in the heap, respectively. The left smaller-than right memory (LSRM) indicates, for each pair of siblings, whether the left sibling is smaller than the right one.

compared to a phase 1 iteration. Although quad-port BRAMs available on most modern FPGAs can solve this issue, a simpler yet more elegant solution is to arrange the memory differently.

The solution is based on the observation that the two reads at every level of the heap are to two siblings. Therefore, by storing the heap in two simple dual-port BRAMs, one left memory (*LM*) containing only the left children and another right memory (*RM*) containing only the right children, we can do both reads and the write in a single clock cycle. An example heap and its corresponding memory configuration are shown in Figure 6.2. The memories are presented vertically for better visualization. Note that the optimization presented in this section is achievable in OpenCL as it provides support for multi-bank BRAM-based memories with independently accessible read/write ports for each bank [Int19]. Also note that although our optimization is somewhat similar to the ones used by the algorithms presented in [Zab11] and [TMA11], the former requires substantially more comparators and the latter uses mixed-width BRAMs.

Lookahead Optimization

Another inefficiency in the second phase of our heapsort algorithm occurs due to control hazards. Indeed, the nodes that must be fetched from the BRAM at every level depend on the result of the comparisons at the parent levels. The negative effects of such hazards scale with the key size. For larger keys, multi-cycle comparisons might be needed resulting in more pipeline stalls.

To overcome this challenge, we augment our dual-memory heap data structure to indicate, for each element in the *LM*, whether it is smaller than its corresponding sibling element in the *RM*. This part of the data structure is represented as the left smaller-than right memory (*LSRM*) shown in Figure 6.2. *LSRM* has a width of 1 bit (as it stores Booleans) and a depth equal to half the cardinality of the input data, therefore needs a negligible amount of BRAM to store. Assuming we can maintain such a data structure, all control hazards may be avoided. Indeed, given that comparison results between all siblings in the heap are available, the path of an element i down the heap can be determined in advance. Hence the pipeline could issue reads to the lower levels of the heap without the need to have the results of the comparisons at the previous levels. Once i finds its proper place, the previous stages of the pipeline should be flushed.

The only remaining question is how to create and maintain such a data structure. At the beginning of phase 1 of the algorithm, the heap is empty, so the *LSRM* property holds. In future iterations of phase 1, every time a new element goes up the heap, we use the dual-memory data structure to compare it, in addition to the parent, with the sibling

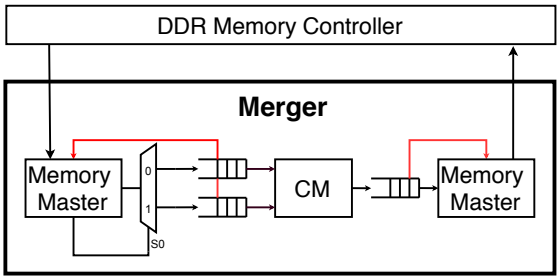


Figure 6.3: *Merger* Architecture

of the parent. Hence in the event the new element takes the place of the parent, we know how the *LSRM* memory should be updated. Moreover, when an element is pushed down, because of the heap property it will be smaller than its new sibling. Thus, with one additional comparator we can maintain the *LSRM* memory property during the first phase of the algorithm. The same trivial analysis can be applied to the second phase of the algorithm, for which one additional comparator is needed to maintain the *LSRM* memory property. Note that this optimization can also be fully implemented in OpenCL by adding a new memory bank for storing the *LSRM* structure.

6.3.3 Merger Architecture

The architecture of the *Merger* is shown in Figure 6.3. A *Memory Master* reads the sorted runs from the FPGA RAM and feeds them through FIFO buffers to a comparator-merger (*CM*). The latter merges the sorted runs before storing the result back into the FPGA RAM through another FIFO buffer-*Memory Master* pair. The architecture can be scaled for merging more than two lists at a time.

The OpenCL implementation consists of 3 modules. Two of them fetch sorted runs from the FPGA RAM feeding them into two inter-kernel channels. The third module merges the keys coming from those channels and writes the result back to the FPGA RAM.

Note that the throughput of the *CM* module decreases linearly with the number of cycles the comparators require. If an n-cycle comparison is required, $n + 1$ *Mergers* will be needed to merge at maximum throughput. This is manageable thanks to the small resource footprint of the *Mergers* (see Section 6.4.3).

6.3.4 Scalability and Resource Adaptability

The proposed architecture is easily scalable and adaptable to different key sizes and FPGA platforms. The number of *Sorters* and *Mergers* can be increased to achieve higher throughput and to take advantage of the full bandwidth of the FPGA RAM. One important functionality of the *Distributors* and the *Merge Scheduler* is to decrease the fanout of the design and ultimately reduce the impact of such up-scaling on the system frequency. The *Sorters* can also be configured to sort smaller (or larger) runs depending on the amount of resources the FPGA can dedicate to this functionality.

Merging is divided into four phases: Strategy, Scheduling, Execution, and Synchronization. Each phase can be fine-tuned to better suit the problem and hardware at hand. For instance, in the absence of high-bandwidth FPGA RAM, the *Merge Strategy* could be modified so it only starts merging after all the sorted runs are available. This decreases congestion on the memory bus, ultimately improving performance.

Impl.	Freq. [MHz]	LU [ALM]	BU [M20K]	ET [s]	Thr. [MB/s]
OpenCL	184	$408,683$ (43.8%)	1762 (15%)	3.77	284

Table 6.1: Characteristics of the OpenCL implementation of the sort-merge algorithm with 16 *Sorters* and 4 *Mergers*. Execution time (ET) and throughput (Thr.) values are for sorting the *Basic Workload*.

6.4 EXPERIMENTS

In this section, we start by presenting benchmarks of our sort-merge architecture developed entirely using OpenCL. We then identify underperforming modules and compare them with their RTL equivalent. Next, we replace them with their RTL equivalent resulting in a hybrid OpenCL-RTL sort-merge architecture. Finally, we compare the OpenCL and the hybrid implementations.

For comparing the different implementations, a data set of 16.5 million randomly generated 512-bit numbers, which we shall call the *Basic Workload* is used. 512-bits is the width of the OpenCL memory bus for our FPGA platform; hence our modules could receive up to one new key per clock cycle without any potential width conversion overhead. The sort-merge implementation used in our benchmarks has 16 *Sorters* configured to sort runs of 1022, 512-bit numbers, and four 2-way *Mergers* for merging the sorted runs.

The implementation runs on the D5005 PCIe-based Intel® FPGA acceleration card supporting a Stratix® 10 FPGA and 32 GB of DDR4 RAM [Cor19a]. We use OpenCL compiler version 19.1 which supports a maximum DDR RAM throughput of 20 GB/s at 333 MHz on our FPGA. The FPGA RAM is divided into 4 channels. OpenCL uses memory interleaving to provide fast access to all 4 channels at the same time [Int19, ZZZ00].

We present logic utilization (LU), BRAM utilization (BU), DDR RAM bandwidth utilization, throughput, and workload execution time (ET) to characterize and compare different implementations. Moreover, the OpenCL compiler provides for each loop in every module, a Loop Initiation Interval (II) and a latency value. II is the estimated number of clock cycles between the launch of successive loop iterations [Int19]. Indeed, based on the amount of dependency between loop iterations, OpenCL can pipeline the hardware needed for an iteration so multiple of them can be launched in parallel, resulting in smaller II values. An II value of 1 is the theoretical best and means that a new loop iteration can launch at every clock cycle. The latency of a loop is the number of clock cycles it takes for an iteration to finish.

It is important to note that the OpenCL compiler first compiles the input C code into RTL before synthesizing it into an image for the FPGA. However, this generated RTL is not "human-readable", especially for medium to large designs. This makes a deep analysis of OpenCL and a perfect comparison with custom RTL practically impossible and leaves us with the aforementioned metrics such as II and latency to characterize and compare OpenCL and RTL code.

6.4.1 OpenCL Sort-Merge Implementation

The characteristics of the OpenCL implementation of our sort-merge algorithm and its performance on the *Basic Workload* are shown in Table 6.1. The entire architecture took us about 3 weeks to implement and optimize.

Impl.	Freq. [MH]	LU [ALM]	BU [M20K]	Thr. [MB/s]
OpenCL	223	$13,725$ $(1,47\%)$	52 (0.44%)	126
RTL	325	$4,488$ (0.48%)	27 (0.23%)	843

Table 6.2: Comparison between the OpenCL and RTL implementations of the *Sorter*. Throughput (Thr.) is for sorting random 512-bit keys producing runs of 1022 numbers.

Both the *Access* and *Distributor* modules have a main loop where they continuously read data and distribute it to the lower layer of the distribution network. Both loops have an II of 1. It signifies that at every clock cycle, they can read and distribute one new element. Hence, they reached maximum theoretical performance.

The *Merge Strategy*, *Scheduler* and *Synchronizer* have IIs of respectively 3, 1, and 1 for their main loops. Only the *Merge Strategy* has an II larger than the minimum theoretical value; but this is fine given that it can still provide jobs to the *Mergers* far faster than they can execute them.

The *Sorter* modules are composed of two nested loops. In the first nested loop, the outer loop accepts a new element which the inner loop places in its correct position in the heap. The second nested loop is similarly structured with the outer loop yielding a new output and the inner loop restructuring the heap for correctness. It is crucial for the inner loops to have the highest theoretical performance. However, both inner loops have an II of 2. Therefore, they are estimated to have a throughput which is two times lower than they were designed to process. They also present an unexpected high latency of 21 and 13 clock cycles. This means that for every new input (resp. output) handled by the outer loop, it takes 21 (resp. 13) clock cycles before even the first iteration of the inner loop is done. Given that the inner loops need to iterate at most as many times as the height of the heap, these additional latencies incur a significant burden. Indeed, for our heaps of 1022 elements, the inner loops require 21 and 13 clock cycles to start, after which they will run for at most $II \times 10 = 20$ clock cycles. This level of overhead is unacceptable.

As for the *Mergers*, their performance-critical main loop compares two numbers to output the smaller of the two while replacing it with a new one. Because of the dependency between consecutive iterations (inherent to the nature of merging) we expect an II of 3. Indeed, the comparison (of two 512-bit numbers) should take approximately two clock cycles and the input/output functionality another, resulting in a 3 clock cycle delay before data for the next iteration is ready. However, OpenCL reports an II of 8 which is approximately three times higher than expected. It also reports a loop latency of 35, which is fine considering that the loop performs a large number of iterations.

Based on our analysis, we decided to use an RTL implementation of the performance-critical *Sorter* and *Merger* modules. The following two sections discuss improvements due to these changes.

6.4.2 RTL Sorters

In this section, we compare the characteristics of a single OpenCL *Sorter* against those of an equivalent RTL implementation, both based on the architecture described in Section 6.3.2. Our RTL *Sorter* has an II of 1 and a latency of 6 clock cycles. This is an improvement over the II of 2 and latency of up to 21 clock cycles for the OpenCL *Sorter*. Based on these values, we expect the RTL implementation to perform at most 2 to 3 times better than the OpenCL implementation.

Impl.	Freq. [MHz]	LU [ALM]	BU [M20K]	Thr. [MB/s]
OpenCL	267	$25,913$ (2.8%)	92 (0.78%)	$1,142$
RTL	302	$1,803$ (0.2%)	43 (0.37%)	$6,500$

Table 6.3: Comparison between the OpenCL and RTL implementations of the *Merger*. Throughput (Thr.) is for merging 2 sorted lists of 16.5 million 512-bit keys.

However, our benchmarks shown in Table 6.2 tell a different story. The RTL *Sorter*, despite using half as much the FPGA resources, reaches more than 6 times the throughput of the OpenCL *Sorter*. After normalizing for frequency, this improvement factor goes down to $4.5\times$, remaining well above our expectations. Given that the II and latency values obtained from the RTL description are accurate, we can only conclude that the OpenCL estimations are off by approximately a factor 1.5. This casts doubts on other modules' II estimations. The *Access*, *Distributor* and *Merger* modules are the only performance-critical components of our design. For the *Access* and *Distributor* modules, additional benchmarks show that they reach an II of 1 as advertised by OpenCL. As for the *Mergers*, they are discussed in length in Section 6.4.3.

Finally, it is interesting to note that each of the OpenCL and RTL *Sorter* designs took approximately 1 week to implement. The unexpectedly short implementation time of the RTL version is mainly thanks to OpenCL. Indeed, OpenCL can automatically generate simulation testbenches for modules written in RTL. This greatly reduces RTL development time as functional verification is a lengthy part of the digital design flow.

6.4.3 RTL Mergers

In this section, we compare the characteristics of a single OpenCL *Merger* against those of an equivalent RTL implementation, both based on the architecture described in Section 6.3.3. The RTL *Merger* has both an II and a latency of 3 clock cycles. The OpenCL *Merger* discussed in Section 6.4.1 achieves an II of 8 and a latency of 35 clock cycles. Based on these values, we expect the RTL implementation to reach approximately 3 times the throughput of the OpenCL implementation.

Table 6.3 shows the results of our benchmarks. The throughput numbers, after normalizing for frequency, indicate that the RTL *Merger* has 5 times the performance of the OpenCL implementation. This is again higher than the expected improvement factor of 3. It could be partially explained by the inaccuracy in the II values reported by OpenCL. Another plausible explanation is the inefficiency of memory accesses. Indeed, merging is a memory-intensive task with sequential burst memory access patterns. It could be that OpenCL is unable to make efficient burst accesses to the FPGA RAM. Further benchmarking dismissed this second hypothesis: a simple OpenCL kernel copying data from the FPGA RAM back to the FPGA RAM with the same access pattern as merging can achieve throughputs of more than 20 GB/s. This observation left us again with the inaccurate estimation of II by OpenCL as the only explanation. This is a major issue, as it affects the predictability of hardware designs in OpenCL.

Table 6.3 also shows that the RTL *Merger* uses approximately 10 times less logic and half as many BRAMs than the OpenCL *Merger*. We believe that these large factors are mainly due to the generic nature of the primitives used by OpenCL to implement various modules.

Finally, the RTL and OpenCL *Mergers* cost us each approximately one working week to implement. Our conclusions regarding development effort for the *Sorters* apply equally well for the *Mergers*.

Impl.	Freq. [MHz]	LU [ALM]	BU [M20K]	ET [s]	Thr. [MB/s]
OpenCL	184	$408,683$ (43.8%)	1762 (15%)	3.77	284
Hybrid	217	$163,848$ (17.5%)	1706 (14%)	1.03	1000

Table 6.4: Comparison between the OpenCL and hybrid OpenCL-RTL implementations of the sort-merge algorithm. Both implementations boast 16 *Sorter* and 4 *Merger* units. Execution time (ET) and throughput (Thr.) are for sorting the *Basic Workload*.

6.4.4 Hybrid OpenCL-RTL Sort-Merge Implementation

Based on our findings from Sections 6.4.2 and 6.4.3, we decided to use the RTL versions of the *Sorter* and *Merger* modules in our now **hybrid** sort-merge implementation. We continue using the OpenCL implementation of the remaining components.

Table 6.4 shows the characteristics of the hybrid sort-merge implementation. For the sake of comparison, we also included the characteristics of the equivalent OpenCL implementation from Section 6.4.1. The hybrid design runs at a higher frequency, uses considerably less logic, and has $3.5\times$ higher throughput. This is a smaller improvement in performance compared to what we obtained for the individual RTL modules in Sections 6.4.2 and 6.4.3. There are two reasons behind this. First, the frequency of the overall architecture is about 30% lower than the individual RTL modules in their own benchmarks. Second, all *Merger* and *Sorter* modules access the FPGA RAM simultaneously. In our sort-merge implementation with 16 *Sorters* and 4 *Mergers*, memory bus congestion can play an important role in degrading system performance. Indeed, our hybrid design saturates the 14 GB/s DDR RAM bandwidth provided by the FPGA platform at a frequency of 217 MHz.

As mentioned earlier, the sort-merge implementation runs at a lower frequency than the individual *Sorter* and *Merger* modules. The amount of drop in frequency seems to be a function of the size of the design. For instance, increasing the number of *Sorters* from 16 to 32 reduces the frequency by about 30MHz. OpenCL has pragmas and compiler flags for controlling the design frequency. However, it seems incapable of reaching timing closure if higher frequencies are requested. Moreover, given that the RTL generated by OpenCL is not human-readable and that it does not provide links between the source code and the generated RTL, it is quite difficult to detect, correct or even influence timing closure or placement and routing at OpenCL level. The OpenCL programming guide suggests decreasing the size of the design or trying different compilation seeds as ways for dealing with failed timing closure [Int19]. The former may result in a loss of functionality or performance and the latter could only solve relatively small timing issues.

Our final remark on the hybrid implementation concerns portability. Intel® OpenCL provides seamless integration of RTL into OpenCL code: the RTL description is wrapped inside a C function that can be called anywhere within the OpenCL code. This means that our architecture remains OpenCL-compatible, keeping most of the advantages of the platform discussed in Section 6.1. However, although OpenCL itself is standardized, the version used by Intel FPGA tools is not universally adopted. This means that code written in OpenCL for Intel FPGAs may result in inefficiencies or not compile at all in tools from other FPGA vendors. Indeed, each vendor has its own extension to OpenCL, with unique pragmas and optimization features added to support FPGA development. RTL languages (e.g., VHDL, Verilog) do not have this issue.

6.5 SUMMARY & DISCUSSION

In this chapter we quantitatively measured the tradeoffs of using OpenCL and RTL for implementing sort-merge primitives. We compared our implementations in terms of performance, FPGA resource utilization, and required development effort. In what follows, we list and discuss our main findings:

- **Performance:** Despite similar development effort, our RTL implementations of the *Sorter* and *Merger* modules perform at least 4 times better than their OpenCL counterparts. However, for some of the simpler modules such as the *Distributors*, the OpenCL implementations reach optimal performance. We can therefore conclude that depending on the complexity of the algorithm, OpenCL may or may not achieve performance comparable to that of hand-crafted RTL. Nonetheless, it is difficult to quantitatively measure the complexity of a design in order to decide whether it should be implemented in OpenCL or RTL. In addition, with OpenCL-generated RTL being hard to read and the compiler reporting inaccurate efficiency measures, it is also difficult to estimate the quality of an OpenCL implementation. Our recommendation for deciding between OpenCL and RTL is based on the nature of the module. We recommend performance-critical modules be primarily implemented in RTL.
- **FPGA Resource Utilization:** The OpenCL vs. RTL benchmarks show that our RTL implementations use less than half the resources (both logic and BRAM) of their OpenCL counterparts. These results are to be considered in applications where FPGA resources are scarce, either due to the size of the design or that of the FPGA. They must also be taken into account in applications that achieve scalability through replication. For instance, with the same FPGA resources, we can implement twice as many RTL *Sorters* than OpenCL ones, resulting (theoretically) in twice the sorting throughput.
- **Development Effort:** Optimizing the OpenCL implementations of the *Sorter* and *Merger* modules took us nearly as much time as efficiently implementing them in RTL. However, RTL development effort was greatly reduced thanks to the testbench generation and simulation capabilities provided by OpenCL for RTL modules.

To put our findings into context, consider the development of a database system accelerator. It has many more performance-critical modules (e.g., filter, aggregation, join, projection) than the simple sort-merge design studied in this chapter. These modules are typically placed in a pipeline where bottlenecks should be avoided (see Section 5.2). Indeed, a slow module placed anywhere in the processing chain slows down the entire pipeline. It is therefore crucial for these modules to be implemented using RTL. When it comes to resource utilization, one must remember that database system accelerators are large systems, so resource efficiency is essential for fitting them on an FPGA (see Section 4.3.4). This gives us another reason to use RTL instead of OpenCL (or HLS in general), given that it almost universally results in considerably lower resource utilization. To conclude, our analysis shows that **a fully RTL-based implementation is more suitable for database system acceleration**. The only potential downfall is increased simulation effort. We can overcome this by using modern HDLs such as Chisel allowing us to write testbenches in high-level languages such as Scala. Moreover, they help us achieve the compile-time flexibility we need to customize the architecture for a particular workload and FPGA platform, as discussed in Section 5.1.

7

RESOURCE-EFFICIENT ACCELERATION OF PIPELINE-BREAKING DATABASE OPERATORS ON FPGAS

7.1 The Case for Resource Efficiency

7.2 Related Work

7.3 Architecture

7.4 Experiments

7.5 Summary

FPGAs are composed of a limited amount of reconfigurable resources. This restricts the number and type of modules (i.e., circuits) that an FPGA can support at the same time. Therefore, resource efficiency is crucial for accelerating large database systems on FPGAs. In this chapter, we propose morphing sort-merge (MSM): a sort-based database system accelerator that achieves resource efficiency by reusing the FPGA's resources to execute different pipeline-breaking database operators at runtime. Our benchmarks show that MSM reaches an average speedup of $5\times$ compared to MonetDB.

Note that parts of the material presented in this chapter have been previously published in DaMoN'21 [MFL+21].

7.1 THE CASE FOR RESOURCE EFFICIENCY

Resource constraints are a major hurdle in system design on FPGAs. In the context of database system acceleration, these constraints are often not a limiting factor for streaming operators (e.g., filter, projection) as they have low complexity and thus require few resources [FMH+20]. The challenge is in implementing the more complex pipeline-breaking database operators. Indeed, current implementations of both hash-based [HSM+13, ABW+16, HANT15] and sort-based [STM+13, CO14] database operators on FPGAs are quite resource-demanding.

The authors of [HSM+13] report a 26% BRAM utilization for hash joins on a Stratix IV FPGA. In [ABW+16], around 20% of a Virtex-6 FPGA's BRAMs are used for hash-based aggregation. The authors of [HANT15] store their hash tables on the FPGA RAM resulting in an 18% BRAM utilization and a whopping 46% logic utilization for hash joins on a Virtex-6 FPGA. The authors of [STM+13] report approximately 24% BRAM utilization for sorting with a Stratix V FPGA while [CO14] reports at least a 50% BRAM utilization for a 12-level merge tree on a Virtex-6 FPGA.

We dedicated Section 4.3.4 of this thesis to a survey on various methods used by existing database system accelerators to overcome FPGA resource limitation challenges. In this chapter, we describe the tradeoffs of these methods and proposes our own technique for efficiently accelerating the sort, aggregation, and join operators by optimizing the usage of FPGA resources.

An FPGA design can achieve *resource efficiency* by reusing its dedicated resources to support different functionality through runtime configurability. *Compile-time parameters* are parameters of the RTL code that often affect the architecture of the design and can only be changed before compilation begins. *Runtime parameters* are control-flow and data-flow parameters that can be altered while the design is running on the FPGA. *Runtime configurability* allows users to change the behavior of the design while it is running on the FPGA, through runtime parameters.

In Section 7.3 we propose a resource-efficient database system accelerator, called MSM. MSM is a set of FPGA modules based on the sort-merge algorithm that can support any of the sort, aggregation, and join operators through **runtime configuration**. We also introduce a few optimization mechanisms (e.g., dynamic distribution and feedback) that improve the performance of MSM on both uniform and highly skewed datasets. MSM also achieves efficient RAM bandwidth utilization, as explained in Sections 7.3.2, 7.3.4, and 7.3.5. Benchmarks in Section 7.4 demonstrate throughput speedups of on average $5\times$ compared to a 28-threaded MonetDB installation.

7.2 RELATED WORK

In Chapter 5, we defined the objective of our research to be the acceleration of pipeline-breaking database operators. Past research listed in Section 4.3 proposes a few efficient techniques for implementing these operators. We summarize and describe the tradeoffs of these techniques below.

Hash-based techniques are commonly used to accelerate the join [PHL18, HSM+13, HANT15] and aggregation [PHL18, WIA14, WFS+19, ABW+16] operators. Hash tables stored on the BRAMs of the FPGA benefit from low-latency and high-throughput, but often lose performance or rely on subsequent processing to deal with overflows and collisions [WIA14, WFS+19]. Indeed, collision resolution using techniques such as linear probing or double hashing makes hash table access times unpredictable, thus greatly reducing the efficiency of the processing pipeline [WTA13, UIO15]. Hash tables stored on the FPGA RAM benefit from latency hiding through pipelining and offer better resistance to overflows and collisions, but are limited by the bandwidth of the RAM which is lowered due to their random access patterns [HANT15, WIA14]. Cache based implementations using constructs such as content addressable memories (CAM) try to strike a balance between the two but are limited due to the complexity of large CAMs negatively affecting the design frequency [ABW+16]. Hash-based sorting attempted in [PHL18] constraints the range of the input keys and may result in sub-optimal memory usage depending on their statistical distribution [Gil04].

Sort-based techniques present a more generic and end-to-end solution compared to hash-based techniques [STM+13, CO14]. Indeed, when operating on sorted tables, pipeline-breaking operators become much simpler to implement, often turning into single-pass algorithms [Sch09]. Moreover, sorting performance is typically far less sensitive to the distribution of data, compared to the performance of a hash table. However, sorting presents some of its own disadvantages. Sort-merge, the commonly used algorithm for sorting large tables may require multiple passes over the data; therefore, it is often memory bound [CO14]. Furthermore, depending on the size and distribution of the input tables, sorting might result in sub-optimal performance on hash-friendly operators such as aggregations and joins [UIO15]. Indeed, the expected algorithmic complexity of hashing ($\mathcal{O}(n)$) is lower than that of sorting ($\mathcal{O}(n \log(n))$). However, the cost of sorting is often amortized across multiple operators through interesting orders [SAC+79, GSDB12].

FPGA reconfiguration (reprogramming) can be used to switch between hash-based and sort-based implementations. For instance, the authors of [UIO15] implement sort-merge join and hash join as separate FPGA images and switch between the two based on a cost model. Approaches based on reconfiguration have a few drawbacks. First, the full reconfiguration of an FPGA takes time in the order of seconds to perform, limiting system performance if the stream of operators requires frequent image swaps. Second, reconfiguration resets all FPGA logic, hence the accelerator can only be dedicated to executing a single operator at a time. Third, reconfiguration might also reset the RAM controllers potentially leading to data corruption [WPC+16]. Therefore, data needs to be reacquired by the FPGA after every image swap. **Dynamic partial reconfiguration** techniques, whereby only a portion of the FPGA chip is reconfigured, could alleviate the latter two limitations but the first one remains [VF18]. The authors of [DZT12] take advantage of this technology to accelerate streaming operators projection and restriction.

We design our accelerator for multi-client data center environments where a continuous stream of concurrent and consecutive queries needs to be executed. The overhead of potentially reconfiguring (a part of) the FPGA before the execution of every query is too high for our application. Among the other two techniques, sort-based implementations

appear to be more adequate for FPGA acceleration. To start, they allow us to efficiently implement all three of our target pipeline-breaking database operators. This is how we justified sort-based query processing in Chapter 5. Furthermore, compared to hash tables that needs to deal with overflows and collisions, sorting follows a more regular execution flow when dealing with a large number of rows or various data distributions. This has two implications. First, sorting has a more predictable performance compared to hash tables whose performance may be significantly impacted by the number of collisions and overflows. Second, sorting is more resource efficient. Indeed, in digital design, the worst-case time of an operation may determine the complexity of the implementation [KM10]. This means that for hash-based techniques, the circuit employed for dealing with collisions/overflows, no matter how rarely used, must still be present and run on the FPGA all the time. Sorting achieves higher resource efficiency by using all of its circuits during normal operations, thanks to its more regular execution flow. Resource efficiently is of course also achieved by sharing the sort pipeline among all three of the pipeline-breaking database operators, as explained earlier in Chapter 5.

In this chapter, we use **runtime configurable** sort-merge primitives to support multiple database operators and to adapt to different data distributions on a single FPGA image. This allows us to achieve an efficient use of FPGA resources. The idea of runtime configuration has been successfully applied in past work [SIOA17, TWN12, WIA14], but often at smaller scale and for single operators (e.g., defining automata for regular expression matching, or predicates for filtering). To the best of knowledge, this is the first work exploring this idea for accelerating distinct resource-intensive pipeline-breaking database operators, and with the goal of achieving both high performance and resource efficiency.

7.3 ARCHITECTURE

The architecture of MSM is based on the sort-merge algorithm, described in detail in Section 4.3.2. As a reminder, the algorithm works as follows. A streaming sort phase transforms a table into small runs of sorted rows. A subsequent merge phase merges the sorted runs, after potentially multiple passes, into the sorted output. Our main contribution to the typical sort-merge algorithm is the design of a sort phase that can morph into an efficient early aggregation phase, and that dynamically adapts to the statistical distribution of the input for higher performance. *Early aggregation* is a streaming algorithm capable of partially aggregating data, but which requires further processing by the merge phase to ensure complete aggregation [Lar02]. The benefit of early aggregation is that it may reduce the amount of data there is to merge, thus improving the performance and memory bandwidth efficiency of the aggregation operator. Early aggregation is discussed in greater detail in Chapter 9.

An overview of our architecture is shown in Figure 7.1. The sort phase of our algorithm is performed by a *Sort-Network* and the merge phase via a multi-way *Merge-Network*. They both access the FPGA RAM to read initial and intermediate data and to store their results. Throughout our architecture, we use modular components with standardized interfaces. This makes it easy to both update previous modules and insert new ones for extending the functionality of the accelerator.

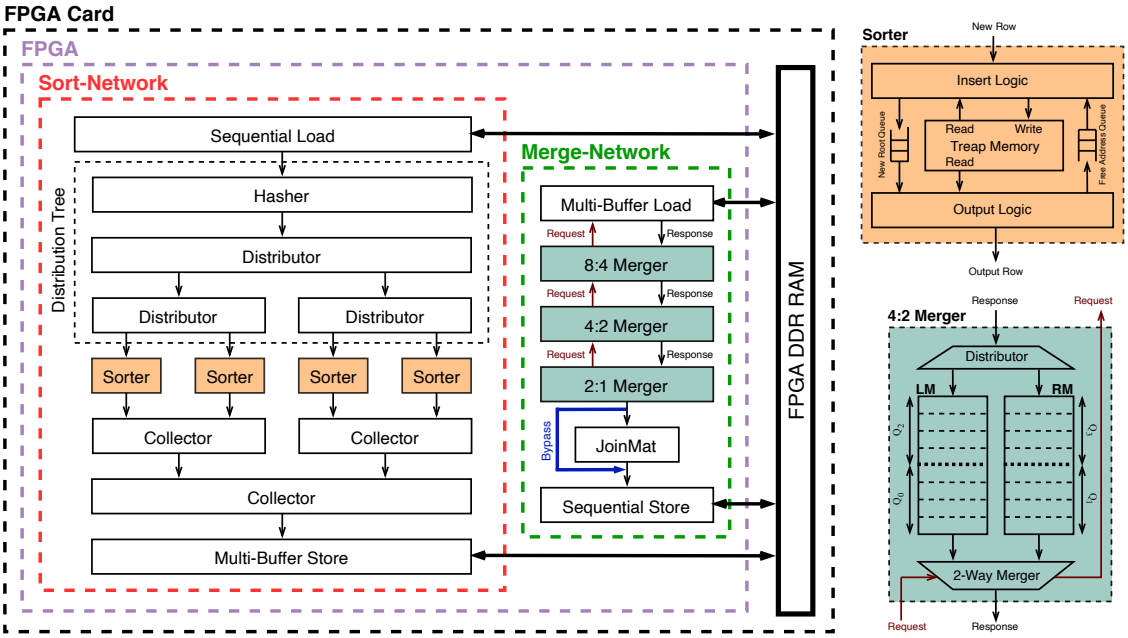


Figure 7.1: Architecture of our database system accelerator MSM

7.3.1 Sorters

A *Sorter* in the *Sort-Network* transforms a stream of rows into sorted runs of c_S rows and can be configured to perform early aggregation at the same time. For sorting, we considered algorithms that offer performance, resource efficiency, and scalability; as previously discussed in Section 6.3.2. Sorting networks and linear sorting were dismissed due to their resource requirement (e.g., $O(c_S \log^2(c_S))$ comparators needed for bitonic sorting networks [Bat68]) and scalability issues. We also dismissed algorithms with relatively high time/space complexity such as insertion sort, selection sort and mergesort. To support **early aggregation**, we considered algorithms and data structures that ensure that an inserting row is compared and aggregated with a potentially existing row with the same key, allowing them to absorb more rows than they output. This is for instance not the case for the heapsort algorithm explored in Chapter 6. With these criteria in mind, we selected the treap data structure as the basis of our *Sorters*, both for its simplicity and to enable some further optimizations discussed below.

A *treap* is a randomized binary tree where each node consists of a key and a random priority. The treap is a **binary search tree (BST) with respect to the key values** and, simultaneously, a **max-heap with respect to the priorities** [MR95]. To insert a new key, the treap is first considered to be a BST. The key is inserted as a leaf of the tree. Next, a random priority is assigned to the key, and left/right rotations are performed until the heap property is satisfied. It can be shown that the expected depth of a treap of c_S nodes is in $O(\log c_S)$ [MR95]. As a result, the treap is statistically expected to be balanced.

In our implementation, each node of the treap represents a database row. The row's key becomes the node's key, and its columns are considered as values attached to the key and stored within the node. The treap is stored in the BRAMs of the FPGA enabling fast random access. It can hold at most c_S nodes, a compile-time parameter bounded by the FPGA's limited BRAM resources. The randomized priorities are generated by a Galois linear-feedback shift register [Lim08]. When the treap becomes full, an in-order traversal of its nodes extracts the rows in **sorted order**, until the treap is empty again. We define

this as the *fill-empty cycle* of the *Sorter*. The treap capacity c_S defines the size of the sorted runs produced by a *Sorter*.

We use treaps for both sorting and early aggregation. Given that a treap is a BST with regards to its keys, the first step in inserting a row ρ_i into the treap is inherently the same as searching for a node representing a row with the same key. If such node does not exist, a leaf will be reached where ρ_i is inserted. Otherwise, the search leads to a node containing row ρ_j with the same key. In this case, depending on the target operator (sort or aggregation), different actions need to be taken:

- **Sort:** The search ignores ρ_j and continues down the tree.
- **Aggregation:** ρ_j is updated with the result of accumulating ρ_i into ρ_j. ρ_i is discarded and the insertion operation terminates.

The runtime configurability of this choice of action translates into a *Sorter's* morphing capabilities, hence our goal of **resource efficiency**. Note that regardless of the operator, a *Sorter* always produces sorted runs which can later be merged to complete sorting or aggregation.

We conclude this section by describing some of the performance improving optimizations applied to our *Sorter* implementation.

Prefetching The BST-like part of the treap insertion process requires the *Sorter* to compare a new key to a sequence of the nodes of the treap starting from the root down to (potentially) a leaf where it should be placed. The comparison result with a given node determines which of its left or right children must be **fetched** and **compared** next. If every comparison requires multiple clock cycles to perform, the *Sorter* can preemptively fetch (from the BRAMs) the children of a node ensuring that they will be ready to compare once the current comparison completes. This results in latency hiding, thus better utilization of the BRAM bandwidth and a faster treap insertion operation. Needless to say, depending on the result of the current comparison, only one of the prefetched children should be compared next. A further optimization can be applied to **pipelined** multi-cycle comparators: in case the prefetched children of a node are received before its comparison ends, they can speculatively start their comparison to the new key as well.

Parallel IO This optimization parallelizes the fill-empty cycles of a *Sorter*. As the *Sorter's* treap becomes full and is being emptied, a new treap can grow in its place, reusing the incrementally freed up memory of the previous instance. The *Sorter* architecture is illustrated in Figure 7.1. A quad-port memory is the centerpiece capable of holding up to c_S treap nodes. Initially, the *Insert Logic* populates a treap. Once full, it notifies the *Output Logic* about the existence of a new treap that needs to be emptied producing a sorted run. It does so by pushing a pointer to the root of the treap into the *New Root Queue*. It then moves on to create a new treap. Every time the *Output Logic* outputs a row, it frees a node by pushing its pointer to the *Free Address Queue* so it can be reused by the *Insert Logic* when inserting a row into the new treap.

Feedback This optimization is based on the observation that nodes closer to the root of a treap are the first to be reached, hence more quickly examined while inserting a new row. It aims at improving the performance of **early aggregation** on highly skewed data by bringing nodes with frequently occurring keys close to the root of the treap. We use feedback as a means to implement this optimization. Each node of the treap maintains a counter for the number of times it has been aggregated, called *hit count*. When the treap becomes full, it is emptied and a new treap is created. The feedback optimization reinserts nodes with a high hit count back into the new treap and assigns to them a **relatively high priority** to ensure that they remain close to the root. This happens automatically thanks to the max-heap and randomization properties of treaps.

7.3.2 Sort-Network

As the *Sorters* sort/aggregate with an expected time complexity in $O(c_S \log c_S)$, they require on average $O(\log c_S)$ clock cycles to process a new row. In order to reach higher throughputs, a sort-network with multiple *Sorters* sharing the load is required. We use n_S to refer to the number of sorters in a sort-network. Figure 7.1 presents the architecture of the sort-network. The *Sequential Load* module reads and streams the rows of a table stored on the FPGA RAM to the *Hasher*. The latter hashes the rows based on their keys and distributes the `(row, hash)` pairs through the *Distributor* modules down to the *Sorters*. The distribution strategy varies depending on the data and is described later in this section. Depending on the operator, the *Sorters* produce sorted and potentially partially aggregated runs of c_S rows. The *Collector* and *Multi-Buffer Store* modules collect the sorted runs and store them in separate buffers on the FPGA RAM so they can be further processed by the merge-network (see Section 7.3.4).

The dynamic distribution strategy implemented by the distribution tree helps MSM achieve lower reduction factors for **early aggregation**. *Reduction factor* is the ratio of the number of rows at the output of the *Sort-Network* over its input. Improved data reduction results in fewer sorted runs being produced, therefore less work for the merge phase. This in turn reduces RAM bandwidth utilization and improves performance. The optimization ensures that rows with identical keys are distributed to the same *Sorter*, so they have a better chance of being aggregated. To achieve this, the *Distributors* can switch between two distribution mechanisms:

- ***Hash-based distribution mechanism*:** The hash-based mechanism distributes a row to a *Sorter* based on the hash of its key, hence guarantees that rows with identical keys are distributed to the same *Sorter*. This allows the n_S *Sorters* of a *Sort-Network* to collectively absorb up to $c_N = n_S \times c_S$ unique keys, where c_N denotes the capacity of the *Sort-Network*.
- ***Availability-based distribution mechanism*:** The availability-based mechanism distributes a row to the first *Sorter* that is available to accept it, ensuring *maximum throughput* by keeping all *Sorters* busy at all times.

The *dynamic distribution strategy* consists of switching between the two mechanisms based on the input data. The *Hasher* regularly and heuristically measures the statistical distribution of the hashed keys. Upon discovering that the input rows can be evenly distributed to the *Sorters* based on the hash of their key, the *Hasher* activates the hash-based distribution mechanism resulting in higher-quality early aggregation. Otherwise, the availability-based distribution mechanism is selected in order to ensure maximum throughput. Note that for the sort operator, the availability-based distribution mechanism is used regardless of the distribution of the data.

7.3.3 X:Y Mergers

An $X : Y$ *Merger*, merges X buffers of sorted rows into Y new sorted buffers. Our implementation of $X : Y$ *Mergers* assumes $Y = X/2$ to be a power of two. Therefore an $X : Y$ merger on input buffers $[I_0, I_1, ..., I_{X-1}]$ produces output buffers $[O_0, O_1, ..., O_{Y-1}]$ where O_i is the result of merging I_{2i} and I_{2i+1}. Figure 7.1 illustrates the architecture of a $4 : 2$ *Merger*. The implementation details are described below. For the sake of brevity, we shall use the term "merger" to refer to an $X : Y$ *Merger* throughout this section.

The mergers use request-response interfaces to interact with other modules and each other:

- **Input Interface:** The merger makes a **request** with ID r. The **response** should contain a burst of rows from I_r that will be buffered in a corresponding BRAM-based on-chip queue Q_r.
- **Output Interface:** The merger receives a **request** with ID r, and **responds** with a burst of new rows from O_r produced by merging on-chip queues Q_{2r} and Q_{2r+1}. If the queues are (nearly) empty, the merger uses the input interface to request new rows for filling them up.

Burst transfers reduce the frequency of requests and with it the impact of the overhead of handling each request. They also result in more efficient RAM transactions. **On-chip queuing** of the input buffers is used for latency hiding. Indeed, the response to a request is not guaranteed to arrive immediately after the request is made. This happens for instance when the requested new rows are on the FPGA RAM which has a high response latency. The on-chip queues Q_{2i} and Q_{2i+1} used to produce a given output buffer O_i are stored in separate BRAM-based memories (LM and RM in Figure 7.1). This enables the parallelized drainage of the queues; therefore, results in higher merging throughput. Note that only 2 memories are required for storing all the on-chip queues.

A 2-*Way Merger* is used in every $X : Y$ *Merger*, as shown in Figure 7.1. It merges two on-chip queues by following a 3-step cycle. First, it waits until the tops of both queues are read, therefore a row is present at each of its 2 inputs. We shall name these rows ρ_i and ρ_s. Next, the keys of ρ_i and ρ_s are compared against each other. Finally, the row with the smaller key is popped from the input and presented at the output. The merger's **morphing capability** to support both sorting and aggregation comes from the action taken when ρ_i and ρ_s have the same key:

- **Sort:** The equality of the keys is ignored. One of ρ_i or ρ_s is arbitrarily forwarded as the output.
- **Aggregation:** ρ_i and ρ_s are both popped from the inputs, accumulated, and the result presented at the output.

It is important to note that in the context of merging, aggregation is done completely as opposed to partially.

7.3.4 Merge-Network

The *Merge-Network* is responsible for merging the sorted runs produced by the *Sort-Network*. The architecture of the *Merge-Network* is shown in Figure 7.1. The *Multi-Buffer Load* module reads from up to n_W sorted buffers on the FPGA RAM to feed a $n_W : n_W/2$ *Merger*. The latter is the first in a sequence of mergers that together are functionally equivalent to a $n_W : 1$ *Merger*. For both the sort and aggregation operators, *JoinMat* (used for joins as explained in Section 7.3.5) is bypassed, and the final sorted or aggregated run is written by the *Sequential Store* module to the FPGA RAM. Depending on n_W and the number of sorted runs produced by the *Sort-Network*, the *Merge-Network* may need to perform multiple merge passes. As explained in Section 7.3.3, the merge results are always sorted, but can also be aggregated if the $X : Y$ *Mergers* are configured for aggregation.

The *Merge-Network* achieves **resource efficiency** through the morphing capabilities of its $X : Y$ *Mergers* and by only requiring $log(n_W)$ of them. This logarithmic resource utilization helps achieve higher n_W, resulting in a more **efficient use of the RAM bandwidth** by reducing the number of merge passes required to sort relatively large tables. A similar optimization is proposed in [CO14].

7.3.5 Join Materialiser (JoinMat)

MSM can perform $M : N$ (equi-)joins where $M < \ell$, with $\ell \in \mathbb{N}_1$ a compile-time parameter. N, however, can be arbitrarily large, only limited by the size of the FPGA RAM. $M : N$ joins are defined in Section 2.2.5.

The execution of $\text{Join}_k(R, S)$ starts by sorting the **union** of both input tables, on **modified keys**. The union of the input tables is sorted by programming the *Sequential Load* module of the *Sort-Network* to read and stream R and then immediately S, together as one large table. *Modified keys* are generated on the fly by shifting a table-ID bit to the least significant position of each row's key. A key k becomes $\overline{k0}$ if from table R and $\overline{k1}$ if from table S. Once sorting is done, the resulting rows are ordered according to their key and within rows with the same key, according to their table-ID (table 0 and then table 1). The modified keys can now be ignored. The *JoinMat* module is activated on the last merge pass. As sorted rows stream into the *JoinMat* module, it stores rows with the same key k_r from table R in a BRAM-based on-chip buffer. Once a row ρ_s from S arrives, its key is compared to k_r and in case they are equal, the join of ρ_s with each of the buffered rows from R is produced. This routine continues until all the rows from S with the same key k_r are processed. The buffered rows from R are then discarded and *JoinMat* restarts the process with new keys from R. Note that the size of the on-chip buffer defines the constant ℓ described above. It is limited by the amount of BRAM that can be dedicated to *JoinMat*. In our experiments in Section 7.4, we set $\ell = 512$, which is largely sufficient for our join benchmarks. Also note that executing *JoinMat* after the last merge pass achieves **RAM bandwidth efficiency** by eliminating the need to store intermediate sort results in the FPGA RAM.

7.4 EXPERIMENTS

In this section, we present the results of our benchmarks comparing the throughput of MSM against that of the state-of-the-art CPU-based database management system MonetDB. The *input (resp. output) throughput*, measured in millions of rows per second (MR/s), is the size of the input (resp. result) table divided by the execution time of the operator. The reported throughput values are the average over benchmarks repeated 5 times after warm-up runs.

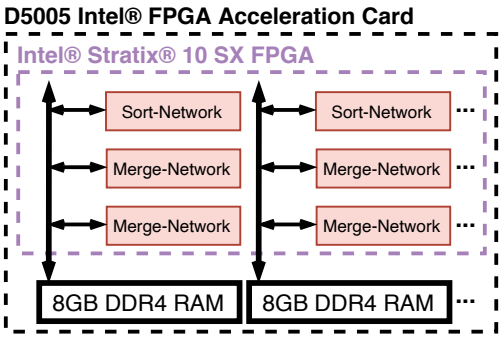


Figure 7.2: Setup of our benchmarking platform. Multiple *Sort-* and *Merge-Networks* are instantiated to improve the performance the system, and to benefit from the four DDR RAM channels provided by the accelerator card.

7.4.1 Experimental Setup

Following our findings in Chapter 6, we decided to implement MSM in Chisel 3.4.2 [Chi21a, BVR+12]. The implementation runs on our PCIe-attached D5005 Intel® FPGA acceleration card with an Intel® Stratix® 10 FPGA and 32 GB of RAM [Cor19a] provided using four 8 GB channels. At a clock frequency of 195 MHz, each RAM channel provides a bandwidth of about 12.5 GB/s for a total of $4 \times 12.5 = 50$ GB/s.

Our software benchmarks use MonetDB v11.39.5 [Mon21] running on an Intel® Xeon® Platinum 8180 CPU @ 2.50GHz (28 cores, 38.5 MB L3 Cache, 6 memory channels) with 376 GB of DDR4 RAM [Int17]. The benchmarks are run by up to 4 clients in parallel, matching the capabilities of our MSM implementation which will be described shortly. MonetDB uses up to 28 CPU threads. Increasing the number of threads beyond this values did not result in further performance improvements. The reported performance numbers from MonetDB do not include query plan generation time but only pure query execution time on warm data, i.e., data cached in memory.

7.4.2 Implementation Description & Tuning

Our implementation of MSM uses a 128-bit data path in order to process database rows that are up to 128 bits wide. The row layout consists of **four 32-bit integer columns**. The keys can span any number of these columns. Non-key columns can be used as values for the aggregation (SUM(), COUNT(), MIN(), and MAX() functions are supported) or as a payload for the sort and join operators. With this level of runtime configurability, we can process datasets commonly used for benchmarking in-memory query processing algorithms [ABW+16, BATÖ13, BLP11, BMK99].

Let us now discuss the parameter tuning process for our implementation. We aim to maximize the capacity of the *Sorters* (c_S) in order to increase the size of the sorted runs, while also keeping BRAM size and latency at reasonable levels. *Sorter* capacity is set to 2048. The *Sort-Network* has $n_S = 32$ *Sorters* in order to achieve *line-rate processing*, i.e., to process one row per clock cycle, therefore maximizing pipeline efficiency. The *Merge-Network* performs 256-way merging so that only **2 merge passes** are necessary to sort up to 130 million rows (or a 2 GB table). Therefore, for large tables the overall throughput of merging is 2× slower than that of producing the sorted runs. To equalize the throughputs, we use 2 *Merge-Network* to process the sorted runs produced by a single *Sort-Network*.

Module	R	Logic Utilization [ALM]		BRAM Utilization [M20K]	
Full System	1	489,602	(52%)	6,558	(56%)
MSM Core	4	102,156	(11%)	1,533	(13%)
Sort-Network	1	73,173	(7.84%)	947	(8%)
Sorter	32	1,413	(0.15%)	24	(0.2%)
Distribution Tree	1	8,401	(0.9%)	4	(0.03%)
Collection Tree	1	14,079	(1.51%)	128	(1.1%)
Merge-Network	2	14,492	(1.55%)	293	(2.5%)
2:1 Merger	1	869	(0.09%)	10	(0.085%)
256:128 Merger	1	1,121	(0.12%)	106	(0.9%)
JoinMat	1	540	(0.058%)	6	(0.051%)

Table 7.1: FPGA resource utilization of our implementation of MSM (with a 128-bit wide data path) and some of its major components. Replication factor (R) is the number of instantiations of a module within its level in the hierarchy (e.g., there are 32 *Sorters* in a *Sort-Network*). The percentage resource utilizations are calculated over the total amount of resources provided by the FPGA.

The benchmarking platform achieves high throughput and support for multiple clients through replication, an idea proposed in Chapter 5. Figure 7.2 shows a high-level view of the platform. Each of the 4 RAM channels is connected to a *Sort-Network* and 2 *Merge-Networks*, together forming an *MSM Core*. Thanks to the 4 *MSM Cores*, we can issue four operators at the same time matching the capabilities of our 4-client MonetDB. This enables us to execute multiple database queries or operators within the same query in parallel. Alternatively, partitioning can be used to execute a single operator using multiple *MSM Cores*. As a general tuning guideline, replication should be practiced until either the RAM bandwidth or the FPGA resources are saturated, and a reasonable frequency is still attainable. MSM is particularly suitable for replication as its morphing capabilities result in less constraints and limitations during operator scheduling. Moreover, MSM's efficient use of FPGA resources allows higher replication factors.

The resource utilization of our entire implementation along with some of its key modules is shown in Table 7.1. All modules run at 195 MHz, which is the highest achievable by the synthesis tools. Throughout our benchmarks, on average less than 62% of the FPGA RAM bandwidth was utilized.

Influence of Architectural Parameters on Resource Utilization

We now present a brief analysis of the most important parameters influencing FPGA resource utilization in our system. The resources allocated to the *Sort-Network* are mostly dedicated to the *Sorters*. Each *Sorter* in our main implementation has a 128-bit wide data path and the capacity to hold $c_S = 2048$ rows. These parameters have a strong impact on its resource utilization, as demonstrated in Table 7.2. Data path width has a greater impact on logic utilization than capacity does. This is because increasing the data path width requires more arithmetic and logic units for handling wider rows (i.e., more columns and/or larger columns), whereas increasing the capacity requires only wider pointer structures and pointer arithmetic units for handling deeper memories. In contrast, capacity has a greater impact on BRAM utilization than data path width does. This is because increasing the capacity also increases the memory overhead of storing larger pointers in the treap data structure kept in the BRAMs of the *Sorter*.

Let us now discuss the resource utilization of the *Merge-Network*. In our implementation, the 256-way *Merge-Network* consists of eight $X : Y$ *Mergers* (256 : 128 *Merger*, 128 : 64

Data Path Width	c_S	Logic Utilization [ALM]	BRAM Utilization [M20K]
128	2048	1413	24
128	4096	1601	48
128	8192	1772	96
128	2048	1413	24
256	2048	2384	37
512	2048	4438	62

Table 7.2: *Sorter* resource utilization as a function of capacity (c_S) and data path width.

Merger, ..., $2:1$ *Merger*) consuming most of its resources. For the sake of brevity, the resource utilization of only the smallest and the largest *Mergers* are provided in Table 7.1. The X and Y parameters of a *Merger* mainly influence its BRAM utilization. This is because each of the X inputs to a $X:Y$ *Merger* needs a dedicated buffer space in BRAM (see Section 7.3.3). The logic utilization, however, is for the most parts unchanged as X varies.

Next, we measure the amount of resource overhead inflicted by runtime configurability on our architecture. To this end, we hardcoded all configurable parts of our implementation, and stripped away modules and optimizations pertaining to the aggregation and join operators (e.g., the *JoinMat* module, the *Sorter* feedback optimization) resulting in a simple sort-merge implementation. Resource utilization numbers show that the runtime configurable implementation supporting all 3 pipeline-breaking database operators requires 7.7% more logic and 1.5% more BRAM resources compared to the stripped-down version supporting only the sort operator. We conclude that support for multiple operators at runtime fully justifies the small additional overhead of the morphing logic.

Finally, let us say a few words about deployability. Indeed, a promise of our research is to ensure that the architecture of the accelerator is (resource-wise) flexible enough to allow deployment on different FPGA platforms. In Section 5.1, we argued that the sort-merge algorithm can be implemented with various area/performance characteristics, allowing the accelerator to adapt to a target FPGA platform, in terms of resources. The results presented in this section confirm our hypothesis. Indeed, the parameters of our *Sorters* (e.g., c_S), *Mergers* (e.g., number of ways), and the overall system (e.g., number of *MSM Cores*) all have tremendous effects on the accelerator's resource utilization and memory bandwidth requirements. They can easily be tuned (in parts also thanks to Chisel) to target a specific workload and FPGA platform.

7.4.3 Sort Benchmarks

The datasets used in our sort benchmarks consist of tables with a single key column and 3 payload columns. The *U-Random* dataset has its keys chosen randomly from a uniform distribution of all possible 32-bit integers. The *U-Sorted* dataset consists of sorted keys from a uniform distribution. All the keys in the *Single Point* dataset are equal. Tables from the U-Sorted and Single Point datasets have a few exceptions (noisy rows) to their order and equality properties to ensure that MonetDB cannot avoid sorting altogether.

The benchmark results are shown in Figure 7.3. The initial ramp-up of the throughputs is due to the operator launch overhead losing its significance against the actual sort time. MSM's sudden decrease in throughput happens when a new merge pass is required, i.e., when the table size is $c_S \times (n_W)^i$ or in our case $2048 \times (256)^i \forall i \in \mathbb{N}$. MSM achieves around 3 to 6 times higher throughput than MonetDB on the U-Random dataset. It performs better on the U-Sorted and Single Point datasets as well, but with MonetDB slowly catching up by benefiting from the existing order in those datasets. MSM's performance shows little variation among different datasets.

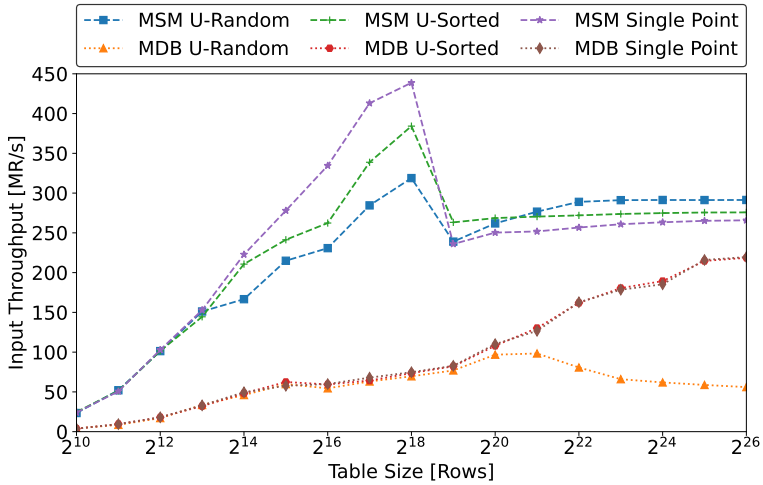


Figure 7.3: The results of our benchmarks of the sort operator comparing the performance of MSM and MonetDB (MDB) on the random uniform (U-Random), sorted uniform (U-Sorted), and single-key (Single Point) datasets.

7.4.4 Aggregation Benchmarks

The datasets used in our aggregation benchmarks consist of tables with a single key column and 3 value columns aggregated using the SUM() function. The keys are selected randomly from a **uniform** or **Zipf** (exponent 0.5) distribution or with a **moving cluster** generation mechanism (with a window size of 1024 values) [CR07]. The latter produces a sequence of keys where those with close values are clustered together. These datasets are commonly used to benchmark the aggregation operator [CR07, MSL+15, ABW+16]. All the tables in our datasets have 2^{26} rows.

The benchmark results are shown in Figure 7.4. MSM performs consistently well on the moving cluster dataset. This is thanks to the fill-empty cycles of the *Sorters*, allowing

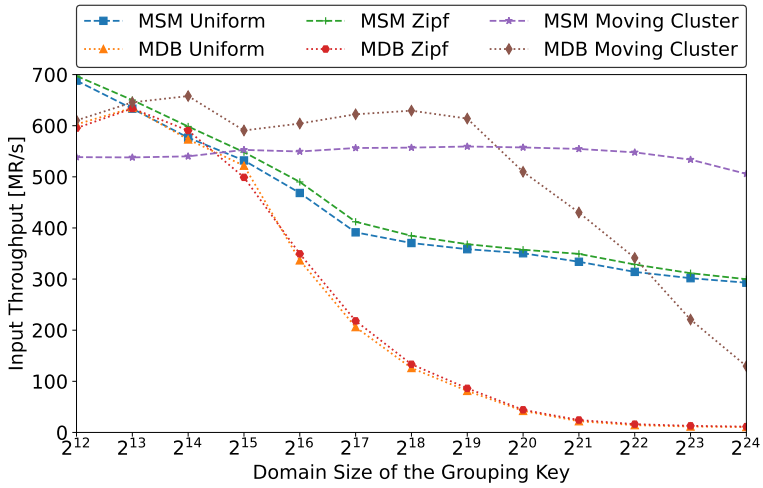


Figure 7.4: The results of our benchmarks of the aggregation operator comparing the performance of MSM and MonetDB (MDB) on the uniform, Zipf, and moving cluster datasets, with keys chosen randomly from the corresponding distributions.

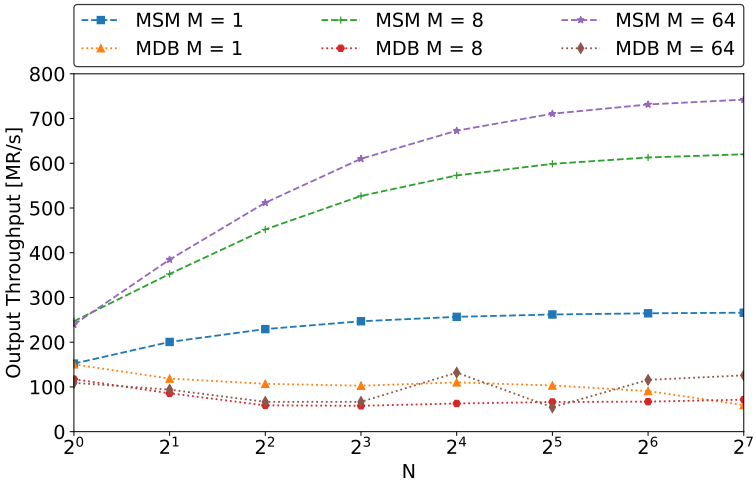


Figure 7.5: The results of our benchmarks of the $M : N$ join operator comparing the performance of MSM and MonetDB (MDB) on uniform datasets with various values for M and N.

them to adapt to distributions where the local range of keys changes over time. The moving cluster distribution exhibits such behavior by choosing the values of the keys based on their position in the table. Benchmarks on the Zipf and uniform datasets manifest an important decrease in performance until a domain size of 2^{17} values is reached, where the curves switch to a slower, more gradual decline. This happens just after the *Sort-Network* capacity $n_S \times c_S = 32 \times 2048 = 2^{16}$ is saturated. The Zipf dataset benefits from the feedback optimization of Section 7.3.1 and the uniform dataset from dynamic distribution introduced in Section 7.3.2. The sudden performance drops of MonetDB happen due to cache effects. MSM performs on average $5\times$ better than MonetDB.

7.4.5 Join Benchmarks

The datasets used in our join benchmarks consist of tables with 64-bit keys (spanning two 32-bit columns) and 32-bit payloads. The join result rows contain the join key, and the payloads of the two joined rows in the remaining two columns. In our benchmarks, MSM performs an $M : N$ join of tables R and S, with the keys of R and S chosen randomly from a uniform distribution with domain size D. The size of the input tables $|R|$ and $|S|$ and the domain size D are chosen so that the size of the cross product of the same-key rows from both tables is exactly $M \times N$, and that the join result contains 2^{26} rows.

The benchmark results are shown in Figure 7.5. MSM performance increases for larger values of M and N. Indeed, joins are executed by buffering same-key rows from R and pairing them, one by one in a loop, with all the rows from S that also have the same key (see Section 7.3.5). Increasing M and N increases the ratio of the join materialization time (proportional to $M \times N$) to the overhead of buffering the rows (proportional to M) and launching the materialization loop (proportional to N); thus effectively improving performance. MSM performs on average $5\times$ better than MonetDB on our join datasets.

7.5 SUMMARY

Given the resource constraints imposed by FPGAs, the quality of data processing units should not only be measured in their performance but also in how efficiently they use FPGA resources. In this chapter, we proposed MSM: a set of runtime configurable modules realizing both goals. **Resource efficiency is achieved** by reusing the same FPGA resources to support multiple pipeline-breaking database operators. This is a major accomplishment as pipeline-breaking operators are among the most popular, yet resource demanding operations executed by database systems. **High performance is achieved** by using parallel optimized sorters with a dynamic distribution mechanism, early aggregation, multi-way merging, and a multi-core design to support multiple clients in parallel. The performance of MSM measured against MonetDB shows relatively high speedups on various datasets. To conclude, MSM is in perfect alignment with the objectives of our work defined back in Chapter 5.

8

KERRAS: COLUMN-ORIENTED WIDE TABLE PROCESSING ON FPGAS

8.1 The Scope of Database System Accelerators

8.2 Related Work

8.3 Key-Reduce Radix Sort (KeRRaS)

8.4 Architecture

8.5 Experiments

8.6 Summary

In Chapter 7, we proposed MSM, a database system accelerator with support for the sort, aggregation, and join operators. MSM is a sort-based accelerator: It starts by sorting its input, before executing additional operators. Using sort-merge with multi-pass merging, it can sort arbitrarily deep tables. However, it is quite limited with regards to the table width. Indeed, the implementation proposed in Chapter 7 can only support tables with up to four 32-bit columns. The number and size of supported columns is a compile-time parameter of MSM and has a large impact on the FPGA resource utilization. Unfortunately, this lack of support for arbitrarily wide tables that are present in many modern database workloads, is ubiquitous in past research on FPGA accelerators.

In this chapter we propose KeRRaS, an abstract sorting algorithm that enables existing sort-based query processors to support arbitrarily wide tables while offering scalability, preserving modularity, and having low resource overhead. We then present an implementation of KeRRaS based on MSM. The new implementation, called MSMK, behaves similarly to MSM on narrow tables and demonstrates good scalability as the number of columns increases. An efficient implementation of KeRRaS requires us to transform our row-oriented accelerator (MSM) into a column-oriented accelerator (MSMK). This brings about a few more advantages (e.g., support for projections) discussed in this chapter.

Note that parts of the material presented in this chapter have been previously published in FCCM'22 [MFM+22].

8.1 THE SCOPE OF DATABASE SYSTEM ACCELERATORS

Database system accelerators trade off generality for performance. Indeed, FPGA-based accelerators are often limited in terms of the types of database operators they can support, the layout (e.g., table depth and width) and distribution of data (including the data types) they can process, and more commonly a combination of these two. These limitations restrict the usability of the accelerator and increase its dependence on the host. The implications are twofold:

- The accelerator is less efficient, as it can only work on a narrow range of workloads.
- An increase in FPGA-host collaboration incurs performance penalties (e.g., by moving data through the PCIe bus) and takes time and bandwidth away from the host by involving it in the execution of some parts of the query.

An important objective of our work is to strike a balance between generality and performance. Indeed, MSM proposed in Chapter 7 supports the most commonly used yet computationally intensive pipeline-breaking database operators, allowing the FPGA to process a wide range of queries. As for the input data, the sort-based MSM is capable of processing tables with an arbitrary number of rows, only limited by the amount of RAM available to the FPGA. The arithmetic logic units (ALUs) used in MSM define the data types supported by the accelerator, and can be updated based on the workload. A crippling limitation of MSM, however, is its lack of support for arbitrarily wide tables. Indeed, the number of supported input columns is a compile-time parameter of MSM and has a large impact on its resource utilization. This is a major hurdle, as support for wide tables is necessary for many database workloads, data warehouses, and scientific computations [RK13]. The main challenge in processing these tables is to deal with a large number of key columns, as value columns can often be processed independently of each other (see Chapter 2).

In this chapter, we propose *key-reduce radix sort* (KeRRaS), an algorithm that enables MSM to process tables with an arbitrary number of key columns; thus, enhancing the use cases and acceleration potential of our platform. Moreover, thanks to the genericity of our solution, it can be applied to many existing **sort-based** database system accelerators. KeRRaS targets the most important primitive used in these accelerators: sorting. Indeed, FPGA-based sorting algorithms often put constraints on the width of the keys. This in turn limits the number of key columns the database system accelerator can process (see Section 8.2). KeRRaS is capable of extending an ordinary (width-limited) FPGA-based sorter to provide it with support for an arbitrary number of key columns. The extension causes minimal resource overhead, preserves modularity, and offers scalability. KeRRaS is described in detail in Section 8.3.

In order to efficiently extend MSM with KeRRaS, we convert it into a **column-oriented** database system accelerator by using interface adapters. Interface adapters are also used to implement the functionality required by KeRRaS. The modified architecture is presented in Section 8.4. We call our extended accelerator MSM with KeRRaS, or MSMK for short. The features of MSMK that are not present in MSM and are enabled by its new architecture are summarized below:

- The column-oriented MSMK can choose which columns to load when processing a table. Therefore, no matter the width of the input table, MSMK can efficiently support queries involving a number of columns that can together fit in its data path.
- If a query involves more columns than MSMK's data path can accommodate (due to its width), KeRRaS is used to process those columns in smaller partitions using an iterative approach.
- After processing a table, the column-oriented MSMK can also choose which columns of the result to store, enabling support for the **projection operator** introduced in Section 2.2.1.

In Section 8.5, we provide an experimental evaluation of MSMK against both MSM and MonetDB. Our benchmarks show that MSMK performs similarly to MSM on narrow tables, and exhibits great scalability with increasing numbers of key columns. MSMK also demonstrates relatively good performance compared to MonetDB on wide tables. We summarize our findings in Section 8.6.

8.2 RELATED WORK

There have been many proposals for designing custom architectures that can accelerate pipeline-breaking database operators. Because of their high architectural complexity and resource demands (e.g., [ABW+16, HANT15, HSM+13, CO14, STM+13]), most research supports only a limited number of these operators at the same time on the FPGA. The authors of [STM+13, MNC09, KT11, MCK17, SKLG16, SCPC15, MFLM20] focus solely on the sort operator, while those in [DZT13, ABW+16, WIA14, YKO+14, MTA09b, WFS+19] target the aggregation, and others in [HANT15, ABW+16, NSJ13, LMM+22] the join operator. Broader research such as [SAS+16] accelerate both the aggregation and join, and [CO14] the sort and join operators. MSM proposed in Chapter 7 supports all three of the sort, aggregation, and join operators. We refer the reader to Chapter 4 for a more comprehensive survey on operator support.

Parameter	Description
n_r	Number of rows in the unsorted table.
n_{kc}	Number of key columns in the unsorted table.
n_{vc}	Number of value columns (payload) in the unsorted table.
n_{bsc}	Number of key columns the base sorter can sort at a time.
n_{it}	Number of iterations.

Table 8.1: Parameters of the KeRRaS algorithm.

In addition, previous studies often impose constraints on the distribution of the data that can be processed by the accelerator. This typically happens in architectures relying on hash-based techniques for operator acceleration. In [WFS+19], hash-based aggregation forwards hash collisions to the host to be later aggregated by the CPU (in software), limiting the distribution of the keys that the FPGA can process alone. Hash-based sorting attempted in [PHL18] bounds the maximum range of the keys and may result in sub-optimal memory usage depending on their statistical distribution [Gil04]. Sort-based approaches like MSM are less sensitive to the distribution and size of the data thanks to a more regular execution flow (see Chapter 7). Finally, sliding-window operator acceleration proposed in [NSJ15, NSJ13, MTA09b] limits the number of rows being considered and, therefore, the scope of the operators. Chapters 4 and 7 present a lot more details regarding past research's support for various data distributions.

The final set of constraints in the state of the art and the main focus of this chapter is the width of the rows or the number of columns that an accelerator can operate on. Indeed, the width of the data path in current architectures often matches that of the widest row they can process. Supporting wider rows requires more of the FPGA resources (bounded by the size of the FPGA) and may result in lower achievable clock frequencies. To name a few examples, the sort operators implemented in [MNC09, SKLG16] and [KT11] can handle 4- and 8-byte keys, respectively. The implementation in [CO14] supports sorts on 8-byte rows, and joins on tables with 4-byte keys and 2-byte values. Finally, the authors of [ABW+16] and [HANT15] assume 8-byte rows and at most 8-byte keys for the aggregation and join operators, respectively; whereas the more flexible architecture presented in [WIA14] can aggregate rows with up to 8×4-byte columns.

Techniques for expanding the number of supported columns typically involve either simply increasing the data path width and/or multi-cycle processing whereby a row is cut into pieces and sent through a narrower data path over multiple clock cycles. Using these techniques in their architectures, the authors of [STM+13] and [MFLM20] support the sorting of rows with up to 40- and 64-byte keys, respectively; and the authors of [WFS+19] support aggregations on 4 key columns, 256 bytes each. These solutions still require entire rows or at least their keys to be loaded on the FPGA chip (e.g., for comparisons) which, given FPGAs' resource limitations, sets a hard limit on the width of the supported rows.

To the best of our knowledge, this is the first work on FPGA-based database system acceleration, proposing an approach for processing arbitrarily wide tables in an iterative way and completely independently of the width of the accelerator's data path.

8.3 KEY-REDUCE RADIX SORT (KERRAS)

In this section, we present KeRRaS, a novel abstract algorithm capable of extending existing sort-based query accelerators, such as MSM, to enable them to process an arbitrary

k_1	k_2	k_3	k_4
3	4	7	5
8	6	1	1
5	7	0	2
3	4	7	1
5	7	0	2
8	6	1	1
5	7	3	9

(a) Table U

k_1	k_2	id
3	4	0
8	6	1
5	7	2
3	4	3
5	7	4
8	6	5
5	7	6

$\Rightarrow$

sk_1	sk_2	sid	ω_{1-2}	κ_{1-2}
3	4	3	0	0
3	4	0	0	2
5	7	2	1	1
5	7	6	1	0
5	7	4	1	1
8	6	1	2	2
8	6	5	2	1

(b) Iteration 1: Reduce $\overline{k_1 k_2}$ into κ_{1-2}

κ_{1-2}	k_3	id
0	7	0
2	1	1
1	0	2
0	7	3
1	0	4
2	1	5
1	3	6

$\Rightarrow$

$s\kappa_{1-2}$	sk_3	sid	ω_{1-3}	κ_{1-3}
0	7	0	0	0
0	7	3	0	3
1	0	4	1	1
1	0	2	1	0
1	3	6	2	1
2	1	5	3	3
2	1	1	3	2

(c) Iteration 2: Reduce $\overline{\kappa_{1-2} k_3}$ into κ_{1-3}

κ_{1-3}	k_4	id
0	5	0
3	1	1
1	2	2
0	1	3
1	2	4
3	1	5
2	9	6

$\Rightarrow$

$s\kappa_{1-3}$	sk_4	sid	ω_{1-4}	κ_{1-4}
0	1	3	0	1
0	5	0	1	4
1	2	4	2	2
1	2	2	2	0
2	9	6	3	2
3	1	1	4	4
3	1	5	4	3

(d) Iteration 3: Reduce $\overline{\kappa_{1-3} k_4}$ into κ_{1-4}

Figure 8.1: Example run of the KeRRaS algorithm with $n_{kc} = 4$ and $n_{bsc} = 2$.

number of **key columns**. The algorithm has a low resource overhead, respects modularity by requiring little to no modifications to the original platform, and is compatible with most existing sort-based database system accelerators. KeRRaS also provides mechanisms to help processing **arbitrarily large payloads**. However, since the way payloads are processed is often specific to a particular accelerator, we shall present concrete payload processing techniques based on these mechanisms on MSM in Section 8.4.3.

KeRRaS is an abstract variant of the forward radix sort algorithm [AN94]. Much like radix sort, it processes wide keys by sorting groups of key columns in an iterative manner. At each iteration, up to n_{bsc} key columns of the unsorted input table are sorted and replaced by a single column, which we shall call the reduced key κ. The reduced key is then used in the subsequent iteration to represent the already-sorted key columns. Following this schema from the most significant to the least significant key columns produces the sorted result. The sorting at each iteration can be performed by any sorting algorithm (e.g., sort-merge, quick-sort, radix sort, ...), which we will refer to as the *base sorter*. n_{bsc} is defined by the base sorter and must be larger than 1 for reduction to take place. KeRRaS works best with column-store databases. Its parameters are summarized in Table 8.1.

Figure 8.1 illustrates KeRRaS through an example. The unsorted table U consists of $n_{kc} = 4$ key columns k_1–k_4 and is shown in Figure 8.1(a). We use the notation $c[i] = v$ to indicate that the row at index i of column c has value v. In the example, we assume that the base

Overhead Column	Definition
ω_{1-j}	$\omega_{1-j}[i] = \begin{cases} 0 & \text{if } i = 0 \\ \omega_{1-j}[i-1] & \text{if } 0 < i < n_r \text{ and } \overline{sk_1...sk_j}[i-1] = \overline{sk_1...sk_j}[i] \\ \omega_{1-j}[i-1]+1 & \text{if } 0 < i < n_r \text{ and } \overline{sk_1...sk_j}[i-1] \neq \overline{sk_1...sk_j}[i] \end{cases}$
κ_{1-j}	$\kappa_{1-j}[sid[i]] = \omega_{1-j}[i], \forall i \in [0, n_r)$ where sid is from the same iteration as κ_{1-j}

Table 8.2: Formal definitions of ω and κ produced by the KeRRaS algorithm. They can both be generated using single-pass algorithms. While ω does not need to be stored in memory, κ must be stored in memory and preserved across KeRRaS iterations.

sorter can sort up to $n_{bsc} = 2$ columns at a time. Figure 8.1(b) shows the first iteration of the algorithm, where the n_{bsc} most significant key columns (k_1 and k_2) along with an incrementing id column attached as a payload are sorted on composite key $\overline{k_1k_2}$. The results of the sort are columns sk_1, sk_2, and payload column sid. Ordering ω_{1-2} starts at $\omega_{1-2}[0] = 0$ and increments for each change in $\overline{sk_1sk_2}$ compared to the previous row. **The ordering is a compressed version of** $\overline{sk_1sk_2}$. Finally, the reduced key κ_{1-2} is obtained by storing the values in ω_{1-2} at indices defined by sid. For instance, the first row of the table with values $sid = 3$ and $\omega_{1-2} = 0$ indicates that the third index (starting from 0) of the reduced key κ_{1-2} should contain the value 0. **The reduced key is a compressed version of** $\overline{k_1k_2}$. ω and κ are formally defined in Table 8.2. In the next iteration of the algorithm, shown in Figure 8.1(c), κ_{1-2} is used to maintain the order imposed by the first iteration. Given that $n_{bsc} = 2$, only a single other column k_3 can now be sorted alongside κ_{1-2}. In this iteration, we simply use $\overline{s\kappa_{1-2}sk_3}$ instead of $\overline{sk_1sk_2sk_3}$ when producing ω_{1-3}. Eventually, by the third iteration shown in Figure 8.1(d), columns sk_1–sk_4 highlighted in blue font represent the fully sorted result.

Note that sid and κ need not be computed in the last iteration of the algorithm, but can prove beneficial for further processing of table U. sid is a bijective mapping of the sorted row indices to the unsorted row indices [Wik22a]. It can be used for ordering potential payload columns. For instance, $sid[0] = 3$ requires the payload at row 3 of table U to move to row 0 of the sorted table. Using this payload processing mechanism, KeRRaS can achieve **stable sorting** if the base sorter itself is stable. κ, which is a compressed version of the keys in the unsorted table, can replace those keys in many situations. For instance, if hashing the keys is necessary to carry out an operation (e.g., hash-based aggregation), κ presents an order-preserving minimal perfect hash function [Jen09].

Note also that KeRRaS accepts all the data types supported by its base sorter. Moreover, large data types can be spread across multiple columns if they are partially reducible, such as strings and unsigned integers that can have their matching bits compared independently and the results combined later.

Lastly, let us discuss an important limitation of our algorithm. During an iteration, effective reduction can only happen if the width of the reduced key column is smaller than the aggregate widths of the key columns used to produce it. More concretely, if the key columns involved in an iteration together contain as many distinct values as can be encoded within their width, no reduction can take place. This limitation can be overcome using partitioning techniques beyond the scope of our work [Rei15].

8.3.1 Time Complexity

In this section, we study the time complexity of KeRRaS. Let us start our analysis by considering the overhead columns. id is generated on-the-fly as the base sorter reads its input table. Both ω and κ are obtained by traversing the sorted columns of their corresponding iteration and performing comparisons among the neighboring rows. The overhead columns are therefore obtained with time complexity in $\mathcal{O}(n_r)$. This equates the time complexity of every iteration of KeRRaS to that of its base sorter. Except in the first iteration, where n_{bsc} key columns are sorted, all other iterations sort $n_{bsc} - 1$ additional key columns as they need to include a reduced key into their computation. The total number of iterations is therefore given by

$$n_{it}(n_{kc}) = \left\lceil \frac{n_{kc} - 1}{n_{bsc} - 1} \right\rceil \tag{8.1}$$

Provided $t_{bs}(n_r)$, the time complexity of the base sorter, KeRRaS is in $\mathcal{O}(n_{it}(n_{kc}) \times t_{bs}(n_r))$.

8.3.2 Space Complexity (Memory Utilization)

The memory footprint of KeRRaS during an iteration equals that of its base sorter sorting n_{bsc} key columns with payload id, plus the storage of κ. Note that ω does not need to be stored as it is a means for producing κ and can be both generated and discarded on-the-fly. Moreover, every new κ can replace its predecessor from the previous iteration (if one exists) in memory. Only κ needs preserving across iterations and, except for the sorted key columns sk_x, all other columns can be discarded. The space complexity of the algorithm is therefore the same as that of its base sorter on the entire table plus a constant number of additional columns, namely sid and κ.

8.3.3 Discussion and Optimizations

KeRRaS presents a convenient way to add support for arbitrarily wide keys to an existing sort-based query accelerator, with the latter acting as the base sorter for the algorithm. The only requirement for the base sorter is that it must be able to sort at least 2 key columns ($n_{bsc} > 1$) with a single payload column at each iteration. We also recommend that the base sorter uses a columnar access and storage model, as it will make it easier to rearrange the columns from iteration to iteration.

After being extended with KeRRaS, the database system accelerator is able to sort any number of key columns within a proportionately large number of iterations. Sort-based operators (e.g., aggregation and join) can also be extended to support large payloads. A few potential approaches are discussed in Section 8.4.3.

KeRRaS is specifically designed to be suitable for **database system** acceleration on **FPGAs**, making it distinct from typical CPU- and FPGA-based sorting algorithms in several ways. First, KeRRaS decouples the width of the rows from that of the accelerator's data path. The latter is defined by the base sorter. Moreover, the KeRRaS-specific overhead columns id, ω, and κ can be generated in a streaming fashion using a simple pipeline structure, which is optimal for implementation on FPGAs.

Second, if used in conjunction with a merge-based sorting algorithm (as its base sorter), KeRRaS can produce the reduced key during the final merge pass of each iteration, requiring no further accesses to the sorted columns stored in the memory. This is an important feature as many of the existing FPGA-based sorters, including MSM, use a variant of the sort-merge algorithm [STM+13, CO14, UIO15, MFL+21].

Additionally, the recommended column-oriented approach requires only the columns targeted by an iteration to be available to the FPGA. The limited amount of RAM in typical FPGA setups (e.g., [Cor20, Cor19a, Bit22]) can therefore be used more efficiently. The algorithm also ensures that each column of the input table is exclusive to a single iteration, with the corresponding sorted column of the final result materialized at the end of that iteration. Therefore, the host-FPGA link only needs to transfer each column once to the FPGA and each sorted column once from the FPGA back to the host.

Finally, unlike typical CPU-based radix sort algorithms that move entire rows every time a set of columns is sorted [CLRS09], the key-reduce approach used in KeRRaS requires only the columns targeted by an iteration to move in the memory. This results in a more predictable and efficient memory bandwidth utilization. Other variants of the radix sort algorithm rearrange pointers to the rows every time a set of columns is sorted. This entails random accesses to the columns involved in the following iterations. KeRRaS, however, pays the cost of rearrangement (via random accesses) once per iteration when producing the single-column reduced key. Our algorithm can therefore run more efficiently on FPGAs, which do not have large caches like CPUs do and therefore exhibit lower random access performance in comparison to CPUs.

KeRRaS also offers a few workload-specific optimizations:

- ***Parallel optimization***: In case the algorithm is run with the single purpose of producing the final reduced key, it can be parallelized and executed by multiple executors (e.g., CPUs, FPGAs). Each executor reduces an exclusive subset of the key columns (i.e., run an independent iteration), and a series of hierarchical passes perform further reductions until the final result is obtained.

- ***Shortcut optimization***: In case the sorted order of the rows can be established solely based on the most significant key columns (e.g., if $\overline{k_1 k_2}$ in our running example was unique for each row), KeRRaS can terminate early and process the remaining key (k_3 and k_4) and payload columns using the last obtained sid as explained earlier in our example.

8.4 ARCHITECTURE

We start this section with a summary of the MSM architecture, highlighting its most relevant components for a KeRRaS implementation. We then present MSMK, an implementation of KeRRaS with MSM as its base sorter. Afterwards, two approaches for processing arbitrarily large payloads on the new platform are proposed. Finally, we discuss some of MSMK's limitations.

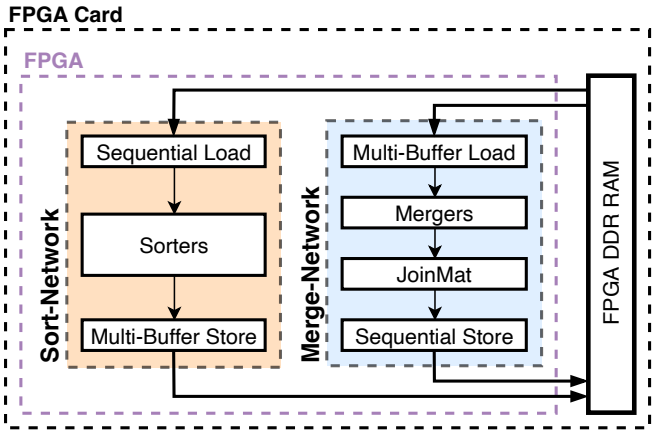


Figure 8.2: Overview of the MSM architecture proposed in Chapter 7.

8.4.1 MSM

The architecture of MSM, which is based on the sort-merge algorithm, is shown in Figure 8.2. The *Sort-Network* is responsible for the sort phase of the algorithm. The *Sequential Load* module reads a stream of rows from the FPGA RAM. The *Sorters* transform the stream of rows into small fixed-size sorted runs. The runs are stored back in the memory via the *Multi-Buffer Store* module. The *Merge-Network* is responsible for the merge phase of the sort-merge algorithm. The *Multi-Buffer Load* provides the *Mergers* access to the sorted runs in the memory. The *Mergers* merge the sorted runs. The result is streamed through the *JoinMat* for potential join processing. The *Sequential Store* module stores the final result back in the FPGA RAM.

Both networks assume a row-store database and only make **full burst accesses** to the memory. The width of the rows and their division into columns are architectural parameters of MSM and define the width of its data path. Each column can be configured as a key or a value column (payload) at runtime. The width of the rows (i.e., the data path width) is defined such that a full burst memory access contains an integer number of rows; therefore, we call it a "row burst access". The host CPU connected through PCIe to the FPGA schedules, launches, and controls operator execution by programming the networks via memory mapped control and status registers (CSR) [HH21].

MSM is particularly suitable for a KeRRaS implementation. The high-performance *Sort-* and *Merge-Networks* constitute the perfect base sorter. Moreover, their additional capabilities can be leveraged to support aggregations and joins on arbitrarily large tables, with minimum effort. Finally, the **row** burst access patterns of the networks can be translated into multiple **column** burst access patterns, efficiently and with no waste of memory bandwidth. This is optimal for KeRRaS that works best with a column-store model.

8.4.2 MSMK: Extending MSM with KeRRaS

KeRRaS was designed with compatibility and ease of integration in mind. Indeed, extending MSM with KeRRaS requires principally changing the host scheduling algorithms that control the *Sort-* and *Merge-Networks*. The algorithms should apply KeRRaS when executing operators on tables wider than supported by MSM alone. As for architectural modifications, the networks should change their interface to support column-store

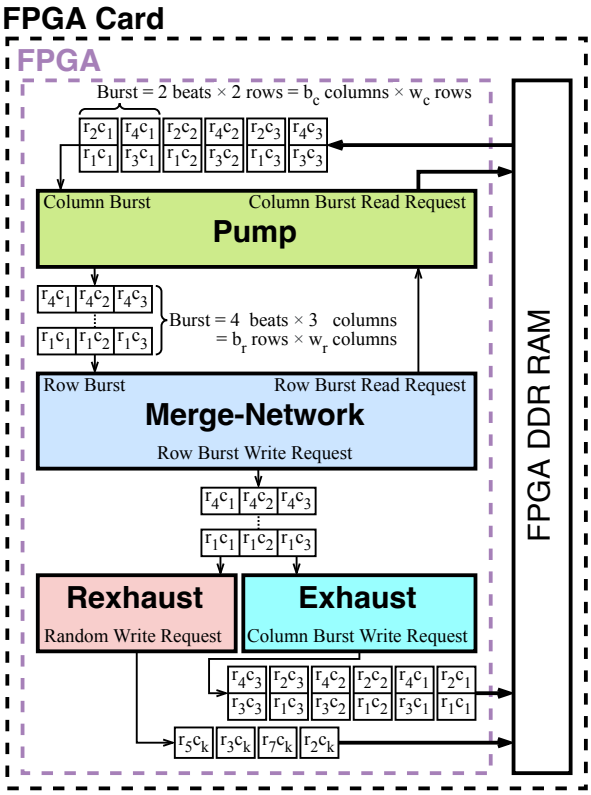


Figure 8.3: Architectural extensions to MSM's *Merge-Network* for implementing the KeRRaS algorithm. The *Pump*, *Exhaust* and *Rexhaust* interface adapters are the only components needed to implement KeRRaS.

databases and be able to generate and store the reduced key. We use interface adapters to implement both functionalities while preserving modularity.

The adapters surrounding the *Merge-Network* are shown in Figure 8.3. The *Sort-Network* requires a subset of the same adapters and is therefore omitted from this figure. The *Merge-Network* and its interface remain unchanged. Upon making a row burst read request, b_r rows (beats) are expected to arrive sometime in the future. Each row has w_r columns. A row burst write request consists of b_r rows that must be written to the memory. The figure contains in-flight rows and columns to exemplify data moving through the interfaces. Table cell $r_i c_j$ is from row i column j. In our example, burst requests read/write 4 rows of 3 columns each; therefore, $b_r = 4$ and $w_r = 3$.

The *Pump* and *Exhaust* adapters convert row burst accesses into column burst accesses. A column burst has b_c beats of w_c rows each. The values of b_c and w_c are chosen to maximize memory access efficiency. b_r is then defined based on their values. In our running example, the memory supports 2-beat burst accesses where each word contains 2 table cells, resulting in $b_c = 2$ and $w_c = 2$. A row burst request is transformed into w_r column burst requests, each bringing back $b_c \times w_c = 4$ cells that are assembled into b_r rows in return. Therefore, b_r must equal $b_c \times w_c$ for maximum bandwidth efficiency. b_r and w_r are architectural parameters of MSM, thus facilitating this type of optimization and parameter tuning. The *Sort-Network* is also extended with a *Pump* and an *Exhaust* adapter.

The *Rexhaust* adapter is responsible for producing and storing the reduced key κ in each iteration of KeRRaS. κ is obtained during the last pass of the *Merge-Network* which produces the final sorted run. While storing this sorted run, the *Rexhaust* is activated generating ω and κ on-the-fly and storing the latter in the memory. This eliminates the need

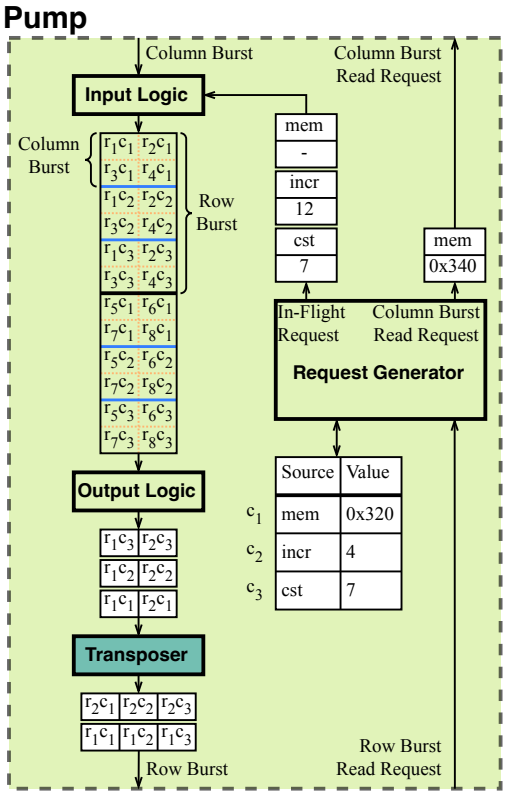


Figure 8.4: Architecture of the *Pump* interface adapter used by MSMK for converting row burst accesses into column burst accesses, as shown in Figure 8.3.

for an additional pass over the result. Note that unlike the final sorted run that is produced sequentially, i.e., in the order of the sorted rows, the reduced key is generated out of order. This is because the location of a particular row in the reduced key is defined by its corresponding value in sid, as explained in Section 8.3. Consequently, the *Rexhaust* is the only module in our architecture that performs random writes. However, as it is activated only once per KeRRaS iteration and to write a single column, it has a minimal negative impact on memory bandwidth efficiency. Note also that the *Sort-Network* does not require a *Rexhaust* adapter, as it only produces the intermediate sorted runs.

Let us now take a closer look at the architecture of the *Pump* shown in Figure 8.4. We shall follow the path of a row burst read request, arriving from the bottom-right of the figure. The *Request Generator* receives the request and transforms it into w_r column burst read requests, with the help of a table containing information about each column. The table, stored in BRAMs, contains a single entry per column indicating its *Source* and an attached *Value*:

- ***mem***: The column resides in the memory. *Value* indicates the address of its first row.
- ***incr***: The column is an arithmetic sequence with increments of $+1$ starting at *Value*. This is useful for generating the id column used in KeRRaS.
- ***cst***: A column of constants, all holding *Value*. This is useful for processing the join operator, described in Section 7.3.5.

The table is filled by the host's scheduling algorithms depending on the needs of the operator. The *Request Generator* computes the effective memory address or increment

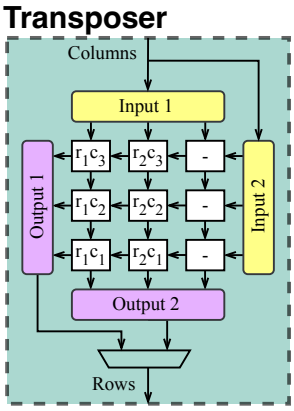


Figure 8.5: Architecture of the *Transposer* used by the *Pump* interface adapter to transpose database columns into rows, as shown in Figure 8.4.

offset of a column based on this table and the position of the requested burst of rows within the stream of data. For columns residing in memory, a column burst read request is made. For every column, an in-flight request is sent to the *Input Logic*.

The *Input Logic* takes various actions based on the nature of an incoming in-flight request:

- ***mem*:** The *Input Logic* waits for the column burst to arrive from the memory, and stores it into the column buffer.
- ***incr* and *cst*:** The *Input Logic* generates and stores the requested rows into the column buffer.

The column buffer, stored in BRAMs, has a width matching that of a memory word and must be capable of holding at least w_r column bursts for a row burst to be producible. In our simple example, it can hold up to 2 row bursts in order to support multiple in-flight row burst requests and benefit from latency hiding. After storing enough data to produce a full row burst, the *Output Logic* is notified. It reads the first beat of each of the column bursts and sends them to the *Transposer*. It then moves onto the second beats, and continues this pattern should a column burst have more beats.

The *Transposer* transforms a set of columns into rows by transposing them. The operation is similar to that of matrix transposition [Wik22i]. The design of the *Transposer* is shown in Figure 8.5. It is based on the universal array architecture [Pan92]. It contains a $w_t \times w_t$ array of registers, each capable of holding a cell of the table. Proper column to row transposition requires the equality $w_t = max(w_r, w_c)$ to hold. The registers forward their data either downwards or to the left, but all in the same direction. The direction alternates every w_t clock cycles. Let us describe a cycle of the universal array to clarify its functionality. During the first w_t clock cycles, $w_r = 3$ columns enter from *Input 1*, moving downwards until occupying their current positions shown in Figure 8.5. The array then switches directions and outputs $w_c = 2$ rows from *Output 1*, while at the same time loading w_r new columns from *Input 2*. The array switches directions again. The rows now exit from *Output 2* while new columns enter from *Input 1*, reaching a full cycle. The benefit of the universal array is its ability to continuously consume, process, and output data. This allows it to achieve the maximum possible throughput constrained by the widths of the input and output data paths.

The *Exhaust* performs the same operations as the *Pump* but in reverse. Therefore, the two modules are quite similar in design and comprise the same important building blocks.

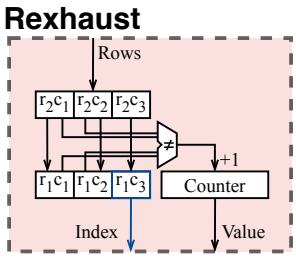


Figure 8.6: Architecture of the *Rexhaust* interface adapter used by MSMK to produce and store the ω and κ columns, as shown in Figure 8.3.

The major operational difference of the *Exhaust* is that it can either store a column in memory or completely discard it if the column is unused. This allows us to efficiently execute the projection operator.

The *Rexhaust* architecture is shown in Figure 8.6. The design compares neighboring rows while keeping track of a counter for producing ω and κ. Indeed, a shift register of depth 2 is used to keep track of adjacent rows. In the figure, the composite keys $\overline{c_1 c_2}$ of the current row r_2 and its predecessor r_1 are being compared. If different, a signal triggers the *Counter* to increment in the next clock cycle. By then, r_2 will be shifted down while r_3 takes its place. Therefore, the value of the *Counter* represents the ordering ω of the oldest row in the shift register. The reduced key is produced by storing the value of the *Counter* at the index specified by the sid column. The latter is represented by c_3 in our example. Finally, note that the last row of ω, i.e., the value of the *Counter* for the last sorted row, indicates the number of unique keys (counting from 0) in the table. This is used for implementing the shortcut optimization introduced in Section 8.3.3.

Finally, let us emphasize some of the important design decisions leading to an efficient KeRRaS implementation. The *Pump*'s latency hiding mechanism and universal array-based *Transposer* architecture allow it to reach very high throughputs. In addition, the data generators of the *Input Logic* reduce the overall memory footprint and memory bandwidth utilization of the platform. Similar optimizations are applied to the *Exhaust*. The *Rexhaust*'s implementation allows it to process a stream of rows by buffering only two rows at a time, which is sufficient for comparing consecutive rows as required for the generation of ω and κ. Indeed, upon the arrival of a new row, the oldest row in the buffer can be discarded. Therefore, the *Rexhaust* has very low resource demands. Moreover, the design is fully pipelined, enabling the processing of a new row every clock cycle.

8.4.3 Payload, Aggregation and Join Processing

MSMK can support payload processing in two ways. The first approach is to use the technique mentioned in Section 8.3 whereby the sid obtained in the last iteration of KeRRaS is used to adjust the payload. Another approach is to use κ as a replacement for all the key columns and sort it with the payload attached. The sort is not part of KeRRaS hence no id processing or κ generation and storage (random memory writes) is necessary, thus resulting in better performance. While the first approach does not require sorting, it moves data with random access patterns. The second approach performs sorting, which is more complex, but with the same sequential access patterns as MSM. Depending on the efficiency of the memory system, the size of the payload, and the number of columns it spans, one or the other approach may be preferred.

We implement the second approach for our experiments in Section 8.5 as using MSM allows us to perform the aggregation and join operators on the payload at the same time. For aggregations, ω of the last iteration can be stored to obtain all the unique keys in their sorted order, matching the result of aggregating the payload. For joins, the keys of both input tables should be reduced together resulting in rows with the same key to have the same reduced key. An incrementing id column can then be tagged along to point to the matching rows of the two tables. Finally, note that as MSM is not a stable sorter, neither of the two payload processing approaches results in stable sorting.

8.4.4 Limitations

KeRRaS is an abstract algorithm hence mostly limited by its underlying base sorter. Let us discuss a few ways in which MSM constrains the use of MSMK. Our MSM implementation fixes the column width at compile time. This in turn limits the number of unique id values and therefore the number of rows MSMK can support at runtime. The implementation used in our experiments in Section 8.5 uses 4-byte wide columns allowing up to 2^{32} rows to be sorted. This is more than enough for our benchmarks. Indeed, our FPGA platform's memory can only hold two 4-byte wide column with this many rows. However, if support for more rows is needed, increasing the column width relaxes this constraint exponentially. Increasing the column width also allows us to support wider (> 4-byte) individual key columns. Although a better solution in this case is to spread those columns, if possible, across multiple smaller ones as described in Section 8.3. Another limitation imposed by the base sorter is the data types supported for each column. Although we use unsigned integers in our experiments, the ALUs used in MSM may be extended to support other data types as well.

8.5 EXPERIMENTS

In this section, we investigate how MSMK performs on various datasets and compare it with the performance of its base sorter, MSM, and the state-of-the-art CPU-based query processor, MonetDB [Mon21]. We use *throughput* in million processed rows per second (MR/s) as a measure of performance in our benchmarks, as we did in Section 7.4.

8.5.1 Experimental Setup

The experimental setup is fairly similar to the one we used in Chapter 7. MSMK is written in Chisel 3.4.3 [BVR+12, Chi21b] which makes it highly parameterizable and thus easy to adapt to various FPGA platforms. The parameters of our main implementation, which we use for scalability analysis and comparison with MonetDB, are $b_r = 64, w_r = 6, b_c = 8$, and $w_c = 8$. They are tuned based on the FPGA memory system characteristics, as explained in Section 8.4.2. The benchmarks run on our PCIe-attached Intel® D5005 FPGA acceleration card with an Intel® Stratix® 10 FPGA and 4 banks (channels) of DDR4 RAM, each with an 8 GB DDR4 RDIMM module [Cor19a]. We instantiate 4 *MSM Cores*, one per memory bank, allowing us to issue up to 4 database operators at a time. Each *MSM Core* includes a single *Sort-Network* and two *Merge-Networks*, as described in Section 7.4.

MSMK's FPGA resource utilization values are shown in Table 8.3. Note that the KeRRaS-related interface adapters (*Pump*, *Exhaust*, and *Rexhaust*) constitute a small percentage

Module	R	Logic Utilization [ALM]		BRAM Utilization [M20K]	
Full System	1	686,809	73.60%	10,474	89.36%
MSM Core	4	151,341	16.22%	2,512	21.43%
Sort	1	102,250	10.96%	1,218	10.39%
Pump	1	1,458	0.16%	14	0.12%
Sort-Network	1	99,374	10.65%	1,183	10.09%
Exhaust	1	1,215	0.13%	21	0.18%
Merge	2	24,545	2.63%	647	5.52%
Pump	1	1,458	0.16%	14	0.12%
Merge-Network	1	20,778	2.23%	607	5.18%
Exhaust	1	1,215	0.13%	21	0.18%
Rexhaust	1	833	0.09%	5	0.04%

Table 8.3: FPGA resource utilization of our implementation of MSMK (with a 192-bit wide data path) and some of its major components. Replication factor (R) is the number of instantiations of a module within its level in the hierarchy. The percentage resource utilizations are calculated over the total amount of resources provided by the FPGA.

of the design's resource utilization. This is consistent with our claim in Section 8.3 that implementing KeRRaS has a low resource overhead. The *Rexhaust*'s BRAM utilization value, is due to the BRAM-based FIFOs we placed around the module for buffering purposes. The design runs at 191 MHz.

We continue comparing the performance of our accelerator with that of MonetDB v11.39.5 [Mon21], a state-of-the-art column-store DBMS [DBRU20]. It runs on an Intel® Xeon® Platinum 8180 CPU @ 2.5 GHz (28 cores, 38.5 MB L3 Cache, 6 memory channels) with 376 GB of DDR4 RAM [Int17]. MonetDB is allowed to use up to 28 threads on a single socket. Increasing the number of threads beyond this value did not result in any further performance improvements for our benchmarks. The MonetDB performance measurements ignore query plan generation, and only include pure operator execution on warm data. By means of multiple clients, we issue 4 operators at a time to match the capabilities of MSMK. The reported performance numbers reflect the aggregate throughput achieved by all 4 *MSM Cores*/MonetDB clients.

8.5.2 Datasets

The parameters of our datasets are listed in Table 8.1, with the exception of n_{bsc} and the addition of r_{kc}. The former is an architectural parameter with value $w_r - 1$, as the base sorter uses one of its columns for processing id. The latter is the domain size of the individual key columns. Indeed, the keys are all chosen randomly from a uniform distribution in the range $[0, r_{kc})$.

As mentioned in Chapter 7, our uniformly distributed datasets are commonly used for studying individual database operators [ABW+16, CR07, MSL+15, MFL+21] and to benchmark sorting algorithms [NS22]. They allow us to analyze the behavior of our system as we gradually increase the number of key or value columns. Moreover, we briefly present some TPC-H based benchmarks demonstrating the capabilities of our platform on standardized workloads [Cou22].

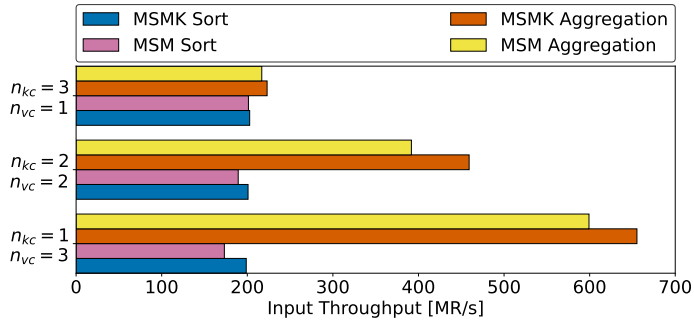


Figure 8.7: The results of the sort and aggregation benchmarks comparing the performance of MSMK and MSM on narrow tables with $n_{kc} + n_{vc} \leq 4$. The benchmark datasets have $n_r = 2^{24}$ and $r_{kc} = 256$.

8.5.3 MSMK vs. MSM

To begin our analysis, we compare the performance of MSMK with its base sorter, MSM, using builds with comparable parameters. Since MSM can only operate on tables with up to 4 columns, the tables in our datasets must satisfy the inequality $n_{kc} + n_{vc} \leq 4$. Some of our most representative results are shown in Figure 8.7. The slightly higher performance of MSMK compared to MSM is mainly due to the *Pumps* and *Exhausts* increasing the memory prefetching and buffering capacity of the accelerator. Our memory bandwidth measurements on benchmarks with similar runtime characteristics show that MSMK uses on average 6% more memory bandwidth compared to MSM.

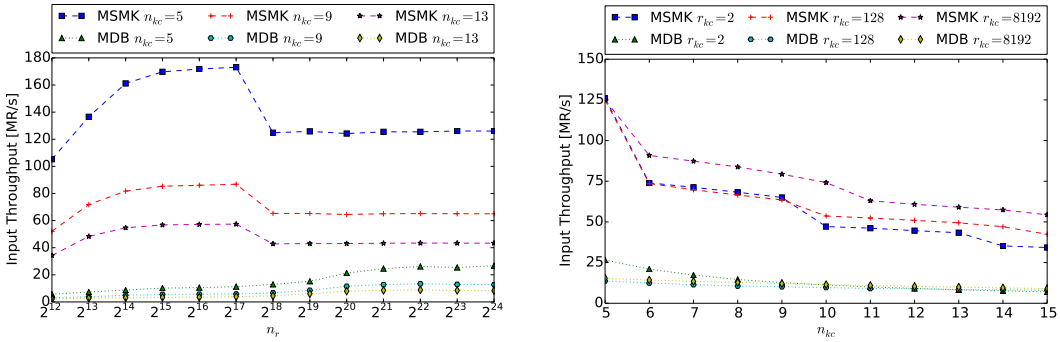
In the case of the sort operator, the number of key and value columns has no significant impact on performance. The slight increase in throughput for larger n_{kc} is due to the increase in the range of keys positively impacting the performance of the *Merge-Networks*.

In the datasets used for benchmarking the aggregation operator, decreasing the number of key columns results in more rows with duplicate keys. The early aggregation optimization implemented by MSM, and thus inherited by MSMK, partially aggregates these rows early in the sort process (see Section 7.3). This results in considerably fewer rows that need to be sorted/merged and thus higher throughputs. The effects of this optimization are visible in the results of our aggregation benchmarks shown in Figure 8.7.

The MSM implementation of the join operator is only capable of processing 2 key columns. The MSMK join benchmarks (omitted from the figure) behave similarly to those from MSM as well.

8.5.4 Payload-Less Benchmarks

In Section 8.5.3, we benchmarked MSMK on narrow tables without producing the reduced key. Producing the reduced key results in a decrease of on average 30% in performance. This is due to the random memory accesses made by the *Rexhaust*, even though it is only activated once per KeRRaS iteration to write a single column. It also emphasizes the importance of the shortcut optimization introduced in Section 8.3.3. A future optimization of the implementation could improve random access performance by either using a small cache, or a different memory technology to store the reduced key. In this section, we focus on wide tables that require KeRRaS for processing. We assume KeRRaS is used for sorting and to produce the final reduced key for future operations.



(a) Throughput of the sort operator as a function of n_r, with $r_{kc} = 2$ and $n_{vc} = 0$

(b) Throughput of the sort operator as a function of n_{kc}, with $n_r = 2^{24}$ and $n_{vc} = 0$

Figure 8.8: The results of the payload-less performance and scalability benchmarks comparing the performance of MSMK and MonetDB (MDB), as the number of rows (n_r) or key columns (n_{kc}) varies.

Figure 8.8(a) shows the results of our benchmarks varying n_r for multiple values of n_{kc}. The observed behavior of the individual MSMK curves mirrors that of the base sorter from Section 7.4.3. The relatively low performance for small n_r is due to the significance of the operator's launch overhead against its actual execution time. The sudden decrease in performance at $n_r = 2^{18}$ happens as an additional merge pass of the sort-merge algorithm is required to merge the many sorted runs. While these properties are inherited from the base sorter, the disparity between the curves with different n_{kc} is due to KeRRaS. Indeed, the datasets with $n_{kc} = 5, 9$, and 13 require $1, 2$, and 3 KeRRaS iterations to be sorted, respectively. Therefore, $n_{kc} = 5$ has approximately twice the throughput of $n_{kc} = 9$, and three times that of $n_{kc} = 13$, in line with our time complexity analysis from Section 8.3.1. MonetDB curves exhibit a similar relationship with respect to n_{kc}.

The results of a fine-grained scalability analysis of MSMK are shown in Figure 8.8(b). In these benchmarks, we vary n_{kc} for multiple values of r_{kc}. The step-like behavior of MSMK is because datasets with more key columns require more KeRRaS iterations to be processed. Indeed, given the parameters of our system, Equation 8.1 becomes $n_{it} = \lceil (n_{kc} - 1)/4 \rceil$, which is reflected through our benchmarks. Moreover, our memory consumption analysis shows that per iteration, MSMK requires the same amount of memory as MSM (including the id column), plus additional memory to store κ. This is also in accord with our space complexity analysis from Section 8.3.2. The three MSMK curves start with the same throughput but deviate from each other due to the shortcut optimization. For instance, at $n_{kc} = 9$, curve $r_{kc} = 8192$ requires a **single** KeRRaS iteration as the first 5 key columns are unique enough to define the sorted order. The remaining key columns are then treated as payload and processed to be aligned with the first 5 sorted key columns. Curves $r_{kc} = 2$ and $r_{kc} = 128$, however, require **two** KeRRaS iterations to define their order, thus resulting in lower throughput. To conclude, note that MSMK achieves a nearly consistent level of performance improvement (approximately $5\times$) compared to MonetDB across all n_{kc} values, indicating that they scale similarly well.

Finally, Figure 8.9 presents the results of our benchmarks for the standalone sort operators extracted from 4 TPC-H queries. The TPC-H dataset has a scale factor of 8 and is stored uncompressed. As the benchmarks show, MSMK performs consistently better than MonetDB. The varying degrees of speedup are mainly due to the behavior of the base sorter (MSM) on different input sizes. Indeed, queries $12, 4, 20$, and 16 require $3, 2, 1$, and 2 **merge passes** of MSM to sort, respectively. This is because the number of rows (n_r) of the tables used in these queries is widely different; ultimately resulting in variations in the performance of MSM and consequently MSMK (see Chapter 7).

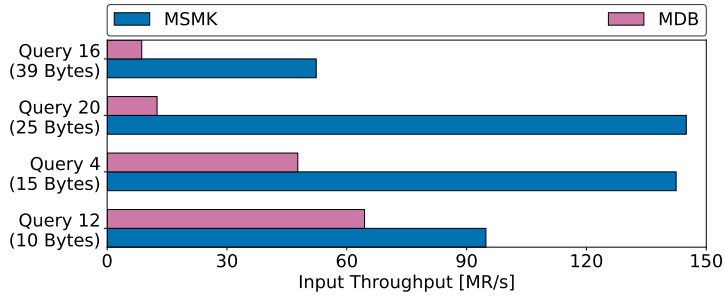
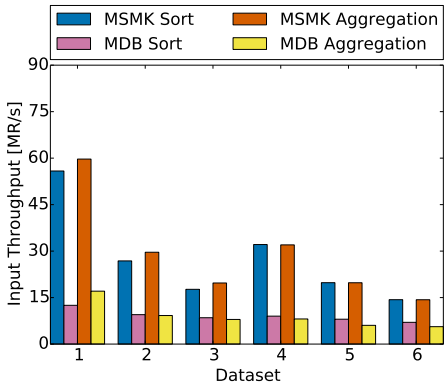
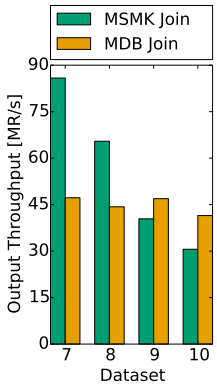


Figure 8.9: The results of the standalone sort operator benchmarks of 4 TPC-H queries with various key lengths (noted below each query) on MSMK and MonetDB (MDB).



(a) Sort and Aggregation Benchmarks



(b) Join Benchmarks

n_r	r_{kc}	n_{kc}	n_{vc}		
			10	30	50
2^{22}	16	5	Dataset 1	Dataset 2	Dataset 3
		10	Dataset 4	Dataset 5	Dataset 6

(c) Sort and Aggregation Dataset Parameters

n_{vc}	n_{kc}	r_{kc}	Join Type	
			2 : 2	1 : 3
1	5	16	Dataset 7	Dataset 8
	10	4	Dataset 9	Dataset 10

(d) Join Dataset Parameters

Figure 8.10: The results of the payload-based sort, aggregation, and join operator benchmarks comparing the performance of MSMK and MonetDB (MDB), and the parameters of the datasets used in these benchmarks.

8.5.5 Payload-Based Benchmarks

In our implementation of KeRRaS, we use the naive sort-based approach described in Section 8.4.3 for processing arbitrarily wide payloads. The approach leverages the existing capabilities of the base sorter (MSM); therefore, requires no extra logic to implement on the FPGA.

The results of our sort and aggregation benchmarks, along with the parameters of the datasets used in these benchmarks are shown in Figures 8.10(a) and 8.10(c). The widths of the keys and payloads are chosen to match the ones in [STM+13], an existing work focused on **large payload** processing on FPGAs. In our benchmarks, increasing the number

Data Path Width			$w_r = 4$	$w_r = 6$	$w_r = 8$
Resource Utilization	Logic		44%	52%	62%
	BRAM		43%	53%	65%
performance	$n_{kc} = 5$	Throughput [MR/s]	74	119	120
		KeRRaS Iterations	2	1	1
	$n_{kc} = 10$	Throughput [MR/s]	32	45	59
		KeRRaS Iterations	5	3	2
	$n_{kc} = 15$	Throughput [MR/s]	23	32	38
		KeRRaS Iterations	7	4	3
	$n_{kc} = 20$	Throughput [MR/s]	16	25	28
		KeRRaS Iterations	10	5	4

Table 8.4: Experiments illustrating the impact of the architectural parameter w_r on the performance and resource utilization of MSMK.

of value columns decreases the throughput improvement factor of MSMK compared to MonetDB. This is expected because our payload processing routine is also based on sorting, which makes payload operations costly. Nonetheless, MSMK achieves on average 3 times higher throughput compared to MonetDB.

The results of our join benchmarks are shown in Figure 8.10(b). Since joins involve two tables and the output may be larger than the input, throughput is measured in the number of result rows per second. The datasets used in our join benchmarks are described in Figure 8.10(d). The input tables are chosen such that for an $M : N$ join, the size of the cross product of the same-key rows from both tables is $M \times N$, and that the tables contain 2^{22} rows in total. Our benchmarks show that for the join operator, MonetDB is much more competitive, with performance surpassing that of MSMK as n_{kc} increases. Indeed, MonetDB uses a hash-based join algorithm to execute our join workloads. MSMK joins by first sorting and then pairing the rows, leading to worse scalability but sorted results. This can prove beneficial in processing chains of operators. For instance, aggregation on a prefix of the join key is faster if the join result is already sorted.

8.5.6 Flexibility

One of the major benefits of our flexible architecture, is that it can be optimized for different workloads. An architectural parameter that is particularly relevant for KeRRaS, is the data path width of its base sorter (w_r). The results of a study examining the impact of this parameter on the performance and resource utilization of MSMK are shown in Table 8.4. In these experiments, we benchmark three MSMK implementations with $w_r = 4$, 6, and 8 on workloads with varying numbers of key columns. Throughput values in MR/s are correlated to the number of KeRRaS iterations required to process the input. According to Equation 8.1, the number of KeRRaS iterations is itself a function of n_{bsc} and thus w_r. Therefore, w_r can be modified to generate an MSMK implementation suitable for processing a particular workload. Minimizing w_r constrained by the target performance reduces the accelerator's resource utilization, allowing the FPGA to accommodate new logic that is useful for the target workload (e.g., filtering, compression, etc.). More importantly, thanks to KeRRaS enabling MSMK to process arbitrarily wide tables (even with the smallest w_r), the database system user can configure the accelerator to have optimal performance for the common case, while still being able to use it for larger (and of course also smaller) tables that appear less frequently in their workloads.

8.6 SUMMARY

Optimal database system acceleration requires the data to be as close and exclusive to the FPGA as possible. This can be achieved by increasing the capabilities of the accelerator. These capabilities include support for multiple operators and various input tables. Although we addressed both areas fairly extensively in Chapter 7, one major limitation remained: the inability of MSM to support arbitrarily wide tables. This became the focus of the current chapter, where we propose KeRRaS as a solution.

KeRRaS is an abstract sorting algorithm that can extend existing sort-based query accelerators to enable them to support arbitrarily wide keys efficiently and with minimal overhead. Moreover, it produces metadata (e.g., reduced key) that can represent the initial data more efficiently or be used for payload processing. An implementation along with some benchmarks were provided to verify these claims. Indeed, MSMK is a minimal (adapter-based) extension of an existing query accelerator (MSM), that can sort an arbitrary number of key columns in a scalable manner. Moreover, a simple payload-processing routine allows MSMK to support the aggregation and join operators on arbitrarily wide tables. Our benchmarks show that MSMK performs equally well as MSM on narrow tables, and demonstrates great scalability with increasing numbers of key columns. Finally, MSMK is a column-oriented accelerator that also supports the projection operator. We showcase more of its capabilities in Chapter 10, where we use our acceleration platform to process TPC-H queries.

9

A STUDY OF EARLY AGGREGATION IN DATABASE QUERY PROCESSING ON FPGAS

9.1 Early Aggregation

9.2 Background & Related Work

9.3 Simulations

9.4 Cache System Architecture

9.5 Experiments

9.6 Summary

The preceding chapters were dedicated to the research and implementation of a novel FPGA-based database system accelerator. After our latest developments in Chapter 8, our accelerator named MSMK is capable of processing the sort, aggregation, and join operators, on arbitrarily large and wide tables. It has a flexible architecture, and can process multiple operators at the same time. With these features, MSMK meets all of our objectives outlined in Section 5.1. We dedicate this chapter to performance improvements. In particular, we focus on the aggregation operator.

Early aggregation is a popular method for improving the performance of the aggregation operator [YL94, Lar02]. We implemented early aggregation in our treap sorters back in Chapter 7. In this chapter, we present a comprehensive survey of various early aggregation algorithms and compare their characteristics. The comparative study leads us to set-associative caches with a low inter-reference recency set replacement policy. They show both great performance and modest implementation complexity compared to some of the most prominent early aggregation algorithms. We also present a novel application-specific architecture for implementing set-associative caches. Benchmarks of our new implementation show speedups of up to $3\times$ for end-to-end aggregation compared MSMK.

Note that parts of the material presented in this chapter have been previously published in FPGA'23 [MMF+23].

9.1 EARLY AGGREGATION

Aggregation is a commonly used, yet computationally expensive database operator [DBRU20]. Most implementations resort to hashing or sorting for intermediates. *Hash-based aggregation* produces a hash table where rows with the same key are inserted into the same bucket, allowing them to be accumulated into their group immediately. *Sort-based aggregation* sorts the rows according to their key if needed, ensuring that those belonging to the same group become neighbors so they can be easily accumulated [GUW09]. Both algorithms are described in detail in Section 4.3.

The runtime of both algorithms increases with the size of the input. Indeed, for hash-based aggregation, a larger input requires more random accesses to the hash table in memory. Sort-based aggregation, often implemented using the sort-merge algorithm (see Section 9.2), needs more merge passes and a larger memory to sort and eventually aggregate larger tables. In these scenarios and to improve the performance of aggregation in general, early aggregation comes to the rescue.

Early aggregation consists of **partially** aggregating the input table using a simple and often streaming algorithm, before complete aggregation with hashing or sorting takes place [YL94]. An example is shown in Figure 9.1. The algorithm produces table S by accumulating the rows belonging to the same group that appear consecutively in the input table. Table S, although smaller than table R, contains duplicate keys and requires complete aggregation to produce table T. However, thanks to the smaller input size, **the cost of complete aggregation is reduced**. The impact of early aggregation is often strong enough that many aggregation algorithms integrate it either directly or indirectly into their execution flow. They employ sophisticated sort-based [YL94, DG20, HNM02] or cache-based [CR07, HNM02, MSL+15] early aggregation algorithms for higher performance. It shall be noted that early aggregation is only applicable for algebraic aggregation functions such as SUM() or MAX(), but not holistic aggregation functions such as MEDIAN() [GCB+97].

K	V
5	2
8	1
3	5
8	2
8	1
2	1
5	4
5	3

(a) Table R

$\xrightarrow[\text{Aggregation}]{\text{Early}}$

K	SUM(V)
5	2
8	1
3	5
8	3
2	1
5	7

(b) Table S

$\xrightarrow[\text{Aggregation}]{\text{Complete}}$

K	SUM(V)
5	9
3	5
8	4
2	1

(c) Table T

Figure 9.1: Example run of the aggregation operator $T = \text{Aggr}_{K,SUM(V)}(R)$, using early aggregation as an intermediate step. Table R is the input of the operator. Table S is the result of early aggregation, and contains duplicate keys that need further processing. Table T is the final result of the aggregation operator.

The goal of this chapter is to explore early aggregation on FPGA-based database system accelerators, fitting the application for several reasons. First, reducing the size of the input table has a positive impact on the ability of the FPGA to accelerate the aggregation operator. Second, early aggregation may reduce the memory footprint of the aggregation operator, enabling the processing of larger input tables on the (often) relatively small amount of DRAM onboard PCIe-attached FPGA cards. Finally, most cache-based early aggregation algorithms can be efficiently implemented in a pipelined manner on FPGAs. We draw upon the extensive body of research on caching in hardware (mostly CPUs) to explore the design space for cache-based early aggregation [HP17].

This chapter is organized as follows. In Section 9.2, we survey existing early aggregation algorithms. We then perform accurate simulations to compare the performance of the ones that are adequate for an FPGA-based implementation. The simulations presented in Section 9.3 lead us to a family of cache-based early aggregation algorithms fit for our application. We then introduce a quantitative methodology for selecting a member of this family for a given accelerator architecture. In Section 9.4, we propose a novel cache architecture enabling an efficient implementation of early aggregation, based on our previous findings. In Section 9.5, we integrate our new cache architecture into MSMK. The resulting platform, called CbMSMK, is benchmarked against both MSMK and MonetDB. Finally, a summary of the chapter is presented in Section 9.6.

9.2 BACKGROUND & RELATED WORK

Efficient early aggregation algorithms can be classified into two major categories: Sort-based and cache-based. Sort-based early aggregation may be used in combination with sort-based aggregation. Cache-based early aggregation can be used with both sort- and hash-based aggregation. In this section, we shall present the state of the art in both categories. Every algorithm is followed by a discussion on the efficiency of a potential FPGA-based implementation. The discussions are based on commonly followed accelerator specifications:

- ***Spec 1*:** Early aggregation may only use on-chip FPGA resources, i.e., logic elements and BRAMs. DRAM or disk utilization often interfere with complete aggregation causing slowdowns down the processing pipeline.
- ***Spec 2*:** If the input rows are narrower or as wide as the query accelerator's data path, an early aggregation algorithm must process the input at near line-rate speeds, i.e., one row per clock cycle.

9.2.1 Sort-Based Early Aggregation

In **sort-based aggregation**, the sort-merge algorithm is commonly used to sort the input. The sorted table is then aggregated at the end of the final merge pass. MSMK, which is based on MSM, relies on this algorithm for executing the aggregation operator.

Sort-based early aggregation intervenes during the **sort phase** of the sort-merge algorithm. Two prominent methods are fixed-length and variable-length sorting. In *fixed-length sorting* [YL94, MFL+21], while loading the rows into the memory, we check if a new row belongs to a group encountered so far. If so, it is immediately accumulated into that group. Once the memory is full, the groups are sorted and output as a fixed-length run. In *variable-length sorting* [YL94, DG20, HNM02], the memory maintains two sorted runs: current and next. Initially, new rows are all inserted into the current run. Once the memory is full we output the (key-wise) smallest group of the current run. Upon arrival of a new row, (1) if it belongs to a group in either of the two runs, it is accumulated into that group. Otherwise, (2) if the row is larger than the previously output group, it is inserted into the current run. Otherwise, (3) the row is inserted into the next run. In cases (2) and (3), the smallest group of the current run is then output at the end. When the current run becomes empty, it switches places with the next run.

MSMK performs fixed-length sorting **along with early aggregation** using the treap data structure. Each treap sorter needs multiple clock cycles to process a single key, thus 32 sorters are used in parallel to achieve line-rate processing. Moreover, the merge phase is optimized to perform on-the-fly aggregation while merging the sorted runs. Variable-length sorting, although **without** early aggregation, is attempted in [STM+13] where the authors also use multiple sorters in parallel to maintain their target throughput. In Section 9.3, we simulate and compare both fixed- and variable-length sort-based early aggregation using 32 parallel sorters.

9.2.2 Cache-Based Early Aggregation

Hash-based aggregation populates a hash table in the main memory (DRAM for both FPGA and CPU). Each row is stored as a group in the bucket corresponding to its hash value [GUW09]. The *hash of a row/group* is the result of applying a hash function **to its key**. The hash table needs to deal with two types of collisions: (1) *Key collisions* occur as rows belonging to the same group, hence with the same key, are mapped to the same bucket. They are resolved by accumulating a colliding row into its group (if present) in the hash table. (2) *Hash collisions* occur if rows with different keys have the same hash value, therefore, also map to the same bucket. They are resolved using techniques such as separate chaining or open addressing [Ski08]. Once every row is processed, all of the groups in the hash table form the result of the aggregation operator.

Cache-based early aggregation is commonly implemented as a scaled down version of hash-based aggregation, where the hash table is stored in a cache. The cache must be designed for fast access, in order to avoid adding too much overhead to the complete aggregation process. For instance, it might reside in the cache of a CPU or on the BRAMs of an FPGA [CR07]. The often small and static structure of the cache may cause some rows to overflow, and thus not meet their matching group. These rows, along with the content of the cache after every row has been processed are sent to a hash- or sort-based aggregation algorithm for complete aggregation.

In the context of early aggregation, the architecture of a cache is based on three important design decisions: (1) *cache structure* is the layout of the cache, (2) *placement strategy* is the mapping of rows to buckets, and (3) *replacement policy* determines a group to

evict in case of a new hash collision in an already-full bucket. Most database-related caching mechanisms use hashing as a placement strategy. MurmurHash [App16] is a non-cryptographic, robust, hash function commonly used in recent database literature (e.g., [MSL+15, KGA17]). Furthermore, it can be pipelined, which results in an efficient FPGA-based implementation [KA16]. Therefore, we shall only focus on studying different cache structures and replacement policies. In what follows, we present the decisions made in past research with regards to both categories. The proposed mechanisms are used in the execution of many database operators (e.g., aggregation and hash join) and at different granularities (e.g., rows or pages). However, note that the cache structures that despite being pipelined/parallelized may require multiple clock cycles for lookup (e.g., they use separate chaining [HSM+13]), replacement, or reorganization [KM10] violate *Spec 2*, hence are excluded from our study. We start by presenting various cache structures.

Cache Structures

An **n-way set-associative cache** has buckets with a maximum capacity of n entries. n is the *degree of associativity* of the cache. Each entry can hold a single group. We shall use the terms "group" and "entry" interchangeably. The buckets are referred to as *sets*. A row is mapped to a single set, defined by its *set index* (i.e., hash of its key). Upon inserting a row into a set, it can be placed in any of its n entries. If all of the entries are full, and the new row does not belong to any of the existing groups, the replacement policy is called. A 1-way set-associative cache is typically referred to as a *direct-mapped cache*. The *size or capacity of a cache* (in general) is the maximum number of groups it can hold at any given point in time. Set-associative caches are commonly used in CPUs [HP17].

An **m-lookup n-way set-associative cache** is an n-way set-associative cache where a row may be placed in m different sets. An implementation would typically hash a row with m different hash functions in order to find its corresponding sets [GS21]. Considering *Spec 2* and given that most modern FPGAs allow at most two simultaneous read accesses to arbitrary lines of BRAMs (see Section 3.1.2) used for storage by the cache (*Spec 1*), we shall limit the scope of our study to 2-lookup n-way set-associative caches only.

A **k-level cache** is a sequence of k caches of any of the types described above, laid out and connected according to their order in the sequence. An element evicted from a cache is inserted into its successor in the sequence. Each cache uses a different hash function [KM10, UIO15]. Note that multilevel caches borrow their name from multilevel hash tables [BK90] and, despite some similarities, are not to be confused with hierarchical caches used in CPUs (e.g., $L1$, $L2$, ...).

Replacement Policies

Let us now discuss some prominent replacement policies. Each policy takes as input a set of cache entries/groups that may accommodate a new row. For instance, in a set-associative cache, the replacement candidates are the entries in the set corresponding to the inserting row. The output of the policy is at most a single entry that may be replaced.

No-replacement is a policy whereby none of the groups in the cache are ever replaced [UIO15]. In *random replacement*, one of the groups is chosen at random [HNM03]. *First-in first-out replacement* (FIFO) selects the group that has been **created** the earliest [EH84]. *Least recently used replacement* (LRU) chooses the group that was **last accessed**

the earliest [EH84, HNM03, Lar02], i.e., not referenced for the longest time. A *group is accessed* if it is created or hit. A *cached group is hit* every time an inserting row belongs to that group.

Least frequently used replacement (LFU) selects the group with the lowest reference count [EH84]. The *reference count of a group* is the number of cache hits into that group.

LRU-2 replacement is a commonly used policy in the family of LRU-K policies [OOW93]. The policy selects the group with the largest 2-distance. The *2-distance* of a group is the distance in time between the last 2 accesses to that group. The notion of *time* is commonly defined as the index of the inserting row in the input table. For instance, a hit while inserting the 5th row of the table into the cache, sets the time of hit to 5.

In *low inter-reference recency set replacement* (LIRS) [JZ02, Bla10, Wik22e], we compute the 2-distance and age of all input groups. The *age of a group in the cache* is the current time minus the time of the last access to that group. The *eviction priority of a group* is the largest of the 2-distance and the age of the group. LIRS selects the group with the largest eviction priority.

Optimal replacement selects the group whose **next appearance** in the stream of rows is the farthest off [HNM03]. Given the policy's need to know about future rows in the stream, it is not feasible to implement and is only used as an upper bound on replacement policy performance.

The replacement policies presented in this section may be classified as follows. *Recency-based policies* such as LRU use access times to make their decision. *Frequency-based policies* such as LFU and LRU-2 select a group based on the frequency of hits into that group. Indeed, higher frequencies result in higher reference counts and often smaller 2-distances. *Mixed policies* such as LIRS consider both recency and frequency. The remaining policies presented in this section belong to their own unique category.

9.3 SIMULATIONS

In this section, we perform simulations comparing the quality of different early aggregation algorithms. The goal is to choose an adequate algorithm for early aggregation on FPGAs. The simulations implemented in Scala use the strategy pattern to combine different cache structures and replacement policies [Wik22h].

9.3.1 Datasets

Our simulations employ synthetic datasets commonly used for benchmarking the aggregation operator [CR07, MSL+15, HNM02, YL94, ABW+16, Lar02]. All of the tables in our datasets have 2^{26} (≈ 64 million) rows. The distributions (of the keys) used in our datasets are uniform (like in TPC-D and TPC-H [CG11]), sorted, heavy-hitter, self-similar, moving cluster, and normal. In the *heavy-hitter* distribution, 50% of the rows have key 1, and the remaining rows have their key chosen uniformly from the remaining range of keys. *Self-similar* is an 80-20 Pareto distribution reflecting extreme natural phenomena and human activities [Wik22g]. The Zipf distribution with exponent 1 is a skewed distribution used to model phenomena such as word occurrences in English texts [YL94]. The moving cluster distribution picks its keys uniformly from a sliding window of size 1024; therefore,

the distribution of the generated keys varies depending on their position in the table. Datasets based on real data are described and used in our experiments in Section 9.5.

Although our simulations and experiments were carried out on all of our datasets, we only show the most representative results that help us compare and study early aggregation algorithms and better highlight their differences.

9.3.2 Metrics

Memory factor is a **parameter** of our simulations, representing the ratio of the number of groups in the input to the number of groups that can be buffered by the partial aggregation algorithm (i.e., its capacity). As an example, a table with any number of rows belonging to one of 10 groups, input to an early aggregation algorithm with a capacity of 50 groups, has memory factor $= 10/50 = 0.2$. The benefit of using memory factor is that it abstracts away the buffer size (i.e., cache size or run size) of the early aggregation algorithm, increasing the applicability of our results. With a memory factor ≤ 1, a **perfect** early aggregation algorithm capable of holding all of its input's groups without overflows should be able to completely aggregate the input. Note that our synthetic data generators can produce a table with only an **approximately** accurate specified number of groups. The memory factors in our graphs, where the capacity of the early aggregation algorithm is kept constant, are accurate within 0.3% of their shown values.

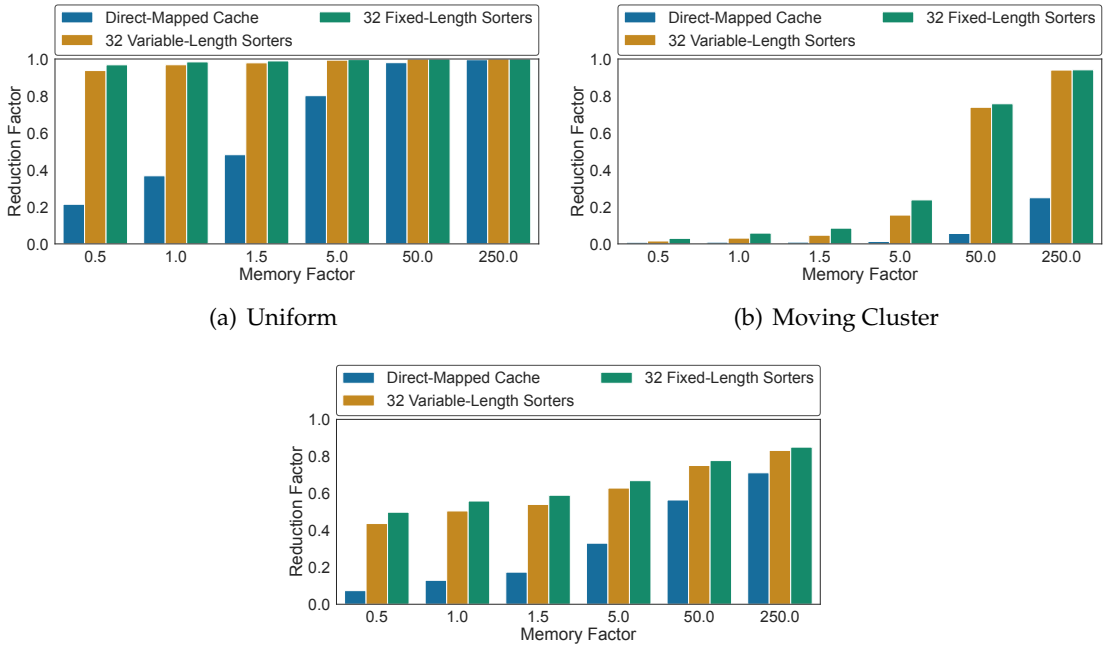
Reduction factor, which is the **result** of our simulations, is a commonly used measure of success for early aggregation. It represents the ratio of the number of rows after early aggregation to the number of rows in the original table. In our simple example in Figure 9.1, reduction factor = $|S|/|R| = 6/8 = 0.75$. Reduction factor can take values between 0 and 1, with **lower values** implying **higher quality** or **higher performance** early aggregation.

9.3.3 Sort-Based Versus Cache-Based Early Aggregation

We start our study by comparing the two main categories of early aggregation algorithms. We compare the two sort-based algorithms with the simplest cache-based algorithm, namely a direct-mapped cache. The results of our simulations are shown in Figure 9.2.

Both sort-based and cache-based algorithms perform poorly on the uniform dataset. Indeed, uniformly distributed data with memory factor > 1 present a bad scenario for any type of buffering or caching mechanism as there is no locality in the data. The sort-based algorithms perform worse than the cache-based algorithm because of their "fill-empty cycles": as soon as a single overflow occurs, the entire buffer of the sorter starts to be drained. This is a fairly drastic action resulting in high (bad) reduction factors. The emptying of a variable-length sorter, however, happens while inserting new rows into the sorter, slightly improving its reduction performance compared to a fixed-length sorter. This observation is valid on all datasets. It should also be noted that although the sort-based algorithms have the same capacity as the direct-mapped cache, they have to split their buffer among their 32 sorters. This is the reason behind the poor performance of the sort-based algorithms on small memory factors.

The moving cluster distribution has high locality, with the distribution of the keys changing depending on their position in the table. In this case, the fill-empty cycles of the



(a) Uniform

(b) Moving Cluster

(c) Self-Similar

Figure 9.2: The results of the simulations comparing sort-based and cache-based early aggregation on various datasets.

sort-based algorithms play to their advantage, allowing them to adapt to the varying distribution. The cache-based algorithm, however, continues to outperform both sort-based algorithms in terms of reduction.

The simulation results on the self-similar dataset present a middle ground between the previous two datasets. All three algorithms achieve good reduction thanks to the skew in the self-similar distribution presenting lots of opportunities for early aggregation.

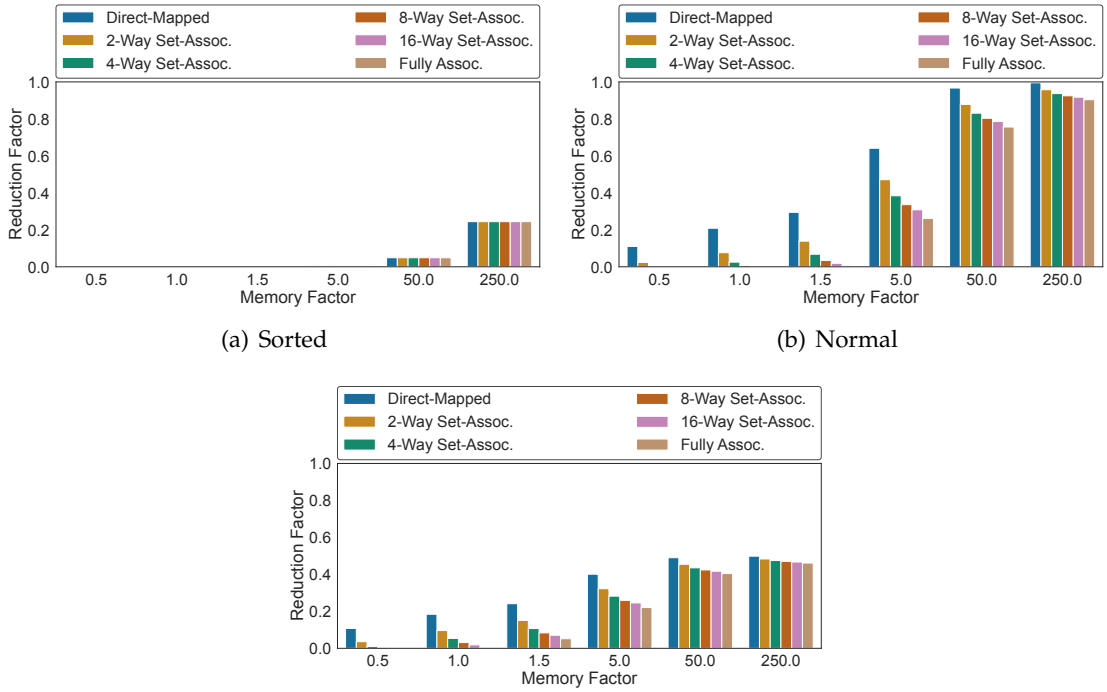
In a nutshell, even the simplest cache-based algorithm achieves lower reduction factors compared to both of the sort-based algorithms. Cache-based early aggregation is the clear winner of this round of our simulations and is therefore the focus of the following sections.

9.3.4 Comparison of Set-Associative Caches

The results of our simulations comparing set-associative caches are shown in Figure 9.3. All cache-based algorithms use the optimal replacement policy, allowing us to focus on the quality of the cache structure alone. The fully set-associative cache, although impractical to implement with large capacities, represents the lower bounds of the reduction factor.

Our set-associative caches perform equally well across all memory factors on the sorted dataset. Indeed, the sorted dataset places all the rows belonging to the same group (perfectly clustered) next to each other, allowing the cache-based algorithms to aggregate them immediately. Similar behavior was observed on the moving cluster dataset.

On both the normal and the heavy-hitter datasets, we see an improvement in the reduction factor as the degree of associativity of the cache increases: an exponential increase



(a) Sorted

(b) Normal

(c) Heavy-Hitter

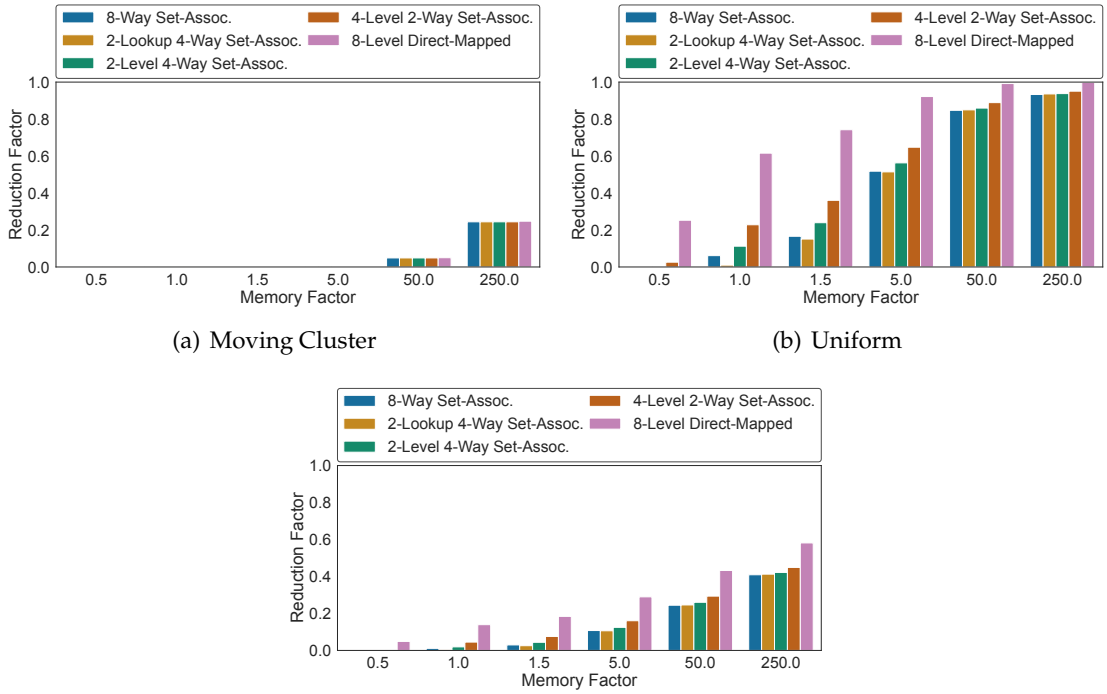
Figure 9.3: The results of the simulations comparing set-associative caches for early aggregation on various datasets. All the caches used in these benchmarks employ the optimal replacement policy.

in the set size results in a logarithmic decrease in the reduction factor. This behavior is representative of our remaining datasets as well. The smaller reduction factors achieved on the heavy-hitter dataset are a result of its larger skew compared to the normal dataset. Another interesting observation is that larger memory factors reduce the impact of the set size on the reduction factor.

Our simulations show that most of the improvements in the reduction factor are achieved by 4- or 8-way set-associative caches. An 8-way set-associative cache performs very close to a fully set-associative cache in most cases. Any further increase in the degree of associativity (i.e., set size) does not seem to have a major impact on the reduction factor, but will continue having hardware complexity implications. Indeed, increasing the degree of associativity requires wider BRAM memories to store the larger sets, more comparators for finding a matching group, and more complex logic for executing any sophisticated replacement policy. These factors affect both the resource utilization and achievable frequency of the design. Therefore, the optimal degree of associativity must be chosen based on the design constraints and the underlying FPGA technology. We shall continue our study with an 8-way set-associative cache as it demonstrates great reduction performance and is a good fit for our FPGA platform (see Section 9.5).

9.3.5 Comparison of Cache Structures

In this section, we compare an 8-way set-associative cache with equivalent caches of different structure. *Two caches are equivalent* if they have the same size and offer the same number of entries for an inserting element. An 8-way set-associative cache maps an element to a single set, allowing it to be placed (inserted or accumulated) into any of the



(a) Moving Cluster

(b) Uniform

(c) Zipf

Figure 9.4: The results of the simulations comparing different cache structures for early aggregation on various datasets. All the caches used in these benchmarks employ the optimal replacement policy.

8 entries in that set. A 2-lookup 4-way set-associative cache allows a row to be placed in any of the 4 entries of 2 distinct sets, for or a total of 8 placement opportunities. A k-level n-way set-associative cache with $k \times n = 8$ also offers 8 placement opportunities to every single row. These algorithms are all compared in Figure 9.4. We continue using the optimal replacement policy in these simulations, as justified in Section 9.3.4.

For the moving cluster dataset, the choice of the cache structure does not have a noticeable impact on the reduction factor. This is due to the clusteredness of the distribution explored in Section 9.3.4. Similar behavior was observed on the sorted dataset.

For both the uniform and Zipf datasets, every k-level cache exhibits worse reduction performance than the 8-way set-associative cache. Moreover, the reduction factor degrades with higher k. This can be explained through the scope of the placement and replacement algorithm. Indeed, when inserting a row into the 8-way set-associative cache, the algorithm has a global view of all the possible entries that may be targeted for placement and/or eviction. In k-level caches, the algorithm has to make decisions with limited visibility locally at each level. This is specially the case for hardware implementations where the cache levels are arranged in a pipeline (each working on a different row) to achieve high throughput. In a software implementation, the algorithm might obtain a global view of the target entries in all levels before making a decision. Other types of multilevel cache structures (e.g., with varying capacities at different levels) are also less efficient for the same reason. Finally, note that a multilevel cache might be used to break down the hardware complexity of an equivalent multi-way set-associative cache (explained in Section 9.3.4). Our novel cache architecture described in Section 9.4 does the same without requiring a multilevel implementation.

The 8-way set-associative cache performs worse than the 2-lookup 4-way set-associative cache for small memory factors ≤ 5, but then catches up, achieving even slightly better reduction factors for memory factors > 5. In deciding between these two cache structures, two aspects should be considered. First, greater importance should be given to larger memory factors. Indeed, even though small memory factors (≤ 5) offer interesting case studies for comparing caching mechanisms, they only represent a small range of possibilities for the number of groups in the input. Second, the hardware complexity of both algorithms should be considered. In this case, data hazard detection and resolution due to collisions are of particular interest. In flat set-associative caches, collisions are detected by comparing the set index of a new row with the set indices of some of the rows already in the cache pipeline. The complexity is greatly increased for m-lookup caches that need to compare each of the m set indices of a new row, with all m set indices of some of the rows already in the cache pipeline. To add to the mix, considering the large width of database rows, multi-cycle ALUs are commonly used in FPGA-based accelerators, including our own MSMK [MFL+21, STM+13]. These ALUs increase the latency (in clock cycles) of an update to a set, further intensifying the complexity of hazard detection and resolution mechanisms. Moreover, if pipeline stalls are used as a means for dealing with hazards, multiple hash functions may result in lower throughputs as the probability of collision is increased. Furthermore, 2-lookup caches need to compute 2 hash functions, and also deal with (rare but possible) situations where both hash functions return the same result. Finally, an efficient implementation of a 2-lookup cache requires quad-port BRAMs that are not available to all FPGAs. To conclude, the slightly improved reduction factors achieved by a 2-lookup cache on small memory factors do not warrant its higher collision rate (potentially causing more stalls) and hardware complexity. We shall therefore continue our study with flat set-associative caches.

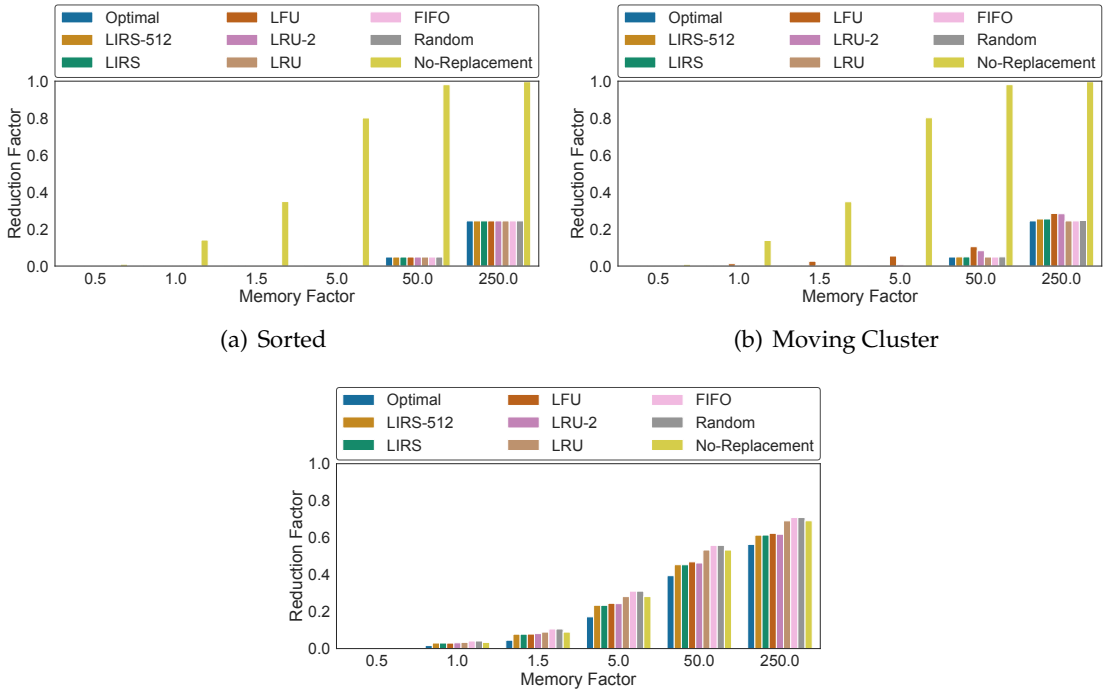
9.3.6 Comparison of Replacement Policies

The results of our simulations comparing the reduction performance of 8-way set-associative caches using different replacement policies are shown in Figure 9.5. All replacement policies, except from LIRS-512, are described in Section 9.2.2. LIRS-512 is an approximation of the LIRS policy introduced in Section 9.4.5. All other replacement policies are implemented accurately, i.e., no approximation was used.

All replacement policies (with the exception of no-replacement) perform equally well on the sorted dataset. The sortedness of the dataset means that there is no benefit in keeping a group in the cache once it has been seen before. The no-replacement policy performs poorly by never evicting a group from the cache.

The moving cluster dataset is a good differentiator for frequency-based replacement policies like LFU and LRU-2. As explained earlier in Section 9.3.1, the moving cluster distribution represents scenarios where the distribution of the grouping key changes depending on our position in the table. Frequency-based policies, however, try to keep **popular groups** (with many hits) in the cache for as long as possible. Unfortunately, a change in the distribution would not immediately affect the popularity of old data in the cache. This explains the poor performance of frequency-based policies on the moving cluster dataset. Recency-based policies adapt faster to a change in the distribution hence perform well on this dataset.

The opposite observation is made on the fairly skewed self-similar dataset. Here, a few groups have many members, making the frequency-based policy of keeping popular groups in the cache very effective. A recency-based policy, however, may evict popular groups if they have not been observed very recently. The FIFO and the random



(a) Sorted

(b) Moving Cluster

(c) Self-Similar

Figure 9.5: The results of the simulations comparing different replacement policies for an 8-way set-associative cache performing early aggregation on various datasets.

replacement policies that belong to neither of the two categories perform poorly on the self-similar dataset as well.

LIRS, a mixed policy relying on both the recency and the frequency of the keys in the stream of rows, performs well across all of our datasets.

In comparing replacement policies, their memory overhead must also be taken into account. For instance, every replacement policy requires one **valid bit** per cache entry indicating if it contains a group. A basic LRU implementation on an 8-way set-associative cache requires 3 additional bits per entry for keeping track of the least-recently used group. Random replacement, on the other hand, requires no additional bits. Given a cache structure and an amount of memory dedicated to the cache, the number of overhead bits affects the depth of the cache. Simply put, more overhead bits result in shallower caches. For the sake of accuracy, we have also performed simulations similar to those seen in this section, but where the depths of our caches are adjusted so they all use (approximately) the same number of memory bits. The datasets remain the same. The results of these simulations are fairly similar to the ones studied in this section. The only interesting observation was made on the uniform dataset. The reduction performance of a cache on uniform data depends strongly on its capacity, resulting in low-overhead policies (e.g., random, FIFO) achieving slightly lower reduction factors compared to high-overhead policies (e.g., LRU-2, LIRS). Apart from this, all of our prior observations remain valid.

9.3.7 Cache Selection Methodology

Based on our simulations, we can confidently claim that the family of n-way set-associative caches with the LIRS replacement policy is a good fit for early aggregation

on FPGAs. The value of n affects both the reduction factor and the complexity of the implementation. Since the cache is typically part of a larger system, we suggest increasing the degree of associativity of the cache as long as the system's target frequency and resource utilization are not violated. Any further increase, specially above 8 ways, is unlikely to be beneficial given the tiny gains in reduction performance.

9.4 CACHE SYSTEM ARCHITECTURE

The most challenging aspect of implementing a cache for early aggregation is dealing with two types of data hazards:

- **Hazards due to key collisions:** A row belonging to the same group as its predecessor needs to wait for the latter to be aggregated (compared and accumulated) into its group in the cache before it aggregates with the result. This will most likely lead to stalls due to the common use of multi-cycle ALUs in database system accelerators (see Section 9.3.5).
- **Hazards due to hash collisions:** Consecutive rows targeting the same cache set require either stalls or forwarding schemes to avoid read-after-write hazards.

Therefore, both key and hash collisions are likely to introduce stalls, resulting in a violation of *Spec 2* from Section 9.2.

Thanks to the nature of our application, we can efficiently eliminate almost all collision-related stalls by factoring collision detection and resolution out of the cache and adding them as preprocessing stages, still performed on the FPGA, just before the cache. Indeed, a cache implemented with a pipeline of depth w (which includes multi-cycle ALUs) does not need to deal with collisions if the colliding rows arrive at least w clock cycles apart. Therefore, the preprocessing stages must **eliminate every collision within all windows of w consecutive rows**. We shall use the term *window key/hash collisions* to refer to these types of collisions. We propose the following preprocessing stages:

1. ***Window aggregation*:** This stage **eliminates window key collisions** by simply aggregating them as they are detected. **No stalls** are added.
2. ***Window collision detection and resolution*:** This step detects and **eliminates window hash collisions** by introducing stalls, in the form of "empty rows".

We must find out the extent to which the stalls introduced by the second preprocessing stage violate *Spec 2*. The number of stalls depends on the number of hash collisions, and therefore the quality of the hash function. According to our simulations using MurmurHash in a 2^{16}-entry 8-way set-associative cache on both synthetic and real datasets (from Sections 9.3.1 and 9.5.3), after window aggregation the ratio of the number of hash collisions within windows of $w = 4$ rows to the number of rows in the input is at most 3.72×10^{-4}. In the worst case scenario where every colliding row has to be stalled by $w - 1$ clock cycles, the cache system can still achieve 99.89% line-rate processing speeds. $w = 4$ is the depth of the cache pipeline used in our experiments in Section 9.5.

An overview of the architecture of the cache system is shown in Figure 9.6. The following sections include a detailed description of the modules involved in the architecture.

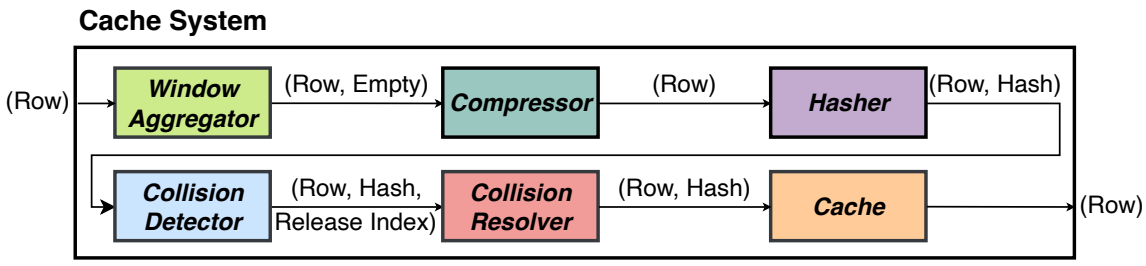
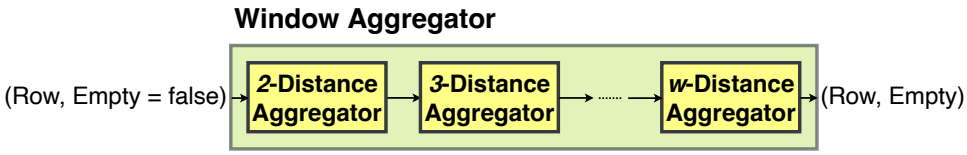


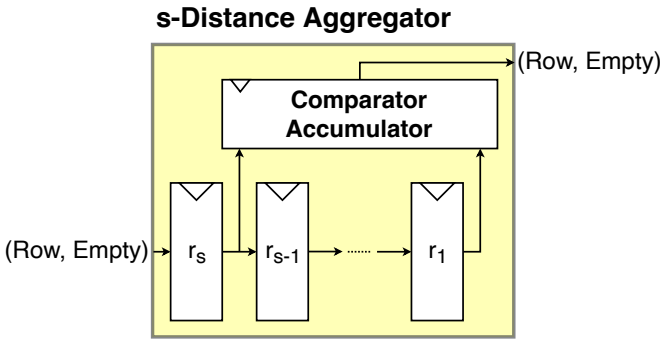
Figure 9.6: Overview of the architecture of the cache system. The pipeline consists of a few preprocessing stages that deal with window key/hash collisions, before the rows are processed by the cache itself.

9.4.1 Window Aggregator

The *Window Aggregator* is responsible for eliminating key collisions in all windows of w consecutive rows. It is composed of a series of *s-Distance Aggregators*, as shown in Figure 9.7(a). An *s-Distance Aggregator* compares and potentially accumulates rows positioned at the two ends of a window of s rows, i.e., they have exactly $s - 2$ other rows in between. Its architecture is shown in Figure 9.7(b).



(a) *Window Aggregator* with window size w



(b) *s-Distance Aggregator*

Figure 9.7: Architecture of the *Window Aggregator* used as a preprocessing stage in the cache system, as shown in Figure 9.6. The module is responsible for eliminating window key collisions.

A sequence of `(row, empty)` tuples arrive from the left. `Empty` is a Boolean flag. `Empty = true` means that the accompanying row is invalid yet considered as a row in a window of rows. It is different from the "valid" signal in handshaking protocols (which we also use) indicating the presence or absence of a row. The rows then progress through a shift register of depth s. The keys of the first (r_1) and the last (r_s) rows in the shift register are compared by the *Comparator Accumulator*:

- If they match, the rows are accumulated, and the result output. The last row is then marked as empty, so it won't be considered for future operations.
- If they do not match, the first row is output.

No matter the number of cycles needed to perform the comparison or the accumulation operations (aggregation for short), the module can function at line-rate speeds. It does not introduce any stalls, nor does it increase the size of the stream. Finally, note that the empty marker is needed for correct functionality. Indeed, if empty rows are simply marked as invalid and not considered as actual rows, the stream can shrink causing an *s-Distance Aggregator* to see different windows than it would have in the absence of its predecessors.

9.4.2 Compressor & Hasher

The *Compressor* removes all the empty rows from the stream. This may undo the work of the previous stage, resulting in window key collisions. The reason we do this is explained in Section 9.4.3.

The *Hasher* computes the hash (of the key) of each row according to the 32-bit MurmurHash 3 function from [App16]. The implementation is pipelined for line-rate processing.

9.4.3 Collision Detector

The stream of rows arriving at this stage may have both window key collisions **and** window hash collisions. The *Collision Detector* calculates the **minimum** number of stalls that must be inserted after each row in order to eliminate both types of collisions. It augments valid rows with **release indices** defining their position within a new collision-free stream of rows. For instance, a stream of `(row, release index)` tuples $[(y, 5), (x, 1)]$ requires row y to be inserted into the cache at least $5 - 1 = 4$ clock cycles after row x.

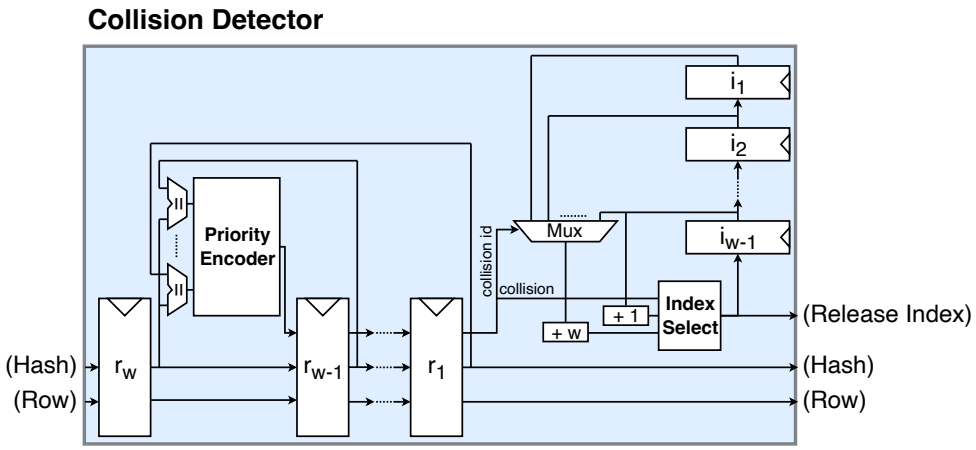


Figure 9.8: Architecture of the *Collision Detector* used as a preprocessing stage in the cache system, as shown in Figure 9.6. The module computes the minimum number of stalls required for each row in order to eliminate all window collisions.

The architecture of the *Collision Detector* is shown in Figure 9.8. Rows, along with the hash of their key are inserted into a shift register of depth w. The last row's (r_w) hash is compared to the hash of all the other rows in the pipeline (as they all belong to the same window). The results of the comparisons are fed into a *Priority Encoder* which produces two information: (1) collision = true if r_w collides with any of the other rows in the window, (2) collision id = the register id of the closest colliding neighbor, if one exists. As both key collisions and hash collisions imply rows producing the same hash, the module detects both types of collisions. The output of the *Priority Encoder* is shifted along its corresponding row.

The release indices are computed as follows. A shift register of depth $w - 1$, storing the previous release indices is initialized at reset to all 0's. When a row arrives at the front of the main shift register (r_1), *Index Select* calculates its release index i_w as follows. If the row causes no collisions $i_w = i_{w-1} + 1$. If a collision was detected, $i_w = max(i_{\text{collision id}} + w, i_{w-1} + 1)$. Once i_w is calculated, it is pushed to the back of the release index shift register, becoming the next i_{w-1}.

Finally, let us explain the reason behind the usage of the *Compressor* after the *Window Aggregator*, via an example. Assume $w = 3$ and consider an input stream of keys $[a, a, a, b, b, b, a, a, a]$. After the *Window Aggregator*, the stream becomes $[\_, \_, a, \_, \_, b, \_, \_, a]$ with "_" representing an empty row (stall). After compression, it transforms into $[a, b, a]$. The *Collision Detector* then recommends converting it into: $[a, \_, b, a]$. All the key collisions are resolved, and the stream is even shorter than its original length. The role of the *Window Aggregator* and the *Compressor* is to ensure that the stalls imposed by the *Collision Detector* **due to key collisions** do not increase the length of the stream beyond its original value. In the absence of these modules, the *Collision Detector* would have imposed a conversion into $[a, \_, \_, a, \_, \_, a, b, \_, \_, b, \_, \_, b, a, \_, \_, a, \_, \_, a]$, a much larger stream violating *Spec 2*.

9.4.4 Collision Resolver

The *Collision Resolver* is simply composed of an index counter and a comparator. It feeds an incoming `(row, hash, release index)` tuple to the cache when `release index` equals the value of the index counter. The index counter is incremented every clock cycle.

9.4.5 Cache

The architecture of the *Cache*, greatly simplified thanks to the preprocessing stages, is shown in Figure 9.9. The hash of a row is the address of its corresponding set in the cache memory. Once the set is fetched, the *Multi Comparator Accumulator* tries to find a group matching the new row. If one is found (*hit* = true), the new row is accumulated into that group and returned as the *result*. The *select* signals specify the index of the matching entry in the set. The *Replacement* module returns the index of an invalid (i.e., currently empty) entry or, if all entries are valid, one chosen for eviction according to the LIRS-512 replacement policy (defined below). The *Update* module gathers and processes these information in order to update the cache memory (both the cache set and replacement information) accordingly. It also forwards evicted or overflowed rows to the next stage in the aggregation pipeline. The depth of the cache pipeline, w, can be trivially increased allowing us to use synchronous memories and multi-cycle ALUs in the design. The preprocessing stages must then be updated to support the larger w.

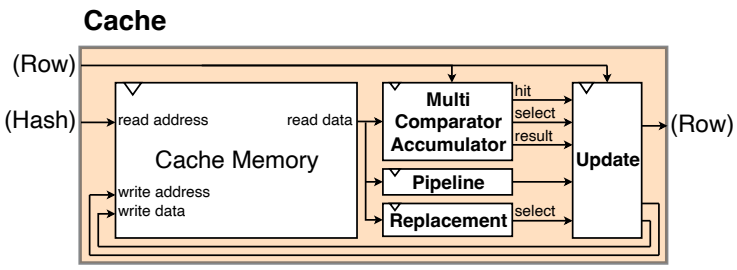


Figure 9.9: Architecture of the cache used for early aggregation in the cache system, as shown in Figure 9.6. The cache expects its input to be free of window key/hash collisions, which is a property ensured by the preprocessing stages described in this section.

The *Cache* also includes control mechanisms for invalidating all of its entries before any row is inserted, and for outputting all of its valid entries after the entire input is processed. Both mechanisms are trivial and, for the sake of clarity, excluded from our schematics.

Let us now briefly discuss our implementation of the LIRS replacement policy. LIRS requires two counters per cache entry to store the times of the last two accesses (insertion or hit) to that entry. The counters must be able to count up to the number of rows in the input. For large tables, this results in a lot of memory overhead. We devised a low-overhead LIRS approximation using reverse aging and saturating counters. The idea is as follows: The counters may be smaller than required by the algorithm. When a cache set is accessed, we emulate the passing of time by decrementing both counters of all of its entries by 1. If a counter holds 0, it may remain at 0. If a particular entry is accessed, we simply set its last access counter to the maximum value it can hold. LIRS-512 is an LIRS approximation using 9-bit counters. The results of our simulations shown in Figure 9.5 demonstrate that it performs almost similarly to the accurate LIRS algorithm, on an 8-way set-associative cache with 2^{16} entries.

9.5 EXPERIMENTS

In this section, we present and benchmark an implementation of our novel cache system. We use MSMK as a basis for both implementing and benchmarking our design. Our modifications to the MSMK architecture are shown in Figure 9.10.

In the *Sorters* module, instead of the 32 treap sorters, we now use 8 heap sorters. Treap sorters were proposed in Chapter 7 for their simplicity and the ability to perform fixed-length sorting along with early aggregation. Heap sorters were proposed in Chapter 6 because of their great sorting performance and low resource utilization. The heapsort implementation used in this section is capable of traversing a heap **two levels** at a time without stalls, thus resulting in even higher performance. This is made possible through the use of simple quad-port BRAMs and the lookahead optimization (introduced in Section 6.3.2) eliminating control hazards while traversing the heap. Our implementation of the heap sorter is on average 4 times faster than that of the treap sorter, yet **unable to perform early aggregation**. Both types of sorters are configured to produce fixed-length runs of 2048 rows.

The task of performing early aggregation is of course now delegated to the *Cache System*. The latter is implemented as a stage in the sort phase, (early-)aggregating rows just before

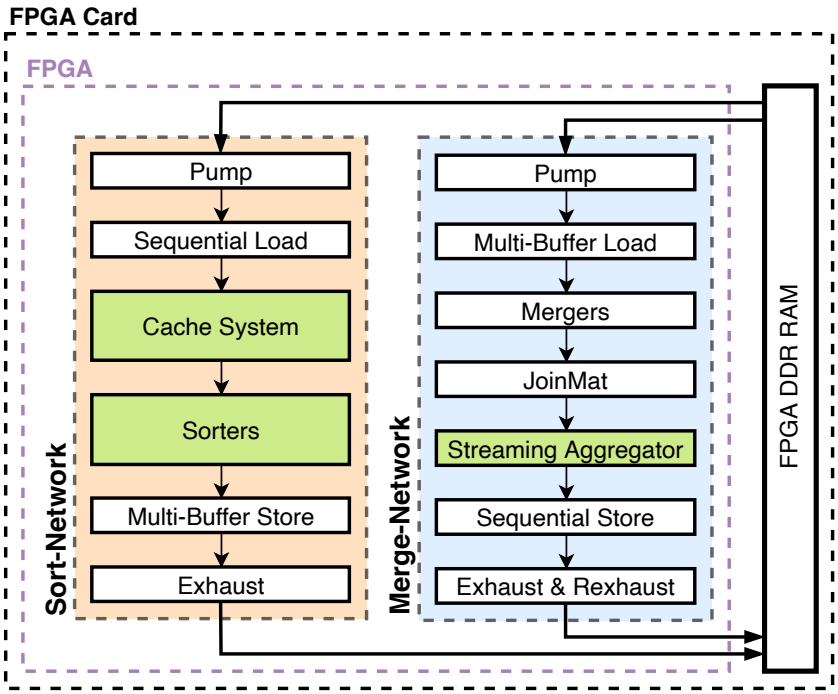


Figure 9.10: Architecture of CbMSMK. The new (*Cache System* and *Streaming Aggregator*) and upgraded (*Sorters*) modules compared to the architecture of MSMK are highlighted in green.

they are distributed to the 8 heap sorters. The cache has enough capacity to hold $32 \times 2048 = 65536$ groups, equivalent to the buffering capacity of the treap sorters in the original MSMK architecture.

Since the heap sorters are not capable of performing early aggregation, they might produce sorted runs with duplicate keys. Given the implementation of the *Mergers* described in Section 7.3.3, they can merge these sorted runs, but may not be able to fully aggregate them on their own. We have therefore inserted a *Streaming Aggregator* module in the *Merge-Network* pipeline. It performs streaming aggregation on the sorted rows as explained in Section 4.3.2. The design of this module is straightforward: it compares neighboring rows and accumulates those with matching keys.

We call our new implementation "Cache-based MSMK " or CbMSMK for short. Our benchmarks in this section compare the performance of CbMSMK with both MSMK and MonetDB [Mon21].

9.5.1 Experimental Setup

We implement CbMSMK in Chisel 3.5.0 [BVR+12, Chi22]. The benchmarks run on our PCIe-attached Intel® D5005 FPGA acceleration card with an Intel® Stratix® 10 FPGA [Cor19a]. Similar to MSMK, the design is configured with a data path wide enough to natively support rows with up to six 32-bit columns (without help from KeRRaS), and uses 2-cycle ALUs for processing these rows at our target 182 MHz frequency. The software competitor, MonetDB v11.39.5, runs on an Intel® Xeon® Platinum 8180 CPU with 376 GB of DDR4 RAM [Int17].

9.5.2 Resource Utilization and Parameter Tuning

Table 9.1 compares the resource utilization of multiple implementations of early aggregation. For sort-based early aggregation, it is hard to distinguish the resource utilization of pure sorting from that of early aggregation. Therefore, we consider sorting as part of early aggregation for both cache-based and sort-based algorithms. Frequency in MHz is the frequency that can be achieved by each implementation of early aggregation, alone. The frequency of the entire (acceleration) platform is discussed below.

Impl.	n	Logic Utilization [ALM]	BRAM Utilization [M20K]	Frequency
MSMK	N/A	86279 (9.25%)	1120 (9.56%)	264
CbMSMK	1	40239 (4.31%)	1063 (9.07%)	289
CbMSMK	2	40903 (4.38%)	1078 (9.20%)	262
CbMSMK	4	41526 (4.45%)	1078 (9.20%)	241
CbMSMK	8	43211 (4.63%)	1074 (9.16%)	223
CbMSMK	16	46657 (5.00%)	1070 (9.13%)	204
AltCache	8	40223 (4.31%)	1052 (8.98%)	149

Table 9.1: FPGA resource utilization of various implementations of early aggregation. n is the degree of associativity in cache-based implementations. The percentage resource utilizations are calculated over the total amount of resources provided by the FPGA.

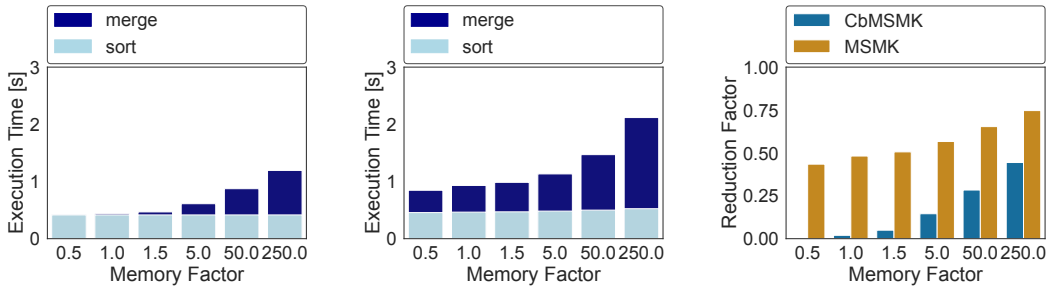
The BRAM utilizations of all the implementations are fairly similar. However, MSMK uses almost double the logic resources compared to any of the CbMSMK implementations, due to its use of the 32 treap sorters. The logic utilization of the CbMSMK implementations increases with n. The AltCache implementation also performs cache-based early aggregation. It has a stall-free cache that uses typical forwarding schemes to deal with collisions, instead of our preprocessing mechanisms. Compared to the CbMSMK implementations, AltCache uses slightly less FPGA resources, but reaches a considerably lower frequency due to its complexity.

We shall follow the methodology described in Section 9.3.7 to select an adequate CbMSMK implementation. The target frequency of our platform (constrained by various modules) is 182 MHz. The synthesis tools are able to reach that frequency for $n \leq 8$. Any further increase in n results in a negative slack, imposing lower frequencies for only slightly better reduction performance. We have therefore decided to **use the implementation with $n = 8$ in our benchmarks**.

To showcase the scalability of our cache system architecture, we generated one more FPGA build with a cache 8 times larger (in capacity) than the one used in our benchmarks, occupying more than half of the BRAMs on our FPGA. The new build could only run at 159 MHz. By increasing the depth of the cache pipeline from 4 to 6, the achievable frequency rose to 247 MHz. This modification cost us only 0.2% more FPGA logic resources, and a decrease in performance from 99.89% to 99.69% line-rate processing speeds.

9.5.3 Datasets

In addition to the synthetic datasets introduced in Section 9.3.1, we also use the following real data in our benchmarks:



(a) CbMSMK Sort and Merge Times

(b) MSMK Sort and Merge Times

(c) Reduction Factors of CbMSMK and MSMK

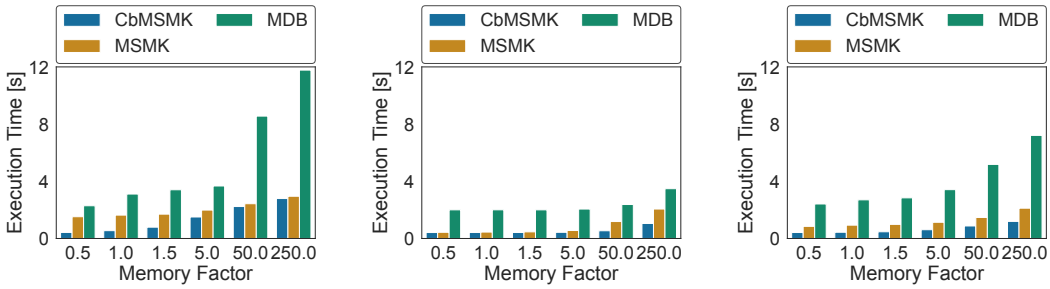
Figure 9.11: The results of the benchmarks comparing the sort times, merge times, and reduction factors of CbMSMK and MSMK on the Zipf dataset.

- **ICE:** A table providing information on the WiFi system onboard the ICE trains in Germany [Deu17]. It holds $22,266,745$ rows. We aggregate the table using either the latency, upload speed, or download speed column as the group key. The resulting queries are called ice_lat, ice_tptx, and ice_tprx, respectively.
- **Political:** A table containing the contributions of individuals to federal committees in the US from 2015 to 2016. It holds $20,459,430$ rows. We aggregate the table using either the city or the zip code column as the group key. The resulting queries are called pol_city and pol_zip, respectively. A fairly similar set of queries is used in [Lar02].

9.5.4 Benchmarks on Synthetic Data

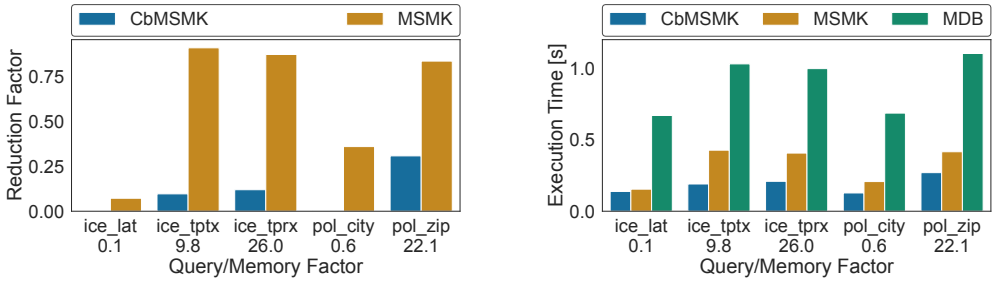
The benchmarks presented in this section compare the execution times of CbMSMK and MSMK performing aggregation on different datasets. **The execution times include both early aggregation and complete aggregation**. Let us first study the key factors contributing to these execution times. Figures 9.11(a) and 9.11(b) show the execution times of both implementations decomposed into sort time and merge time, on the Zipf dataset. Sort times include early aggregation. First note that the sort times across all memory factors and implementations are approximately the same. This is due to the fact that both sort pipelines are built to process data at near line-rate speeds and that they need to read their input table once (all tables have the same number of rows). The merge times, however, depend on the amount of data there is to merge, which is directly proportional to the reduction factors of the early aggregation algorithms. Moreover, the *Merge-Network* is implemented with optimizations (e.g., aggregation while merging) that affect merge times as well. The reduction factor also determines the decrease in the **memory footprint** of the merge phase. An early aggregation algorithm with better reduction performance allows an FPGA (with a limited amount of RAM) to process larger tables. The reduction factors on the Zipf dataset are shown in Figure 9.11(c).

The total execution times of our implementations are shown in Figure 9.12. On the uniform dataset, CbMSMK runs more than $3\times$ faster than MSMK for small memory factors. The difference in execution times almost disappears for larger memory factors. The opposite is true on the moving cluster dataset, where the two platforms have similar performance for small memory factors, while the performance of CbMSMK surpasses that of MSMK as the memory factor increases. The Zipf dataset presents a scenario where CbMSMK keeps a steady lead in performance across all memory factors. These behaviors can be explained through our analysis of the reduction performance in Section 9.3.3.



(a) Uniform (b) Moving Cluster (c) Zipf

Figure 9.12: The results of the benchmarks comparing the performance of CbMSMK, MSMK, and MonetDB (MDB) on synthetic data.



(a) Reduction (b) Execution Times

Figure 9.13: The results of the benchmarks comparing the performance of CbMSMK, MSMK, and MonetDB (MDB) on real data.

MonetDB performance numbers are also included on our plots as a reference point for comparison. In all of our benchmarks, CbMSMK is significantly faster than MonetDB.

It is worth noting that although the cache system reduces the execution time of the aggregation operator, it does not improve the overall system throughput (in million rows per second). Indeed, the throughput of a system is limited by the slowest element in its processing pipeline. In our case, the *Sort-Network* is the slowest element, and has unchanged running times for the reasons mentioned above. This is why our benchmarks in this chapter focus on execution time rather than throughput like in Chapters 7 and 8.

9.5.5 Benchmarks on Real Data

The results of our benchmarks on real data are shown in Figure 9.13. CbMSMK achieves much better reduction factors compared to MSMK across all datasets. With respect to execution times, both platforms perform well on data with small memory factors, while CbMSMK runs up to $2\times$ faster on data with larger memory factors. Finally, observe that CbMSMK runs on average $5\times$ faster than MonetDB.

9.6 SUMMARY

The objective of this part of our work was to study early aggregation in the context of database system acceleration on FPGAs. We started the chapter with a guided survey

on different early aggregation algorithms, suitable for an efficient FPGA-based implementation. We then performed software simulations comparing the most promising of these algorithms in terms of their reduction performance. Set-associative caches with the LIRS replacement policy demonstrated great performance with modest implementation complexity. We then proposed a novel architecture for implementing these caches. The architecture can achieve near line-rate throughputs despite data dependencies typically caused by collisions. We also presented a low-overhead approximation of the LIRS algorithm. Benchmarks revealed that CbMSMK, our new accelerator equipped with our novel cache-based early aggregation architecture, performs at least as well as MSMK, with great speedups (of up to $3\times$) achieved in many scenarios for the aggregation operator.

10

THE FULL PICTURE

10.1 System Architecture

10.2 Benchmarks

10.3 Meeting the Objectives

In this chapter, we present an overview of the final version of our acceleration architecture, CbMSMK. We then provide some single-operator and system-level TPC-H benchmarks, showcasing the ability of our platform to process simple and complex queries. Finally, we revisit the objectives and goals outlined in Chapter 5 and discuss how CbMSMK effectively meets these goals.

10.1 SYSTEM ARCHITECTURE

A high-level depiction of our acceleration platform is shown in Figure 10.1. The FPGA card is connected to the host computer via PCIe, as an accelerator type platform described in Section 4.2. Since we target the acceleration of in-memory database systems (see Section 2.1.2), the host has the initial database stored in its RAM, which must be transferred to the FPGA for processing. The FPGA hosts multiple *CbMSMK Cores*, each containing at least one instance of both the *Sort-* and *Merge-Networks*. This allows each core to be able to independently process all of the operators supported by our accelerator. The networks are configured and launched by the scheduling algorithms running on the host CPU.

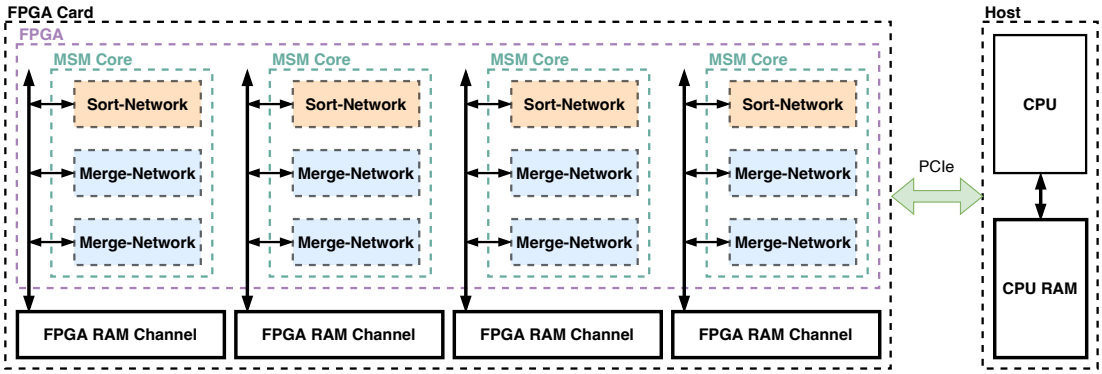
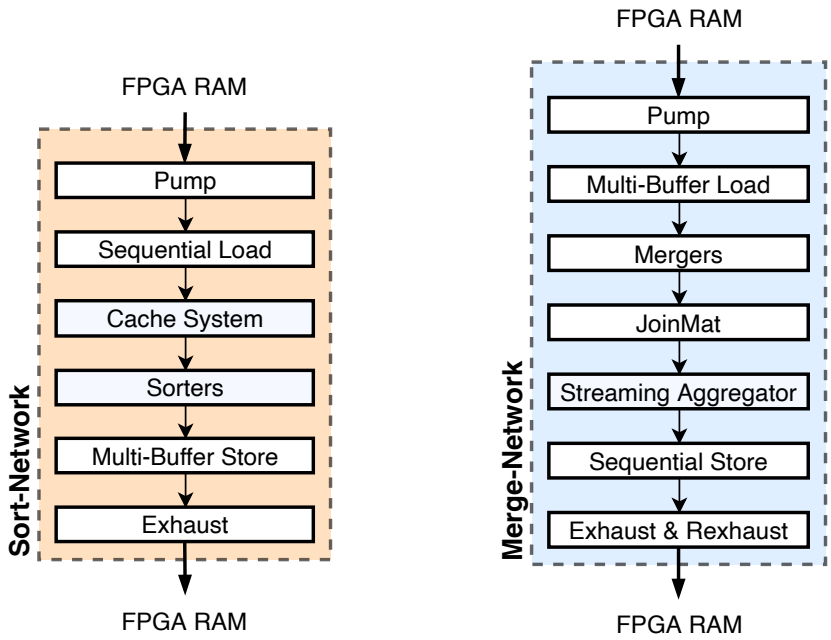


Figure 10.1: Architecture of the CbMSMK acceleration platform.

The block diagrams of the *Sort-* and *Merge-Networks* are presented in Figure 10.2. They can together run the sort-merge algorithm, in order to process the sort, aggregation, and join database operators. By using column-oriented load and store techniques, they can also execute the projection operator. Let us briefly describe the flow of data through the networks.

The input data, in the form of a database table, is initially stored column by column on the FPGA RAM. Indeed, CbMSMK is a column-oriented database accelerator. The *Pump* and *Exhaust* modules convert row accesses (reads/writes) made by the remaining modules in the networks into column accesses before forwarding them to the FPGA RAM. They also perform projections by filtering out the columns that are not part of the input or result of the query.

The *Sort-Network* is principally responsible for executing the sort phase of the sort-merge algorithm. The *Sequential Load* module reads the input table, row by row (thanks to the *Pump*), streaming each row into the accelerator's processing pipeline. The *Cache System* is only activated for the aggregation operator. It partially aggregates the input, hoping to reduce its size and thus the runtime of the entire aggregation operator, as described in Chapter 9. The (potentially partially aggregated) rows are then forwarded to the *Sorters*. The latter is composed of a set of 8 parallel heap sorters, proposed in Chapters 6 and 9.



(a) Sort-Network

(b) Merge-Network

Figure 10.2: Architecture of the *Sort-* and *Merge-Networks* used in CbMSMK for executing the sort-merge algorithm. The latter is used as a basis for running our target pipeline-breaking database operators.

They then transform the stream into runs of sorted rows. The sorted runs are stored in the FPGA RAM by the *Multi-Buffer Store* module.

The *Merge-Network* is principally responsible for executing the merge phase of the sort-merge algorithm. The *Multi-Buffer Load* module provides the *Mergers* access to the sorted runs stored on the FPGA RAM. The (128-way) *Mergers* merge the sorted runs, which after potentially multiple passes turn into the sorted table. During the intermediate passes, the newly produced (larger) sorted runs are directly stored on the FPGA RAM by the *Sequential Store* module, bypassing both the *JoinMat* and *Streaming Aggregator*. During the final merge pass, the *JoinMat* can be activated to materialize the result of a join operator, as described in Section 7.3.5. The *Streaming Aggregator*, introduced in Section 9.5, can also be activated to aggregate the sorted table.

Note that CbMSMK is built for flexibility. The configuration presented in this section (with 4 *MSM Cores* each with 1 *Sort-* and 2 *Merge-Networks*, 8 heap sorters per *Sort-Network*, etc.) is the one we commonly used in the previous chapters and should not be considered as the final product of our work. The flexibility of our architecture has been thoroughly examined in Sections 7.4.2, 8.5.6, and 9.5.2. We briefly touch upon this subject again in Section 10.3.

10.2 BENCHMARKS

In this section, we present the results of a set of summary benchmarks for our final acceleration platform, CbMSMK. The experimental setup consists of a PCIe-attached Intel®

(a) Sort Operator

(b) Aggregation Operator

(c) M:N Join Operator

Figure 10.3: The results of the benchmarks comparing the performance of CbMSMK and MonetDB on the sort, aggregation, and join operators with various datasets. These datasets are generated similarly to the ones in Chapter 7.

D5005 FPGA acceleration card with an Intel® Stratix® 10 FPGA [Cor19a] and 4×8GB of DDR4 RAM for the CbMSMK benchmarks, and an Intel® Xeon® Platinum 8180 CPU with 376 GB of DDR4 RAM [Int17] for the MonetDB v11.39.5 benchmarks.

The results of our single-operator benchmarks are shown in Figure 10.3. The datasets used in these benchmarks are similar to the ones described in Chapter 7. CbMSMK achieves up to approximately 6, 23, and 13 times higher performance compared to the state-of-the-art MonetDB on the sort, aggregation, and join operators; respectively. We refer the interested reader to Section 7.4 for a detailed analysis on the behavior of these benchmarks.

The results of our benchmarks on four TPC-H queries are shown in Figure 10.4. The datasets were generated using 3 separate scale factors. In TPC-H, a *scale factor* of x indicates that the size of the underlying database is approximately x GB. The datasets are preprocessed before executing the TPC-H queries:

- All date values were converted into their corresponding Unix time integer values [Wik23].
- All decimal values were converted into integers. Care was taken to ensure that this loss of precision did not affect the overall computational flow or the size of the final result.

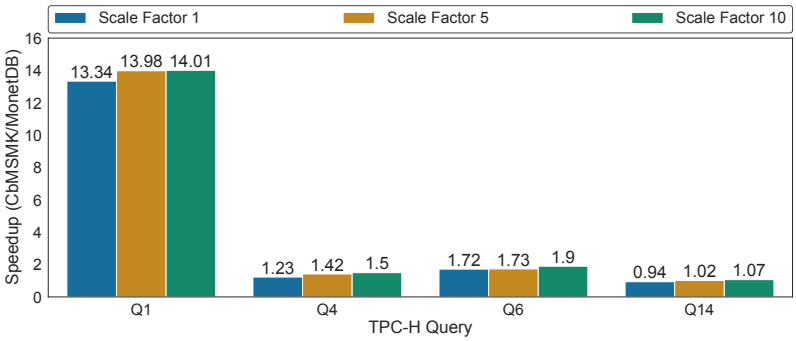


Figure 10.4: The results of the benchmarks comparing the performance of CbMSMK and MonetDB on the TPC-H dataset with various scale factors.

As many of these queries contain basic filter and arithmetic operators, we added a few static filter stages in the CbMSMK processing pipeline. With this, the FPGA is capable of executing all 4 of our TPC-H queries on its own. Indeed, the queries contain a mix of the projection, filtering, sort, aggregation and join operators on multiple tables with various sizes and layouts, all of which are supported by our accelerator.

There are two primary reasons for the relatively small speedups of CbMSMK compared to MonetDB on the TPC-H benchmarks. First, TPC-H queries are composed of chains of pipeline-breaking database operators that must be executed one after another. Our accelerator currently stores the results of each operator in the FPGA RAM, before passing them to the next one. This increases the overall runtime required to execute the queries. Second, the aggregation and join operators in the TPC-H queries are not particularly well-suited for acceleration by CbMSMK. For instance, aggregation is often performed on tables with only a few grouping keys and joins are typically 1:1 or 1:N equi-joins, both of which result in lower speedups as shown by our single-operator benchmarks in Figure 10.3.

Another notable observation is that the speedups tend to increase for larger scale factors. Indeed, increasing the size of the data reduces the overhead of launching the operators on the FPGA, compared to the execution time of the operators themselves.

Although further research and development will be necessary to optimize CbMSMK for processing chains of pipeline-breaking operators (e.g., by forwarding the result of one operator to another) and different operator configurations, our platform managed to perform as well as, or even significantly better than the state-of-the-art MonetDB on the TPC-H benchmarks. More importantly, thanks to its wide array of capabilities, CbMSMK demonstrated the ability to process these complex queries independently, without the need for external support from the host. Indeed, we managed to tackle one of the greatest challenges of using accelerators: data movement. Thanks to the capabilities of CbMSMK, the FPGA can now own, keep, and process the data locally, only transferring the results of the queries back to the host. With some additional development (e.g., adding support for the filter operator and more data types), it would be possible for the host system to almost completely offload the processing of intensive analytical queries to the FPGA, freeing itself from this computational burden.

10.3 MEETING THE OBJECTIVES

The objectives of our work were defined in Section 5.1. Below is a categorized list of these objectives, and how CbMSMK has been successful in meeting them:

- **Operator support:** CbMSMK is capable of accelerating all three of our target pipeline-breaking database operators, namely sort, aggregation, and join. Additionally, it supports the projection operator. All operators can process arbitrarily deep and wide tables. CbMSMK currently supports the processing of integers and strings, but has the potential to support additional data types by extending the ALUs.

- **Resource efficiency:** All pipeline-breaking database operators share the sort-merge pipeline, achieving resource efficiency through reuse. Moreover, CbMSMK takes advantage of the columnar storage of data to execute the projection operator, without incurring additional resource utilization.

- **Modularity and extensibility:** All the processing modules in the CbMSMK pipeline have a data interface based on the standard handshaking protocol [ARM20], which allows them to transfer one database row per clock cycle and exert backpressure as needed. This makes it easy to add or remove modules to or from the pipeline.

- **Flexibility:** CbMSMK is both designed and implemented (in Chisel) with flexibility in mind. Indeed, the architecture can be configured to fit various resource constraints and workloads. The number of *MSM Cores*, the number and type of networks within each *MSM Core*, the capacity and structure of the *Cache System*, the number and capacity of the sorters in the *Sort-Network*, the number of ways supported by the *Mergers*, and the capacity of the *JoinMat* are a few of the highly impactful, yet configurable parameters of the architecture. Even the storage model (i.e., row storage and column storage) of the accelerator can be modified by adding/removing the interface adapters (i.e. *Pumps/Exhausts*).

- **Support for multiple clients:** The multi-core design of CbMSMK allows it to support multiple clients at the same time. Moreover, each *MSM Core* is capable of executing all of the operators supported by our accelerator. This facilitates the scheduling and assignment of client queries to the *MSM Cores* on the FPGA.

- **Support for FPGA accelerator card platforms:** CbMSMK is optimized to run on accelerator card platforms. Indeed, our implementation on the Intel® D5005 FPGA acceleration card receives its data from the host memory, processes it, and returns the result back to the host. The architecture also takes advantage of the multiple RAM channels provided by the accelerator card.

In summary, although CbMSMK still has room for improvement, it currently satisfies our objectives and goals and conforms to our requirements. In Chapter 11, we list and discuss further directions for our research.

Part III

Conclusion

11

SUMMARY AND OUTLOOK ON FUTURE RESEARCH

11.1 Summary

11.2 Future Work

With the growing prevalence of hardware acceleration platforms (e.g., GPUs, FPGAs) in both on-premise and cloud setups, the development of custom architectures for database system acceleration has gained significant traction. Database system acceleration is a vast area of research that involves both hardware and software innovations and optimizations. Our work attempts to push the boundaries in this domain. In this chapter, we reiterate the important milestones and contribution of our work. We then discuss a few directions for future research.

11.1 SUMMARY

In Chapter 1, we explained the motivation for our research by highlighting the significance of performance in database query processing. We also discussed how the slowing down of Moore's law makes it increasingly challenging to meet the performance demands of modern database systems. We then outlined a set of requirements, based on industry needs, for a database system accelerator with practical applications. Essentially, the accelerator must support multiple operators, with priority given to the ones whose acceleration results in greater end-to-end system performance improvements. The architecture of the accelerator must be extensible, flexible, and optimized for in-memory database system acceleration. Finally, the design must provide opportunities to support multiple simultaneous clients.

In Chapters 2 and 3, we gave a brief introduction on the topics of database query processing and FPGA development. The goal of these chapters was to familiarize the reader with both domains and present the advantages and challenges of integrating the two. Past research on the incorporation of FPGAs into query processing pipelines were discussed in Chapter 4. There, we observed that there is a lack of acceleration platforms that align with the requirements defined in Chapter 1.

Chapter 5 marks the start of our contributions. There, we leveraged our knowledge of past research in order to convert the requirements defined in Chapter 1 into concrete objectives and goals for the architecture of our database system accelerator. The architecture was then defined in broad terms, left to be further refined and detailed in the subsequent chapters.

Database systems are typically implemented by software engineers using traditional languages such as C/C++. Therefore, the first objective of our research was to determine whether HLS tools can be leveraged to continue this trend on FPGAs. In Chapter 6, we compared Intel® FPGA SDK for OpenCL, which is a fairly mature HLS tool, with handcrafted RTL code in VHDL. The results of our study showed that while an OpenCL implementation may provide comparable performance to an RTL implementation in some situations, complex designs generally perform significantly better when implemented in RTL. Moreover, the FPGA resource utilization of OpenCL designs is almost always crucially worse than that of those written in RTL. Based on these results, we decided to use RTL rather than OpenCL for implementing our accelerator.

In Chapter 7, we introduced MSM, the first version of our database system accelerator. MSM uses the sort-merge algorithm to execute all three of the sort, aggregation, and join operators. These operators are particularly relevant to our work as they are both popular and computationally demanding. By accelerating them, we can significantly improve the overall performance of analytical queries, which is essential for meeting our requirements. The sort-merge pipeline is shared by all three operators, resulting in a resource-efficient design. This allowed us to instantiate multiple acceleration cores (*MSM Cores*), enabling the FPGA to process multiple queries at the same time; which helped us

meet another important part of our requirements. MSM is shown to have on average $5\times$ higher throughput (in million processed rows per second) than the state-of-the-art MonetDB for the sort, join, and aggregation operators.

A crucial requirement for our accelerator is that it must be able to support multiple, high-impact operators. This not only pertains to the number of operators supported, but also the size and layout of the tables they can accept as input. MSM is capable of processing tables with different data types and an arbitrary number of rows. However, it has a limitation on the width, or number of columns, of tables it can support. This is a limitation commonly found in past research. In Chapter 8, we proposed KeRRaS, an abstract sort algorithm that enables existing sort-based accelerators to support arbitrarily wide tables. MSMK is the result of integrating KeRRaS into MSM. The performance of the new architecture is similar to MSM on narrow tables and scales well as the table width increases. Furthermore, MSMK is a column-oriented accelerator (as opposed to the row-oriented MSM), which allows it to support the projection operator and a broader range of queries in general.

Having already met our requirements from Chapter 1, we decided to further investigate the FPGA-accelerated aggregation operator with the aim of improving its performance. In Chapter 9, we conducted a comparative analysis of various early aggregation algorithms for an FPGA-based implementation. An early aggregation algorithm is a low-complexity algorithm that can help reduce the size of the input to an aggregation operator, with the aim of decreasing its overall running time. Our study concluded that set-associative caches with the LIRS replacement policy perform very well and have a reasonable level of implementation complexity when used for early aggregation on FPGAs. Consequently, they were integrated into MSMK, resulting in a new architecture called CbMSMK. In our benchmarks, CbMSMK was able to run the aggregation operator up to $3\times$ faster than MSMK.

Finally, in Chapter 10, we provided an overview of the complete CbMSMK architecture. We then presented the results of some single-operator and TPC-H query benchmarks evaluating the accelerator in its final form. In the end, we revisited the objectives of our thesis, and discussed how they were met by CbMSMK.

11.2 FUTURE WORK

In this section, we briefly outline some potential directions for future research that could enhance our work in terms of both performance and applicability.

The first category of improvements relates to the capabilities of the platform. Indeed, by expanding these capabilities, the FPGA can accelerate a broader range of queries and a greater portion of each query. This makes the FPGA even more independent of the host system, further freeing up the CPU and reducing the need for host-FPGA data transfers. Below are a few suggestions in this direction:

- **Filtering:** Filtering is typically a computationally-inexpensive, yet very commonly used operator in database workloads. The main benefit of integrating filtering into our architecture would be to avoid round trips between the FPGA and its host. Given the size (e.g., large Boolean expressions) and complexity (e.g., arithmetic operations, regular expression matching, etc.) of many filter predicates, a

practical filter implementation must have a high degree of runtime configurability. Many FPGA-based architectures for filtering have been proposed in the literature [DZT13, SJT+12, SMT+14, SMT+12, WIA14]. We are currently integrating a tiny-CPU-based implementation of filtering into our platform. It is composed of multiple domain-specific CPUs with small instruction and data memories stored in BRAMs. They can be programmed to evaluate a wide range of predicates.

- **Other operators:** Support for additional streaming and pipeline-breaking database operators would help alleviate some more of the limitations of our database system accelerator. The challenge in this direction is that generality can come at the cost of performance. Therefore, it is incredibly important to strike a balance between the two. Thankfully, our architecture allows for many more operators to be supported without compromise. For instance, different types of anti-joins can be implemented by simply tweaking the *JoinMat* module in the *Merge-Network*.

- **Data types:** Currently, CbMSMK can process tables with integer and string columns. Additional data types can be supported by extending the ALUs used in the design. Fortunately, many database-specific data types, such as the Decimal data type defined in TPC-H, can be implemented on FPGAs very efficiently.

Another category of improvements concerns the performance of the system, based on its architecture. This has been a major theme of the thesis, and will likely remain so in future work. Below are a few suggestions for improving the operator- and query-level performance of CbMSMK:

- **General optimizations:** Further studies and investigations into various architectures for processing database operators are necessary. They can help improve the performance of our existing modules (e.g., improving the pipeline-efficiency of our heap sorters) or result in more efficient alternatives.

- **Partitioning:** Using table partitioning techniques, we can offload a single operator to multiple *MSM Cores*. This allows the accelerator to have higher single-operator performance in case the number of concurrently executing operators is smaller than that of the *MSM Cores*. Partitioning, even within a single core, can lead to better performance as it can break the problem down into smaller pieces. For instance, if the number of unique keys in each of the partitions of a table is smaller than the capacity of the *Cache System* in CbMSMK, a single pass of the *Sort-Network* may be able to fully aggregate the input.

- **Advanced interconnects:** Improved on-chip interconnects can help reduce the number of memory accesses and lower memory bandwidth requirements by moving data between the modules within the FPGA, instead of using the FPGA RAM as a buffer. For example, consider a chain of two pipeline-breaking database operators O_1 and O_2. To execute the chain, CbMSMK must store the result of O_1 in the FPGA RAM before it is loaded back up for executing O_2. A significant optimization would be to forward the result of O_1, obtained after the last pass of the *Merge-Network*, directly to the *Sort-Network* executing O_2. This can greatly improve the query-level performance of CbMSMK.

The features and characteristics of an FPGA platform have significant implications on the types of architectures that it can support, as well as the performance of an implementation of those architectures on the platform. Some areas that have yet to be explored in this field include:

- **Memory system:** Analytical database processing is inherently a relatively memory-intensive task. Fortunately, new and upcoming FPGA platforms offer increasingly better memory systems. For instance, FPGAs with larger and faster BRAMs and external memories such as high bandwidth memory (HBM) are becoming increasingly prevalent [WZTD19]. A promising area of research is to update and adapt CbMSMK to take advantage of these new memory technologies.
- **Scaling:** Similar to traditional database systems, FPGA-based database processing also has the potential to be scaled out and executed by a network of interconnected FPGAs. Microsoft is using this type of infrastructure to speed up web search and convolutional neural networks [CCP+16, PCC+15, ORK+15]. Farview is another platform using a similar setup for database system acceleration [KKK+22].
- **Support for additional platform types:** Accelerator card platforms are particularly well-suited for accelerating in-memory database systems, which has been the focus of our work. However, our architecture can be easily adapted to other types of platforms (e.g., network processors and smart storage platforms), thus increasing the applicability and scope of our research.

A number of unexplored feature- and performance-related areas of research pertain to end-to-end query execution. Indeed, despite all of its features, CbMSMK remains a **query engine** accelerator that can only **execute** the operators scheduled by the host. However, the other tasks typically performed by a DBMS (e.g., query parsing, query execution plan generation and optimization, etc.) are currently done manually with the architecture of the accelerator in mind. Further research is needed to both improve and automate these tasks:

- **DBMS integration:** A crucial next step for our research would be to integrate CbMSMK into a DBMS. This would involve a thorough examination to identify which components of the DBMS will be affected by the integration, and how they should be modified to accommodate it. As CbMSMK gains new capabilities and becomes increasingly independent from its host, it is likely that more extensive updates to the underlying DBMS will be necessary. Eventually, it may become more beneficial to research and develop an FPGA-specific DBMS from the ground up. Related works in this area are discussed in [BGB+18].
- **Near-FPGA scheduling:** In the current state of our platform, the host is responsible for scheduling and synchronizing the execution of the operators in a query execution plan on the FPGA. All the commands necessary for configuring and launching the FPGA must be transmitted through the PCIe bus, resulting in performance penalties. An alternative approach would be to transfer the query execution plan to a processing unit, such as a hard processor system (HPS), on the FPGA, which would then orchestrate the execution of its operators on the *Sort-* and *Merge-Networks*. This would reduce the scheduling and synchronization overheads as the HPS would be closer (in latency) than the host to the networks on the FPGA.

Finally, we would like to propose more extensive benchmarking as a requirement for the future development of our acceleration platform. Indeed, as this thesis primarily focused on accelerating various pipeline-breaking database operators, our benchmarks were specifically tailored to measure the performance of those operators. Although we did provide some TPC-H benchmarks to demonstrate the capabilities of our platform, a deeper analysis of the performance of our accelerator on complete queries is necessary to guide further development.

BIBLIOGRAPHY

[AAA+22] Daniel Abadi, Anastasia Ailamaki, David G. Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter A. Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Y. Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh M. Patel, Andrew Pavlo, Raluca A. Popa, Raghu Ramakrishnan, Christopher Ré, Michael Stonebraker, and Dan Suciu. The seattle report on database research. *Commun. ACM*, 65(8):72–79, 2022.

[ABW+16] Ildar Absalyamov, Prerna Budhkar, Skyler Windh, Robert J. Halstead, Walid A. Najjar, and Vassilis J. Tsotras. FPGA-accelerated group-by aggregation using synchronizing caches. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*, pages 11:1–11:9. ACM, 2016.

[AG12] Indu Arora and Anu Gupta. Cloud databases: a paradigm shift in databases. *International Journal of Computer Science Issues (IJCSI)*, 9(4):77, 2012.

[AKN12] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, 2012.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, volume 30 of *AFIPS Conference Proceedings*, pages 483–485. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967.

[AMH08] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 967–980. ACM, 2008.

[AN94] Arne Andersson and Stefan Nilsson. A new efficient radix sort. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 714–721. IEEE Computer Society, 1994.

[ANA10] Abhinav Agarwal, Man Cheuk Ng, and Arvind. A comparative evaluation of high-level hardware synthesis using Reed-Solomon decoder. *IEEE Embed. Syst. Lett.*, 2(3):72–76, 2010.

[ANS+14] Oriol Arcas-Abella, Geoffrey Ndu, Nehir Sönmez, Mohsen Ghasempour, Adrià Armejach, Javier Navaridas, Wei Song, John Mawer, Adrián Cristal, and Mikel Luján. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*, pages 1–8. IEEE, 2014.

[App16] Austin Appleby. MurmurHash3. `https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp`, 2016.

[ARM20] ARM. AMBA AXI and ACE protocol specification version D: Handshake process. `https://developer.arm.com/documentation/ihi0022/d/AMBA-AXI3-and-AXI4-Protocol-Specification/Single-Interface-Requirements/Basic-read-and-write-transactions/Handshake-process?lang=en`, 2020. (Accessed on 12/04/2022).

[AWS17] Inc. Amazon Web Services. Amazon EC2 F1 instances, customizable FPGAs for hardware acceleration are now generally available. `https://aws.amazon.com/about-aws/whats-new/2017/04/amazon-ec2-f1-instances-customizable-fpgas-for-hardware-acceleration-are-now-generally-available/`, 04 2017. (Accessed on 11/11/2022).

[Azu22] Microsoft Azure. NP-series - Azure virtual machines. `https://learn.microsoft.com/en-us/azure/virtual-machines/np-series`, 09 2022. (Accessed on 11/11/2022).

[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/`, 04 2021. (Accessed on 11/11/2022).

[BAS04] Ravi Budruk, Don Anderson, and Tom Shanley. *PCI express system architecture*. Addison-Wesley Professional, 2004.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[BATÖ13] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multicore, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, 2013.

[BBC+12] Ronald Barber, Peter Bendel, Marco Czech, Oliver Draese, Frederick Ho, Namik Hrle, Stratos Idreos, Min-Soo Kim, Oliver Koeth, Jae-Gil Lee, Tianchao Tim Li, Guy M. Lohman, Konstantinos Morfonios, René Müller, Keshava Murthy, Ippokratis Pandis, Lin Qiao, Vijayshankar Raman, Richard Sidle, Knut Stolze, and Sandor Szabo. Business analytics in (a) blink. *IEEE Data Eng. Bull.*, 35(1):9–14, 2012.

[BDD+10] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura M. Haas, Renée J. Miller, and Nesime Tatbul. SECRET: A model for analysis of the execution semantics of stream processing systems. *Proc. VLDB Endow.*, 3(1):232–243, 2010.

[BGB+18] Andreas Becher, Lekshmi B. G., David Broneske, Tobias Drewes, Bala Gurumurthy, Klaus Meyer-Wegener, Thilo Pionteck, Gunter Saake, Jürgen Teich, and Stefan Wildermann. Integration of FPGAs in database management systems: Challenges and opportunities. *Datenbank-Spektrum*, 18(3):145–156, 2018.

[BHS+14] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-accelerated database systems: Survey and open challenges. *Trans. Large Scale Data Knowl. Centered Syst.*, 15:1–35, 2014.

[Bit22] BittWare. Agilex FPGA card with QSFP-DD and MCIO. `https://www.bittware.com/files/IA-840F_datasheet.pdf`, 7 2022. (Accessed on 12/25/2022).

[BK90] Andrei Z. Broder and Anna R. Karlin. Multilevel adaptive hashing. In David S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 43–53. SIAM, 1990.

[Bla10] Vladimir Blagojevic. Infinispan eviction, batching updates and LIRS. `https://infinispan.org/blog/2010/03/30/infinispan-eviction-batching-updates`, 2010. (Accessed on 08/15/2022).

[BLP11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 37–48. ACM, 2011.

[BMK99] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 54–65. Morgan Kaufmann, 1999.

[Bro96] Stephen Brown. FPGA architectural research: a survey. *IEEE Design & Test of Computers*, 13(4):9–15, 1996.

[Bru09] Jake Brutlag. Speed matters for Google web search. `https://services.google.com/fh/files/blogs/google_delayexp.pdf`, 06 2009. (Accessed on 11/10/2022).

[BVR+12] Jonathan Bachrach, Huy Vo, Brian C. Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanovic. Chisel: constructing hardware in a Scala embedded language. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 1216–1225. ACM, 2012.

[CCP06] Deming Chen, Jason Cong, and Peichen Pan. FPGA design automation: A survey. *Found. Trends Electron. Des. Autom.*, 1(3), 2006.

[CCP+16] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, pages 7:1–7:13. IEEE Computer Society, 2016.

[CDL11] Alexandre Cornu, Steven Derrien, and Dominique Lavenier. HLS tools for FPGA: faster development with better performance. In Andreas Koch, Ram Krishnamurthy, John McAllister, Roger F. Woods, and Tarek A. El-Ghazawi, editors, *Reconfigurable Computing: Architectures, Tools and Applications - 7th International Symposium, ARC 2011, Belfast, UK, March 23-25, 2011. Proceedings*, volume 6578 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2011.

[CDN11] Surajit Chaudhuri, Umeshwar Dayal, and Vivek R. Narasayya. An overview of business intelligence technology. *Commun. ACM*, 54(8):88–98, 2011.

[CER18] CERN. Key facts and figures – CERN data centre. `https://information-technology.web.cern.ch/sites/default/files/CERNDataCentre$_{K}$eyInformation$_{0}$1June2018V1.pdf`, 06 2018. (Accessed on 11/10/2022).

[CG11] Alain Crolotte and Ahmad Ghazal. Introducing skew into the TPC-H benchmark. In Raghunath Othayoth Nambiar and Meikel Poess, editors, *Topics in Performance Evaluation, Measurement and Characterization - Third TPC Technology Conference, TPCTC 2011, Seattle, WA, USA, August 29-September 3, 2011, Revised Selected Papers*, volume 7144 of *Lecture Notes in Computer Science*, pages 137–145. Springer, 2011.

[Chi21a] Chisel development team. Chisel release 3.4.2. `https://github.com/chipsalliance/chisel3/tree/e84a934e753738cc7472e1f94e38523d8ec172ff`, 02 2021. (Accessed on 12/14/2022).

[Chi21b] Chisel development team. Chisel release 3.4.3. `https://github.com/chipsalliance/chisel3/tree/2554adfdae7933e7b0cf62ca71a6cb6b0c576f46`, 4 2021. (Accessed on 12/08/2021).

[Chi22] Chisel development team. Chisel release 3.5.0. `https://github.com/chipsalliance/chisel3/tree/v3.5.0`, 1 2022. (Accessed on 08/21/2022).

[Chu08] Pong P. Chu. *FPGA Prototyping by VHDL Examples*. John Wiley & Sons, Ltd, 2008.

[Clo21] Huawei Cloud. FPGA-accelerated ECSs. `https://support.huaweicloud.com/en-us/productdesc-ecs/en-us$_{t}$opic$_{0}$069206563.html`, 11 2021. (Accessed on 11/11/2022).

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[CO14] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In Vaughn Betz and George A. Constantinides, editors, *The 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA '14, Monterey, CA, USA - February 26 - 28, 2014*, pages 151–160. ACM, 2014.

[Cod69] E. F. Codd. Derivability, redundancy and consistency of relations stored in large data banks. *Research Report / RJ / IBM / San Jose, California*, RJ599, 1969.

[Com22] Scala Development Community. The Scala programming language. `https://scala-lang.org/`, 2022. (Accessed on 11/18/2022).

[Con12] EuroBen Contributors. Convey systems. `https://www.euroben.nl/reports/web13/convey.php`, 2012. (Accessed on 11/25/2022).

[Cor10] Intel Corporation. Stratix IV GX FPGA development board reference manual. https://www.intel.com/programmable/technical-pdfs/654380.pdf, 11 2010. (Accessed on 11/25/2022).

[Cor14] Intel Corporation. OpenCL™ 2.0 shared virtual memory overview. https://www.intel.com/content/www/us/en/developer/articles/technical/opencl-20-shared-virtual-memory-overview.html, 09 2014. (Accessed on 11/24/2022).

[Cor17] Intel Corporation. Fitter resource usage summary report. https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/mapIdTopics/mwh1465496451103.htm, 2017. (Accessed on 11/11/2022).

[Cor18a] Intel Corporation. Intel® Stratix® 10 GX/SX product table. https://cdrdv2.intel.com/v1/dl/getContent/652463, 2018. (Accessed on 11/17/2022).

[Cor18b] Oracle Corporation. In-memory column store architecture. https://docs.oracle.com/en/database/oracle/oracle-database/21/inmem/in-memory-column-store-architecture.html#GUID-EEA265EE-8FBA-4457-8C3F-315B9EEA2224, 2018. (Accessed on 11/13/2022).

[Cor19a] Intel Corporation. Intel FPGA programmable acceleration card D5005 data sheet. https://www.intel.com/content/www/us/en/programmable/documentation/cvl1520030638800.html, 11 2019. (Accessed on 11/11/2022).

[Cor19b] Intel Corporation. Intel® FPGA programmable acceleration card D5005 data sheet: DDR4 SDRAM. https://www.intel.com/content/www/us/en/docs/programmable/683568/current/ddr4-sdram.html, 11 2019. (Accessed on 11/11/2022).

[Cor20] Intel Corporation. Intel programmable acceleration card (PAC) with intel Arria 10 GX FPGA data sheet. https://www.intel.com/content/www/us/en/docs/programmable/683226/current/introduction-rush-creek.html, 10 2020. (Accessed on 12/25/2022).

[Cor21a] Intel Corporation. Intel® Optane™ persistent memory (Pmem). https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html, 2021. (Accessed on 11/19/2022).

[Cor21b] Intel Corporation. Intel® Stratix® 10 DX FPGA overview. https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10/dx.html, 2021. (Accessed on 11/11/2022).

[Cor21c] Intel Corporation. Supported operational modes in Intel® Stratix® 10 devices. https://www.intel.com/content/www/us/en/docs/programmable/683832/21-2/supported-operational-modes-in-devices.html, 08 2021. (Accessed on 11/17/2022).

[Cor22a] Intel Corporation. ALM. https://www.intel.com/content/www/us/en/docs/programmable/683699/current/alm.html, 03 2022. (Accessed on 11/17/2022).

[Cor22b] Intel Corporation. Intel® Stratix® 10 embedded memory configurations. https://www.intel.com/content/www/us/en/docs/programmable/683423/21-4/embedded-memory-configurations.html, 04 2022. (Accessed on 11/17/2022).

[Cor22c] Intel Corporation. Intel® Stratix® 10 embedded memory features. https://www.intel.com/content/www/us/en/docs/programmable/683423/21-4/embedded-memory-features.html, 04 2022. (Accessed on 11/17/2022).

[Cor22d] Intel Corporation. Logic elements. https://www.intel.com/content/www/us/en/docs/programmable/683777/current/logic-elements.html, 09 2022. (Accessed on 11/16/2022).

[Cou22] Transaction Processing Performance Council. TPC-H homepage. https://www.tpc.org/tpch/, 2022. (Accessed on 11/30/2022).

[CP16] Ren Chen and Viktor K. Prasanna. Accelerating equi-join on a CPU-FPGA heterogeneous platform. In *24th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016, Washington, DC, USA, May 1-3, 2016*, pages 212–219. IEEE Computer Society, 2016.

[CR07] John Cieslewicz and Kenneth A. Ross. Adaptive aggregation on chip multiprocessors. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 339–350. ACM, 2007.

[CSPJ03] Seonil Choi, Ronald Scrofano, Viktor K. Prasanna, and Ju-wook Jang. Energy-efficient signal processing using FPGAs. In Steve Trimberger and Russell Tessier, editors, *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2003, Monterey, CA, USA, February 23-25, 2003*, pages 225–234. ACM, 2003.

[Dat03] C. J. Date. *An introduction to database systems (8. ed.)*. Addison-Wesley-Longman, 2003.

[DBRU20] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. Quantifying TPC-H choke points and their optimizations. *Proc. VLDB Endow.*, 13(8):1206–1220, 2020.

[Deu17] Deutsche Bahn Fernverkehr AG. WIFI on ICE. https://data.deutschebahn.com/dataset/wifi-on-ice.html, 12 2017. (Accessed on 12/30/2022).

[DG20] Thanh Do and Goetz Graefe. Sort-based grouping and aggregation. *CoRR*, abs/2010.00152, 2020.

[DGN23] Thanh Do, Goetz Graefe, and Jeffrey Naughton. Efficient sorting, duplicate removal, grouping, and aggregation. *ACM Trans. Database Syst.*, 47(4), 01 2023.

[DKM+12] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. CPU DB: recording microprocessor history. *Commun. ACM*, 55(4):55–63, 2012.

[DSB19] Manish Deo, Jeffrey Schulz, and Lance Brown. Intel® Stratix® 10 MX devices with samsung* HBM2 solve the memory bandwidth challenge. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01264-stratix10mx-devices-solve-memory-bandwidth-challenge.pdf, 2019. (Accessed on 11/11/2022).

[DZS13] Luka Daoud, Dawid Zydek, and Henry Selvaraj. A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing. In Jerzy Swiatek, Adam Grzech, Pawel Swiatek, and Jakub M. Tomczak, editors, *Advances in Systems Science - Proceedings of the International Conference on Systems Science 2013, ICSS 2013, Wroclaw, Poland, September 10-12, 2013*, volume 240 of *Advances in Intelligent Systems and Computing*, pages 483–492. Springer, 2013.

[DZT12] Christopher Dennl, Daniel Ziener, and Jürgen Teich. On-the-fly composition of FPGA-Based SQL query accelerators using a partially reconfigurable module library. In *2012 IEEE 20th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2012, 29 April - 1 May 2012, Toronto, Ontario, Canada*, pages 45–52. IEEE Computer Society, 2012.

[DZT13] Christopher Dennl, Daniel Ziener, and Jürgen Teich. Acceleration of SQL restrictions and aggregations through FPGA-Based dynamic partial reconfiguration. In *21st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2013, Seattle, WA, USA, April 28-30, 2013*, pages 25–28. IEEE Computer Society, 2013.

[EH84] Wolfgang Effelsberg and Theo Härder. Principles of database buffer management. *ACM Trans. Database Syst.*, 9(4):560–595, 1984.

[Epr18] Eproav. Cubix Xpander Fiber 8. `https://eproav.com/cubix-xpander-fiber-8-xf8-4u-rp`, 2018. (Accessed on 11/25/2022).

[Fae17] Christian Faerber. Acceleration of Cherenkov angle reconstruction with the new Intel Xeon/FPGA compute platform for the particle identification in the LHCb upgrade. In *Journal of Physics: Conference Series*, volume 898, page 032044. IOP Publishing, 2017.

[FCP+11] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, 2011.

[FdSJ19] Gabriel Fornari and Valdivino Alexandre de Santiago Júnior. Dynamically reconfigurable systems: A systematic literature review. *J. Intell. Robotic Syst.*, 95(3-4):829–849, 2019.

[FfI22] Technische Universität München Fakultät für Informatik, Lehrstuhl III: Datenbanksysteme. HyPer: Hybrid OLTP & OLAP high-performance database system. `https://hyper-db.de/`, 2022. (Accessed on 10/18/2022).

[FMH+20] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. In-memory database acceleration on FPGAs: a survey. *VLDB J.*, 29(1):33–59, 2020.

[FML+12] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA database – An architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[FMM12] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *FPGA Architectures: An Overview*, pages 7–48. Springer New York, New York, NY, 2012.

[Fra14] Phil Francisco. IBM PureData system for analytics architecture. *IBM Redbooks*, 2014:1–16, 2014.

[GAT+15] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. Amazon Redshift and the case for simpler data warehouses. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1917–1923. ACM, 2015.

[GCB+97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[Gil04] William F. Gilreath. Hash sort: A linear time complexity multiple-dimensional sort algorithm. *CoRR*, cs.DS/0408040, 2004.

[GK19] Shubham Gandhare and B. Karthikeyan. Survey on FPGA architecture and recent applications. In *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, pages 1–4, 2019.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[Gra06] Goetz Graefe. Implementing sorting in database systems. *ACM Comput. Surv.*, 38(3):10, 2006.

[Gro16] Stanford University's VLSI Research Group. CPU DB: - looking at 40 years of processor improvements | a complete database of processors for researchers and hobbyists alike. `http://cpudb.stanford.edu/download`, 06 2016. (Accessed on 11/10/2022).

[GS92] Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, 1992.

[GS21] Prajith Ramakrishnan Geethakumari and Ioannis Sourdis. A specialized memory hierarchy for stream aggregation. In *31st International Conference on Field-Programmable Logic and Applications, FPL 2021, Dresden, Germany, August 30 - Sept. 3, 2021*, pages 204–210. IEEE, 2021.

[GSDB12] Ravindra Guravannavar, S. Sudarshan, Ajit A. Diwan, and Ch. Sobhan Babu. Which sort orders are interesting? *VLDB J.*, 21(1):145–165, 2012.

[GUW09] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[HAMS08] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 981–992. ACM, 2008.

[HANT15] Robert J. Halstead, Ildar Absalyamov, Walid A. Najjar, and Vassilis J. Tsotras. FPGA-based multithreading for in-memory hash joins. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[HCGL15] Kenneth Hill, Stefan Craciun, Alan D. George, and Herman Lam. Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA. In *26th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2015, Toronto, ON, Canada, July 27-29, 2015*, pages 189–193. IEEE Computer Society, 2015.

[HH21] Sarah Harris and David Harris. *Digital Design and Computer Architecture: RISC-V Edition*. Morgan Kaufmann, 2021.

[HNM02] Sven Helmer, Thomas Neumann, and Guido Moerkotte. Early grouping gets the skew. *Technical reports*, 2, 2002.

[HNM03] Sven Helmer, Thomas Neumann, and Guido Moerkotte. Estimating the output cardinality of partial preaggregation with a measure of clusteredness. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 656–667. Morgan Kaufmann, 2003.

[HP17] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach, 6th Edition*. Morgan Kaufmann, 2017.

[HS76] Ellis Horowitz and Sartaj Sahni. *Fundamentals of data structures*. Computer Science Press, 1976.

[HSM+13] Robert J. Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh W. Asaad, and Balakrishna Iyer. Accelerating join operation for relational databases with FPGAs. In *21st IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2013, Seattle, WA, USA, April 28-30, 2013*, pages 17–20. IEEE Computer Society, 2013.

[HWFH08] Omar Hammami, Zhoukun Wang, Virginie Fresse, and Dominique Houzet. A case study: Quantitative evaluation of C-based high-level synthesis systems. *EURASIP J. Embed. Syst.*, 2008, 2008.

[IEE18] IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group. IEEE standard for SystemVerilog–unified hardware design, specification, and verification language. *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pages 1–1315, 2018.

[IEE19] IEEE Design Automation Standards Committee. IEEE standard for VHDL language reference manual. *IEEE Std 1076-2019*, pages 1–673, 2019.

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[Inc18] Terasic Inc. DE5-Net FPGA development kit: User manual. `https://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=526&FID=49e768f77939a23529c519ae385dde6a`, 06 2018. (Accessed on 11/25/2022).

[Inc19] Terasic Inc. Intel® Stratix® 10 DX FPGA development kit. `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=248&No=1258&PartNo=1#contents`, 2019. (Accessed on 11/19/2022).

[Inc20] Khronos® Group Inc. OpenCL overview. `https://www.khronos.org/opencl/`, 2020. (Accessed on 11/11/2022).

[Inc22] Amazon Inc. Columnar storage - Amazon Redshift. `https://docs.aws.amazon.com/redshift/latest/dg/c_columnar_storage_disk_mem_mgmnt.html`, 2022. (Accessed on 10/18/2022).

[Ins22] Hasso Plattner Institut. Hyrise: The open-source in-memory research DBMS. https://hpi.de/plattner/projects/hyrise.html, 2022. (Accessed on 11/11/2022).

[Int13] Intel Corporation. Implementing FPGA design with the OpenCL standard. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01173-opencl.pdf, 11 2013. (Accessed on 11/11/2022).

[Int17] Intel Corporation. Intel® Xeon® Platinum 8180 processor. https://www.intel.com/content/www/us/en/products/sku/120496/intel-xeon-platinum-8180-processor-38-5m-cache-2-50-ghz/specifications.html, 2017. (Accessed on 12/27/2022).

[Int18a] Intel Corporation. Intel® FPGA SDK for OpenCL™ standard edition getting started guide. https://www.intel.com/content/www/us/en/docs/programmable/683678/18-1/standard-edition-getting-started-guide.html, 09 2018. (Accessed on 11/11/2022).

[Int18b] Intel Corporation. Swarm64's S64DA* delivers fast, innovative data analytics solution with Intel® FPGAs. https://www.intel.com/content/dam/www/public/us/en/documents/partner-alliance/solution-briefs/swarm64-fpga-solution-brief.pdf, 2018. (Accessed on 11/21/2022).

[Int19] Intel Corporation. Intel® FPGA SDK for OpenCL™ pro edition: Programming guide. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl$_{\text{p}}$rogramming$_{\text{g}}$uide.pdf, 12 2019. (Accessed on 02/23/2020).

[Int22] Intel Corporation. FPGA architecture overview. https://www.intel.com/content/www/us/en/develop/documentation/oneapi-fpga-optimization-guide/top/introduction-to-fpga-design-concepts/fpga-architecture-overview.html, 09 2022. (Accessed on 11/16/2022).

[Int23] Intel Corporation. Intel® FPGA add-on for oneAPI base toolkit: Program FPGAs faster. https://www.intel.com/content/www/us/en/developer/tools/oneapi/fpga.html#gs.mvt6sd, 2023. (Accessed on 01/17/2023).

[Jen09] Bob Jenkins. Order-preserving minimal perfect hashing. in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., April 2009. Available from: https://www.nist.gov/dads/HTML/orderPreservMinPerfectHash.html (accessed 25 December 2022).

[JMS+08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stanley B. Zdonik. Towards a streaming SQL standard. *Proc. VLDB Endow.*, 1(2):1379–1390, 2008.

[JZ02] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In Richard R. Muntz, Margaret Martonosi, and Edmundo de Souza e Silva, editors, *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2002, June 15-19, 2002, Marina Del Rey, California, USA*, pages 31–42. ACM, 2002.

[KA16] Kaan Kara and Gustavo Alonso. Fast and robust hashing for database operators. In Paolo Ienne, Walid A. Najjar, Jason Helge Anderson, Philip Brisk, and Walter Stechele, editors, *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*, pages 1–4. IEEE, 2016.

[KBB+15] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A modern, open-source SQL engine for hadoop. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[KGA17] Kaan Kara, Jana Giceva, and Gustavo Alonso. Fpga-based data partitioning. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 433–445. ACM, 2017.

[KHD+20] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. High bandwidth memory on FPGAs: A data analytics perspective. In Nele Mentens, Leonel Sousa, Pedro Trancoso, Miquel Pericàs, and Ioannis Sourdis, editors, *30th International Conference on Field-Programmable Logic and Applications, FPL 2020, Gothenburg, Sweden, August 31 - September 4, 2020*, pages 1–8. IEEE, 2020.

[KKK+22] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. Farview: Disaggregated memory with operator off-loading for database engines. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022.

[KM10] Adam Kirsch and Michael Mitzenmacher. The power of one move: Hashing schemes for hardware. *IEEE/ACM Trans. Netw.*, 18(6):1752–1765, 2010.

[KMK+19] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Howard Katz, Jonathan Bachrach, and Krste Asanovic. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. *IEEE Micro*, 39(3):56–65, 2019.

[Knu73] Donald E. Knuth. *The art of computer programming, Vol. 3: sorting and searching*. Addison-Wesley, 1973.

[KS16] Christoforos Kachris and Dimitrios Soudris. A survey on reconfigurable accelerators for cloud computing. In Paolo Ienne, Walid A. Najjar, Jason Helge Anderson, Philip Brisk, and Walter Stechele, editors, *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*, pages 1–10. IEEE, 2016.

[KSC+09] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proc. VLDB Endow.*, 2(2):1378–1389, 2009.

[KT11] Dirk Koch and Jim Tørresen. FPGASort: a high performance sorting architecture exploiting run-time reconfiguration on FPGAs for large problem sorting. In John Wawrzynek and Katherine Compton, editors, *Proceedings of the ACM/SIGDA 19th International Symposium on Field Programmable Gate Arrays, FPGA 2011, Monterey, California, USA, February 27, March 1, 2011*, pages 45–54. ACM, 2011.

[KTB+12] Dirk Koch, Jim Tørresen, Christian Beckhoff, Daniel Ziener, Christopher Dennl, Volker Breuer, Jürgen Teich, Michael Feilen, and Walter Stechele. Partial reconfiguration on FPGAs in practice - tools and applications. In Gero Mühl, Jan Richling, and Andreas Herkersdorf, editors, *ARCS 2012 Workshops, 28. Februar - 2. März 2012, München, Germany*, volume P-200 of *LNI*, pages 297–319. GI, 2012.

[Lar02] Per-Åke Larson. Data reduction by partial preaggregation. In Rakesh Agrawal and Klaus R. Dittrich, editors, *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 706–715. IEEE Computer Society, 2002.

[Lim08] S. S. Limaye. *VHDL: A Design Oriented Approach*. McGraw-Hill Publ., 2008.

[LMM+22] Robert Lasch, Mehdi Moghaddamfar, Norman May, Süleyman Sirri Demirsoy, Christian Färber, and Kai-Uwe Sattler. Bandwidth-optimal relational joins on FPGAs. In Julia Stoyanovich, Jens Teubner, Paolo Guagliardo, Milos Nikolic, Andreas Pieris, Jan Mühlig, Fatma Özcan, Sebastian Schelter, H. V. Jagadish, and Meihui Zhang, editors, *Proceedings of the 25th International Conference on Extending Database Technology, EDBT 2022, Edinburgh, UK, March 29 - April 1, 2022*, pages 1:27–1:39. OpenProceedings.org, 2022.

[MA18] Noor Mohammedali and Michael Opoku Agyeman. A study of reconfigurable accelerators for cloud computing. In *Proceedings of the 2nd International Symposium on Computer Science and Intelligent Control*, ISCSIC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[MCK17] Susumu Mashimo, Thiem Van Chu, and Kenji Kise. High-performance hardware merge sorter. In *25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017, Napa, CA, USA, April 30 - May 2, 2017*, pages 1–8. IEEE Computer Society, 2017.

[MFL+21] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, Norman May, and Akash Kumar. Resource-efficient database query processing on FPGAs. In Danica Porobic and Spyros Blanas, editors, *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, pages 4:1–4:8. ACM, 2021.

[MFLM20] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, and Norman May. Comparative analysis of OpenCL and RTL for sort-merge primitives on FPGA. In Danica Porobic and Thomas Neumann, editors, *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*, pages 11:1–11:7. ACM, 2020.

[MFM+22] Mehdi Moghaddamfar, Christian Färber, Norman May, Wolfgang Lehner, and Akash Kumar. FPGA-based database query processing on arbitrarily wide tables. In *30th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2022, New York City, NY, USA, May 15-18, 2022*, page 1. IEEE, 2022.

[Mic22] Microsoft Learn contributors. Columnstore indexes: Overview - sql server. `https://learn.microsoft.com/en-us/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-ver16`, 11 2022. (Accessed on 01/23/2023).

[Miy16] Takefumi Miyoshi. Synthesijer. `http://synthesijer.github.io/web/`, 2016. (Accessed on 11/18/2022).

[MK19] Onisimo Mutanga and Lalit Kumar. Google earth engine applications. *Remote. Sens.*, 11(5):591, 2019.

[MMF+23] Mehdi Moghaddamfar, Norman May, Christian Färber, Wolfgang Lehner, and Akash Kumar. A study of early aggregation in database query processing on FPGAs. In Paolo Ienne and Zhiru Zhang, editors, *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA 2023, Monterey, CA, USA, February 12-14, 2023*, pages 55–65. ACM, 2023.

[MNC09] Rui Marcelino, Horácio C. Neto, and João M. P. Cardoso. Unbalanced FIFO sorting for FPGA-based systems. In *16th IEEE International Conference on Electronics, Circuits, and Systems, ICECS 2009, Yasmine Hammamet, Tunisia, 13-19 December, 2009*, pages 431–434. IEEE, 2009.

[Mon21] MonetDB B.V. MonetDB. `https://www.monetdb.org/`, 2021. (Accessed on 12/14/2022).

[Moo98] Gordon E. Moore. Cramming more components onto integrated circuits. *Proc. IEEE*, 86(1):82–85, 1998.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[MSC+16] Yufei Ma, Naveen Suda, Yu Cao, Jae-sun Seo, and Sarma B. K. Vrudhula. Scalable and modularized RTL compilation of convolutional neural networks onto FPGA. In Paolo Ienne, Walid A. Najjar, Jason Helge Anderson, Philip Brisk, and Walter Stechele, editors, *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*, pages 1–8. IEEE, 2016.

[MSL+15] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. Cache-efficient aggregation: Hashing is sorting. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1123–1136. ACM, 2015.

[MTA09a] René Müller, Jens Teubner, and Gustavo Alonso. Data processing on FPGAs. *Proc. VLDB Endow.*, 2(1):910–921, 2009.

[MTA09b] René Müller, Jens Teubner, and Gustavo Alonso. Streams on Wires - A query compiler for FPGAs. *Proc. VLDB Endow.*, 2(1):229–240, 2009.

[MTA10] René Müller, Jens Teubner, and Gustavo Alonso. Glacier: a query-to-hardware compiler. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 1159–1162. ACM, 2010.

[MTA12] René Müller, Jens Teubner, and Gustavo Alonso. Sorting networks on FPGAs. *VLDB J.*, 21(1):1–23, 2012.

[Mul19] David Mulnix. Intel® Xeon® processor scalable family technical overview. `https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html`, 06 2019. (Accessed on 11/19/2022).

[NMG+15] Katayoun Neshatpour, Maria Malik, Mohammad Ali Ghodrat, Avesta Sasan, and Houman Homayoun. Energy-efficient acceleration of big data analytics applications using FPGAs. In *2015 IEEE International Conference on Big Data (IEEE BigData 2015), Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 115–123. IEEE Computer Society, 2015.

[NS22] Chris Nyberg and Mehul Shah. Sort benchmark home page. `http://sortbenchmark.org/`, 2022. (Accessed on 12/27/2022).

[NSJ13] Mohammedreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. Flexible query processor on FPGAs. *Proc. VLDB Endow.*, 6(12):1310–1313, 2013.

[NSJ15] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. Configurable hardware-based streaming architecture using online programmable-blocks. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 819–830. IEEE Computer Society, 2015.

[NSP+16] Razvan Nane, Vlad Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Dean Brown, Fabrizio Ferrandi, Jason Helge Anderson, and Koen Bertels. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 35(10):1591–1604, 2016.

[OLG+05] John D. Owens, David Luebke, Naga K. Govindaraju, Mark J. Harris, Jens H. Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In Yiorgos Chrysanthou and Marcus A. Magnor, editors, *26th Annual Conference of the European Association for Computer Graphics, Eurographics 2005 - State of the Art Reports, Dublin, Ireland, August 29 - September 2, 2005*, pages 21–51. Eurographics Association, 2005.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 297–306. ACM Press, 1993.

[ORK+15] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2(11):1–4, 2015.

[OSC+11] Neal Oliver, Rahul R. Sharma, Stephen Chang, Bhushan Chitlur, Elkin Garcia, Joseph Grecco, Aaron Grier, Nelson Ijih, Yaping Liu, Pratik Marolia, Henry Mitchel, Suchit Subhaschandra, Arthur Sheiman, Tim Whisonant, and Prabhat Gupta. A reconfigurable computing system based on a cache-coherent fabric. In Peter M. Athanas, Jürgen Becker, and René Cumplido, editors, *2011 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2011, Cancun, Mexico, November 30 - December 2, 2011*, pages 80–85. IEEE Computer Society, 2011.

[OSKA17] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid CPU-FPGA databases. In *25th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2017, Napa, CA, USA, April 30 - May 2, 2017*, pages 211–218. IEEE Computer Society, 2017.

[Pan92] Sethuraman Panchanathan. Universal architecture for matrix transposition. *IEEE Proceedings: Computers and Digital Techniques*, 139(5):387–392, January 1992.

[PCC+15] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*, 35(3):10–22, 2015.

[Pet19] Alex Petrov. *Database Internals: A Deep Dive into How Distributed Data Systems Work*. " O'Reilly Media, Inc.", 2019.

[PHL18] Johns Paul, Bingsheng He, and Chiew Tong Lau. Query processing on OpenCL-Based FPGAs: Challenges and opportunities. In *24th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2018, Singapore, December 11-13, 2018*, pages 937–945. IEEE, 2018.

[PRE21] SAP PRESS. What is SAP HANA? a guide to in-memory computing with SAP | SAP PRESS. `https://learning.sap-press.com/sap-hana`, 2021. (Accessed on 11/24/2022).

[Red11] Martin Reddy. Chapter 7 - performance. In Martin Reddy, editor, *API Design for C++*, pages 209–240. Morgan Kaufmann, Boston, 2011.

[Rei15] John H. Reif. Forward radix sort. `https://users.cs.duke.edu/~reif/courses/alglectures/littman.lectures/lect05/node37.html`, 2015. (Accessed on 12/25/2022).

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[RK13] Margy Ross and Ralph Kimball. *The data warehouse toolkit: the definitive guide to dimensional modeling*. John Wiley & Sons, 2013.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-Store: A column-oriented DBMS. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564. ACM, 2005.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, pages 23–34. ACM, 1979.

[Sak14] Sherif Sakr. Cloud-hosted databases: technologies, challenges and opportunities. *Clust. Comput.*, 17(2):487–502, 2014.

[SAS+16] Behzad Salami, Oriol Arcas-Abella, Nehir Sönmez, Osman S. Unsal, and Adrián Cristal Kestelman. Accelerating hash-based query processing operations on FPGAs by a hash table caching technique. In Carlos Jaime Barrios Hernández, Isidoro Gitler, and Jaime Klapp, editors, *High Performance Computing - Third Latin American Conference, CARLA 2016, Mexico City, Mexico, August 29 - September 2, 2016, Revised Selected Papers*, volume 697 of *Communications in Computer and Information Science*, pages 131–145, 2016.

[SB09] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*. oreilly, 2009.

[SBJS15] Jeffrey Stuecheli, Bart Blaner, C. R. Johns, and M. S. Siegel. CAPI: A coherent accelerator processor interface. *IBM J. Res. Dev.*, 59(1), 2015.

[Sch09] Nicole Schweikardt. One-pass algorithm. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1948–1949. Springer US, 2009.

[Sch22] Thomas Schulte. Need PCI Express 5.0 for your next FPGA design? check out Intel® Agilex™ I-series and M-series FPGAs. `https://community.intel.com/t5/Blogs/Products-and-Solutions/FPGA/Need-PCI-Express-5-0-for-your-next-FPGA-design-Check-out-Intel/post/1385046`, 05 2022. (Accessed on 11/24/2022).

[SCPC15] Ajitesh Srivastava, Ren Chen, Viktor K. Prasanna, and Charalampos Chelmis. A hybrid design for high performance large-scale sorting on FPGA. In Michael Hübner, Maya B. Gokhale, and René Cumplido, editors, *International Conference on ReConFigurable Computing and FPGAs, ReConFig 2015, Riviera Maya, Mexico, December 7-9, 2015*, pages 1–6. IEEE, 2015.

[SE17] SAP SE. Columnar and row-based data storage. `https://help.sap.com/docs/SAP_HANA_PLATFORM/6b94445c94ae495c83a19646e7c3fd56/bd2e9b88bb571014b5b7a628fca2a132.html?version=2.0.01`, 2017. (Accessed on 11/11/2022).

[SE22] SAP SE. What is SAP HANA? `https://www.sap.com/uk/products/technology-platform/hana/what-is-sap-hana.html`, 2022. (Accessed on 11/11/2022).

[SFJ+19] Rym Skhiri, Virginie Fresse, Jean-Paul Jamont, Benoît Suffran, and Jihene Malek. From FPGA to support cloud to cloud of FPGA: State of the art. *Int. J. Reconfigurable Comput.*, 2019:8085461:1–8085461:17, 2019.

[Sie21] Siemens AG. C++/systemc synthesis. `https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis/hls/c-cplus/`, 02 2021. (Accessed on 01/17/2023).

[SIOA17] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 403–415. ACM, 2017.

[SJT+12] Mohammad Sadoghi, Rija Javed, Naif Tarafdar, Harsh Singh, Rohan Palaniappan, and Hans-Arno Jacobsen. Multi-query stream processing on FPGAs. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 1229–1232. IEEE Computer Society, 2012.

[Ski08] Steven Skiena. *The Algorithm Design Manual, Second Edition*. Springer, 2008.

[SKLG16] Wei Song, Dirk Koch, Mikel Luján, and Jim D. Garside. Parallel hardware merge sorter. In *24th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016, Washington, DC, USA, May 1-3, 2016*, pages 95–102. IEEE Computer Society, 2016.

[SMT+12] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh W. Asaad. Database analytics acceleration using FPGAs. In Pen-Chung Yew, Sangyeun Cho, Luiz DeRose, and David J. Lilja, editors, *International Conference on Parallel Architectures and Compilation Techniques, PACT '12, Minneapolis, MN, USA - September 19 - 23, 2012*, pages 411–420. ACM, 2012.

[SMT+14] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Bernard Brezzo, Sameh W. Asaad, and Donna Dillenberger. Database analytics: A reconfigurable-computing approach. *IEEE Micro*, 34(1):19–29, 2014.

[STM+13] Bharat Sukhwani, Mathew Thoennes, Hong Min, Parijat Dube, Bernard Brezzo, Sameh W. Asaad, and Donna Dillenberger. Large payload streaming database sort and projection on FPGAs. In *25th International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2013, Porto de Galinhas, Pernambuco, Brazil, October 23-26, 2013*, pages 25–32. IEEE Computer Society, 2013.

[STM+15] Bharat Sukhwani, Mathew Thoennes, Hong Min, Parijat Dube, Bernard Brezzo, Sameh W. Asaad, and Donna Dillenberger. A hardware/software approach for database query acceleration with FPGAs. *Int. J. Parallel Program.*, 43(6):1129–1159, 2015.

[Syn21] SynaptiCAD. Testbencher pro and reactive test bench – golden reference models. `https://www.syncad.com/web_manual_testbencher/test_bench_generator_main_index.html?golden_reference_models.htm`, 2021. (Accessed on 11/18/2022).

[Tea19] MonetDB Team. Memory footprint | MonetDB docs. `https://www.monetdb.org/documentation-Sep2022/admin-guide/system-resources/memory-footprint/`, 2019. (Accessed on 11/24/2022).

[Tea22a] MonetDB Team. Products | MonetDB solutions. `https://www.monetdbsolutions.com/products`, 2022. (Accessed on 11/24/2022).

[Tea22b] PostgreSQL Development Team. PostgreSQL: Database file layout. `https://www.postgresql.org/docs/current/storage-file-layout.html`, 2022. (Accessed on 11/24/2022).

[The08] The Khronos Group Inc. Khronos launches heterogeneous computing initiative. `https://khr.io/b1`, 6 2008. (Accessed on 11/11/2022).

[THSW15] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. General incremental sliding-window aggregation. *Proc. VLDB Endow.*, 8(7):702–713, 2015.

[TMA11] Jens Teubner, René Müller, and Gustavo Alonso. Frequent item computation on a chip. *IEEE Trans. Knowl. Data Eng.*, 23(8):1169–1181, 2011.

[Tri15] Stephen Trimberger. Three ages of FPGAs: A retrospective on the first thirty years of FPGA technology. *Proc. IEEE*, 103(3):318–331, 2015.

[TSJ+10] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. Hive - a petabyte scale data warehouse using Hadoop. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005. IEEE Computer Society, 2010.

[TW13] Jens Teubner and Louis Woods. *Data Processing on FPGAs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[TWN12] Jens Teubner, Louis Woods, and Chongling Nie. Skeleton automata for FPGAs: reconfiguring without reconstructing. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 229–240. ACM, 2012.

[UIO15] Takanori Ueda, Megumi Ito, and Moriyoshi Ohara. A dynamically reconfigurable equi-joiner on FPGA. *IBM Tehnical Report RT0969*, 2015.

[Vah10] Frank Vahid. *Digital Design with RTL Design, Verilog and VHDL*. Wiley Publishing, 2nd edition, 2010.

[VF18] Kizheppatt Vipin and Suhaib A. Fahmy. FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications. *ACM Comput. Surv.*, 51(4):72:1–72:39, 2018.

[VMB19] Pirmin Vogel, Andrea Marongiu, and Luca Benini. Exploring shared virtual memory for FPGA accelerators with a configurable IOMMU. *IEEE Trans. Computers*, 68(4):510–525, 2019.

[Wak21] John F. Wakerly. *Digital Design: Principles and Practices, 5th edition*. Pearson, 2021.

[WFS+19] Satoru Watanabe, Kazuhisa Fujimoto, Yuji Saeki, Yoshifumi Fujikawa, and Hiroshi Yoshino. Column-oriented database acceleration using FPGAs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*, pages 686–697. IEEE, 2019.

[WIA14] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex - an intelligent storage engine with support for advanced SQL off-loading. *Proc. VLDB Endow.*, 7(11):963–974, 2014.

[Wik22a] Wikipedia contributors. Bijection. `https://en.wikipedia.org/w/index.php?title=Bijection&oldid=1128043142`, 2022. [Online; accessed 25-December-2022].

[Wik22b] Wikipedia contributors. Cloud database. `https://en.wikipedia.org/w/index.php?title=Cloud_database&oldid=1120962476`, 2022. [Online; accessed 11-November-2022].

[Wik22c] Wikipedia contributors. In-memory database. `https://en.wikipedia.org/w/index.php?title=In-memory_database&oldid=1123276502`, 2022. [Online; accessed 24-November-2022].

[Wik22d] Wikipedia contributors. Interleaved memory. `https://en.wikipedia.org/w/index.php?title=Interleaved_memory&oldid=1119679344`, 2022. [Online; accessed 19-November-2022].

[Wik22e] Wikipedia contributors. Lirs caching algorithm. `https://en.wikipedia.org/w/index.php?title=LIRS_caching_algorithm&oldid=1119087145`, 2022. [Online; accessed 29-December-2022].

[Wik22f] Wikipedia contributors. List of social platforms with at least 100 million active users. `https://en.wikipedia.org/w/index.php?title=List_of_social_platforms_with_at_least_100_million_active_users&oldid=1120940377`, 2022. [Online; accessed 10-November-2022].

[Wik22g] Wikipedia contributors. Pareto distribution. https://en.wikipedia.org/w/index.php?title=Pareto$_{d}$istribution&oldid=1125821066, 2022. [Online; accessed 29-December-2022].

[Wik22h] Wikipedia contributors. Strategy pattern. https://en.wikipedia.org/w/index.php?title=Strategy$_{p}$attern&oldid=1106787350, 2022. [Online; accessed 29-December-2022].

[Wik22i] Wikipedia contributors. Transpose. https://en.wikipedia.org/w/index.php?title=Transpose&oldid=1126080877, 2022. [Online; accessed 26-December-2022].

[Wik23] Wikipedia contributors. Unix time. https://en.wikipedia.org/w/index.php?title=Unix$_{t}$ime&oldid=1131996869, 2023. [Online; accessed 7-January-2023].

[WPC+16] Ze-ke Wang, Johns Paul, Hui Yan Cheah, Bingsheng He, and Wei Zhang. Relational query processing on OpenCL-based FPGAs. In Paolo Ienne, Walid A. Najjar, Jason Helge Anderson, Philip Brisk, and Walter Stechele, editors, *26th International Conference on Field Programmable Logic and Applications, FPL 2016, Lausanne, Switzerland, August 29 - September 2, 2016*, pages 1–10. IEEE, 2016.

[WTA13] Louis Woods, Jens Teubner, and Gustavo Alonso. Less watts, more performance: An intelligent storage engine for data appliances. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1073–1076. ACM, 2013.

[WZTD19] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Day. Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance. https://www.xilinx.com/support/documentation/white$_{p}$apers/wp485-hbm.pdf, 07 2019. (Accessed on 11/11/2022).

[Xil11a] Xilinx, Inc. ML505/ML506/ML507 evaluation platform: User guide. https://docs.xilinx.com/v/u/en-US/ug347, 05 2011. (Accessed on 11/25/2022).

[Xil11b] Xilinx, Inc. Virtex-II Pro and Virtex-II Pro X platform FPGAs: Complete data sheet. https://docs.xilinx.com/v/u/en-US/ds083, 06 2011. (Accessed on 11/25/2022).

[Xil19] Xilinx, Inc. Swarm64 PostgreSQL accelerator: The easiest path to faster performance & scalability. https://www.xilinx.com/publications/solution-briefs/swarm64$_{s}$olutionbrief.pdf, 2019. (Accessed on 11/21/2022).

[Xil21] Xilinx, Inc. Vivado design suite user guide: High-level synthesis. https://www.xilinx.com/content/dam/xilinx/support/documents/sw$_{m}$anuals/xilinx2020$_{2}$/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirectives, 05 2021. (Accessed on 11/11/2022).

[Xil22a] Xilinx, Inc. Vitis high-level synthesis user guide. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Introduction, 2022. (Accessed on 01/17/2023).

[Xil22b] Xilinx, Inc. Vitis libraries. https://docs.xilinx.com/r/en-US/Vitis$_{L}$ibraries, 2022. (Accessed on 11/21/2022).

[Xil22c] Xilinx, Inc. Vitis_Libraries/database at main on Xilinx/Vitis_Libraries GitHub. https://github.com/Xilinx/Vitis$_{L}$ibraries/tree/main/database, 2022. (Accessed on 11/21/2022).

[YKO+14] Masato Yoshimi, Ryu Kudo, Yasin Oge, Yuta Terada, Hidetsugu Irie, and Tsutomu Yoshinaga. Accelerating OLAP workload on interconnected FPGAs with flash storage. In *Second International Symposium on Computing and Networking, CANDAR 2014, Shizuoka, Japan, December 10-12, 2014*, pages 440–446. IEEE Computer Society, 2014.

[YL94] Weipeng P. Yan and Per-Åke Larson. Data reduction through early grouping. In John E. Botsford, Ann Gawman, W. Morven Gentleman, Evelyn Kidd, Kelly A. Lyons, Jacob Slonim, and J. Howard Johnson, editors, *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research, October 31 - November 3, 1994, Toronto, Ontario, Canada*, page 74. IBM, 1994.

[Zab11] Wojciech M. Zabolotny. Dual port memory based heapsort implementation for FPGA. In *Symposium on Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments (WILGA)*, 2011.

[ZZZ00] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In Andrew Wolfe and Michael S. Schlansker, editors, *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 33, Monterey, California, USA, December 10-13, 2000*, pages 32–41. ACM/IEEE Computer Society, 2000.

LIST OF FIGURES

1.1 The frequency of Intel CPUs as a function of their release year. The data in this graph is sourced from the Stanford CPU DB platform [DKM+12, Gro16]. 17

2.1 Tables R and S and examples of database operators applied to them: (a) Table R, (b) Table S, (c) Projection operator, (d) Filter/Selection operator, (e) Sort operator, (f) Grouping step used in the aggregation operator, (g) Aggregation operator, (h) Join operator. 24
(a) Table R 24
(b) Table S 24
(c) $\text{Project}_{k_1,k_2}(R)$ 24
(d) $\text{Filter}_{k_2>40\&v_1<10}(S)$ 24
(e) $\text{Sort}_{k_1,k_2}(S)$ 24
(f) Grouping R based on k_1 and k_2 24
(g) $\text{Aggr}_{k_1,k_2,sum(v_1),max(v_2)}(R)$ 24
(h) $\text{Join}_{k_1,k_2}(R,S)$ 24
2.2 Example SQL Query on tables R and S from Figure 2.1. 27
2.3 Logical query plan of the SQL query in Figure 2.2. 27

3.1 Simplified Structure of an FPGA. 30
3.2 Simplified structure of a logic element. 31
3.3 Alternative representations and the FPGA implementation of a single-bit half adder $y = a + b$: (a) Truth table, (b) XOR gate representation which can be implemented on ASICs using CMOS transistors, (c) FPGA implementation using a logic element. 31
(a) Truth table 31
(b) Gate representation 31
(c) Logic element implementation 31
3.4 Simplified structure of an FPGA's programmable interconnect. Connection boxes provide configurable routing between the logic elements and the programmable interconnect. Switch boxes define the topology of the interconnect. 33
3.5 The steps in the design process for a digital system on an FPGA. 34
3.6 Example FPGA card, based on the Intel® Stratix® 10 DX FPGA development kit [Inc19]. 38

4.1 Various types of FPGA acceleration platforms used for database query processing. The CPU and its RAM together form the host (see Section 3.4). The FPGA and its RAM are placed on an FPGA card. Storage refers to permanent storage, such as hard disk drive (HDD) or solid-state drive (SSD), commonly referred to as the *disk* in database system literature. 44
(a) Accelerator Card 44
(b) Coprocessor 44
(c) Smart Storage 44
(d) Network Processor 44

4.2 Examples of sort-based and hash-based aggregation and join operator execution: (a)Table R, (b)Table S, (c)Sorted table R, (d)Sorted table S, (e)5-entry hash table using hash function $h(x) = x\%5$ and containing all rows of R, (f)Aggregation of R, (g)Join of R and S. 47
(a) Table R 47
(b) Table S 47
(c) $\text{Sort}_k(R)$ 47
(d) $\text{Sort}_k(S)$ 47
(e) Hash Table of R 47
(f) $\text{Aggr}_{k,sum(v)}(R)$ 47
(g) $\text{Join}_k(R, S)$ 47

5.1 Overview of the architecture of our accelerator. The *Sort-* and the *Merge-Network* implemented on the FPGA execute the sort and the merge phases of the sort-merge algorithm, respectively. They are both pipelined, and read and write their data from/to the FPGA RAM. 58

6.1 Sort-merge architecture used for comparing RTL (VHDL) and HLS (OpenCL) methodologies in the context of database system acceleration. . 64
6.2 An example of the memory model used by our heapsort algorithm. Left memory (LM) and right memory (RM) are used to store the left and right siblings in the heap, respectively. The left smaller-than right memory (LSRM) indicates, for each pair of siblings, whether the left sibling is smaller than the right one. 66
6.3 *Merger* Architecture 67

7.1 Architecture of our database system accelerator MSM 77
7.2 Setup of our benchmarking platform. Multiple *Sort-* and *Merge-Networks* are instantiated to improve the performance the system, and to benefit from the four DDR RAM channels provided by the accelerator card. 82
7.3 The results of our benchmarks of the sort operator comparing the performance of MSM and MonetDB (MDB) on the random uniform (U-Random), sorted uniform (U-Sorted), and single-key (Single Point) datasets. 85
7.4 The results of our benchmarks of the aggregation operator comparing the performance of MSM and MonetDB (MDB) on the uniform, Zipf, and moving cluster datasets, with keys chosen randomly from the corresponding distributions. 85
7.5 The results of our benchmarks of the $M : N$ join operator comparing the performance of MSM and MonetDB (MDB) on uniform datasets with various values for M and N. 86

8.1 Example run of the KeRRaS algorithm with $n_{kc} = 4$ and $n_{bsc} = 2$. 93
(a) Table U 93
(b) Iteration 1: Reduce $\overline{k_1 k_2}$ into κ_{1-2} 93
(c) Iteration 2: Reduce $\overline{\kappa_{1-2} k_3}$ into κ_{1-3} 93
(d) Iteration 3: Reduce $\overline{\kappa_{1-3} k_4}$ into κ_{1-4} 93
8.2 Overview of the MSM architecture proposed in Chapter 7. 97
8.3 Architectural extensions to MSM's *Merge-Network* for implementing the KeRRaS algorithm. The *Pump*, *Exhaust* and *Rexhaust* interface adapters are the only components needed to implement KeRRaS. 98
8.4 Architecture of the *Pump* interface adapter used by MSMK for converting row burst accesses into column burst accesses, as shown in Figure 8.3. . . . 99
8.5 Architecture of the *Transposer* used by the *Pump* interface adapter to transpose database columns into rows, as shown in Figure 8.4. 100
8.6 Architecture of the *Rexhaust* interface adapter used by MSMK to produce and store the ω and κ columns, as shown in Figure 8.3. 101

8.7 The results of the sort and aggregation benchmarks comparing the performance of MSMK and MSM on narrow tables with $n_{kc}+n_{vc} \leq 4$. The benchmark datasets have $n_r = 2^{24}$ and $r_{kc} = 256$. 104
8.8 The results of the payload-less performance and scalability benchmarks comparing the performance of MSMK and MonetDB (MDB), as the number of rows (n_r) or key columns (n_{kc}) varies. 105
(a) Throughput of the sort operator as a function of n_r, with $r_{kc} = 2$ and $n_{vc} = 0$ 105
(b) Throughput of the sort operator as a function of n_{kc}, with $n_r = 2^{24}$ and $n_{vc} = 0$ 105
8.9 The results of the standalone sort operator benchmarks of 4 TPC-H queries with various key lengths (noted below each query) on MSMK and MonetDB (MDB). 106
8.10 The results of the payload-based sort, aggregation, and join operator benchmarks comparing the performance of MSMK and MonetDB (MDB), and the parameters of the datasets used in these benchmarks. 106
(a) Sort and Aggregation Benchmarks 106
(b) Join Benchmarks 106
(c) Sort and Aggregation Dataset Parameters 106
(d) Join Dataset Parameters 106

9.1 Example run of the aggregation operator $T = \text{Aggr}_{K,SUM(V)}(R)$, using early aggregation as an intermediate step. Table R is the input of the operator. Table S is the result of early aggregation, and contains duplicate keys that need further processing. Table T is the final result of the aggregation operator. 111
(a) Table R 111
(b) Table S 111
(c) Table T 111
9.2 The results of the simulations comparing sort-based and cache-based early aggregation on various datasets. 116
(a) Uniform 116
(b) Moving Cluster 116
(c) Self-Similar 116
9.3 The results of the simulations comparing set-associative caches for early aggregation on various datasets. All the caches used in these benchmarks employ the optimal replacement policy. 117
(a) Sorted 117
(b) Normal 117
(c) Heavy-Hitter 117
9.4 The results of the simulations comparing different cache structures for early aggregation on various datasets. All the caches used in these benchmarks employ the optimal replacement policy. 118
(a) Moving Cluster 118
(b) Uniform 118
(c) Zipf 118
9.5 The results of the simulations comparing different replacement policies for an 8-way set-associative cache performing early aggregation on various datasets. 120
(a) Sorted 120
(b) Moving Cluster 120
(c) Self-Similar 120
9.6 Overview of the architecture of the cache system. The pipeline consists of a few preprocessing stages that deal with window key/hash collisions, before the rows are processed by the cache itself. 122

9.7 Architecture of the *Window Aggregator* used as a preprocessing stage in the cache system, as shown in Figure 9.6. The module is responsible for eliminating window key collisions. 122
(a) *Window Aggregator* with window size w 122
(b) s-*Distance Aggregator* 122
9.8 Architecture of the *Collision Detector* used as a preprocessing stage in the cache system, as shown in Figure 9.6. The module computes the minimum number of stalls required for each row in order to eliminate all window collisions. 123
9.9 Architecture of the cache used for early aggregation in the cache system, as shown in Figure 9.6. The cache expects its input to be free of window key/hash collisions, which is a property ensured by the preprocessing stages described in this section. 125
9.10 Architecture of CbMSMK. The new (*Cache System* and *Streaming Aggregator*) and upgraded (*Sorters*) modules compared to the architecture of MSMK are highlighted in green. 126
9.11 The results of the benchmarks comparing the sort times, merge times, and reduction factors of CbMSMK and MSMK on the Zipf dataset. 128
(a) CbMSMK Sort and Merge Times 128
(b) MSMK Sort and Merge Times 128
(c) Reduction Factors of CbMSMK and MSMK 128
9.12 The results of the benchmarks comparing the performance of CbMSMK, MSMK, and MonetDB (MDB) on synthetic data. 129
(a) Uniform 129
(b) Moving Cluster 129
(c) Zipf 129
9.13 The results of the benchmarks comparing the performance of CbMSMK, MSMK, and MonetDB (MDB) on real data. 129
(a) Reduction 129
(b) Execution Times 129

10.1 Architecture of the CbMSMK acceleration platform. 132
10.2 Architecture of the *Sort-* and *Merge-Networks* used in CbMSMK for executing the sort-merge algorithm. The latter is used as a basis for running our target pipeline-breaking database operators. 133
(a) Sort-Network 133
(b) Merge-Network 133
10.3 The results of the benchmarks comparing the performance of CbMSMK and MonetDB on the sort, aggregation, and join operators with various datasets. These datasets are generated similarly to the ones in Chapter 7. . 134
(a) Sort Operator 134
(b) Aggregation Operator 134
(c) M:N Join Operator 134
10.4 The results of the benchmarks comparing the performance of CbMSMK and MonetDB on the TPC-H dataset with various scale factors. 135

LIST OF TABLES

2.1 An example of a table in a relational database. 22

4.1 Past academic and industrial work on database system acceleration on FPGAs, with a focus on pipeline-breaking database operators. Cells containing n.d. indicate that the corresponding attribute has not been disclosed. . 43

6.1 Characteristics of the OpenCL implementation of the sort-merge algorithm with 16 *Sorters* and 4 *Mergers*. Execution time (ET) and throughput (Thr.) values are for sorting the *Basic Workload*. 68
6.2 Comparison between the OpenCL and RTL implementations of the *Sorter*. Throughput (Thr.) is for sorting random 512-bit keys producing runs of 1022 numbers. 69
6.3 Comparison between the OpenCL and RTL implementations of the *Merger*. Throughput (Thr.) is for merging 2 sorted lists of 16.5 million 512-bit keys. 70
6.4 Comparison between the OpenCL and hybrid OpenCL-RTL implementations of the sort-merge algorithm. Both implementations boast 16 *Sorter* and 4 *Merger* units. Execution time (ET) and throughput (Thr.) are for sorting the *Basic Workload*. 71

7.1 FPGA resource utilization of our implementation of MSM (with a 128-bit wide data path) and some of its major components. Replication factor (R) is the number of instantiations of a module within its level in the hierarchy (e.g., there are 32 *Sorters* in a *Sort-Network*). The percentage resource utilizations are calculated over the total amount of resources provided by the FPGA. 83
7.2 *Sorter* resource utilization as a function of capacity (c_S) and data path width. 84

8.1 Parameters of the KeRRaS algorithm. 92
8.2 Formal definitions of ω and κ produced by the KeRRaS algorithm. They can both be generated using single-pass algorithms. While ω does not need to be stored in memory, κ must be stored in memory and preserved across KeRRaS iterations. 94
8.3 FPGA resource utilization of our implementation of MSMK (with a 192-bit wide data path) and some of its major components. Replication factor (R) is the number of instantiations of a module within its level in the hierarchy. The percentage resource utilizations are calculated over the total amount of resources provided by the FPGA. 103
8.4 Experiments illustrating the impact of the architectural parameter w_r on the performance and resource utilization of MSMK. 107

9.1 FPGA resource utilization of various implementations of early aggregation. n is the degree of associativity in cache-based implementations. The percentage resource utilizations are calculated over the total amount of resources provided by the FPGA. 127

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, April 28, 2023