Exploring the Limits of Query Pushdown for SQL Acceleration on FPGAs

by

Yüksel Yönsel

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday September 21, 2021 at 11:00 AM.

Student number:	5128501	
Thesis number:	Q&CE-CE-MS-2021-08	
Project duration:	October 20, 2020 – September 21, 2021	
Thesis committee:	Dr. Zaid Al-Ars,	TU Delft, supervisor
	Dr. Claudia Hauff,	TU Delft
	Dr. Joost Hoozemans,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

There has been an increasing interest in moving computation closer to storage in recent years due to significant improvements in memory technology. FPGAs were proven to be an exciting candidate for accelerating database workloads since they provide an energy-efficient, reconfigurable and high-performance computation platform. Therefore, FPGAs are widely used as attached accelerators on data-centric applications.

Database operations usually run on large volumes of data, which creates an I/O bottleneck when processing them on CPUs. Therefore, recently, researchers have been investigating query pushdown techniques during a database load operation. A well-known columnar storage format, Apache Parquet, provides an efficient way to store a database. In addition, current big data processing engines provide functionalities for pushing filter operation down to the parquet reading stage.

This study explores the boundaries of pushing down analytic queries to the parquet reader stage by using FPGAs. An extended roofline analysis is performed on a proof-of-concept hardware design. The analysis shows that peak performance is achieved via a storage-attached accelerator once a high bandwidth interface is introduced. Furthermore, using multiple FPGAs with flash storage while interfacing them with OpenCAPI or PCI switch enables higher performance for aggregation since aggregation is shown to be I/O bound.

The thesis introduces Apache Spark integration of the proof-of-concept query pushdown for parquet reading operations. Apache Spark implements several layers of parallelism to achieve higher speed-ups. However, the concurrency and parallelism for a single FPGA instance for multi-threaded Apache Spark applications requires synchronization on a constrained resource represented by a single FPGA. Therefore, this work suggests a way to achieve synchronization with a single FPGA instance.

The present work shows that for a single Spark thread, a maximum end-to-end application speed-up of 3.88x and a kernel speed-up of 7.24x are achieved. As a result, the throughput of TPC-H Query 6 can be increased up to 3.8 GB/s. Furthermore, FPGA can perform better than CPU until Spark is configured to run on 7 CPU threads. Then, for the scaled-up multi-threaded Spark application with six CPU threads, the FPGA can achieve 1.13x end-to-end application speed-up and a kernel speed-up of 13.19x.

Preface

This thesis is the conclusion of my eleven months of work with the Accelerated Big Data Systems group. At every small step and hurdle, I had a chance to learn new ways to improve myself. In addition, it was an excellent opportunity to conduct innovative research and push state-of-art while meeting great people with exceptional knowledge.

Firstly, I would like to thank my supervisor, Dr. Zaid Al-Ars, for his endless help and motivation towards my work. Even though I encountered some problems during the thesis, he always welcomed my questions and had inspirational solutions. Moreover, I want to thank Dr. Joost Hoozemans and Àkos Hadnagy for their support of my work and for sharing valuable ideas. I have learned so much from the amazing people in Accelerated Big Data Systems Group. These experiences will always last with me and help me overcome challenges that life will throw at me in the future.

Finally, I would like to give my gratitude towards my dear parents, Sibel and Hasan. They were always there for me when I needed them. They always kept encouraging me to improve myself and push my limits. Thank you for always supporting me!

Yüksel Yönsel
Den Haag, September 2021

Contents

List of Figures ix

List of Tables xi

List of Listings xii

List of Acronyms xiii

1 Introduction 1
- 1.1 Context . . . 1
- 1.2 Research Questions and Contributions . . . 2
- 1.3 Outline . . . 3

2 Background 5
- 2.1 Fundamentals of Databases . . . 5
- 2.2 Database Storage Models . . . 8
- 2.3 System Interconnects . . . 8
- 2.4 Projects . . . 9
  - 2.4.1 Apache Spark . . . 9
  - 2.4.2 Apache Parquet . . . 11
  - 2.4.3 Apache Arrow . . . 12
  - 2.4.4 Fletcher . . . 12
- 2.5 Benchmarks . . . 13
  - 2.5.1 Transaction Processing Performance Council - TPC . . . 13
  - 2.5.2 TPC-H . . . 14
  - 2.5.3 TPC-DS . . . 15
- 2.6 Related Work . . . 16

3 Case Study On TPC-H and TPC-DS 19
- 3.1 Motivation . . . 19
- 3.2 A Brief Complexity Analysis . . . 19
- 3.3 Discussion . . . 22

4 Extending Parquet File Reading Operation on Apache Spark 23
- 4.1 Apache Spark Parquet Scan Operation . . . 23
- 4.2 Injecting custom Parquet File Reader . . . 25
- 4.3 Concurrency and Parallelism . . . 25
- 4.4 Preliminary Conclusion . . . 28

5 Hardware Design 31
- 5.1 Parquet Reading on FPGA . . . 31
  - 5.1.1 Reading Multiple Columns . . . 32
  - 5.1.2 Integrating computation in the Architecture . . . 33
- 5.2 Hardware Implementation of TPC-H Query 6 . . . 33
  - 5.2.1 TypeConverter . . . 34
  - 5.2.2 FilterStream . . . 35
  - 5.2.3 MergeOp . . . 35
  - 5.2.4 Aggregation . . . 35
- 5.3 Discussion . . . 36

6 An Extended Roofline Analysis for Parquet Decoding Operation on FPGAs 39
6.1 Parameters of Roofline Analysis . . . 40
6.1.1 Byte Operations . . . 40
6.1.2 Input/Output . . . 41
6.1.3 Area . . . 41
6.1.4 Attainable Computational Performance per FPGA . . . 42
6.1.5 CI - Computational Intensity . . . 42
6.2 Roofline . . . 43
6.2.1 Constructing Computational Roof . . . 43
6.2.2 Constructing I/O Roof . . . 44
6.2.3 Roofline Model . . . 44
6.3 System Design . . . 47
6.3.1 Analysis on state-of-art system architectures . . . 47
6.3.2 Preliminary Conclusions . . . 49

7 Profiling Results 53
7.1 Experiment Setup . . . 53
7.2 Architecture A . . . 53
7.2.1 Throughput Analysis . . . 54
7.2.2 Single Parquet File . . . 54
7.2.3 Multiple Parquet Files . . . 54
7.3 Architecture B . . . 55
7.3.1 Throughput Analysis . . . 56
7.3.2 Single Parquet File . . . 56
7.3.3 Multiple Parquet Files . . . 57
7.4 Large Tables . . . 57
7.5 Discussions . . . 58

8 Conclusion and Future Research 63
8.1 Conclusion . . . 63
8.2 Future Research . . . 64

Bibliography 65

A Summary Metrics for Different Cluster Setups 71

B TPC-H/DS Analysis Results 73

List of Figures

1.1 The placement of FPGA in system architectures [29] . . . 2

2.1 Database Types . . . 6
2.2 Example Business Type Relational Database Schema . . . 6
2.3 System Interconnect Trends . . . 9
2.4 The components of Apache Spark . . . 10
2.5 An example lineage graph of the transformation in 2.1 . . . 11
2.6 The layout of a parquet file . . . 12
2.7 The architecture of Fletcher [49] . . . 13
2.8 TPC Benchmark Timeline [54] . . . 13
2.9 TPC-H business environment [21] . . . 14
2.10 The TPC-H Schema [21] . . . 15
2.11 TPC-DS Components [20] . . . 16

3.1 Complexity Analysis on TPC-H Queries . . . 21
3.2 Complexity Analysis on TPC-DS Queries . . . 21

4.1 Apache Spark *FileScan* Operator Details . . . 24
4.2 Extending the physical plan of Apache Spark . . . 25
4.3 Complete Software Design . . . 27
4.4 Software Architecture of *ColumnScheduler* . . . 29
4.5 Software Architecture of Thread Safe Thread Pool . . . 30

5.1 Higher Level design proposed by Van Leeuwen 2019 [55] . . . 31
5.2 Architectural overview of Parquet to Arrow Converter Accelerator [50] [55] . . . 32
5.3 Example computation pushed down parquet reader with multiple parquet reader instances 33
5.4 Proposed changes on Parquet Reader interface - Taking ArrayWriter out of datapath . . . 33
5.5 Kernel Hierarchy . . . 34
5.6 Query 6 accelerator Architecture . . . 34
5.7 ReduceStage module [32] . . . 35
5.8 ReduceStream module [32] . . . 35
5.9 Accumulator logic with Hash Table implementation . . . 36
5.10 The architecture of Hash Table . . . 37
5.11 Architecture of query pushdown parquet reader instance . . . 37
5.12 Architecture with multiple query pushdown reader instances which can parallelize upto 3 Row Groups . . . 38

6.1 Basis of the Roofline Model for different applications [22] . . . 39
6.2 Simple block design for realigning of 2 bytes input data stream . . . 40
6.3 Computational Intensity of seperate implementations with datapath of 64 bits 200 MHz design . . . 43
6.4 Attainable Performance (per FPGA) of seperate implementations with datapath of 64 bits 200 MHz design . . . 44
6.5 I/O roof per interconnect . . . 45
6.6 Roofline Analysis . . . 46
6.7 Application Runtime Path . . . 47
6.8 Roofline model for SmartSSD . . . 48
6.9 Roofline model for Power9 . . . 49
6.10 The throughput projection . . . 51

7.1 Throughput calculation for different native thread counts 54
7.2 Total Size of 0.686 GB 55
7.3 Total Size of 1.4 GB 55
7.4 Total Size of 6.9 GB 56
7.5 Total Size of 14 GB 56
7.6 Total Size of 21 GB 57
7.7 Architecture of kernel which can process 4 elements per cycle 58
7.8 Throughput calculation for different native thread counts 59
7.9 Total Size of 0.686 GB 59
7.10 Total Size of 1.4 GB 60
7.11 Total Size of 6.9 GB 60
7.12 Total Size of 14 GB 60
7.13 Total Size of 21 GB 61
7.14 Runtimes 61

B.1 TPC-H Queries with ORDER BY Operation 73
B.2 TPC-H Queries with JOIN operation 74
B.3 TPC-DS Queries with ORDER BY Operation 74
B.4 TPC-DS Queries with JOIN operation 75

List of Tables

6.1 Byte operations per cycle for each component to be used for CI(Data bus length in bytes=N) 40
6.2 Utilization Report of query pushdown parquet reader implementation 42
6.3 Scalability of the important components 42
6.4 Available LUTs for different FPGA families 47

7.1 Utilization Report of query pushdown parquet reader implementation with epc=4 57

A.1 Summary Metrics for local cluster setup which has 1 worker node with 12 cores 71
A.2 Summary Metrics for local cluster setup which has 3 worker node with 4 executors . . . 71

List of Listings

2.1 An example set of operations 10
3.1 An example SQL code visualizes operators(red) and operands(black) 19
4.1 An example API calls for parquet file reading of TPC-H lineitem Table 23
4.2 Physical Plan generated by Spark API 24
4.3 An example task for FPGA execution 27
5.1 An example code to obtain Columnar Chunks 32
5.2 TPC-H Query 6 34

List of Acronyms

ABS Accelerated Big Data Systems.

AXI Advanced eXtensible Interface.

BRAM Block RAMs.

CI Computational Intensity.

CLB Configurable Logic Block.

CPU Central Processing Unit.

CSV Comma-separated values.

DAG Directed Acyclic Graph.

DRAM Dynamic random access memory.

DSM Decomposition Storage Model.

FPGA Field Programmable Gate Array.

HLS High Level Synthesis.

I/O Input/Output.

LUT LookUp Table.

OLAP Online Analytical Processing.

OLTP Online Transactional Processing.

PAX Partition Attributes Across.

PCI-E Peripheral Component Interconnect Express.

QPI QuickPath Interconnect.

RDD Resilient Distributed Datasets.

RLE Run-length encoding.

RPC Remote procedure call.

SQL Structured Query Language.

TPC Transaction Processing Council.

VHDL VHSIC Hardware Description Language.

1 Introduction

1.1. Context

The rate of improvement in processor speed has exceeded the rate of improvement of memory technology throughput over the last two decades. This phenomenon is defined as the *memory wall* [58]. To address this problem, researchers have proposed several ways to overcome this limitation. One of which is introducing an ever-increasing memory hierarchy, in addition to faster, smaller on-chip memories, so that the communication speed increases.

While the *memory wall* has been around for more than two decades, processor architectures have evolved faster to demonstrate higher performance. Since 2005, system designers have increased the processor core count on a single chip to exploit Moore's Law Scaling. Nevertheless, Esmaeilzadeh et al. [27] have shown that the end of multicore-scaling is expected to be reached by a technology integration of 8nm due to the need to keep more than 50% of chip area unpowered because of heat. Therefore, there has been an increasing interest in heterogeneous computing platforms for energy and performance considerations [61], [29].

In the context of big data, bringing data from storage has always been challenging as it was mostly technology bound, and the volume of big data was more extensive than the volume associated with other workloads. Previous research has shown that for transaction processing workloads, 65% of the node idle times are due to storage bottlenecks [43]. Since OLAP workloads require processing in large amounts, it is no coincidence that it will have ample storage bottlenecks. In order to overcome these bottlenecks, FPGA-based database systems are designed in several different models depending on the workload characteristics, as illustrated in Figure 1.1 [29]. FPGA as a bandwidth amplifier, in Figure 1.1a, is desirable once FPGA is used as a decompressor, filter, or doing early aggregations. This kind of configuration prevents excess copies of data sources while providing a low latency stream processing solution. FPGA as I/O attached accelerator, as seen in Figure 1.1b, creates an opportunity to run computationally intensive applications on FPGAs. The drawback of such a configuration is that the source data should be copied to the device's memory. Finally, FPGA as a co-processor, shown in Figure 1.1c, configures FPGA as a distinct processor mapping the shared memory with the host CPU. This kind of architecture has the benefit of eliminating data copies. Moreover, newer coherent interconnect types between the host CPU and FPGA enable high bandwidth communication.

All these chokepoints of big data and memory have paved our way into pushing code into data storage instead of loading the data from disk. Therefore, current systems employ solutions closer to storage and use FPGA as an intermediate platform to filter or aggregate data closer to the memory. The vision of the Accelerated Big Data systems group in TU Delft is to improve the knowledge on possible ways to accelerate the different types of workloads and their extensibility to the most well-known big data frameworks. There exist individual efforts in our group to compile and run single operators such as parquet file reading [50], filtering [32], [42], aggregations [46], sorting [65], joins, etc, on FPGAs. Nevertheless, integrating the hardware design of analytic workloads with the hardware design of file readers has not been treated in much detail. Running aggregations in the earlier stages of the big data pipeline will decrease data copies and fully utilize the underlying storage bandwidth. Designing FPGA accelerators, which implement few distinct operators, and integrating them with current well-known

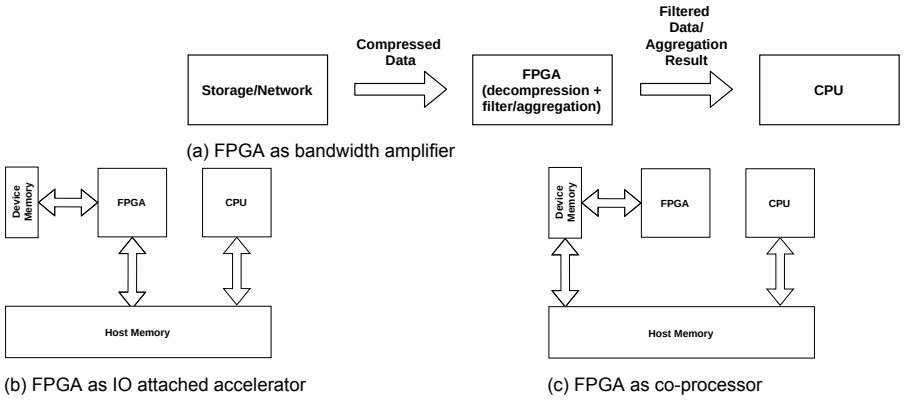


Figure 1.1: The placement of FPGA in system architectures [29]

frameworks creates a big challenge. Therefore, there is always a question remaining whether or not there is a limit on pushing filters or aggregations to deeper levels of the big data pipeline.

There are several attempts to model the performance of a hardware design to find the area and performance-optimized solution[22]. Moreover, several studies proposed a conceptual FPGA intensive server configuration [29], [30]. These studies investigate the possible FPGA intensive server architectures based on commercially available products [16], [38]. However, there is still a challenge in estimating the performance gains of different FPGAs placed in different server configurations. This study aims to characterize the computational intensity of analytic queries and estimate the performance achievements on FPGA intensive system architectures. The analysis in this thesis goes beyond the commercially viable products and discusses the next generation interconnect types.

1.2. Research Questions and Contributions

This thesis will examine how query pushdown is used until the beginning of the query processing pipeline and the limitations of such pushdown operations in terms of state-of-the-art accelerators. Therefore, the thesis will address the following two research questions.

1. How can query pushdowns be integrated into state-of-art parquet decoders in dataflow hardware designs?
2. How much of a query can we accelerate on FPGAs? What is the bounding factor for query pushdown?

The contributions of this thesis work can be summarized as follows:

A Case Study on TPC Decision Support Benchmarks
TPC Decision Support Benchmarks are well-known for profiling database systems. This work investigates TPC Decision Support Benchmarks in terms of their acceleration opportunities and implementation complexities. This case study is mainly performed to decide which query to design as a part of this thesis work.

Apache Spark integration of pushed down query accelerators
Apache Spark distributes the workload to corresponding executors. This thesis profiles the workload division for the File Scan operator and Filter pushdown File Scan operator and proposes a software integration for the accelerated query pushdown design, using a single FPGA instance and low-level multithreading APIs.

A proof of concept pushdown accelerator design
Peltenburg et al. proposes an extensible parquet to arrow converter design using FPGAs [50]. This study implements a dataflow hardware design for TPC-H Query 6 and integrates both solutions to implement an FPGA accelerated query pushdown kernel.

An extended roofline analysis for parquet decoder operations and query pushdown
Da Silva et al. proposes an extended roofline analysis for HLS design tools [22]. In this work,

a derivate of that implementation is used for estimating peak performance of parquet decoding and query pushdown operation for different system interconnects with several different FPGA instances, namely, VU37P, VU19P, KU15P. The analysis is performed on state-of-art accelerator configurations to estimate peak performance achievements of parquet reading and query pushdown designs. This study reveals the bottlenecks of storage-attached ingestion of Parquet files, bandwidth amplification, and query pushdown.

1.3. Outline

The thesis is composed of the following chapters:

Background
This section introduces the necessary information about the frameworks and techniques used in the thesis.

Case Study On TPC-H and TPC-DS
A quantitative analysis of TPC Decision Support Benchmarks is presented.

Hardware Design
Dataflow Hardware Design of pushdown operation is investigated.

Extending Parquet File Reading Operation on Apache Spark
Apache Spark integration of such accelerator is described.

An Extended Roofline Analysis for Parquet Decoding Operation on FPGAs
The proposed peak performance estimation models are investigated, and a system prototype is discussed for the current state of art parquet decoding operator.

Profiling Results
Results section summarizes the profiling results of such operation on Power9.

Conclusion and Future Research
This section concludes the thesis and discusses the limitations and future research directions for query pushdown operation.

2 Background

2.1. Fundamentals of Databases

In simple terms, a database is a collection of information that should provide a proper volume and fast access to big data. A *database management system* is a software that enables the manipulation and storage of the information stored in databases. In 1956, IBM introduced the first computer to use random access disk drive, which made relational databases possible as it was the pioneer of finding the data in the storage and manipulating it in the order of seconds [37]. In the 1960s, Charles W. Bacman, a database software pioneer, had introduced the first generalized database management systems and relational data structures [10],[11]. Afterward, IBM has introduced the world's first commercial database management system, called Information Control System and Data Language/Interface [14]. In 1970, Edward F. Codd, an IBM fellow, has introduced a relational model for representing data in large shared banks [17]. The 1970s were innovative in terms of *relational databases*. In the 1980s, commercial relational database systems started to be seen. The well-known commercial databases are RIM, RBASE 5000, PARADOX, OS/2 Database Manager, Dbase, and Watcom SQL. In 1986, ANSI standardized SQL as a standard query language. Moreover, object-oriented databases were introduced in the late 1980s. The 1990s were very influential as the Internet was introduced. This enabled remote access to servers. The businesses started to employ client-server models for their databases. XML was introduced in 1997. This enabled vendors to integrate XML to their databases [13]. In the 21st century, the three main companies have dominated the market for databases: Oracle, IBM, Microsoft. The applications of databases have become prominent and have inclined towards storing large amounts of structured-unstructured datasets. All of this effort led to a 12 billion dollar industry. In the next sections, the different types of databases will be investigated.

Database Types

A database can be designed concerning specific characteristics, which can be briefly classified as a database model, the business objective, stakeholders, and location as illustrated in Figure 2.1.

Relational Databases A relational database stores the data into different tables which can be related to one another. Different related tables can be merged using their unique ids. The relational databases can communicate to each other by using Structured Query Language *SQL*. SQL provides a mathematical language that manipulates the relational data structures. This kind of structure creates some benefits, namely, flexibility, reduced redundancy, and consistency. A perfect example can be given by a business-related database which is visualized in Figure 2.2. A company holds a table of products that will be shipped to the customers. There are subsequently 3 columns, *product_id*, *price* and *discount*. Also, the same company runs a part table which holds the information about the production cost of each product and has the following columns: *product_id*, *cost*. By stating that this *product_id* is a unique id, it can be seen that these tables are related. The company may use this information to calculate revenue by creating another table resulting from the joining of these two tables.

The definition of the relational model is beyond this example. As an example provides a basic view on relational databases, we can see that relational databases store data in tales. Each Table organizes

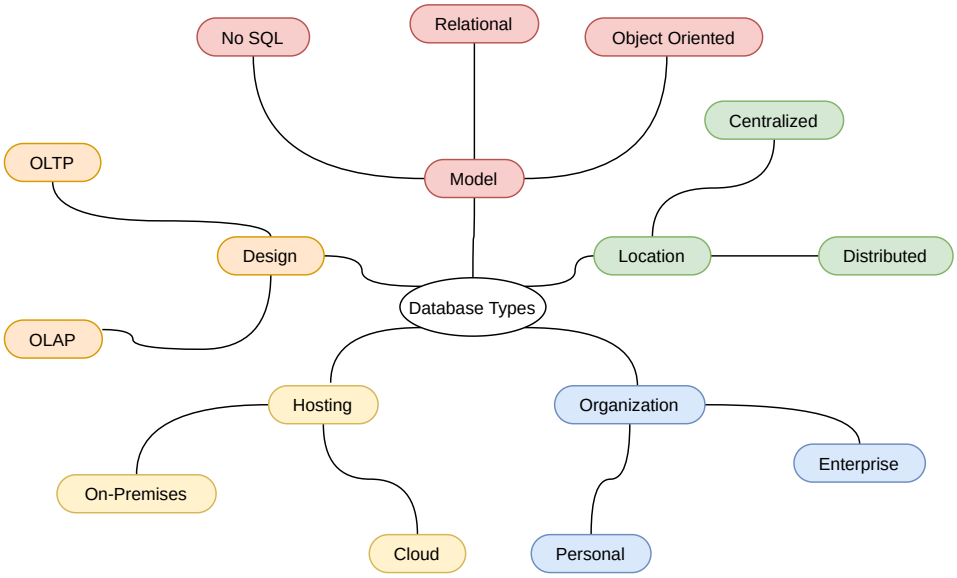


Figure 2.1: Database Types

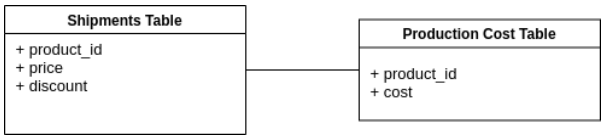


Figure 2.2: Example Business Type Relational Database Schema

data in columns which are *produc_id*, *cost* etc.

We will now examine the terminology to understand architectural design decisions employed for relational databases. Codd defines a relation to be a collection of sets in which n-tuples reside [17]. A domain is, in a way subset of a relation. A domain is considered to be non-redundant if it is simple.

Active Domain is a pool of values represented at some instant is called active domain.

Primary Key is defined as a domain that holds that relation's unique set of elements (tuples).

Foreign Key is used when we express cross-references between different or same relations. The foreign key of one relationship is not the primary key of its relation but the primary key of other relations.

Non-simple domains are the type of domains that hold non-atomic values as elements. In other words, they can hold relations.

Knowing basic terminologies, we can move on with schema models.

Normal Forms of relational databases - 3NF Schema

A relation whose domains include non-simple domains is worth normalizing due to the simplicity of storing primitive values, simple data structures in memory. We define the degree of normal form once we normalize the relation. *1NF*, first normal form, suggests that every relation has a primary key and single attribute. *2NF*, second normal form, is satisfied when the relation is *1NF* and non-key elements are fully dependent on the primary key. Finally, a relation is *3NF* once it is *2NF* and there are no dependencies between non-key elements. By transitioning from the first level to the third level of normal form, the redundancy of a relation gets minimized. This implies that there will be fewer insertion, deletion, and update anomalies in the database.

Star Schema

Star Schema consists of fact tables and dimension tables. In simpler terms, fact tables are larger and at the center of schema topology.

Snowflake Schema
Snowflake schema resembles star schema. The only difference is that dimension tables are normalized.

We will revisit these schema models while discussing benchmarks in the following sections.

Non-relational Databases(NoSQL) NoSQL databases are non-relational databases that can handle extensive, unstructured data different than traditional relational tables. The most common database models are key-value, column, document, and graph databases.

Key-value store model does not define any query language. Instead, the data manipulation is done via the store, retrieve and delete commands. As the name suggests, this type of database can be modeled as a Hash-Map or associative row, in which key is a string of characters and the value is any data type, opaque to storage. As one can imagine, the key-value is not well-defined by any schema, making this type of database more flexible.

Document-oriented databases is in a way extended key-value store which stores the data in groups called collections. The database can be stored in JSON, XML, or YAML formats. This type of database offers flexibility in the same way that key-value store offers. Apache CouchDB and Mongo DB can be provided as an example to this type of database.

Graph database structure big data into graph vertices and nodes. This type of database offers a better representation of data as it provides a more profound relationship structure than relational database models.

Column database stores data into columns and rows in a similar way the relational databases. There exists ”column-families,” which holds logically related columns for each row and a ”key.”

Object-Oriented Databases This type of database resembles the application of objects used in object-oriented programming languages. The data stored in an object-oriented database is persistent. Object-oriented databases do not have tables, rows, or keys as in relational databases. Instead, the data is stored as objects that can represent complex data structures on a disk. Relational database model stor.es atomic and primitive types of data structures, such as ”utf-8”, ”int.” Therefore, object-oriented databases can have fast queries over more complex data structures. MongoDB Realm is one of the examples of document databases [44].

Centralized Databases Centralized databases are stored and maintained on a single location, mainframe computer. The other nodes connect to this single location over a network to access the database.

Distributed Databases In this type of database, the data is stored across multiple nodes. These multiple computers can be

Online Transactional Processing - OLTP Databases An online Transactional database is a type of database where large numbers of database transactions occur in real-time over the Internet. A *database transaction* is defined as any change performed within a database such as insertion, deletion, or query [26]. Furthermore, the database transactions should be indivisible in such a way that all actions should be reflected appropriately in the database or nothing happens [33]. Therefore, we define several characteristics for database transactions:

- **A**tomicity. The transaction should be deterministic. It either runs all actions successfully or fails.
- **C**onsistency. Once the *End of Transaction* is reached, the returned result does not affect the consistency of a database.
- **I**solation. Each transaction runs independently from other transactions.
- **D**urability. Once a transaction commits its result back to the database, the system will keep the result even a failure occurs.

If a particular system supports a transaction, that transaction is ACID-compliant for that system.

Online Analytical Processing - OLAP Databases OLAP is a software performing multidimensional analysis at high speeds on large volumes of data [25].OLAP server stores the data in the form of front-end packages. This type of database is started from a need of consolidating, viewing, and analyzing multidimensional data in a way that makes sense for enterprise analysts [18]. OLAP offers a multidimensional database named OLAP cube. A single table is represented in multiple layers, while each layer adds a different concept regarding the stored table in OLAP Cube. The multidimensional, Relational and Hybrid OLAP databases are the branches of OLAP. Relational OLAP offers relational tables instead of organizing data in OLAP Cube. This type of OLAP gives the ability to perform complex queries on more significant amounts of data. Hybrid OLAP is the hybrid of these two OLAP types, as the name suggests. It includes the high-performance feature of Multidimensional OLAP and the complexity of Relational OLAP in the same database.

Cloud Databases Cloud databases attempt to build, manipulate and deploy the database in a cloud environment. Amazon Web Services, Oracle, Microsoft Azure, and IBM are few well-known cloud database examples. Cloud databases offer a low-cost, flexible, easy to manage service for storing data.

On-Premises Databases These types of databases are private databases that are maintained within a particular enterprise. The company sets up a database according to the performance and power usage of its peak loads. In some cases, this type of database offers less flexibility than the cloud databases because the company handles the server installation costs. On the other hand, the enterprise can scale up or down quickly by leaving installation or maintenance cost calculations to the cloud provider.

2.2. Database Storage Models

The layout of data stored in a disk carries enormous importance in terms of the performance of a query. The traditional database schemes use the N-ary Storage Model (slotted pages), which stores from the beginning of the page and uses an offset table at the end of the page to keep track of the records [51]. Another type of storage model is DSM which is not common and relatively old. DSM is a transposed storage model which partitions attributes vertically into relations [19]. However, newer database storage models make use of PAX layout for the sake of utilizing memory bandwidth. PAX layout groups the value of each attribute of each page. PAX layout offers decreased memory stall times of NSM by 75% and executes range-selection queries 17-25% faster [5].

2.3. System Interconnects

The volume of data grows exponentially worldwide. Therefore, how big data can be smartly stored carries considerable importance. While storage and compute platforms achieve peak speed-ups, the interconnect type is essential for creating a low overhead high-speed communication channel. In this section, the system interconnects types will be investigated. The memory technologies are beyond the scope of this thesis. Therefore, this work does not engage with several different memory architectures and their peak performance specifications.

A computing system may have the best possible configuration and technology which provides low latency and high bandwidth. However, the bounding factor for the performance will be the type of connection between the memory and compute module. This computer module may be an FPGA or CPU. The interconnect will mostly be dependent on the limitations of the interface architecture of our compute module. There are specific requirements one should expect from an interface:

- The memory interconnect should have low latency and low overhead.
- The interface should be able to scale so that it can support communication on multiple channels
- The interface should consume low power.
- The system architecture should be flexible as it should support different platforms and applications.
- The interconnect should have low cost.

State-of-the-art interconnects technologies are summarized in the following paragraphs.

PCI-E Intel introduced the PCI Express bus in 2004. The target was to design an interface that could transfer graphics data faster than the I/O technologies of that era. It is, in a way, the successor of the PCI bus, which was introduced by intel back in 1993. PCIe is a serialized point-to-point interconnect protocol that also provides a high-speed, scalable data transportation [41]

PCIe doubles its bandwidth every 3 years [64]. PCIe has several different form factors which are briefly x1, x2, x4, x8, x16. As the form factor increases, the bandwidth also increases by the multiples of the form factor.

OPENCAPI OpenCAPI is Open Interface Architecture that allows processors to attach to I/O devices, accelerators, memories. OpenCAPI is a high-bandwidth, low latency interface also used in accelerated Power9 servers. The peak bandwidth of OpenCAPI is 25 Gbyte/s. A POWER9-based system can use OpenCAPI 3.0 to coherently communicate between the CPU and an FPGA (e.g. Xilinx KU60, VU3P, VU37P). The system uses 8 OpenCAPI channels, which theoretically provide a peak bandwidth of 25 GB/s.

Fang et al. investigates the bandwidth of interconnect trends over time as visualized in Figure 2.3 [29]. The increase in bandwidth of PCI-E and DRAM is less than that of storage and network. Fang et al. explain the reduction of DRAM bandwidth increase by the rise in chip area cost. This graph has significant implications for Chapter 6 as the scaling trend of storage bandwidth will have a positive effect on increasing the system performance.

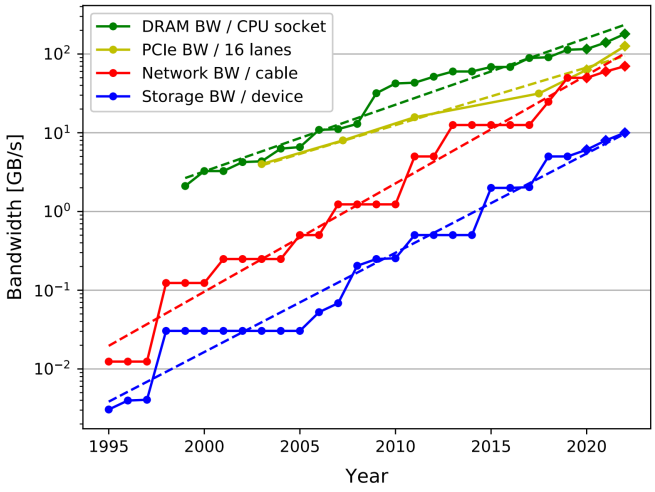


Figure 2.3: System Interconnect Trends

2.4. Projects

In this section, the frameworks and third-party projects used in the scope of this thesis are described.

2.4.1. Apache Spark

Apache Spark is the unified engine for distributed big data processing. Apache Spark was initially developed at UC Berkeley's AMPLab in 2009, open-sourced in 2010 under the BSD license. Until then, Apache Spark has been used extensively by enterprises across many different industries. The programming model of Apache Spark is similar to the MapReduce framework but extends it with Resilient Distributed Databases, "RDDs." Apache Spark can process many different workloads and use the same kind of optimizations for the different workloads with a unified API. This enables users to develop applications quickly and extend Spark to implement different processing workloads, such as streaming machine learning.[63]

Execution Model Apache Spark initializes a SparkContext, which runs a set of processes on a cluster. The components of Apache Spark are visualized in Figure 2.4. Spark Context can connect to several cluster managers, namely, MESOS, YARN, standalone. The cluster manager is responsible

for resource allocation. First, it assigns executor processes on worker nodes to SparkContext, then SparkContext sends the execution code to the nodes.

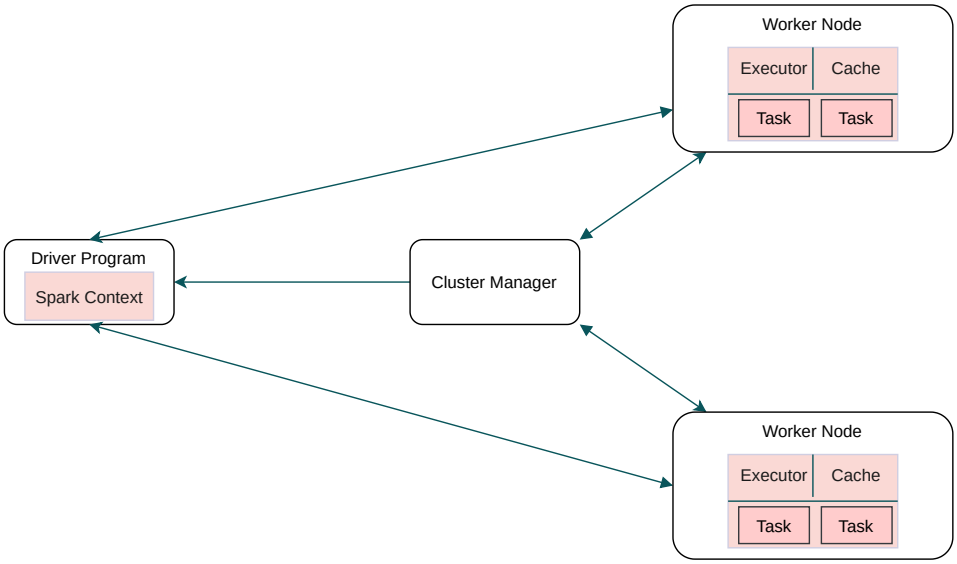


Figure 2.4: The components of Apache Spark

These are important terminology to remember

Spark Application
Application is an instance of Spark Context.

Job
Spark jobs are create every time a *collect()* action is called.

Stage
Apache Spark divides jobs into stages, which is a step in the physical plan. There are two types of stages in Apache Spark, namely the map stage and the result stage. The number of stages is dependent on the underlying partitions and the necessity to do any shuffles.

Task
Task is a computation on a single data partition in a stage.

DAG
Directed Acyclic Graph is a set of vertices and edges representing the RDD and operation applied on RDD. A detailed visualization is provided at Figure 2.5.

RDD As explained previously, RDDs are the primary abstraction for the programming model of Apache Spark. Before RDDs were developed, most of the big data processing frameworks(e.g., MapReduce) were used to save intermediate results between different computations to an external storage system. However, this was creating a huge overhead due to serialization, disk I/O, etc. Therefore, Apache Spark proposed RDDs to persist intermediate results in memory and offer fault tolerance explicitly.

RDD is defined as a read-only, partitioned collection of records [62]. The most important feature RDD presents is that it stores transformation to create itself from other RDDs or datasets in stable storage. This is called the *lineage of RDD*. A lineage graph is directed acyclic graph which is illustrated in Figure 2.5. This way, the program knows how to recreate RDD in case of failure. RDD can be partitioned across multiple machines, and the programmers can explicitly declare which RDDs to keep in memory between different computations.

```
1 rdd1 = spark.parquet.read(”hdfs://”)
2 rdd2 = rdd1.filter(_.contains(”x”))
3 rdd3 = rdd2.map(_.split('\t'))
```

Listing 2.1: An example set of operations

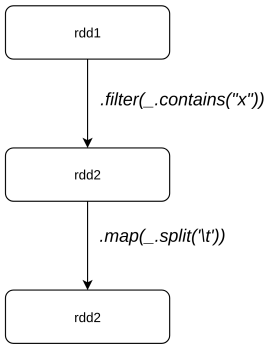


Figure 2.5: An example lineage graph of the transformation in 2.1

DAGs in apache spark can be divided into multiple stages so that tasks can run the same, either fine or coarse-grained stages.

Spark SQL and DataFrames Spark SQL is an Apache Spark module built upon Shark and enables relational processing within Apache Spark API [7]. Spark SQL enables its users to mix relational queries and procedural algorithms. The DataFrames are the distributed collection of structured records. Spark SQL proposes the DataFrame API to perform relational operations on both external data sources and on built-in distributed collections.

Spark SQL designs an extensible query optimizer named Catalyst for supporting a large variety of analytic workloads. The extensibility of Spark SQL is an important feature that this thesis is built upon. There are two main parts contributing to Catalyst: *trees* and *rules*. A tree consists of node objects which may or may not have children. Rules are used to manipulate the tree nodes. One can simply run a transformation on tree branches. In the Chapter 4, we will examine the Spark SQL extensions for parquet reader operation.

2.4.2. Apache Parquet

Apache Parquet is a columnar storage format available to any project in the Hadoop ecosystem [2]. Apache Parquet provides an efficient storage format for storing both nested or flat schema types. Furthermore, the Parquet file can be built by different encodings and compressions depending on the use case. Apache Parquet stores the in PAX model introduced in Section 2.2. In the following sections, we will discover the details necessary in this thesis.

Parquet File Organization As seen from Figure 2.6, parquet file is designed hierarchically. Briefly, there are multiple levels of parallelization. Firstly, parallelization can be achieved via multiple files. Then, MapReduce and Apache Spark frameworks can parallel multiple row groups in a single file. The deeper level of parallelization is done by columnar chunks which are residing in each row group. The final and deepest level of parallelization is at the page level. Each column has at least one page which holds the data associated with that column. The data residing in pages can have compression or be encoded by several different options.

Metadata There are three types of metadata: file metadata, column(chunk) metadata, and page metadata. The metadata is serialized via Thrift Compact Protocol. The metadata holds significant information regarding the size, file offsets, number of values, encoding type, compression type, column data type, and column statistics. The column statistics include the maximum and minimum value of the particular column, null counts, distinct counts. The statistics are held per row group and page. They are helpful for pushing down predicates.

Encoding The latest Parquet project has ten different encodings: plain, plain dictionary, RLE, bit packed, delta binary bit packed, delta length byte array, delta byte array, RLE dictionary, and byte stream split. The scope of this thesis is on plain and delta. Plain encoding, VarInt encoding, is used for primitive types as well as array types. It is the default encoding type. Moreover, delta encoding is

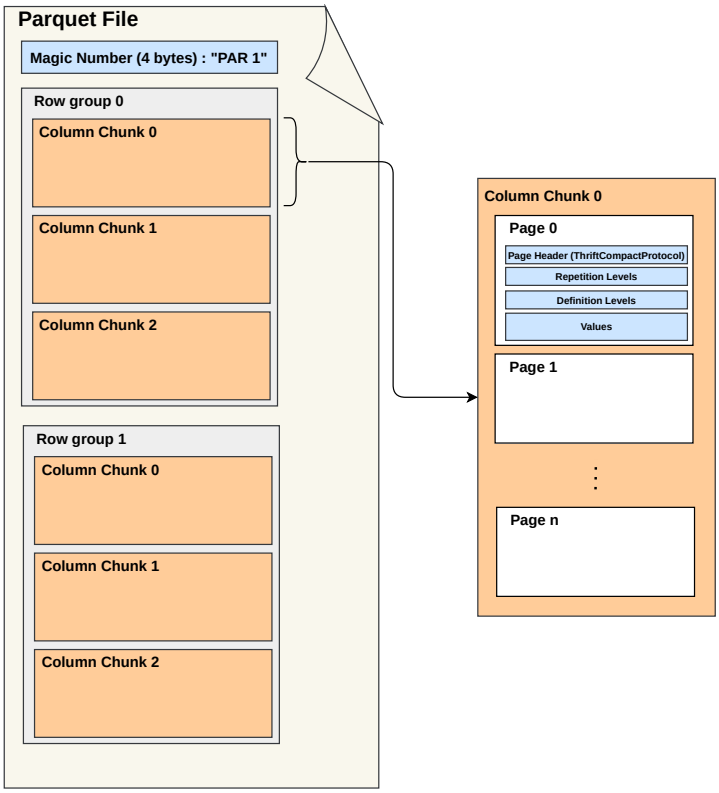


Figure 2.6: The layout of a parquet file

particularly helpful in storing the data in a smaller size. In delta encoding, the value is represented in a way that it is the difference between the current value and the previous value.

Compression Compression is an important technique helping the user to store large amounts of data. Apache Parquet in Apache Arrow project enables using several different compression schemes: snappy, brotli, gzip. In this project, the analysis in Chapter 6 uses snappy compression. The compression happens on the page level.

A dataset can be stored in multiple parquet files. The user can partition the dataset into chunks and store it in different files. The work in this thesis is designed for Parquet files with V1 and metadata serialized with Thrift V2.

2.4.3. Apache Arrow

Apache Arrow is a multi-language development platform for in-memory analytics. Apache Arrow leverages modern hardware with Arrow columnar format for moving data faster. Apache Arrow libraries introduce features such as Zero-copy shared memory and RPC-based data movement, reading or writing file formats as parquet, CSV and in-memory data analytics, and query processing [8]. In addition, the Apache Arrow format enables processing efficiency by leveraging scanning and iterating on large columnar chunks.

2.4.4. Fletcher

Fletcher is a framework developed by the ABS group in TU Delft and enables integrating FPGA accelerators with Apache Arrow back-end. Fletcher generates an easy-to-use hardware interface from Arrow Schemas. One can connect his/her accelerator kernel to these interfaces without spending too much time implementing interfaces for different interconnect types. The architectural overview of Fletcher is depicted in Figure 2.7. Fletcher provides important components such as *ArrayWriter*, *ArrayReader* to read and write to Arrow buffers.

The Fletcher runtime provides an easy-to-use software interface that is supported on different platforms. The user can easily copy Arrow Tables between the host memory or accelerator memory.

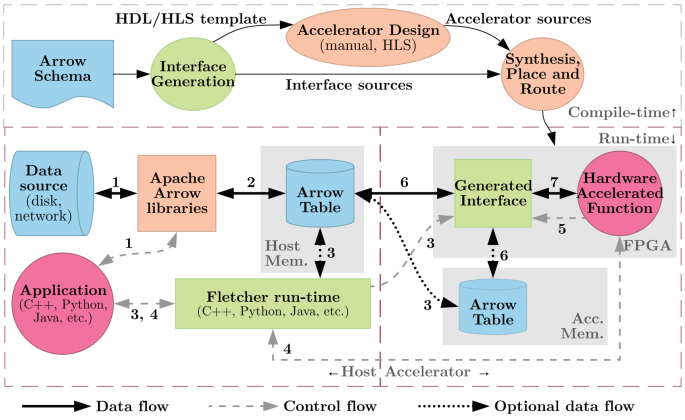


Figure 2.7: The architecture of Fletcher [49]

Moreover, the user can poke or peek MMIO register to control fletcher generated Hardware Accelerated Functions. By the time this thesis is written, fletcher has support for the following FPGA platforms:

- Amazon EC2 F1
- Xilinx Alveo
- Intel OPAE
- OpenPOWER SNAP (CAPI 2.0)
- OpenPOWER OC-Accel (OpenCAPI 3.0)

2.5. Benchmarks

2.5.1. Transaction Processing Performance Council - TPC

The Transaction Processing Performance Council is a non-profit corporation that aims to define vendor-neutral transaction processing benchmarks. There are 20 full members and four associate members, and seven professional affiliates. They are aimed to complement or help fulfill the TPC's mission.

TPC provides many different active or stale benchmarks for different perspectives of database systems. Figure 2.8 illustrates these benchmarks with their corresponding year. The active TPC benchmarks are denoted by the color green, whereas stale ones are coded as brown. TPC also contains supporting benchmarks, provided by blue in Figure 2.8.

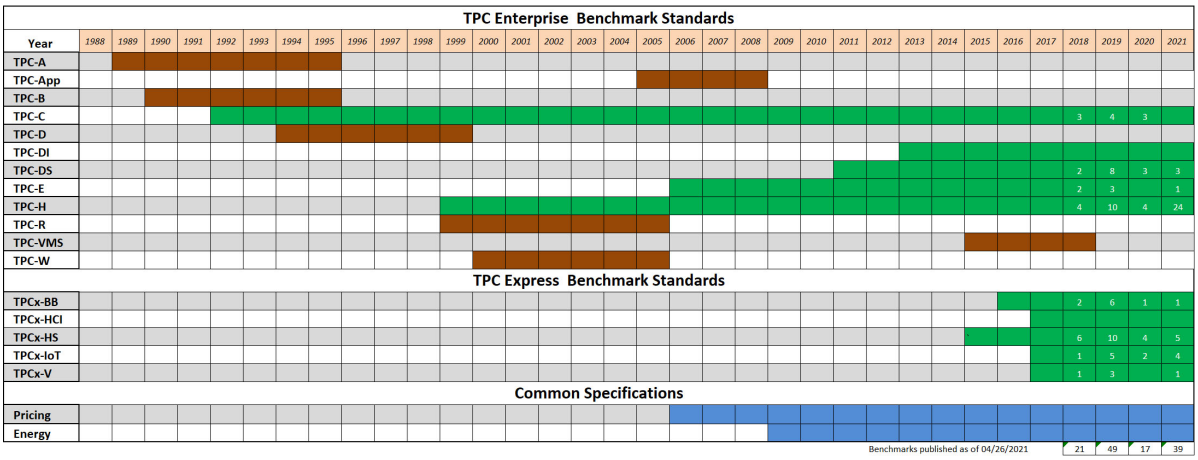


Figure 2.8: TPC Benchmark Timeline [54]

In this thesis, we concentrate mainly on decision support benchmarks, more specifically TPC-DS and TPC-H. The reason is that the scope of this work is integrated towards accelerator architectures for running database load and query operations. TPC Decision Support benchmarks consist of TPC-DS, TPC-H, and TPC-DI. The decision support benchmarks intend to provide a fair comparison between various vendor implementations by including controlled and repeatable queries evaluating the decision support systems.

2.5.2. TPC-H

TPC-H benchmark consists of business-oriented ad-hoc queries. TPC-H benchmark employs following conditions [21]:

- It gives answers to real-world questions;
- It consists of queries far more complex than OLTP transactions;
- It includes a large variety of operators and selectivity constraints;
- The queries in this benchmark are executed on a database server complying with the specific population and scaling requirements;
- The queries should generate intense activity on the part of the database.

TPC-H benchmark does not concentrate on the creation and retrieval of the data. Rather, this benchmark focuses on exercising system functionalities of complex business analysis applications. The business environment of TPC-H is illustrated in Figure 2.9

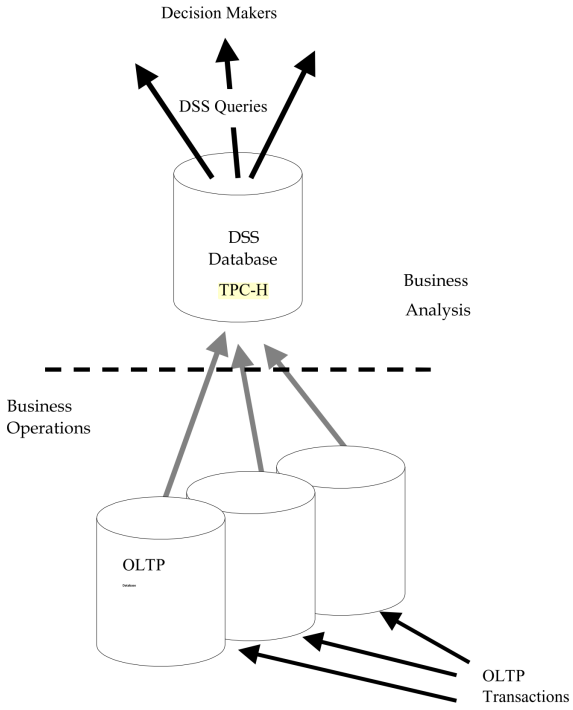


Figure 2.9: TPC-H business environment [21]

Database Architecture There exists 8 different tables which is visualized in the Figure 2.10. The number above each table represents the number of rows of each table. The scale factor (SF) is provided at the generation of the database. The schema of the TPC-H benchmark is modeled as the 3NF schema. Therefore, as discussed in Section 2.1, one can doubt the reality of the TPC-H database due to not investigating update, insertion, and deletion anomalies.

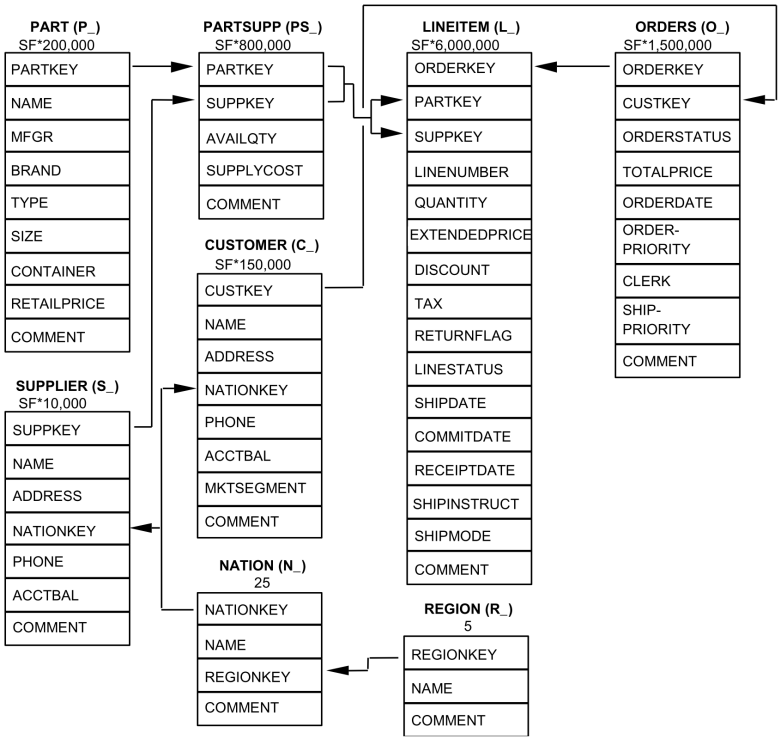


Figure 2.10: The TPC-H Schema [21]

Queries There are 21 different queries in TPC-H. Each query has:

- a business question, which illustrates the concept;
- functional query definition, which is in SQL-92;
- substitution parameters, which describes the value generation to complete the query syntax
- query validation, which describes how to confirm the query against qualification database [21].

For these types of needs, the TPC-H standard provides a query generation program to create the queries.

2.5.3. TPC-DS

TPC-DS benchmark is the successor of TPC-H and TPC-R benchmarks. The main focus areas of TPC-DS are:

- multiple snowflake schemas with shared dimensions to reveal the indexing techniques and query query optimizers of modern DSS system;
- ETL-like data maintenance;
- Sub-linear scaling of non-fact tables;
- Ad-hoc, iterative and extraction queries;
- More representative skewed database content [45].

In Figure 2.11, the components of TPC-DS benchmarks are illustrated. ETL operation is injected between the DSS database and files storing the dataset. In TPC-H benchmark, discussed in previous section, this type of operation is only visible in several queries and usually assumed that the data is clean [45].

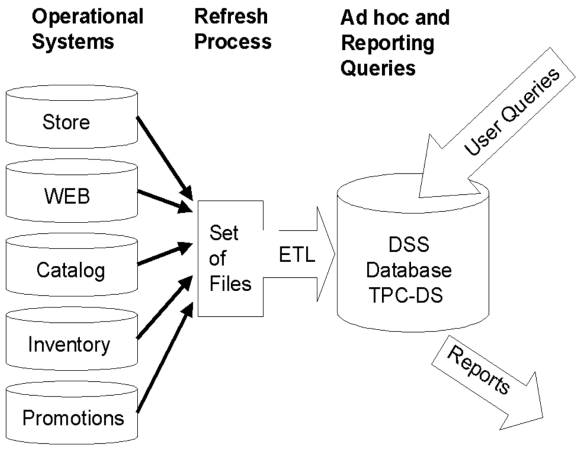


Figure 2.11: TPC-DS Components [20]

Database Architecture TPC-DS database includes seven fact tables that model the sales and sales return process for an organization that employs three primary sales channels. The database is designed as a hybrid of star schema and 3rd Norm Factor schema.

Queries TPC-DS queries contain two essential components, namely, user queries and data maintenance queries, as illustrated in Figure 2.11. TPC-DS offers 99 distinct SQL user queries with OLAP extensions. There are four different categories: reporting, ad-hoc, Iterative OLAP, and ETL. The result type and quantity are known for reporting queries, enabling data placement methods to optimize query execution. However, ad-hoc queries disable these optimizations. The ”data placement optimizations” were the chokepoint of the TPC-H benchmark, which enabled the users to optimize the execution in advance. That is why the TPC-DS benchmark offers a more realistic comparison.

2.6. Related Work

Accelerating big data analytics workloads over FPGAs has been seen over the literature for foreseeable reasons, such as achieving hardware parallelism, dynamic reconfigurability, and low power consumption [29]. Moreover, there are commercially available hardware acceleration engines such as IBM Netezza[35], AWS-FGPA [6], Azure FPGA [9], Intel Open FPGA Stack [38]. There are many academic efforts in terms of hardware acceleration of database operations as they prove to achieve higher throughput and better energy efficiency [34], [29], [52] [40]. The performance improvements of FPGA accelerators can be classified in terms of the type of the operator used for the kernel, what type of accelerator is used for design, Power9 system or AWS-FPGA and what kind of database is used to integrate the design: Postgresql, Apache Spark, MySQL.

Since Apache Spark provides an extensible architecture and optimized execution model which can parallelize workloads in Spark RDDs, there have been several studies investigating acceleration opportunities of Spark SQL workloads on FPGAs. Wasai Technology has proven that TPC-DS query 55 can be accelerated up to 50x for dataset size of 300 GB with Arrow-based accelerator design [15]. Nonnenmacher was able to accelerate Spark SQL workload, which consists of regular expression matching filter and aggregation, on FPGAs by achieving 13x speed-up. Hoozemans et al. investigate the FPGA acceleration of big data analytics and surveys the programming practices[34]. Ziener et al. investigate FPGA-based dynamically reconfigurable SQL processing and reports performance estimations for individual operators [66].

PostgreSQL has existed for almost 30 years. There are many database solutions already integrated with PostgreSQL. Swarm64 offers a database accelerator engine that extends PostgreSQL. They have reported that on 1TB of the dataset, TPC-H benchmarks can be accelerated by more than 7x to 60x [1]. Becher et al. proposes ReProvide as an FPGA+CPU acceleration framework and achieves up to 40% reduction in execution time [12]. ReProvide is integrated with PostgreSQL.

Sidler et al. present a hardware-accelerated database with CPU+FPGA platform with shared memory by extending MonetDB[52]. Woods et al. demonstrated a storage engine, IBEX, pushing down

complex query operators to storage [57]. IBEX uses a host system with MySQL and accelerates TPC-H query 13 by 11x, query 5 by 2.5x. The IBEX storage engine connects FPGA to SSD by SATA and the host CPU. Sun et al. introduces a row-based storage engine implemented on AWS F1 instances and shows 2.8x computation speed-up for TPC-H Query 6 [53]. The integration is done on MySQL.

In terms of moving FPGAs to the storage and making use of Smart SSDs, Do et al. proposes a query pushdown with Smart SSDs using CPU and achieve 1.7x speed-up for TPC-H Query 6 over traditional SSDs[24].

This thesis also investigates the scheduling of FPGA resources in a multi-threaded environment. This effort is relatively more straightforward than exposing processing units on FPGA as CPU threads, but it is necessary to review state-of-art solutions. Kara et al. propose PipeArch that can run threads on FPGA resources and enables context-switching [39]. Centaur provides a framework to run FPGA operators independently and concurrently [48].

3

Case Study On TPC-H and TPC-DS

3.1. Motivation

Hardware Design for FPGAs introduces a challenging environment due to the complexity of the design process. Therefore, it is crucial to analyze the queries one desires to accelerate before jumping into the design process. As addressed in Section 2.6, there are numerous efforts in accelerating different queries from well-known benchmarks. The scope of this thesis is to demonstrate the possible real-life performance once FPGAs execute a full query (as opposed to accelerating a query by offloading a part of it to FPGA). Therefore, in the following sections, we will employ a case study of some of the best-known benchmarks to evaluate their ability for acceleration.

Finally, the integration of such an operation with Apache Spark will be discussed.

To analyze a query in terms of its complexity and the required effort to implement it on custom hardware, we should define a metric independent of the implementation of the accelerator and defines operators as a black-box entity. Moreover, this metric we use should be independent of the language we are using. The main reason for such independence is that we are operating on heterogeneous computing platforms, which may require multiple different languages. Software architects measure the programming complexity of the software by using several different tools. One of them is *Halstead Measurements*, which Maurice Halstead introduced in 1977.

3.2. A Brief Complexity Analysis

Straightforwardly, software consists of operators and operands. To estimate the complexity of such a program, we should analyze not the part of the program but the overall source code. Therefore, this analysis provides a static way of defining the vocabulary, testing time, programming effort, etc. The operators and operands in a SQL program can be visualized in the following Listing 3.1.

```
1 select
2   sum (l_extendedprice * l_discount)
3 from
4     parquet. l_plain.prq
5 where
6   l_shipdate >= date '1994-01-01'
```

Listing 3.1: An example SQL code visualizes operators(red) and operands(black)

The main components of Halstead measures are the following:

$n1$ Number of unique operators

$n2$ Number of unique operands

$N1$ Number of total occurance of operators

$N2$ Number of total occurance of operands

Halstead measures define the metrics based on these components. A related metrics to our use case can be summarizes as follows:

Difficulty(D)
The difficulty level of the program.

$$D = \frac{n1}{2} \times \frac{N2}{n2} \tag{3.1}$$

Volume(V)
The memory space used by the program.

$$V = (N1 + N2) \times \log(n1 + n2) \tag{3.2}$$

Estimated Program length (L)
This metric is defined as the total weighted summation of the operands and operators.

$$L = n1 \log n1 + n2 \log n2 \tag{3.3}$$

Programming Effort (PrE)
The difficulty level is proportional to the number of unique operators.

$$PrE = D * V \tag{3.4}$$

Once we think about dataflow design architectures, we can represent each operator by a block and create a dataflow graph to represent the design. Then, this would make easier to understand the critical path of our design. Yet, this is hard when we try to choose from 120 different queries (99 TPC-DS + 22 TPC-H). Therefore, the software designers also make use of a complexity measure called *Cyclomatic complexity*, which is defined by the Equation 3.5. In queries we desire to accelerate, there is only one query and accelerated kernel. Hence, we can take it as one and equation becomes as Equation 3.6.

$$Complexity = (NumberofEdges) - (NumberofNodes) + 2(NumberofDisconnectedParts) \tag{3.5}$$

$$Complexity = (NumberofEdges) - (NumberofNodes) + 2 \tag{3.6}$$

There are two assumptions we make. First, there is no parallelism, and analysis is done before the SQL optimizer.

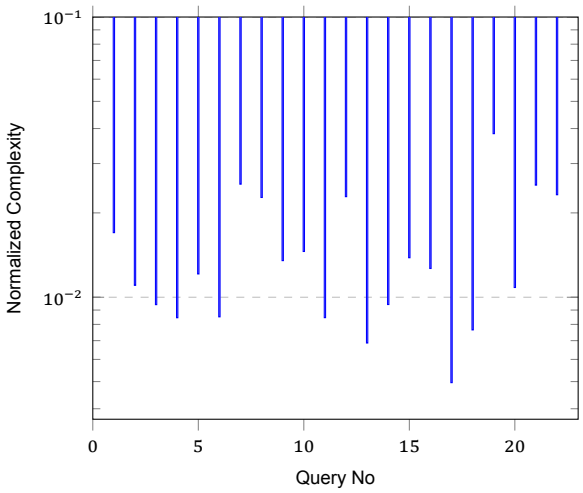
This kind of effort is essential to estimate a total query acceleration development effort for decision support benchmarks. Moreover, it provides a solid understanding of the operator densities of each query in TPC decision support benchmarks. Usually, we understand the performance of our hardware design once we have the synthesis and simulation results. We can obtain the critical path delay and area from synthesis results. However, this is very hard in scaling for 120 different queries as it involves designing possibly 120 different accelerators. We can use the metrics explained above to get an estimation of their performance.

This study combines the metrics explained above and creates a metric to measure both computational complexity and complexity of the control flow, *Cyclomatic Complexity*. We will define computational complexity in a way Halstead defines programming effort. The only difference is that the sort and join operations will be explicitly logged as their complexity is dependent on the problem size. The complexity of joins is calculated by taking the complexity of a HashJoin operator, which is $O(n)$. For the complexity of sorts, we will use a merge sorter [65], which has the complexity of $O(n \log n)$. Therefore, the discussion will be made according to both dependences. The formula is given by Equation 3.8.

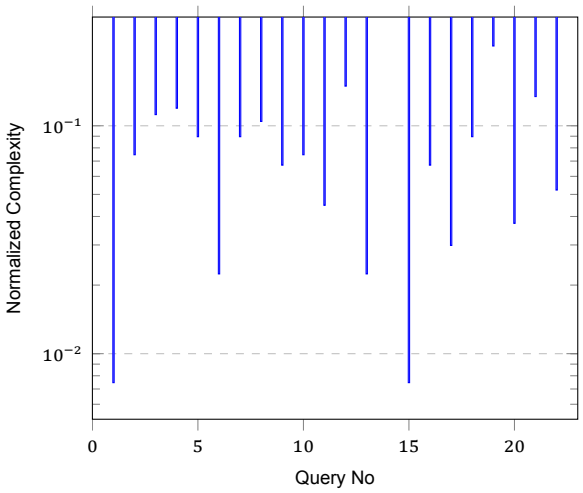
(3.7)

$$Computational\ Complexity = (\frac{n1}{2} \times \frac{N2}{n2}) \times (N1 + N2) \times \log(n1 + n2) \tag{3.8}$$

$$Cyclomatic\ complexity = (Number\ of\ bitwise\ operations\ in\ WHERE) + 1 \tag{3.9}$$

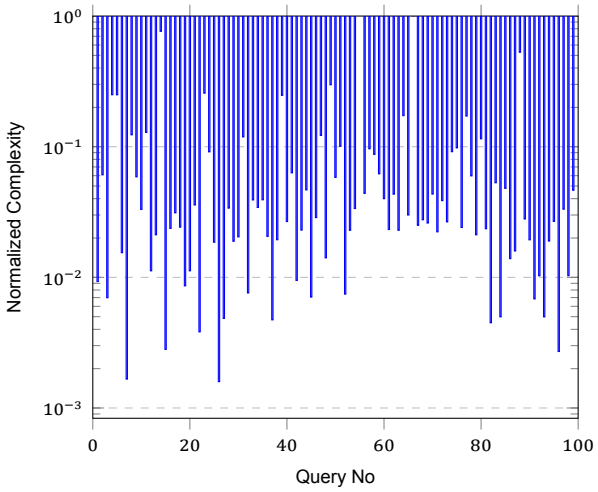


(a) Computational Complexity of TPC-H Queries

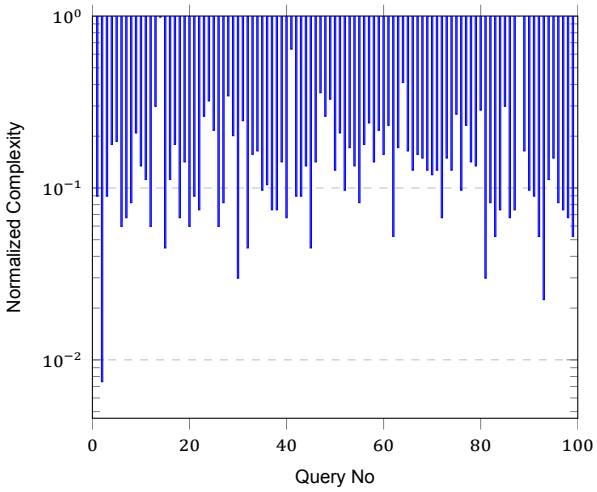


(b) Cyclomatic Complexity of TPC-H Queries

Figure 3.1: Complexity Analysis on TPC-H Queries



(a) Computational Complexity of TPC-DS Queries



(b) Cyclomatic Complexity of TPC-DS Queries

Figure 3.2: Complexity Analysis on TPC-DS Queries

The complexity analysis depicted in Figure 3.1 and 3.2 are normalized by a joint function. This implies that the complexity values of each dataset are normalized under the same maximum and minimum. The following equation is applied for normalization.

$$norm_h = \min(TPC - H(n)) \tag{3.10}$$

$$norm_d = \min(TPC - DS(n)) \tag{3.11}$$

$$norm = \frac{x - \min(norm_h, norm_d)}{\max(norm_h, norm_d) - \min(norm_h, norm_d)} \tag{3.12}$$

As visible in Figure 3.1a, query 16 and 6 gives the least possible computational complexity, whereas query 18 has the highest computational complexity. The cyclomatic complexity of query 18 is the highest amongst TPC-H queries as visible in Figure 3.1b. On the contrary, TPC-H query 1, 15 offers the least cyclomatic complexities. In terms of TPC-DS queries, one thing to remember is that they have much higher complexities than TPC-H queries. What can be seen in Figure 3.2a is that TPC-DS query 54 has the least computational complexity, whereas query 65 is the highest amongst both benchmarks. The peak cyclomatic complexity occurs in query 87, whereas the minimum is query 1, as depicted in Figure 3.2b.

3.3. Discussion

Our initial objective was to find the query with the most straightforward implementation and the best acceleration capability. This implies that the chosen query should have the least computational and cyclomatic complexity. Also, this query should no have any join or sort operations for the sake of the design time of such dataflow hardware. That is why *TPC-H Query 6* is chosen. TPC-H Query 6 is a no-join query with single aggregation. The query is more I/O intensive than the others as it does not have any group by or order by operators. Restating the objective of this thesis, we are investigating the capabilities of FPGAs with pushing down aggregation to file reading operation. Therefore, I/O intensive queries should provide us with a better acceleration as the file copies will take a significant portion of the execution time.

4

Extending Parquet File Reading Operation on Apache Spark

Apache Spark can process several different types of big data formats. However, the focus of this thesis is the parquet file format. Therefore, in this section, we will investigate the parquet reading operation of Apache Spark. As explained in section 2.4.2, the parquet file has a hierarchical logical design that enables several different levels of parallelism. Therefore, the amount of parallelism depends on the type and size of the parquet file being read. Moreover, cluster configuration also affects the reader performance as expected.

In the following sections, firstly, the operations of Apache Spark API will be explained for parquet file reading operation. Then, we will discuss how possible extensions to the parquet file reader API can be done. In the second section, the proposed architecture will be revealed. Finally, we will discuss how to make use of the available parallelism.

4.1. Apache Spark Parquet Scan Operation

Apache Spark can read or write parquet files. Due to the focus of this thesis, the parquet reading operation will be under examination. The Listing 4.1 shows a simple parquet file reading example with Apache Spark API. First, *lineitem_schema* represents the schema of the parquet file we are reading. Then, in the next *Spark. Read* call, and we read the parquet file in the DataFrame. As explained in section 2.4.1, DataFrame API distributed a collection of rows with the same schema.

```
1  val lineitem_schema = StructType(Seq(
2      StructField("l_orderkey",IntegerType,false),
3      StructField("l_partkey",IntegerType,false),
4      StructField("l_suppkey",IntegerType,false),
5      StructField("l_linenumber",IntegerType,false),
6      StructField("l_quantity",DoubleType,false),
7      StructField("l_extendedprice",DoubleType,false),
8      StructField("l_discount",DoubleType,false),
9      StructField("l_tax",DoubleType,false),
10     StructField("l_returnflag",StringType,false),
11     StructField("l_linestatus",StringType,false),
12     StructField("l_shipdate",DateType,false),
13     StructField("l_commitdate",DateType,false),
14     StructField("l_receiptdate",DateType,false),
15     StructField("l_shipinstruct",StringType,false),
16     StructField("l_shipmode",StringType,false),
17     StructField("l_comment",StringType,false)
18 ))
19 val lineitemdf = spark.read.schema(lineitem_schema).parquet(config.file_path).
       createOrReplaceTempView("lineitem")
```

Listing 4.1: An example API calls for parquet file reading of TPC-H lineitem Table

Apache Spark enables reading multiple files in case the user provides a directory path. Therefore, the same kind of operations as listed in Listing 4.1 will automatically extract partitioning information from paths. Apache Spark API provides the following properties for parquet files:

Predicate Pushdown Once the user enables this optimization, the Spark API pushes a filter into the parquet reader.

Snappy compression Spark API can operate on compressed or decompressed parquet files.

Merging schema Once multiple files are being read. Spark can merge schemas of these files.

Being backward compatible Spark can handle older parquet files.

Low-level Overview

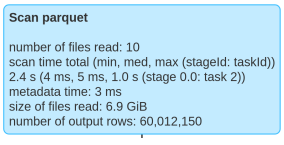
In order to analyze the possibility of extensions to Spark API, we should further examine how Spark handles parquet files and distributes the workload. As explained in Section 2.4.1, Spark SQL generates the code after a number of optimizations. Before generating the code, Spark SQL translates the query into a physical plan. The physical plan of Parquet reading operation is provided in Listing 4.2.

```
1 +- FileScan parquet [l_quantity#4,l_extendedprice#5,l_discount#6,l_shipdate#10] Batched: true
    , DataFilters: [isnotnull(l_quantity#4), isnotnull(l_shipdate#10), isnotnull(l_discount
    #6), (l_shipdate#10 >= 87..., Format: Parquet, Location: InMemoryFileIndex[file:/dev/shm
    /20sf/l_plain.prq], PartitionFilters: [], PushedFilters: [IsNotNull(l_quantity),
    IsNotNull(l_shipdate), IsNotNull(l_discount), GreaterThanOrEqual(l_shipda..., ReadSchema:
     struct<l_quantity:double,l_extendedprice:double,l_discount:double,l_shipdate:date>
```

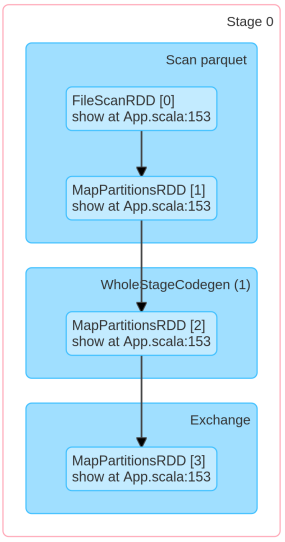
Listing 4.2: Physical Plan generated by Spark API

Apache Spark generates a FileScan operator to read parquet files as seen in Listing 4.2. *FileSourceScanExec* node scans the node from HadoopFsRelations. The scan node supports reading in both *ColumnarBatch* and *InternalRow* formats. As seen in Figure 4.1, once the scan operator starts, the *doExecute()* call is performed and Spark performs reading and mapping the scan results in partitions. In order to read the metadata values, which hold important information about the parquet file, Apache Spark uses Hadoop internals. After reading metadata, Apache Spark gets the number of row groups and parquet files to parallelize tasks.

Spark executes *doExecute()* to map *InternalRows* into RDDs. The *doExecute()* call returns a *FileScanRDD* object which creates a wrapper for *Reader*. Reader is created by the *buildReaderWithPartitionValues()* call. *buildReaderWithPartitionValues()* extends to an iterator for *RDD[InternalRow]*. Spark



(a) *FileScan* node details regarding to File Type, size and quantity



(b) *FileScan* Operator internal structure being run by tasks

Figure 4.1: Apache Spark *FileScan* Operator Details

History server shows the parquet scan operators file statistics as provided in Figure 4.1a. Individual parquet scan stage, shown as Stage 0 in Figure 4.1b, will be replaced by fletcher scan stage for accelerated workers.

4.2. Injecting custom Parquet File Reader

Nonnenmacher 2020 [46] proposes a solution for injecting accelerated file reading, filter, and aggregation operations as Apache Spark Rule. Since we are trying to emulate this behavior only on a single accelerated node with file reading and query execution, it is possible to use his work as a baseline to inject one single node that runs an FPGA instance. The higher-level overview of the injected custom node is presented in Figure 4.2. The way we inject the file reader is that we build *FletcherParquetReaderIterator*, which interfaces to *Iterator<InternalRow>*, by overriding the call *buildReaderWithPartitionValues()* as depicted in Figure 4.3.

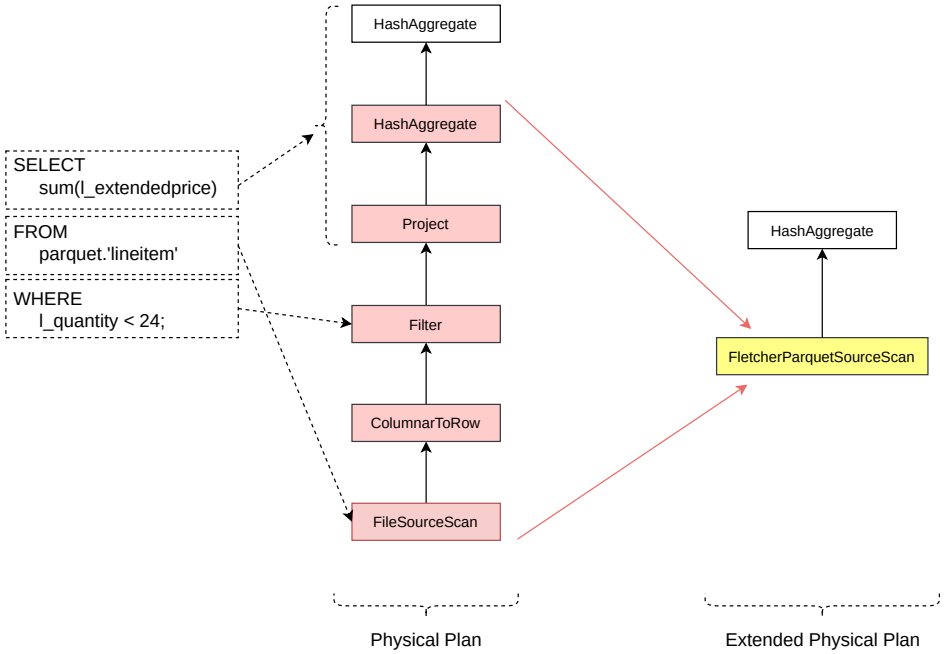


Figure 4.2: Extending the physical plan of Apache Spark

4.3. Concurrency and Parallelism

To scale the application up, one should discuss the opportunities to increase the system's parallelism and concurrency. Before moving on with the architectural mechanisms to achieve such systems, we should define concurrency and parallelism. *parallelism* is defined as at least two threads executing simultaneously. *Concurrency* is a more generalized form of parallelism which can include time-slicing [47].

Every higher-level framework integrates solutions of its own to achieve concurrency and parallelism. Luppes 2021 [42] studied acceleration for DASK distributed workers and argued that single node setups are more preferred as they have higher community exposure. In other words, community prefers to use single worker setups.

The focus of this thesis is on Apache Spark. For this reason, the concurrency and parallelism of Apache Spark with accelerated workers will be discussed. Apache Spark achieves optimizing distributed applications on constrained resources very well. However, the concurrency creates complexity to software design all by itself. Nevertheless, Apache Spark can handle concurrency for multiple workers successfully while abstracting user from underlying complexity.

Parallelism in Apache Spark

The parallelism in Spark can be achieved via several different levels. The user can achieve

parallelization by using Apache Spark API and Scala Concurrency API. As described in Section 2.4.1, Apache Spark distributes the RDDs in the cluster. One level of parallelism is achieved by *parallelizing partitions*. The user can assign multiple executors to each partition and achieve higher throughput for their application.

Concurrency in Apache Spark

There are two different levels of execution to consider once we examine the concurrency in Apache Spark.

Firstly, Apache Spark can schedule resources between different Spark applications. As described in Section 2.4.1, each Spark application having their own context and independent set of executors. The *cluster manager* can statically or dynamically allocate resources to applications. In static resource allocation, each application holds the data distributed by a fixed amount during their runtime. In dynamic resource allocation, the resources are distributed as if they were in static allocation, but they can be shared across other applications if there is no application running on them.

Secondly, the concurrency can be achieved within an application. This is achieved by scheduling multiple parallel jobs on a single spark context. The user can define *spark.scheduler.pool* per context and *fairly* submit jobs in these pools.

In our context, we are trying to achieve heterogeneous resource scheduling within Apache Spark's File Reading operation. The underlying resource is FPGA. In order to optimize parallelized accelerated workers(FPGA), the kernel design should achieve computations with high throughputs. FPGA design should be highly optimized and used without any overhead. We will address the dataflow hardware design of this operation in the later chapters. However, software integration is the topic of this chapter. The designer should be fully aware of the software calls without blocking operations, introducing too much overhead. Copying the data between CPUs and FPGAs is one of these operations with too much overhead. One way to overcome this is by overlapping FPGA computation with data transfers between CPU and FPGA.

Moreover, the designer can configure the cluster to schedule FPGA resources for Spark applications (each running Spark Context). More specifically, the cluster setup should have one or more FPGAs and multiple Spark applications dynamically or statically allocate them. In the case of users configuring these multiple applications to run on multiple FPGAs so that each application has one FPGA and CPU, the control is easy to achieve. Whereas, if the cluster has only one FPGA and multiple Spark Contexts, the concurrency is very hard to achieve. The reason is the following. FPGAs use system interconnects such as OPENCAPI, DMA, and MMIO for communicating with CPUs. Therefore, multiple processes writing or reading the control registers on FPGA require keeping track of these resources by the *cluster manager*. One can argue that it should be plausible to run in *local-cluster mode*, which means running multiple worker nodes on a single machine. The older versions of Apache Spark enables creating multiple worker instances on a single machine. However, it is deprecated in newer versions of Spark on a single machine based on the discussion of no use case exists. Therefore, it is desirable to run workers on all resources in a cluster-mode setup and divide those resources on multiple executors. In order to prove this reasoning, an example application is provided in Appendix A.

The scope of this thesis is to achieve concurrency on local worker threads on a single Spark Context and propose a proof-of-concept solution for extending this design for cluster setups with multiple accelerators. In the next section, the concurrent software architecture with single FPGA multiple Spark threads will be investigated.

Concurrency and Parallelism FPGAs

The concurrency of the FPGA concerns the control of the FPGA and memory. As explained in Section 2.4.4, fletcher provides an abstraction for controlling registers and memory actions. In order to achieve concurrency, an extensible native wrapper for fletcher runtime is proposed in this section. In order to implement the desired functionalities, features of the C++ language are used. The functionalities can be summarized as such:

- The wrapper should provide the abstraction and interfaces to be usable with fletcher runtime.
- To achieve the high throughput that system interconnect offers, the wrapper should use more CPU resources and parallelism.

- The design should have properties to schedule multiple tasks once the resources are a constraint.
- The threads launched at the beginning of the application should not stay idle or idle for a minimal amount of time.
- Preemption is not a requirement. In simple terms, kernel wait times are relatively small(less than 1000ms). Therefore, there is no need to introduce any other overheads from context switches.

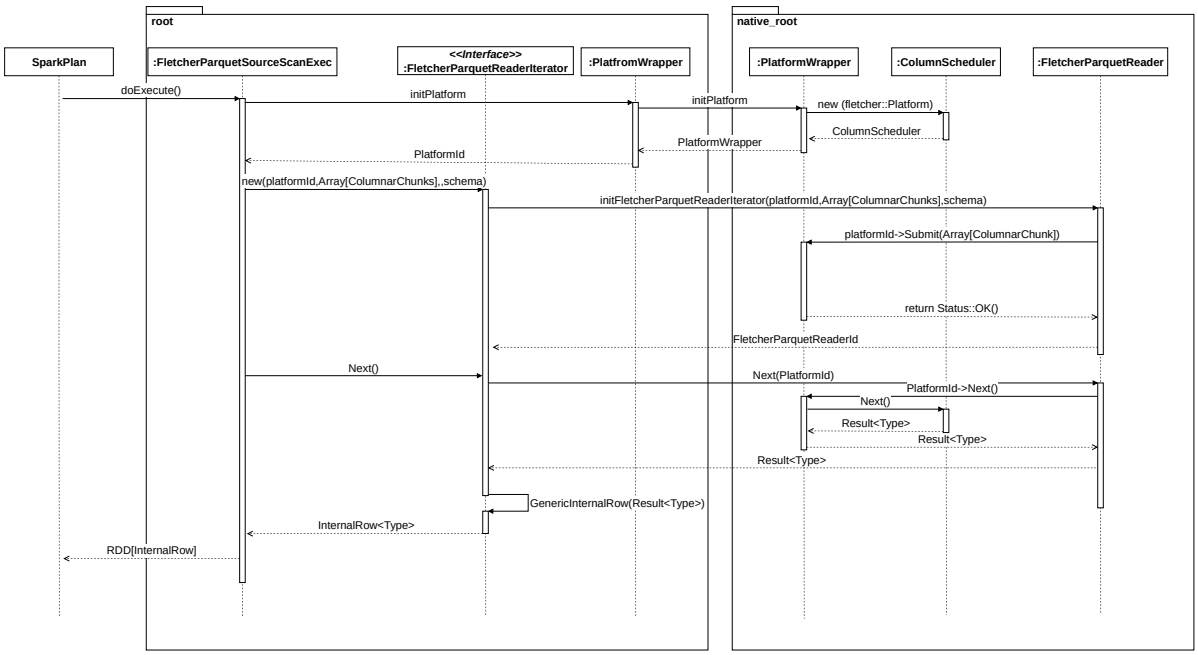


Figure 4.3: Complete Software Design

There are 3 main components of Native Software API which are visualized in Figure 4.3. These components are *FletcherParquetReaderIterator*, *PlatformWrapper*, *ColumnScheduler*.

ColumnScheduler The main objective of this class is to provide the concurrency features addressed earlier. Several different components help to manage this multithreaded environment.The overall software architecture for *ColumnScheduler* is provided in Figure 4.4.

Before moving forward, we should remember the terminology this section makes use of. In this section, once we define threads, it is meant the system-level threads. Tasks are the minor component of execution that threads run and have a *promise of a future*. A simple task for FPGA runtime is provided in Listing 4.3.

```
1 double Task(Platform platform)
2 {
3     Reset(platform);
4     setFPGARegisters(platform, regs);
5     Start(platform);
6     WaitForFinish(platform);
7     return ReadResult(platform);
8 }
```

Listing 4.3: An example task for FPGA execution

ColumnScheduler instance implements three different design decisions: Spinlocks, Thread Safe Fletcher Platform, Thread Safe Task Queue and Thread Pool.

Spinlock

When multiple threads call *platformWriteMMIO()* API Call to manipulate action registers, they cannot do it at the same time for the sake of eliminating race conditions. Therefore, the idea is to use synchronization between multiple tasks. There are several ways to achieve synchronized

access, such as mutexes and semaphores. We should explicitly address that we do not seek preemption, as stated earlier. Moreover, since mutexes usually yield the thread and wait for preemption from other threads, using mutexes would introduce more overhead. Therefore, we only need a *spinning* lock, waiting for another thread to release the lock. This is achieved via boost *spinlock* implementation.

Thread Safe Fletcher Platform
It is important to ensure that the *spinlocks* are carefully injected into the fletcher platform. In terms of this effort, the spinlock is locked only before accessing MMIO registers and unlocked after access to MMIO registers is completed.

Thread Safe Task Queue
The architecture of this mechanism is visualized in Figure 4.5. There are two different implementations of Thread Safe queues for thread-local queueing and global queueing. The implementations are very similar to the ones introduced in the literature [56].

Task Queue
Task Queue is a global queue that holds the idle tasks to be executed in the future.

Work Stealing Task Queue
One of the requirements of this implementation was to keep FPGA always busy. If a thread finishes with the execution of the task assigned, it pops the next task that is waiting to be administered. For this kind of mechanism, this thesis introduces a work-stealing task queue using C++/JNI constructs. This queue implementation is thread-local.

Thread Pool
Thread Pool is the main wrapper that handles this queuing logic. By *ThreadPool* class, *ColumnScheduler* can *Submit* tasks without handling any logic. There are few important class functions to keep in mind. Firstly, *Submit(&Task)* submits the task to thread-local queues if they exist. If those queues do not exist, it submits the task to the global queue. *Submit(&Task)* returns future of the task back to *ColumnScheduler*. Secondly, *run pending task* handles the task assigned to the threads. The hierarchy of assigning tasks is as follows:

1. Use thread local queue
2. Use global queue
3. Steal from other threads local queue

PlatformWrapper A single FPGA instance should initialize a single platform. Platform construct in both Fletcher and here is defined as a medium to attach to the FPGA card through low-level software API. For instance, in *oc-accel, Power9*, reading snap card information via multiple *snap_card_ioctl* calls and attaching action by *snap_attach_action* call, should only be done once in a single application run. This class implementation, as visualized in Figure 4.3, is responsible for initializing only a single platform throughout the runtime of an application. It is imperative to suppress the copy operation of Platform wrapper instances. In case of any copy of *PlatformWrapper* instance, higher-level software APIs will act as it has attached to multiple FPGAs.

As seen in Figure 4.3, *PlatformWrapper* has a Scala wrapper to call it when necessary. The Scala wrapper is designed as a companion object so as to emulate *PlatformWrapper* static. Apache Spark API calls for the read operation attaches to the Scala companion object of single *PlatformWrapper* and submits the task to the same *ColumnScheduler*. In this way, we achieve to parallelize files that hold different partitions in Spark Context.

4.4. Preliminary Conclusion

In this section, the software design of query pushdown for parquet reading operation is identified. In summary, this chapter describes two main design challenges. Firstly, o accelerated node should be injected instead of the Apache Spark parquet scan operator while keeping the functionalities that the Apache Spark parquet scan operator offers. These functionalities are summarized as follows:

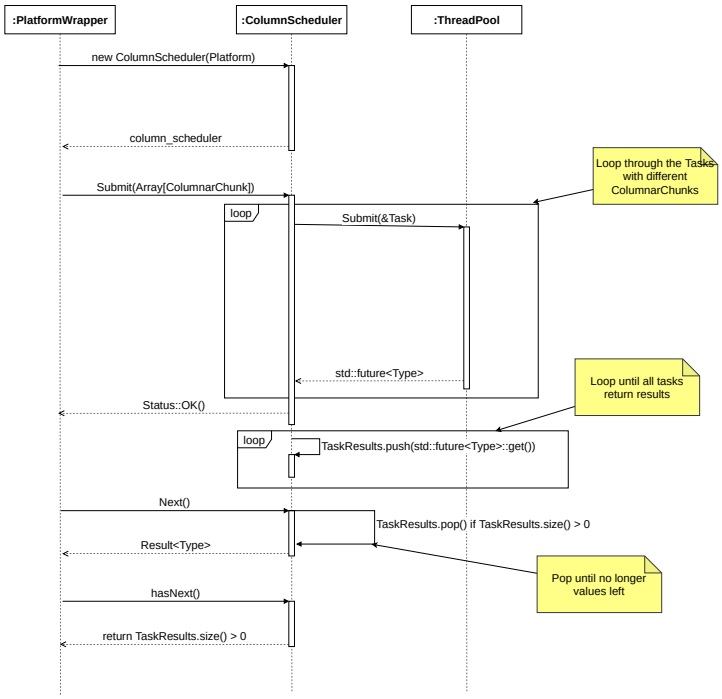


Figure 4.4: Software Architecture of *ColumnScheduler*

- Parsing metadata of parquet file using Hadoop internals
- Parallelizing the parquet scan workload with respect to row groups and file quantity
- Preserving the partitioning logic of Apache Spark parquet source scan operator

We were able to accomplish to propose a design, preserving these functionalities. The only difference of the proposed implementation from Spark's parquet scan operator's implementation is that the parallelization of workload is achieved via the number of files on Apache Spark Plan. Nevertheless, parallelization of row groups is transitioned to one level deeper to fletcher processor native program as depicted by *ColumnScheduler* class in Figure 4.3.

The second challenge was to create an FPGA wrapper that can schedule Spark tasks to FPGA. For this purpose, a concurrent and parallel platform wrapper,*PlatformWrapper* and *ColumnScheduler*, has been proposed. This module helps design to achieve higher throughput by using multithreading on FPGA runtime. Using multithreaded native runtime software, Apache Spark was able to submit tasks to the native thread pool.

In the chapter that follows, the hardware design of such pushdown implementation will be discussed.

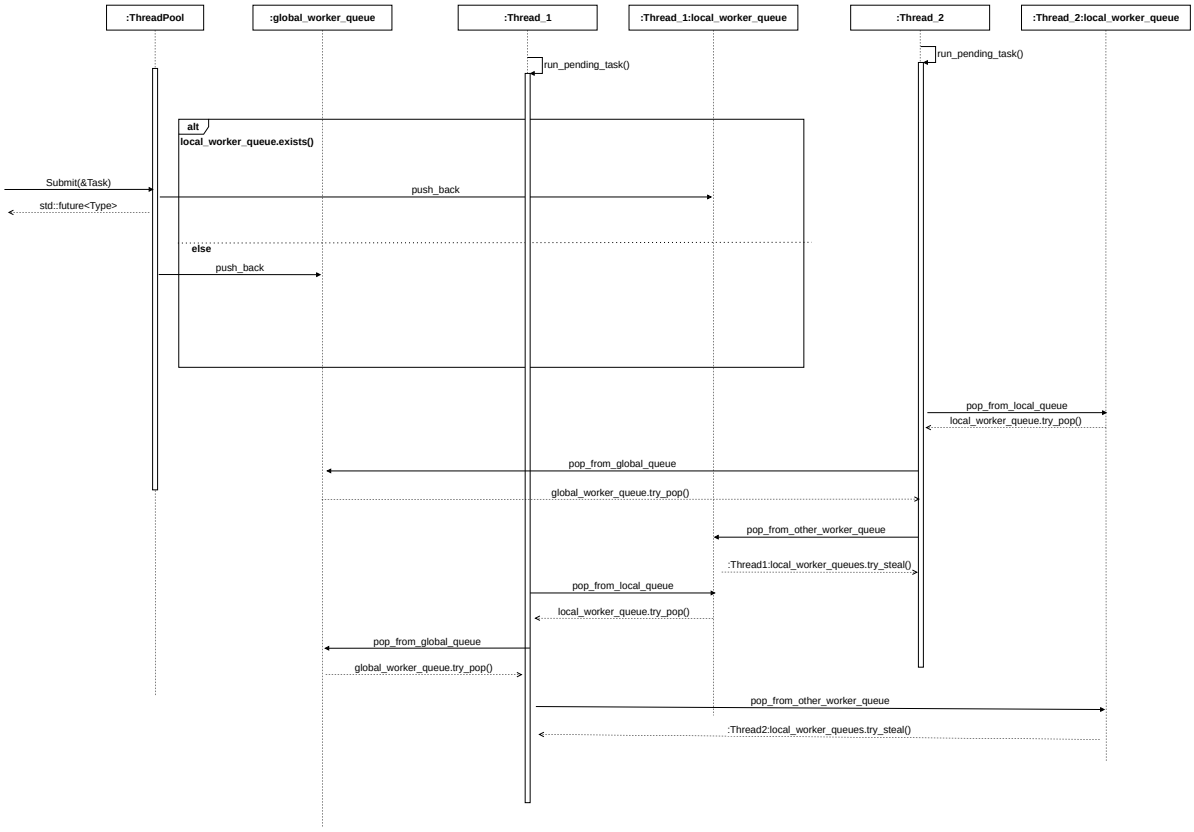


Figure 4.5: Software Architecture of Thread Safe Thread Pool

5

Hardware Design

This chapter investigates the hardware design in two different sections. Firstly, we will investigate the extensions to the state-of-art parquet reader. Then, we will design TPC-H Query 6 on hardware.

5.1. Parquet Reading on FPGA

Peltenburg et al. and Van Leeuwen 2018 propose a high throughput parquet to arrow converter on FPGAs [50], [55]. This design is particularly important as it provides an implementation to overcome parquet reading bottlenecks while injecting FPGA in between the host CPU and storage as depicted in Figure 5.1. The overall architecture is visible in Figure 5.2. The design achieves 12 GB/s end-to-end throughput by utilizing less than 5% of the resources at FPGA. Briefly, there exist eight different components in the architecture.

Ingester: This module is responsible for loading pages from memory with large bursts.

Aligner: The Aligner includes a pipelined barrel shifter and history buffer. The aim is to preserve the alignment of the data stream.

Metadata Interpreter Metadata Interpreter parses the metadata which Apache Thrift Serialization Protocol generates. The implementation offers interpretation for versions 1 and 2. The page metadata parsed in this module includes valuable information such as uncompressed size, compressed size, number of values.

Values Decoder: The values decoder decodes the data of the column. There exist two crucial components, namely Decompressor and decoder. **Decompressor** is a snappy decompressor wrapper that decompresses the compressed files. **Decoder** module decodes the column data in case any kind of encoding feature is present in the parquet file. Different kinds of encoding schemes are presented in Section 2.4.2. The supported types are *PLAIN*, *BIT_PACKED DELTA* and *MIXED*.

Fletcher Array Writer: This module is part of the fletcher framework [49], acting as a DMA engine to write hardware streams to in-memory arrays in Arrow format.

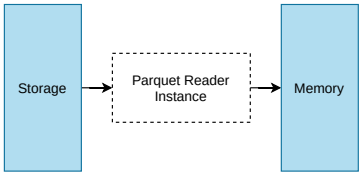


Figure 5.1: Higher Level design proposed by Van Leeuwen 2019 [55]

The architecture can decode parquet files with the configurations explained above. This architecture allows Parquet to Arrow conversion of only one column. This research investigates capabilities of this

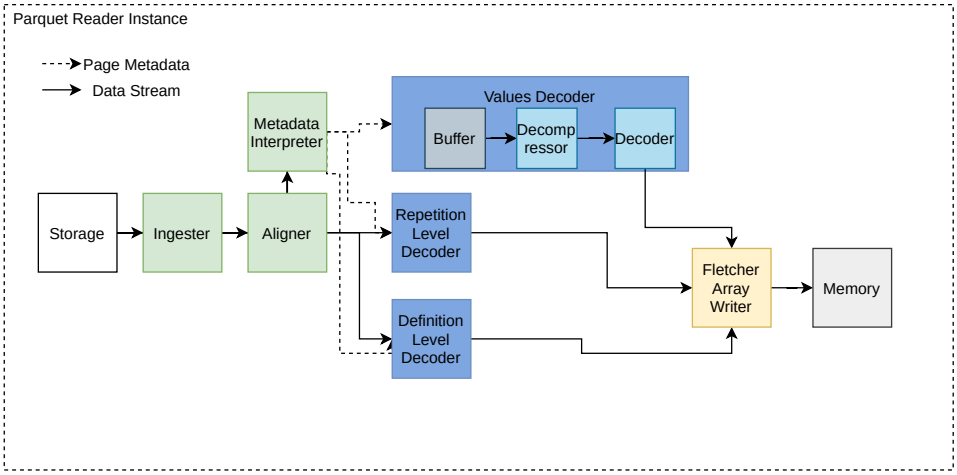


Figure 5.2: Architectural overview of Parquet to Arrow Converter Accelerator [50] [55]

hardware implementation to extend it for multiple columnar readers. Moreover, a query pushdown integration is proposed. The parquet column reader hardware implementation only uses 5% of the FPGA resources which means one can simply add more of these columnar readers in order to extend the architecture.

5.1.1. Reading Multiple Columns

The Accelerated Big Data Systems group, in collaboration with Teratide, has been working on the scalability of the architecture in Figure 5.2 so that the overall parquet reading operation utilizes hardware parallelism by operating on multiple columns. There are two ways to extend the current implementation to run multiple columns, namely sequential or parallel.

First, as explained in Section 2.4.2, there are columnar chunks in each row group. The software can *sequentially* call the FPGA instances to decode these columnar chunks back to back. The Arrow API offers an abstraction for reading the file offsets and sizes of each columnar chunk. An example code is provided in Listing 5.1. In line 1, we open the parquet file via *ParquetFileReader::OpenFile()* API call without creating any scan operators. Each row group can be accessed with the *RowGroup()* API call. We can read the metadata of each columnar chunk in this way.

```
1  auto pqFileReader = parquet::ParquetFileReader::OpenFile(input_path, true);
2  for (int i = 0; i < num_row_groups; ++i) //num_row_groups is obtained from parsed metadata
3  {
4      rowGroupMetadata.push_back(pqFileMetadata->RowGroup(i)); //pqFileMetadata is the pointer
       to file metadata
5  }
6  for (auto &r : rowGroupMetadata)
7  {
8      for (int i = 0; i < num_columns; ++i)
9      {
10       int64_t fpo = r->ColumnChunk(i)->file_offset();
11       size_t buffer_size = r->ColumnChunk(i)->total_uncompressed_size();
12       int num_val = r->ColumnChunk(i)->num_values();
13       auto device_addr = read_file(fpo, r->ColumnChunk(i)->total_uncompressed_size(),
       buffer_size);
14     }
15 }
```

Listing 5.1: An example code to obtain Columnar Chunks

Second, one can implement multiple converters on FPGA and send these chunks to decode in *parallel*. This implementation offers higher utilization of FPGA resources. Moreover, add one more level of parallelism, a multithreaded data communication between FPGA and CPU can be proposed. In this way, the program also utilizes more CPU sources and decreases idle times. Van Leeuwen 2019 [55] proposes an architecture for multiple parquet readers sharing the same Fletcher generated UserCoreController. Nevertheless, for higher reconfigurability, multiple core controllers can also be

integrated into the design. There has been progressing in our group in terms of multiple parquet readers with their controllers. The following Figure 5.3 illustrates the higher-level design of such efforts.

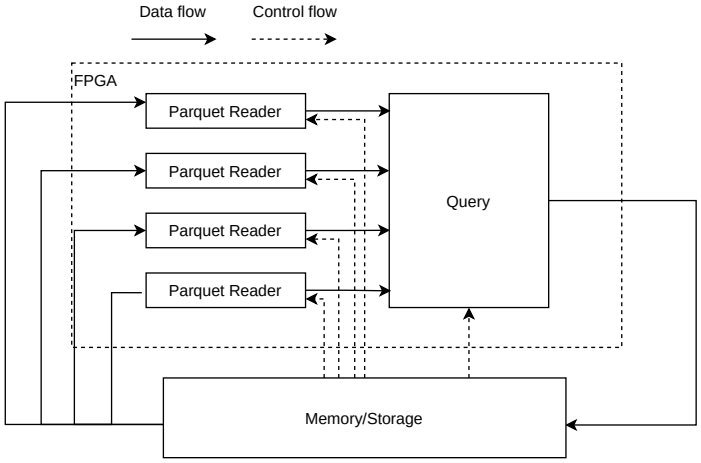


Figure 5.3: Example computation pushed down parquet reader with multiple parquet reader instances

5.1.2. Integrating computation in the Architecture

Inherently, the design streams are decoded and decompressed data right after the Values decoder. Therefore, one can add a compute kernel in between the values decoder and array writers. Since *Parquet Reader Instance* is interfaced to memory by *Fletcher Array Writer*, we should take out the *Fletcher Array Writer* and output the data, command, and unlock streams of the column read by the corresponding *Parquet Reader Instance*, as seen in Figure 5.4. Fletcher already provides an interface

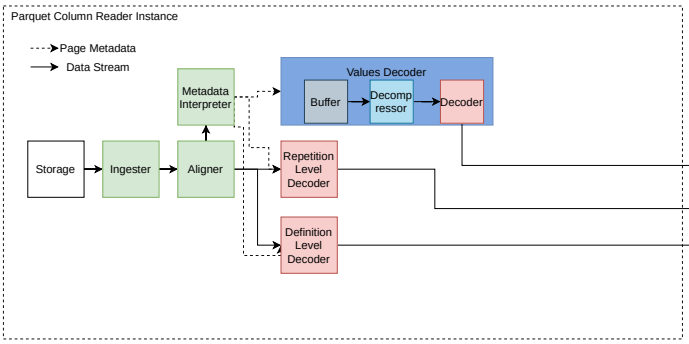


Figure 5.4: Proposed changes on Parquet Reader interface - Taking ArrayWriter out of datapath

and kernel wrappers named *AxiTop*, *Mantle*, *Nucleus* (top level to bottom level). Normally, *AxiTop* interfaces to an AXI interface. However, for the input stream, we do not need such an interface. We will connect the data stream output of the parquet reader to our *Compute Kernel* directly without implementing any *Fletcher Array Reader*. The hierarchical changes are visualized in Figure 5.5. The color yellow indicates the added wrapper for interfacing the data stream of *Parquet Reader Instance* to *Compute Kernel*.

5.2. Hardware Implementation of TPC-H Query 6

In this thesis, TPC-H query 6 is aimed to be benchmarked and implemented on FPGA for the reasons explained in Section 3. The overall hardware architecture of Query 6 is provided in Figure 5.6. Several necessary modules are contributing to the acceleration of analytic queries. These components are also helpful in other kinds of analytic queries.

Figure 5.6 provides a higher level architecture of query listed in Listing 5.2.

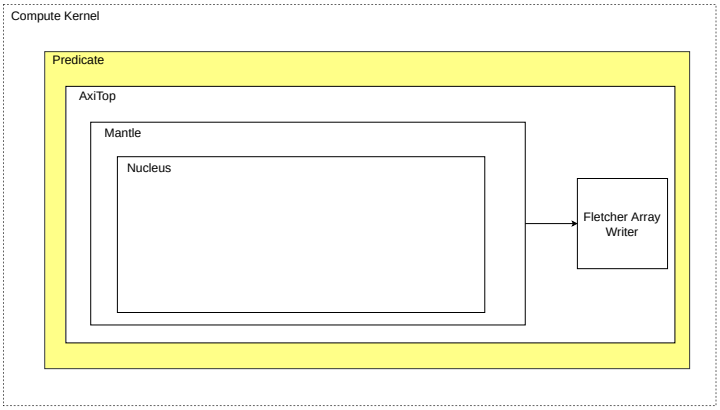


Figure 5.5: Kernel Hierarchy

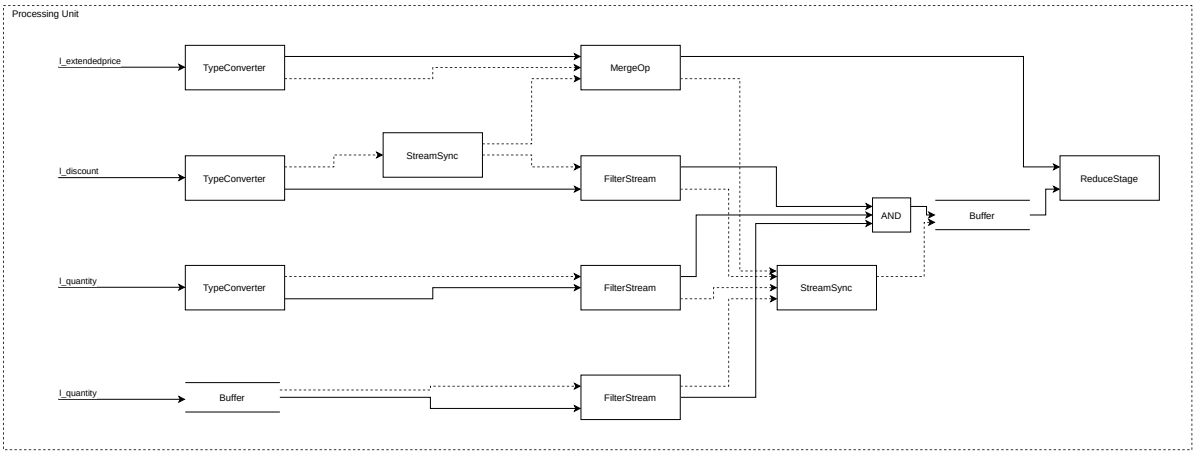


Figure 5.6: Query 6 accelerator Architecture

```
1   select
2     sum(l_extendedprice * l_discount) as revenue
3   from
4      parquet.`lineitem.parquet`
5   where
6     l_shipdate >= 8900
7     and l_shipdate < 9500
8     and l_discount between .06 - 0.01 and .06 + 0.01
9     and l_quantity < 24;
10
```

Listing 5.2: TPC-H Query 6

The following sections will lay out a more detailed explanation of the components.

5.2.1. TypeConverter

The analytic queries mostly rely on floating-point arithmetic to calculate the average, sum, or multiplication on different dataset columns. Floating-point operations are expensive in FPGA. Consequently, it is more efficient to use a circuit employing fixed-point arithmetic. To be able to use fixed-point arithmetic, there should be a *TypeConverter* before computation modules to convert the data from floating point to fixed point representation. In this thesis, the Xilinx[60] or FloPoCo[23] floating to fixed point converters can be used in *TypeConverter* module. These projects are chosen as they offer highly configurable, pipelined datapath for floating-point arithmetic and type conversions.The results in Chapter 7 are obtained by using Xilinx Converter.

5.2.2. FilterStream

One of the main components of SQL queries is *WHERE* statement which adopts a set of filters on a database such as regular expression, arithmetic expressions on primitive types. This module aims to provide a wrapper that can hold the filters and customize filters either in runtime or synthesis.

5.2.3. MergeOp

Analytic projections require merging different columns in some cases depicted in Equation 5.1. In dataflow design architectures, the operations can be performed on each stream separately easily with *FilterStream* operator. Yet, when we do operations that require merging we have to synchronize the streams.

$$(a + b)\ as\ x \tag{5.1}$$

$$avg(a * (b + c))\ as\ y \tag{5.2}$$

5.2.4. Aggregation

So far, we have seen the ways to apply expression matching operations, arithmetic operations. The final module we will visit is the *ReduceStage* module, which is responsible for aggregating the streams either generated by *MergeOp* module or streamed by the input. An extensible baseline reduction module is designed by Hadnagy 2020[32], as visualized in Figure 5.7 and 5.8. There are several extensions applied to this module to support multiple aggregation operators. This study extends that module to support multi-column aggregations combined by *GROUP BY* statement by using a Hash-Table implementation.

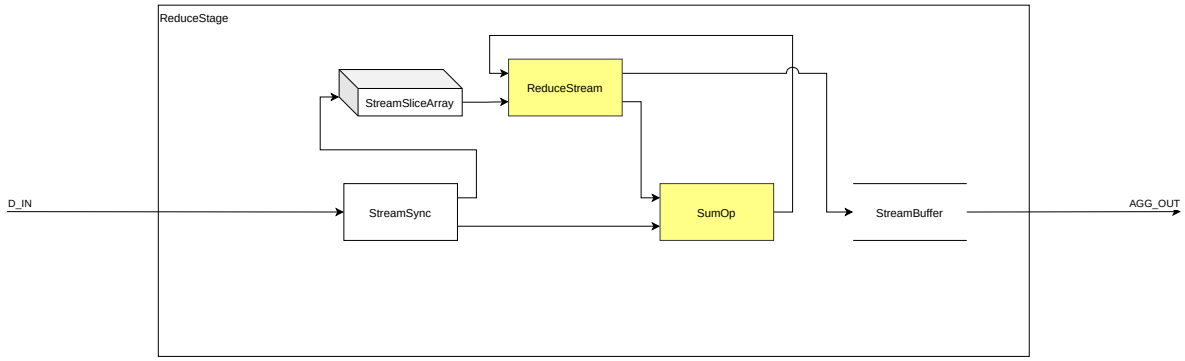


Figure 5.7: ReduceStage module [32]

The extensions done on the *ReduceStage* module are presented by yellow blocks in Figure 5.7. *SumOp* module runs a fixed point summation as the datapath is in fixed point. The main changes in *ReduceStage* package are under *ReduceStream* package.

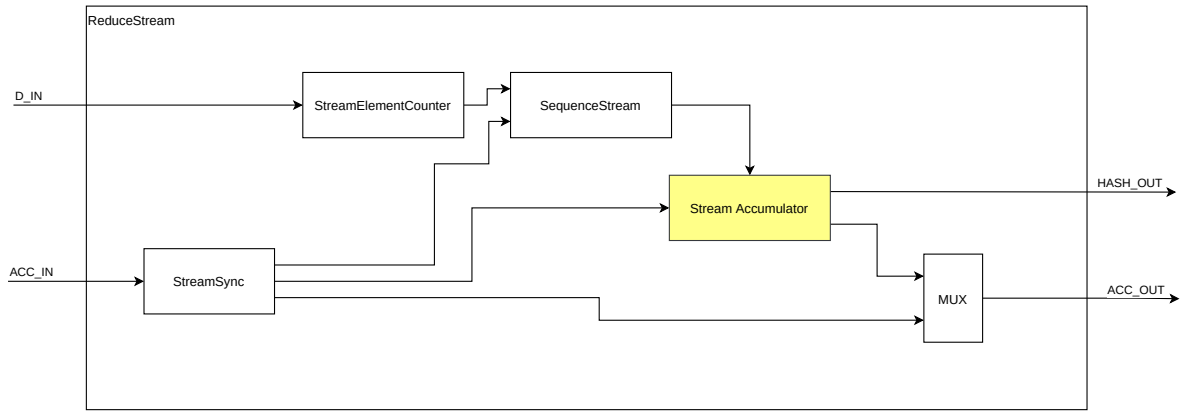


Figure 5.8: ReduceStream module [32]

The *ReduceStream* package, as visualized in Figure 5.8, uses a single accumulator register to hold the accumulated, aggregated value. However, to do this with multi-column grouped aggregations, one should add more registers or a table. Various studies have implemented a hash table and content addressable memory on intermediary storage (BRAMs) to surround a logic that undertakes the grouping operation [66], [4].

Introducing Hash Table for multiple groupings This thesis proposes a hash table implementation for a multi-column textitGroupBy operation. The architecture is provided in Figure 5.10. There are two states of aggregation operation, namely, build and probe. In the build state, the aggregates and keys are streamed to the *HashTable*. In *HashTable*, the keys are stored in *bit_table* and data is stored in *hash_table*. The idea behind two different tables is that if one desires to implement sort on keys, the sort operation can be performed on the elements of *bit_table*. The keys, provided by the *StreamAccumulator*, are hashed by a hash function which can be modulo32, modulo128. *Controller* handles the logic to update the *hash_table*. If a new key is introduced to the *HashTable*, the key is appended to the *bit_table*, *hash_pointer* is incremented. Then, the values are written to the *hash_table*, which is indexed by a hashed key. The keys can be separate columns in the dataset.

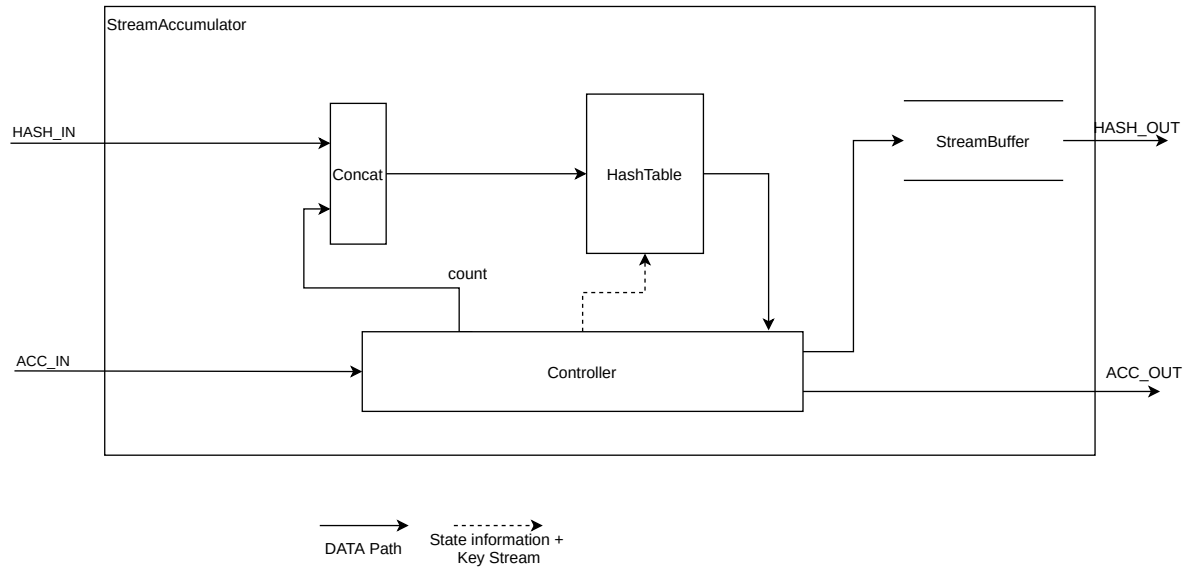


Figure 5.9: Accumulator logic with Hash Table implementation

5.3. Discussion

This chapter presents the integration efforts of query computation with parquet reading on FPGA. The baseline parquet reading design was proposed by Van Leeuwen 2019 [55]. We have seen that the integration is feasible and further extensible. In Chapter 4, we have explained that we can parallelize row groups by using concurrent, multithreaded Thread Pools on native software. These parallelization efforts were the seed of increasing the interconnect bandwidth utiilization that this project runs on. The proposed design in Figure 5.3 uses around 10 % of the FPGA fabric as depicted in Chapter 6. In other words, one can add more of these resources and obtain a higher level of parallelism. Keeping in mind that these columnar readers and Compute Kernels implement their *AxiMMIO* module and *UserCoreController*. More specifically, they can be controlled separately by the Software API designed in Chapter 4. Figure 5.12 proposes an architecture to use more FPGA resources by attaching multiple all-query pushdown parquet readers. The query pushdown parquet reader design, as illustrated in Figure 5.11, bundles four parquet column readers and a single query. Software APIs can connect these all-query pushdown parquet readers and run them in parallel.

The underlying setup is FPGA as a co-processor in Figure 5.12. Nonetheless, one can extend this to run in FPGA as an amplifier setup. In this case, each parquet reader instance should decode the file metadata and get columnar chunks on their own. We will investigate such setup in Section 6.

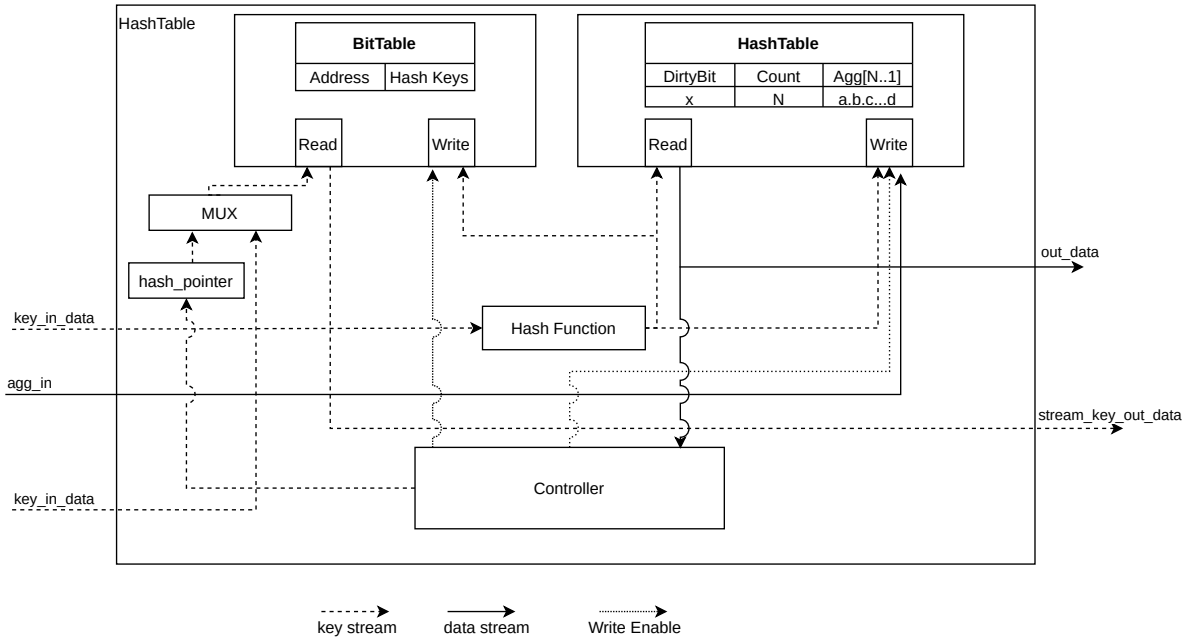


Figure 5.10: The architecture of Hash Table

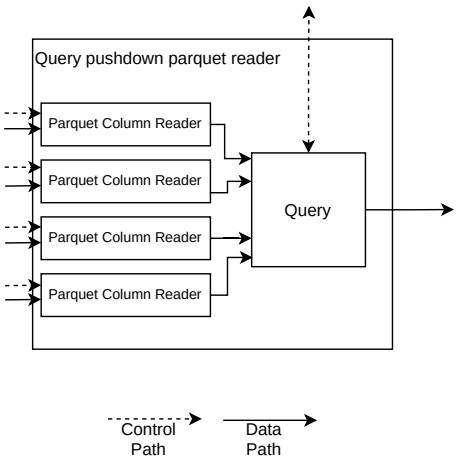


Figure 5.11: Architecture of query pushdown parquet reader instance

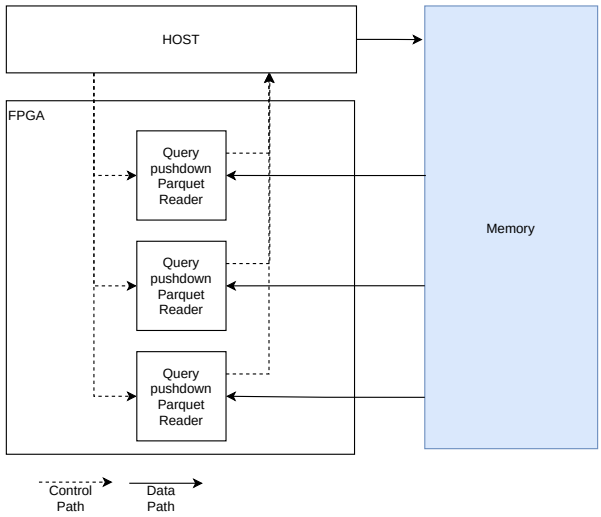


Figure 5.12: Architecture with multiple query pushdown reader instances which can parallelize upto 3 Row Groups

6

An Extended Roofline Analysis for Parquet Decoding Operation on FPGAs

In this chapter, we will examine the extended roofline model for parquet reading operation on FPGAs. The aim is to explore the boundaries of FPGA implementation of parquet reading operation and query pushdown. The roofline model consists of several essential components, such as the computational intensity of the kernel, bandwidth of the technology, clock frequency. An example roofline model is provided in Figure 6.1. Blue lines are the I/O throughput of the system, and red lines are the computation throughput. Dotted lines show the ceiling, whereas thick lines present the roof. We examine byte operations for the design for two different applications. The first application is I/O-bound, and the second application is compute-bound.

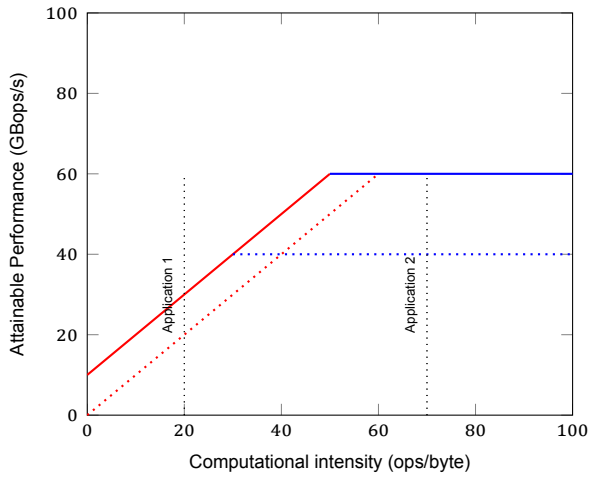


Figure 6.1: Basis of the Roofline Model for different applications [22]

There are multiple ways of formulating a roofline model for hardware design. Da Silva et al. proposes a formulation of constructing a roofline for HLS designs and integrate that solution for Xilinx ISE 14.4[22]. In this thesis, we extend this work for VHDL design. In order to construct the roofs we utilize the area utilization on FPGA fabric and available bandwidth of the technology used as the bounding factor.

$$Scalability = \left\lfloor \frac{Available\ Resources}{Resource\ Consumption\ per\ Design} \right\rfloor \tag{6.1}$$

$$Attainable\ Performance\ FPGA = Scalability \times Attainable\ performance\ per\ Design \tag{6.2}$$

$$Attainable\ Performance\ = \min\{(CI \times BW), (Attainable\ Performance\ FPGA)\} \tag{6.3}$$

As seen from the Equation 6.3, we define several different independent variables for Roofline model. Computational intensity, I/O bandwidth are two independent variables. Resource Consumption depends indirectly on the computational intensity and is calculated using synthesis results. There are five main designs analyzed in this chapter:

- Parquet reader for uncompressed files
- Parquet reader for compressed files
- TPC-H Query 6 hardware design
- Query Pushdown Parquet reader for uncompressed parquet files(Hybrid Uncompressed)
- Query Pushdown Parquet reader for compressed parquet files(Hybrid Compressed)

In the next sections, we will investigate the ways to obtain these variables.

6.1. Parameters of Roofline Analysis

This study extends Roofline model, proposed by Da Silva et al. [22], for HDL language as the design, explained in section 5, is implemented in VHDL. This means that byte operations and the input-output size of each architectural component should be calculated without the help of any tool.

6.1.1. Byte Operations

An example block design is provided in Figure 6.2. *D_IN* represents the input data stream. Regardless of the protocol it uses, the important factor for analysis is the byte operations on the data stream. If we demonstrate it with the example in Figure 6.2, which illustrates the aligning of an elementary data stream. It includes one buffer. We may define that if the downstream is not blocked, there will be a continuous flow of data. The buffer may hold unprocessed data, which adds a delay. In this example, data width of *d_in* and *d_out* is 2 bytes. The design exchanges one block per data per cycle. In total, the byte operations per cycle are $2 * (k + 1)$ as one cycle per exchange and k cycles per delay in the buffer. In Figure 6.2, the buffer has 2 data, so that means k will be 2. In this case, this module processes $6\frac{bytes}{cycle}$

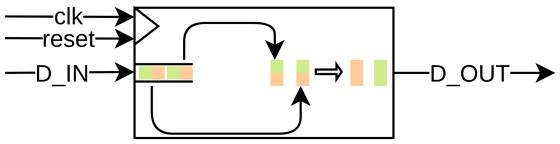


Figure 6.2: Simple block design for realigning of 2 bytes input data stream

The design includes multiple buffers constructing from BRAMs. Estimating the runtime behavior of each buffer is out of the scope of this thesis. Therefore, we will assume that at time t, the number of bytes processed in the buffer equals the number of bytes on the data bus. Table 6.1 shows the byte operations per module. These are obtained by the same analysis described earlier and simulation results.

Component	Number of byte operations per cycle (N):
Interpreter	30
Decoder	6N
Ingester	2N
Aligner	4N
Decompressor(Snappy)	31[1][28]

Table 6.1: Byte operations per cycle for each component to be used for CI(Data bus length in bytes=N)

Predicate(Query) design implements 3 Floating-point to Fixed-point converters, one multiplication, three filterings, and one aggregation. Each multiplier processes one 64-bit wide element per 8 cycles that makes N bytes per cycle for input with the width of N bytes. Converters take nine cycles for 64-bit inputs, hence makes 0.88N per byte operations per cycle. Each filter processes a single element per cycle, which corresponds to N bytes per cycle. Aggregation is tricky, as it includes a stream accumulator(either in the form of a table or register) with sequential logic, updating one element per cycle. Also, there is a parallel fixed-point summation in the aggregator. However, for the sake of easiness, we can take it as N bytes per cycle. In summary, the byte operations per cycle are 2.88N bytes per cycle for TPC-H Query 6.

In case we want to analyze byte operations for filtering, we can simply say that it is N bytes per cycle. The reason is that it uses the same filter operation as TPC-H Query 6, except that it also has two different stream synchronizers.

6.1.2. Input/Output

Another significant part of the computational intensity is the number of inputs, outputs, and memory accesses. For the sake of the easiness of the analysis, AXI Width is the chosen parameter. In the scope of the proposed design in this thesis, the width of the AXI bus used is 1024 bits.

The memory accesses for our design:

- For parquet reading operation, the input and output have a width of 128 bytes.
- Query operation will have four elements(3 floats and one date32) streamed into it. The design will have 128 bytes input and 8 bytes output (aggregation result). It writes the output to the corresponding MMIO register
- Query pushdown parquet reading operation will have 128 bytes input and no output.

6.1.3. Area

As denoted in Equation 6.2, scale argument expects the utilization information of the kernel. In the scope of this thesis, there are five different types of algorithms, namely compressed baseline FPGA parquet reader, uncompressed baseline FPGA parquet reader, compressed query pushdown FPGA parquet reader, uncompressed query pushdown FPGA parquet reader, and TPC-H Query 6. We will investigate the FPGA utilization of these designs. The scale will be calculated as denoted in Equation 6.4. *module* property is defined as the component under analysis. A simple example for *module* can be the 64-bit column reader.

$$Scalability = \frac{1 - utilization\ overheads}{(\textbf{\textit{module}}\ utilization)} \tag{6.4}$$

What can be clearly seen in Table 6.2 is that the parquet reader for a single column (either 64 bits or 32 bits) occupies only 0.78% of the FPGA fabric. We will use this as a baseline once we extend the estimations for multiple columnar readers. Moreover, it should be noted that the synthesized kernel consists of 12 single-column parquet readers and three predicates (query) instances. Therefore, the utilization is *fletcher_axi_top* is 24.58%.

The design is synthesized for the platform with oc-accel and OpenCAPI Snap. Therefore, the framework occupies some area to implement the board support package that it provides and the interface for CAPI, memory-mapped registers. The total area it occupies is that 5.87% as given in Table 6.2. In order to calculate the scaling of each component, we will use the formula in Equation 6.4 and put *utilization overheads* as 0.0587. The resulting Table 6.3 presents the results for several different designs in the scope of this thesis. One important reminder is that the numbers are the ceiling of the fractions.

Single column parquet reader - Compressed and Uncompressed

Single 64 bit column reader utilizes around 2.16% of the FPGA fabric. Whereas 32-bit columnar readers use 1.46% of the available BRAMs. For the sake of convenience and extensibility of our analysis, we will take the utilization of 64-bit columns. Since the design has a lot of buffers, it makes sense to use BRAMs for the scalability analysis.

[1]The current wrapper only supports 64 bit datapath

Module	CLB LUTs(Used)	BRAM Tiles(Used)
bsp	2.65%(34569)	0.84%(17)
cfg	0.10%(1263)	0%
oc_func	27.69%(361001)	35.42%(714)
cfg	0.03%(433)	0%
fw_afu	27.66%(360570)	35.42%(714)
desc	<0.01%(31)	0%
snap_core	3.08%(40099)	1.79%(36)
fletcher_axi_top	24.58%(320497)	33.63%(678)
extended_col(64 bit)	0.78%(10161)	2.16%(43.5)
shipdate_col(32 bit)	0.72%(9449)	1.46%(29.5)
predicate_instance	4.43%(57703)	3.27%(66)

Table 6.2: Utilization Report of query pushdown parquet reader implementation

Module	Scalability
Parquet Reader Single Column(Uncompressed)	$\frac{(1-0.0587)}{0.0216} \cong 43$
Parquet Reader Single Column(Compressed)	$\frac{(1-0.0587)}{(0.142+0.0216)} \cong 5$
Predicate(Query 6) Design	$\frac{(1-0.0587)}{0.0443} \cong 21$
Predicate(Query 6) pushdown Parquet Reader Design(Uncompressed)	$\frac{(1-0.0587)}{(4*0.0216+0.0443)} \cong 6$
Predicate(Query 6) pushdown Parquet Reader Design(Compressed)	$\frac{(1-0.0587)}{(4*(0.0216+0.142)+0.0443)} \cong 1$

Table 6.3: Scalability of the important components

Predicate(Query) Design

Predicate kernel implements either query or filters pushdown. As expected, pushing down queries will use more FPGA resources. Therefore, we will use the utilization of query pushdown (TPC-H Query6) for scalability analysis.

Predicate(Query) pushdown Design - Compressed and Uncompressed

We are aiming to extend this analysis for query pushdown parquet reading operations. The implemented SQL query has four columnar readers and one predicate(query) design. The hybrid implementation in Table 6.3 shows the scalability of such query.

6.1.4. Attainable Computational Performance per FPGA

Attainable performance per FPGA is related to the number of byte operations per second constrained by the scalability of the design. Figure 6.3 shows the attainable performance for a single FPGA and five different designs, which are used in this thesis. Firstly, for the baseline column reader for parquet files, the attainable performance is higher, as shown in Figure 6.3.

6.1.5. CI - Computational Intensity

CI variable shows the complexity of an algorithm. It is easier to construct this variable in higher-level languages by investigating loop bodies before the software compiles. Da Silva et al. [22] proposes the following equality 6.5 for HLS designs to calculate CI.

The computational intensity of a model shows the byte operation of a model per byte accessed. Therefore, the modifications on a program by increasing the data locality can increase the computational intensity as it can process more elements per iteration. In this study, the researcher examines the increase of data locality by changing the synthesized design. The change is to make the design sustain more elements per cycle. In this way, keeping the input-output width the same, one can change elements processed per iteration (cycle).

$$CI = \frac{\#Byte\ Operations}{\#Inputs(Bytes) + \#Outputs(Bytes)} \tag{6.5}$$

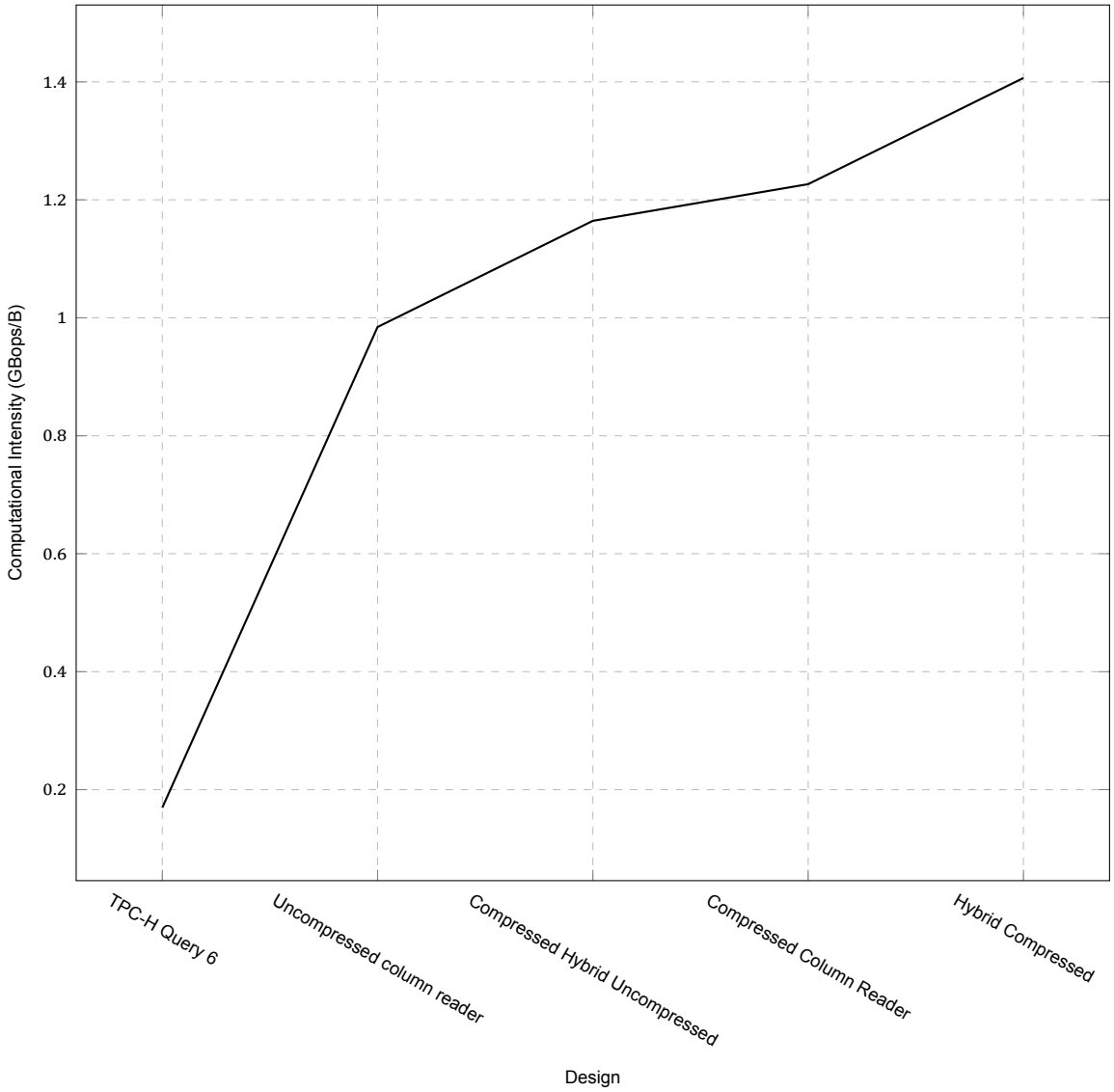


Figure 6.3: Computational Intensity of seperate implementations with datapath of 64 bits 200 MHz design

6.2. Roofline

6.2.1. Constructing Computational Roof

In order to construct the computational roof, we will iterate on different designs. Normally, roofline analysis on software is carried out on the operations of the loop body. As the number of parallel columnar readers increases, the number of bytes processed per cycle stays the same. Therefore, it makes sense to calculate the computational intensity of different hardware designs. Before calculating CI, we should calculate the attainable performance per FPGA, which also puts scalability in the equation. Figure 6.4 illustrates computational roof estimation. One important observation is that uncompressed parquet readers have the highest attainable performance and scales better. The reason is that the utilization of uncompressed parquet files is less than 3%. Therefore, scalability becomes the bounding factor for designs having high computational intensity. One should interpret the effect of scalability relation in the following way. Scalability is a measure that multiplies the roof for estimating attainable computational performance. A statement about scalability can be made as follows: *"The application can attain more computational performance if scalability gets higher while keeping CI the same."*.

In this thesis, the attainable peak performance per FPGA is dependent on computational intensity. Figure 6.4 shows the results of computational intensity per parallel modules

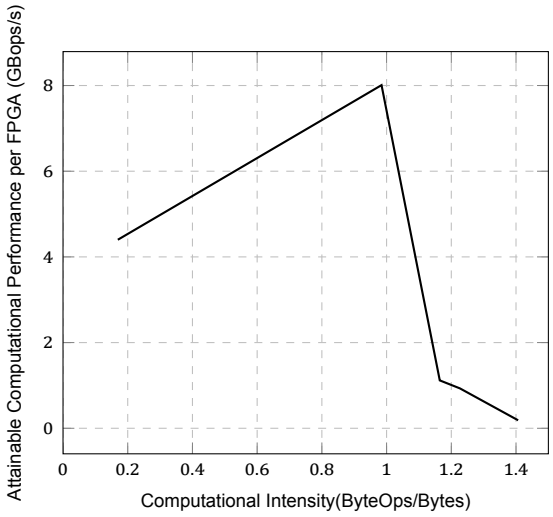


Figure 6.4: Attainable Performance (per FPGA) of seperate implementations with datapath of 64 bits 200 MHz design

6.2.2. Constructing I/O Roof

So far, we have constructed to compute roof for our model, which is independent of the system design. Now, we will examine the effect of interconnect on the Roofline and construct the I/O roof of the Roofline. We take unidirectional bandwidth as analytics query processing happens in one direction. We will investigate several different interconnect types:

- x16 PCI-E Gen. 3 with 32 GB/s peak bandwidth.
- x8 PCI-E Gen. 3 with 16 GB/s peak bandwidth.
- x16 PCI-E Gen. 4 with 64 GB/s peak bandwidth.
- x8 PCI-E Gen. 4 with 32 GB/s peak bandwidth.
- x16 PCI-E Gen. 5 with 128 GB/s peak bandwidth.
- x8 PCI-E Gen. 5 with 64 GB/s peak bandwidth.
- FPGA point-to-point 100 GB/s peak bandwidth.[16]
- OpenCAPI with 50 GB/s peak bandwidth. (Oc-accel)
- DDR4 per DIMM with 50 GB/s peak bandwidth.
- DDR5 per DIMM with 101 GB/s peak bandwidth.

Inherently, this analysis constructs a baseline for system architecture. Therefore, it is beyond the scope of this thesis to discuss whether or not the interconnect IP is available for the chosen FPGA platform. Equation 6.6 shows the bandwidth calculation for our design.

$$IO\_ROOF = CI \times IO\ Bandwidth \tag{6.6}$$

Figure 6.5 depicts the technology dependent portion of our roofline model. It can be seen from the figure that we obtain highest speed with the on-chip memory attached configuration. Then, the best possible performance comes from the CPU+FPGA shared memory configuration.

6.2.3. Roofline Model

Putting it all together, we obtain the following model in Figure 6.6. Figure 6.6 illustrates the roofline model of three different FPGA models. The idea of sweeping through different FPGAs is to estimate the performance concerning the scaling of hardware designs. Moreover, the results will give an idea about the performance improvements of different architectures. Table 6.4 lists the analyzed FPGA families.

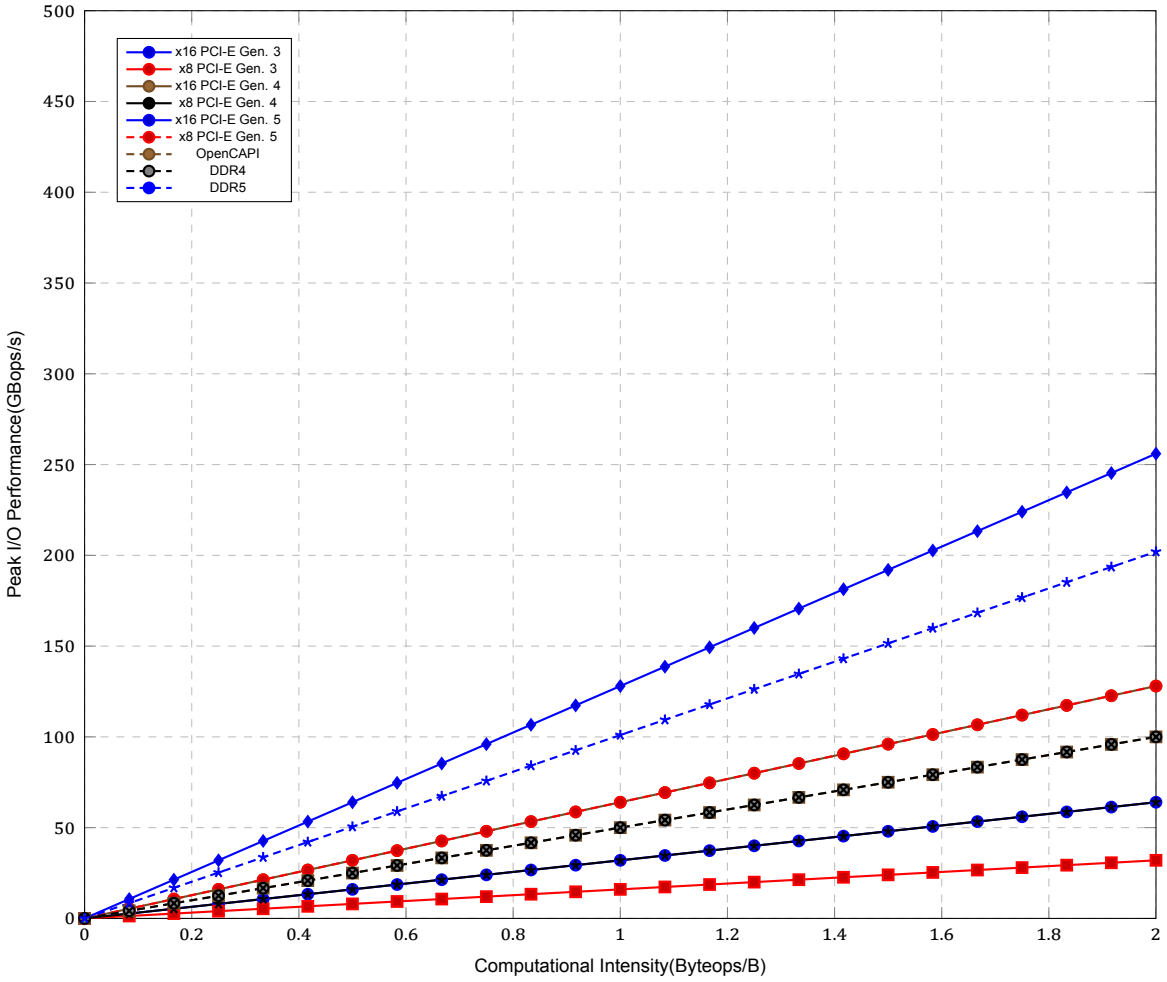


Figure 6.5: I/O roof per interconnect

Multiplicative factor denotes the constant for multiplying the *scalability* of a design. The reason is that FPGAs do not have the same programmable logic size, which affects the *scalability* of our design. In return, this effect will be seen on peak performance. One important aspect to mention is that the analysis assumes that the only limitation comes from BRAM Tiles for parquet readers and CLB LUTs for TPC-H Query 6.

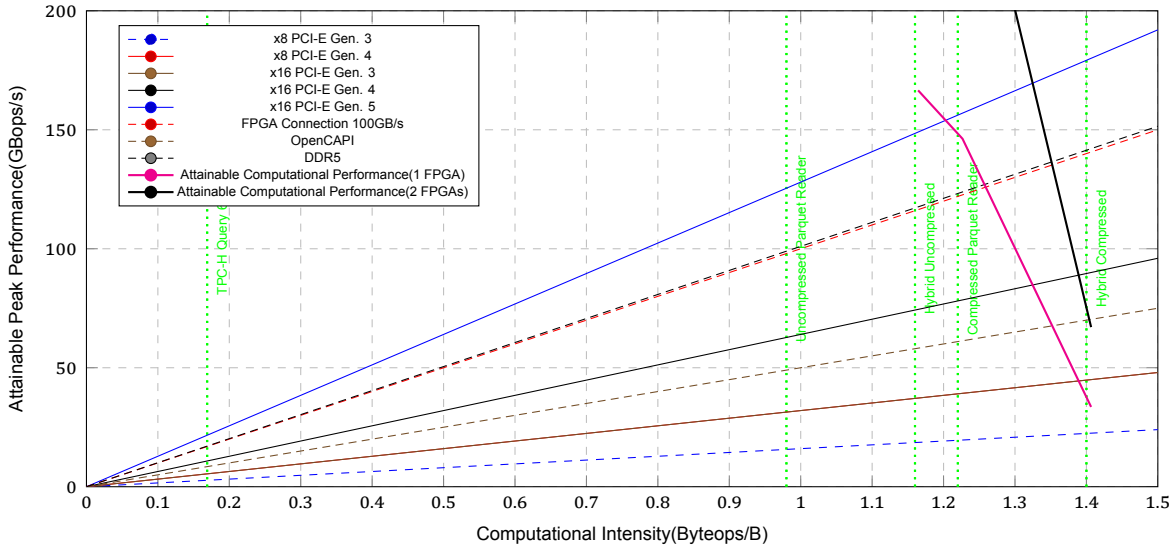
The compute roof is way higher and is capable of reaching almost 1000 GB/s. Therefore, the compute roofs are clipped in Figure 6.6. In this analysis, we are interested in the intersections of the I/O roof and compute roof. Moreover, it is significant to show whether the design is compute-bound or I/O bound.

Firstly, the analysis in Figure 6.6a shows the peak performance estimations on VU37P FPGAs. The peak performance observed is for computational intensity slightly larger than 1.3 with PCI-E Gen. 5.

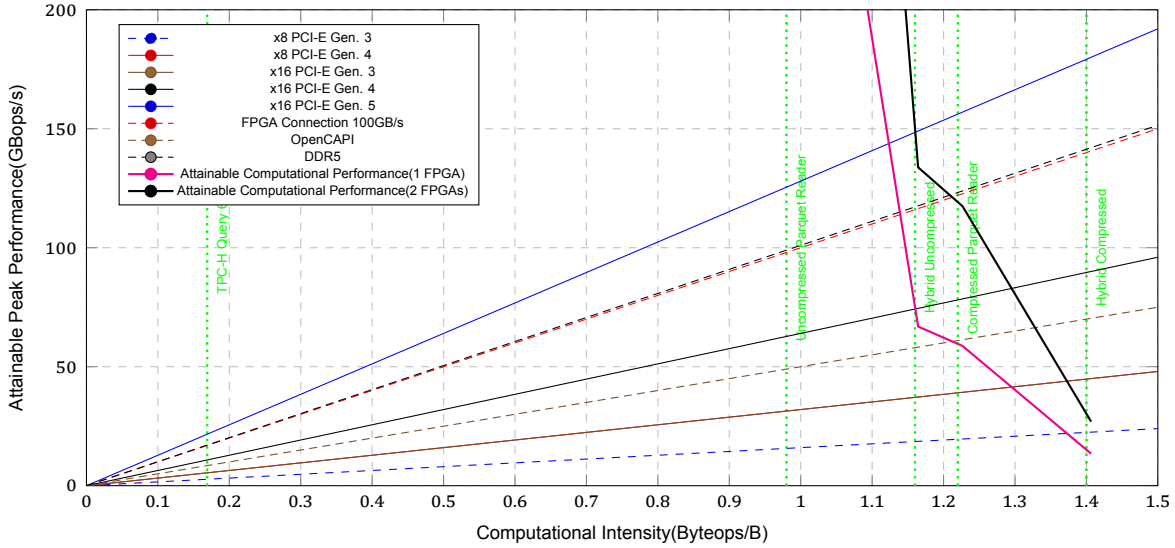
Then, Figure 6.6b depicts the performance for KU15P, which has the smallest chip area amongst all. One should expect that the peak performance should decrease as *scalability* decreases. As seen from Figure 6.6b, the peak performance achieved is at most.

Finally, 6.6c shows the performance characteristics for VU19P FPGA. Scalability increases the compute roof, which will result in all of the designs being I/O bound. This result is different once the interface type is PCI-E Gen. 5 or FPGA to FPGA connection.

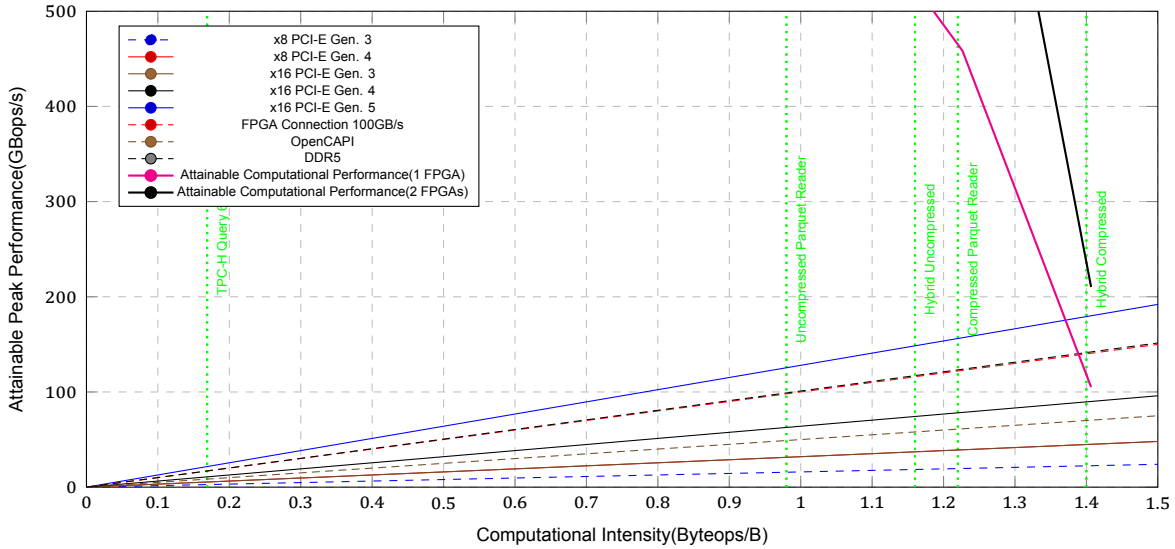
TPC-H Query 6 has computational intensity of 0.69, which is the lowest compared to the others. The analysis in Figure 6.6 shows that query itself is always I/O bound. Moreover, Uncompressed parquet reading operation has computational intensity of 0.98 and also always I/O bound for tested configurations.



(a) Attainable performance characteristics of Compute and I/O roofs for VU37P



(b) Attainable performance characteristics of Compute and I/O roofs for KU15P



(c) Attainable performance characteristics of Compute and I/O roofs for VU19P

Figure 6.6: Roofline Analysis

FPGA	Available LUTs	Multiplicative factor
KU15P	523000	2.49
VU19P	4086000	0.319
VU37P	1303680	1

Table 6.4: Available LUTs for different FPGA families

6.3. System Design

Understanding the capabilities of the underlying hardware design, we can now propose a system design for query pushdown operation. The lifetime of an analytic query starts from the file, which is stored on a disk. One of the purposes of this thesis is to understand the limitations of pushing down an end-to-end analytic query to parquet reading operation.

The purpose is to read aggregations once the parquet file is read. Therefore, FPGA should be on the datapath between CPU and memory. Fang et. al. presented 3 different architectures as depicted in Figure 1.1. The best possible configuration is FPGA as a bandwidth amplifier. Inherently, we do want to hold large datasets (1TB). Hence, a non-volatile memory connected by a multi-channel PCI-E bus can be one type of configuration. Then, FPGA can be attached to the CPU by multi-channel OpenCAPI. There are commercial examples of such systems [35], [16], [36].

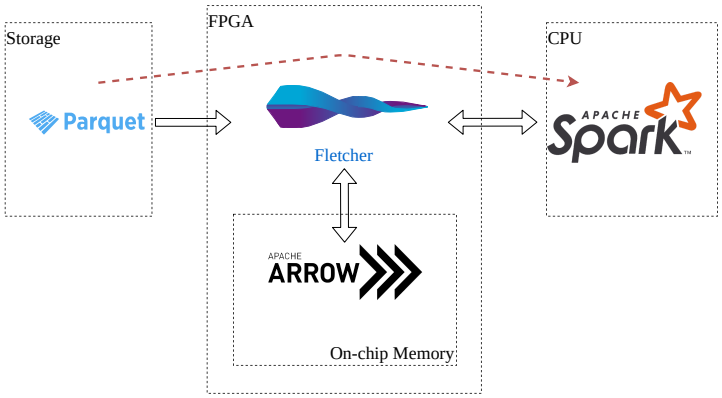


Figure 6.7: Application Runtime Path

Early filtering and aggregation of datasets prevent extra loads or copies of data. Integrating this concept with FPGAs, we achieve high throughput, low power analytic server. The work in ABS is concentrated on communicating in Arrow format to accelerate workloads. Arrow format, as described in Section 2.4.3, avoids serialization overheads. Hence, communicating by Arrow tables creates much room for improvement in terms of system design. Fletcher also provides necessary wrappers (reading, writing) for such tables. Inherently, we want to achieve the programming model visualized in Figure 6.7. The red dotted line represents the runtime architecture in this thesis.

In the following sections, we will investigate the performance of several system designs.

6.3.1. Analysis on state-of-art system architectures

So far, the broader roofline analysis and its components have been introduced. This thesis concentrates on a design which has 2 components, namely *parquet column reader* and *query*. As a query module, TPC-H Query 6 is the focus of this thesis. TPC-H 6 reads the *lineitem* table in TPC-H benchmarks. *lineitem* table is the largest table in the benchmark with 16 columns, constructed by 3NF Schema. It has been stated before that the objective is to decode multi-column parquet files in Arrow Tables and hold them in on-chip memory. It is desired to use HBM, which provides up to 470GB/s bandwidth to hold Arrow Tables. Therefore, selective access and partial decompression, if the file is compressed, are needed to decode only the columns filtered by the query. This thesis proposes a software solution for selective access and partial decompression in Chapter 4. Nonetheless, it is no harm to assume that once FPGA decodes the parquet file, it has the offsets and sizes of each column required for selective access.

FPGA as Bandwidth Amplifier: SmartSSD Computational Storage Drive Xilinx powers the computational storage drives, SmartSSDs, which consists of flash memory, and Xilinx Kintex Ultrascale+ KU15P FPGAs [59]. The interconnect to the host interface is Single Port PCIe Gen 3.0 x4. Smart SSDs have NAND flash memory capable of reading 800000 transfers per second, with a peak bandwidth of 6.4 GB/s. As SmartSSDs use different kinds of FPGA than the design proposed in the thesis, the scalability will be changed regarding Kintex KU3P, which has approximately 300K available LUTs for acceleration [59]. It should be noted that Xilinx XCVU37P has 1303680 available LUTs. Therefore, the scalability will have multiplicative factor of $\frac{1303680}{300000} = 4.34$. More specifically, if the design has utilization of 1 before, it will have 4.34 in Kintex KU15P. The storage capacity is a maximum of 3.84 TB.

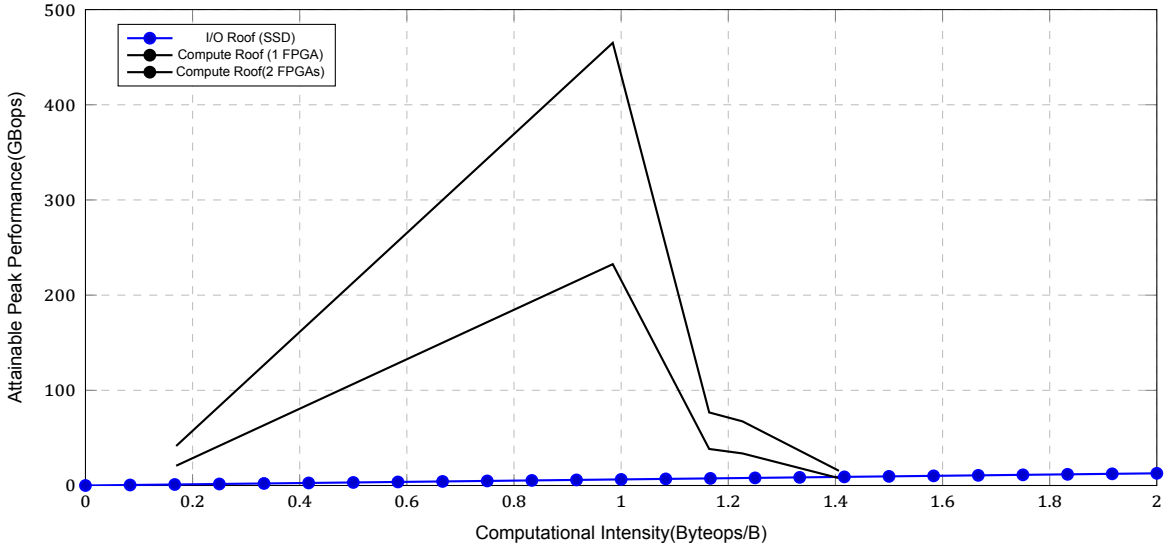


Figure 6.8: Roofline model for SmartSSD

The roofline model for SmartSSD is depicted in Figure 6.8. One important result to address is that the peak performance is lower than the other interconnect types. This implies that peak performance is bounded by PCI Gen 3 x4 bandwidth for SmartSSDs. One other thing to mention is that the scalability factor for Kintex KU15P is lower than other FPGAs. One can propose to use SmartSSDs to read the parquet files and use another FPGA instance (coupled by either on-chip memory or more channels of PCI-E) to run queries.

FPGA as Co-processor: OpenCAPI There are memory types sharing the memory between the host CPU and FPGA. One example is OpenCAPI. The brief roofline analysis on different interconnects, including OpenCAPI, is provided in Figure 6.6. OpenCAPI provides a coherent interface to attach FPGA as a co-processor. It can provide 25 Gbit/s per channel. This section analyzes the roofline model for Power9, which provides 25 GB/s (8 channels) OpenCAPI interconnect.

Figure 6.9 depicts the roofline analysis on Power9. 2 FPGAs dual-channel OpenCAPI achieves the highest peak performance.

FPGA as Co-processor: Intel QPI Intel offers Xeon and FPGA Accelerator Platform for handling data center workloads [31]. The architecture consists of QPI, which enables up to 25.6 GB/s interconnect bandwidth. At this instant, one can state that it is not ideal for analyzing such a platform as the FPGA vendor changes and the software used for hardware design. However, since the interconnect bandwidth is roughly the same as OpenCAPI, we can reach the same conclusions for this interface.

FPGAs as Co-Processor: PCI-E Switch So far, we have investigated several different interconnect types and their roofline analysis with query pushdown and parquet reading operation. Some servers utilize FPGA to FPGA communication by PCI-E Gen 3. x16 switch, which can provide up to 100 GB/s communication [16]. The roofline analysis in Figure 6.6 shows 100GB/s FPGA Connection. It is visible

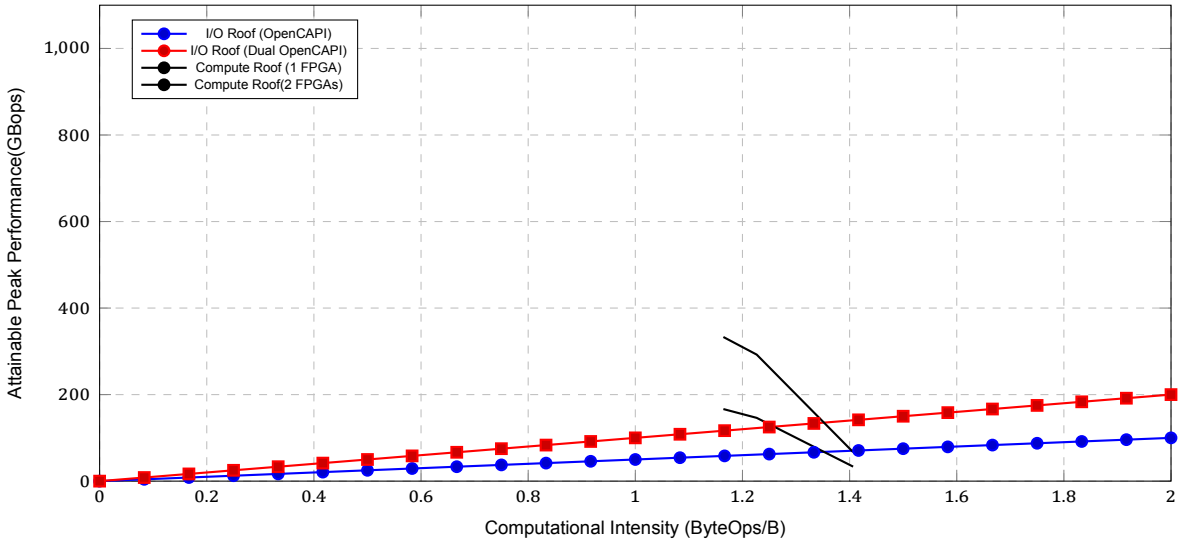


Figure 6.9: Roofline model for Power9

in Figure 6.6a that only hybrid, compressed design is compute-bound. Whereas Figure 6.6b depicts that the design for parquet reading with compressed files offers the highest possible performance gain.

6.3.2. Preliminary Conclusions

In summary, the well-known interconnect technologies are investigated for parquet reading operation and an example ad-hoc query(TPC-H Query 6). We have seen that pushing code down to parquet reading has better performance with high throughput interconnect types. The reason is that the hardware design is I/O bound in almost all cases except FPGA to FPGA communication cases. In the SmartSSD case, the FPGA does not have enough area to fit many analytic queries and parquet readers. Moreover, SmartSSD has the interconnect type as the bounding factor. 100GB/s FPGA to FPGA connection is the only interconnect type that is closer to make uncompressed hybrid design (CI equals 1.16) I/O bound as seen in Figure 6.6b. On the other hand, we have seen that decompressor has lower scalability, resulting in a lower attainable performance for compressed hybrid design.

Using former analysis, we project the throughput numbers for different technologies. Figure 6.10 represents the projected throughput numbers for each technology, while computational intensity is swept from 1 to 1.4. The vertical lines show the technology bounds as interpreted from Roofline analysis in Figure 6.6. More specifically, once the computational intensity exceeds those vertical thresholds, the design is compute-bound for the technology specified by the vertical bound. For instance, we can observe the behavior of the system with 1 VU19P FPGA, 100GB/s link. The applications with CI less than 1.9 are I/O bound for this system. Therefore maximum theoretical attainable throughput is 100 GB/s if the application runs perfectly and saturate the interconnect bandwidth. Whereas applications with CI higher than 1.9 are compute-bound. This will result in bandwidth decrease until the throughput specified by the following formula $\frac{Roofline}{CI}$. The analysis in this chapter also includes the next-generation interfaces such as PCI-E Gen. 5. Nevertheless, many FPGA links still use PCI-E Gen. 3 communication, which is also analyzed in this chapter. It has been shown that the hardware designs interfaced to host CPU or other FPGA by PCI-E Gen. 3 are I/O bound. The ad-hoc query analyzed in this thesis, TPC-H Query 6, is not computationally intensive for any known interconnect to be compute-bound. That is why computational intensity, which is smaller than 1, is not included in Figure 6.10. The proposed 100 GB/s link increases the performance for I/O bound applications for simple aggregations.

In summary, this chapter analyzes the characteristics of five different hardware designs, namely, parquet reader for compressed parquet files, parquet reader for uncompressed parquet files, TPC-H Query 6, query pushdown parquet reader for compressed parquet files, and query pushdown parquet reader for uncompressed parquet files. The analysis revealed that the ad-hoc queries with basic aggregation operations and filters are I/O bound. Using 100 GB/s FPGA to FPGA communication link can reveal high-performance gains in case the aggregation. There are commercial database servers

available for such communication [16]. The performance of parquet reader design is heavily dependent on the type of parquet file being read. SmartSSDs are I/O bound for any design. The best possible performance for parquet readers is achieved using 2 VU19P FPGAs with a dual-channel OpenCAPI link. This analysis reveals that the performance gains become significant once the computation platform is linked via a high bandwidth next-generation channel to the storage. That is why FPGA as co-processor architecture enables better performance achievements and is used for profiling query pushdown operation in Chapter 7. We observe that the most significant limitation with current accelerators is the interconnect bandwidth. Moreover, the scalability decreases once the design complexity increases. This will result in lower attainable computational peak performance. Therefore, another limitation of query pushdown is scalability.

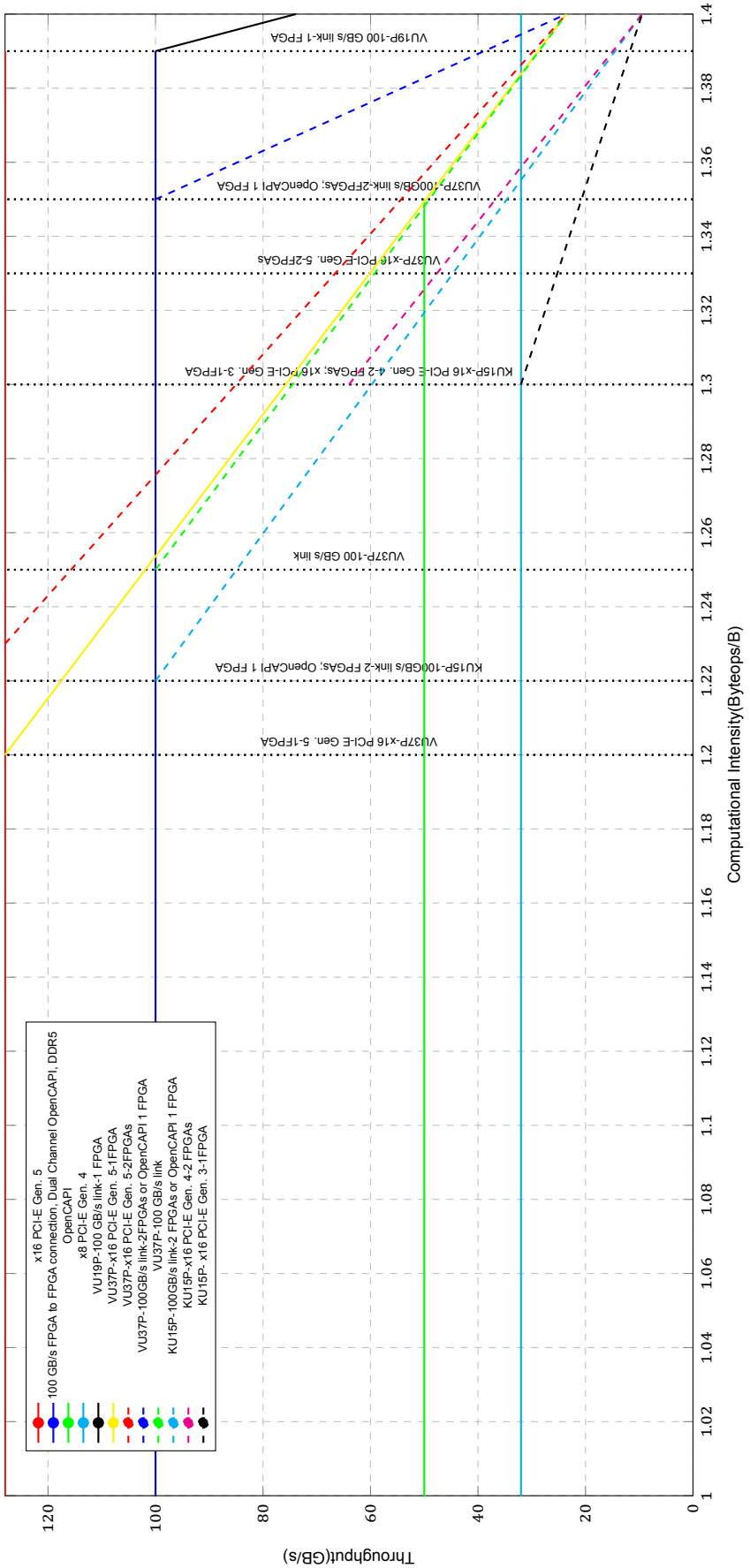


Figure 6.10: The throughput projection

7

Profiling Results

7.1. Experiment Setup

In this experiment, Q&CE power9 server is used [3]. Power9 system with two CPUs and 128GB DDR4 memory. The FPGA accelerator card ADM-PCI-9H7 is used and interface with OPENCAPI. There are 44 total cores and 176 threads in the system. Benchmarks are done on the TPC-H lineitem table. The database generator of TPC benchmarks generates the table. TPC benchmark database generator creates a *.tbl* file. Afterward, by using Apache Arrow C++ API, a parquet file is generated. Then, to make this parquet file compatible with FPGA implementation, the parquet-mr package with Java API is used to create Parquet files with version 1. The parquet files have *PLAIN* encoding and are decompressed. The experiment is done on the following configurations:

- Single parquet file with the scale factor of 1.
- Single parquet file with the scale factor of 2.
- Directory with ten different parquet files, each having a scale factor of 1.
- Directory with ten different parquet files, each having a scale factor of 2.
- Directory with 20 different parquet files with half of them being scale factor of 1 and another half 2.

The application is run on the single worker setup with multiple cores. The core size is swept from 1 to 10. It is hard to estimate the execution times of applications with multiple cores. We will investigate them by calculating the average runtime per task. We will also calculate throughput numbers with average runtime per task. The experiments are conducted 11 times successively. The first run of each experiment is not counted since it includes task deserialization times that should not reflect the overall performance. The runtimes are the average of 10 runs.

The results will be plotted regarding several different execution times, namely, FPGA, Accelerated Spark(FPGA), Vanilla Spark. FPGA shows the total FPGA execution time for that query. Accelerated Spark(FPGA) is the total Spark runtime, including aggregating the partitions for Query 6, parsing metadata of the parquet files, and path of the files. Vanilla Spark is the Spark execution time without any acceleration.

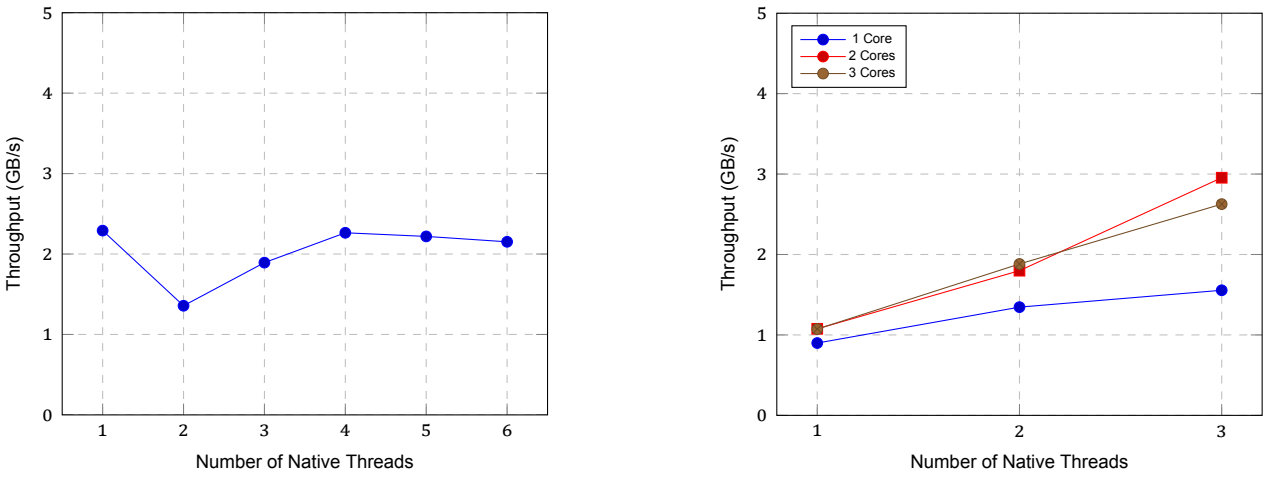
As a reference for following experiments, it should be noted that Spark core means CPU Threads dispatched by Apache Spark. In the next sections, each configuration will be tested and discussed.

7.2. Architecture A

The first architecture, which was benchmarked, is with 3 query pushdown parquet readers, as depicted in Figure 5.12. Each module can stream a single element per cycle.

7.2.1. Throughput Analysis

As discussed earlier in Section 4, scheduling on constrained resources, which is in our case FPGA, is complex. Moreover, each thread reading and writing MMIO registers require a locking mechanism to ensure the registers are being set correctly. In this section, we will investigate the effect of the increase in thread count in *Native ColumnScheduler* native object on FPGA runtime throughput. Figure 7.1 visualizes the change in the throughput as number of native threads (Figure 7.1a) and Spark threads (Figure 7.1b) change. We observe that highest achieved throughput is slightly less than 3 GB/s for software architecture with 3 native threads and 2 Spark Cores.



(a) Single file with 1.4 GB

(b) 10 files each having 1.4 GB size

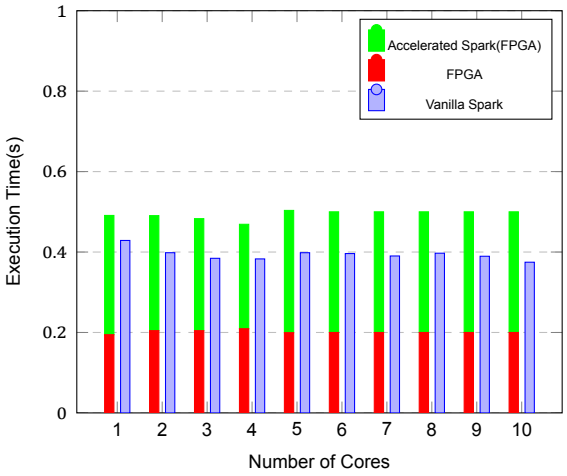
Figure 7.1: Throughput calculation for different native thread counts

7.2.2. Single Parquet File

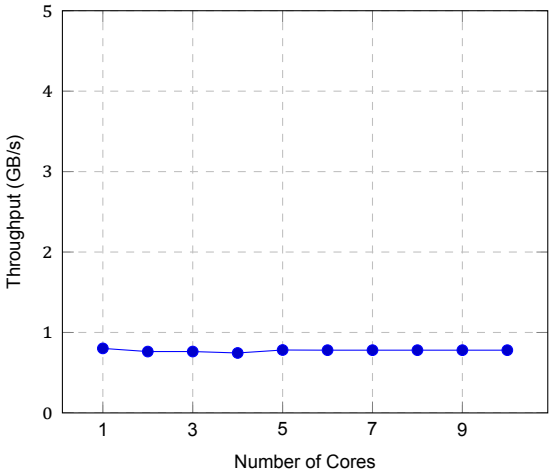
The experiments are conducted on single parquet files with 703MB(scale factor of 1) and 1.4 GB (scale factor of 2). The parallelization of the File Reading operation is heavily dependent on the parquet file. As explained in Section 2.4.2, the row groups are created on the creation of the parquet file. Figure 7.2 illustrates the single parquet file that has 0.686 GB size, reading with aggregation. 0.686 GB has a single row group. As seen in Figure 7.2a, for a single row group, both vanilla and accelerated Spark cannot parallelize the parquet file reading and aggregation operations. Figure 7.2b depicts the throughput of almost 3.5 GB/s for each accelerated task. On the other hand, the other parquet file with the size of 1.4 GB is visualized in Figure 7.3. Parquet file with 1.4 GB of size has two row groups. Therefore, the reader should expect execution time gets lower as the core size increases up to 2 cores. As the proposed design in this thesis parallelizes row groups by *NativeScheduler*, the researcher observes no change in execution time for accelerated Spark in Figure 7.3a. Nevertheless, it should be noted that the throughput for each accelerated task, depicted in Figure 7.3b, decreases by almost 20%. The reason is that each Spark core pushes its task to *Native TaskPool*. This creates a scheduler overhead. Therefore, the throughput decreases.

7.2.3. Multiple Parquet Files

It is also desirable performance-wise to load dataset from multiple parquet files. In this section, we will use the configurations explained in the setup. The first experiment is performed on a directory with 10 different parquet files each having 0.686 GB of data as Figure 7.4 illustrates the results. Figure 7.4a plots the execution times of a directory with 6.9 GB. Each Spark task pushes jobs to the *Native ColumnScheduler* in the form of single parquet files. By the core count increases, the execution time decreases. One of the most important observations that can be interpretted from Figure 7.4b, the throughput decreases after Spark core count reaches to three. The reason is that *Native ColumnScheduler* implements three native threads. By the core count reaches three, the FPGA is busy with execution so the tasks coming from Spark are pushed in global queue. Hence, it creates a scheduler overhead.

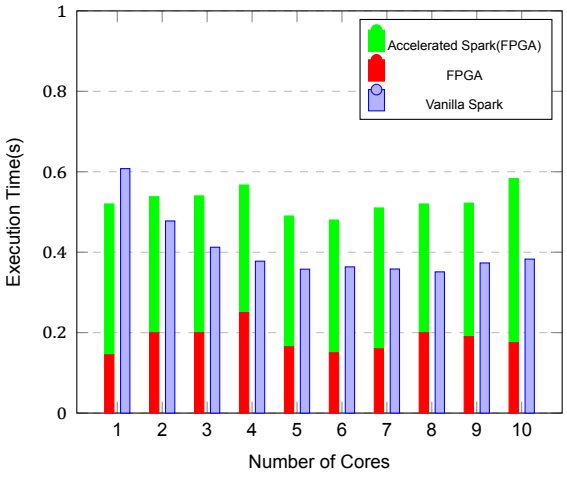


(a) Runtimes

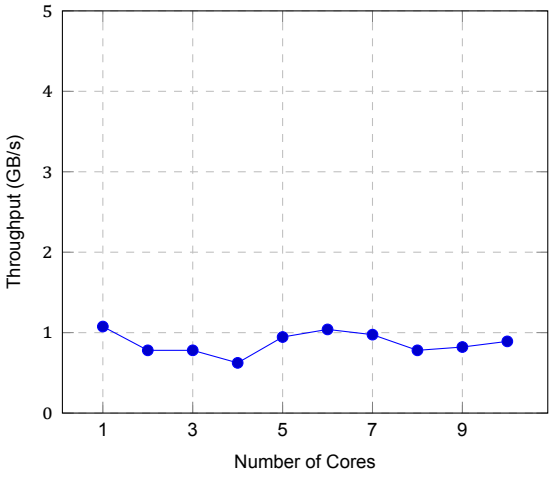


(b) Throughput per accelerated task

Figure 7.2: Total Size of 0.686 GB



(a) Runtimes



(b) Throughput per accelerated task

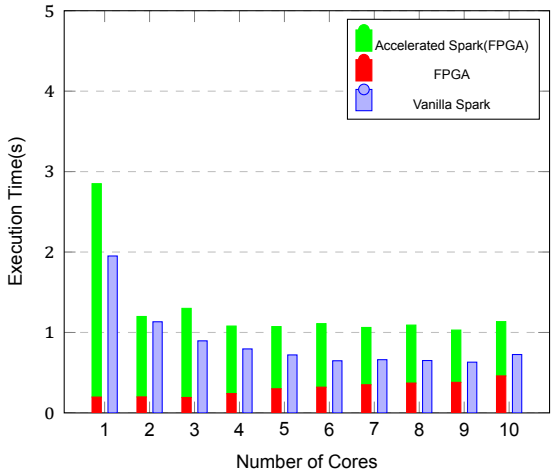
Figure 7.3: Total Size of 1.4 GB

Figure 7.5 shows the benchmark results for a directory with ten different parquet files, each having 1.4 GB of size and two row groups. The execution times decrease by half once the core count increases to two, as shown in Figure 7.5a. Two components are contributing to the execution time of accelerated workers. The first one contributes to the Spark computations such as aggregating partitions and reading metadata, file path. Secondly, FPGA execution contributes to the total runtime. We observe that FPGA execution slows down significantly after a single core. Since each parquet file has two row groups, after core size equals 1, the queue size becomes larger than 3. In this case, the scheduler overheads start to contribute to FPGA execution time. Figure 7.5b depicts the decrease in throughput due to scheduler overheads.

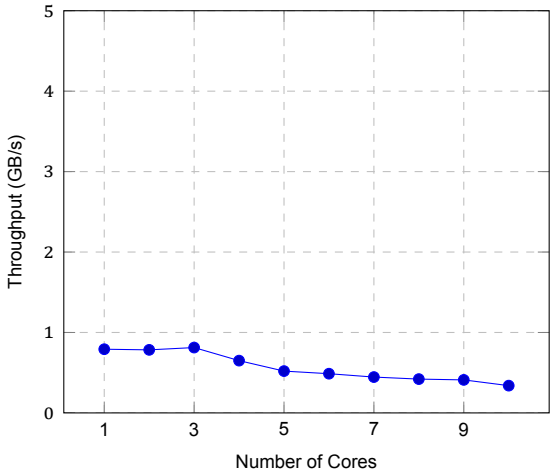
The final data source that we ran our experiments on was a directory of size 21 GB that includes ten files of size 0.686 GB and 1.4 GB, as shown in Figure 7.6. Figure 7.6a plots that the Spark runtimes increase in case of the number of tasks increases. In this case, the application creates 20 tasks for each file. As a result, the FPGA throughput can reach slightly over 6 GB/s as seen in Figure 7.6b.

7.3. Architecture B

The second architecture which was is benchmarked is with 3 query pushdown parquet reader, each module processing 4 elements per cycle. The formulation behind this setup is to increase the through-

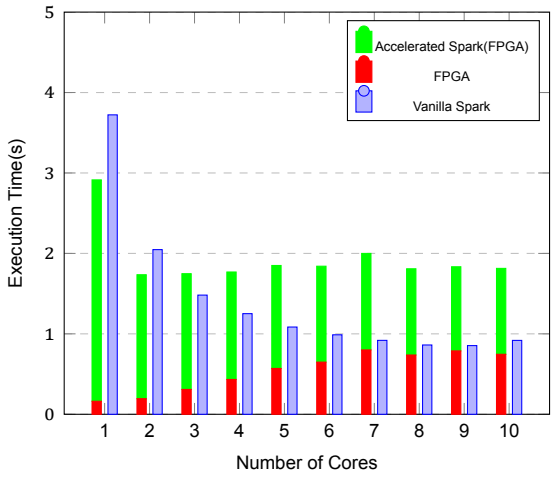


(a) Runtimes

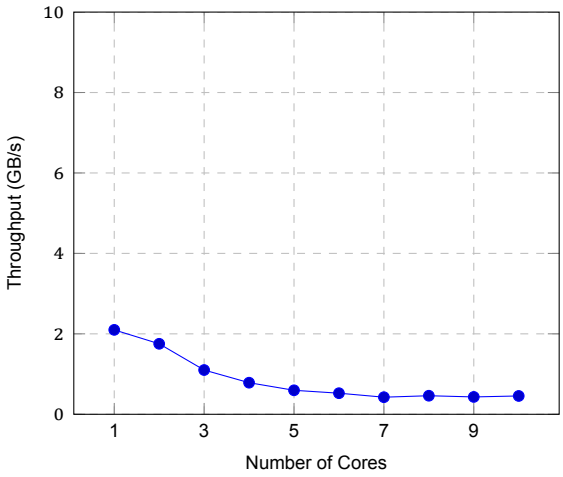


(b) Throughput per accelerated task

Figure 7.4: Total Size of 6.9 GB



(a) Runtimes



(b) Throughput per accelerated task

Figure 7.5: Total Size of 14 GB

put while fitting in FPGA. The overall architecture is provided in Figure 7.7. The synthesis report for single elements per cycle was provided in Chapter 6. Table 7.1 shows the utilization report for design that can process 4 elements per cycle. Since board support package and configuration packages of OC-ACCEL are the same, we do not report them again.

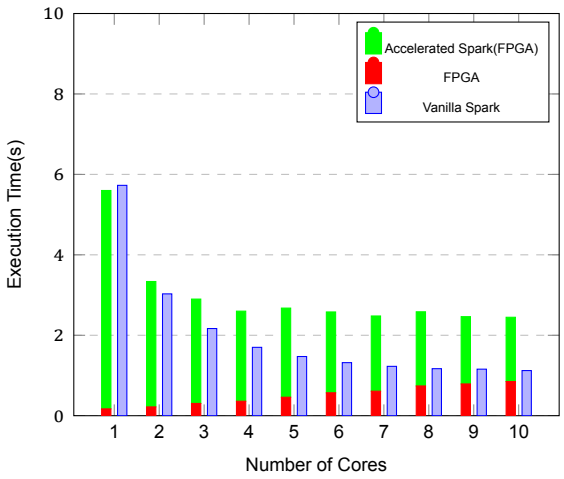
The same set of experiments are conducted in this section. This experiment shows the increase in throughput by increasing the parallelism in the hardware.

7.3.1. Throughput Analysis

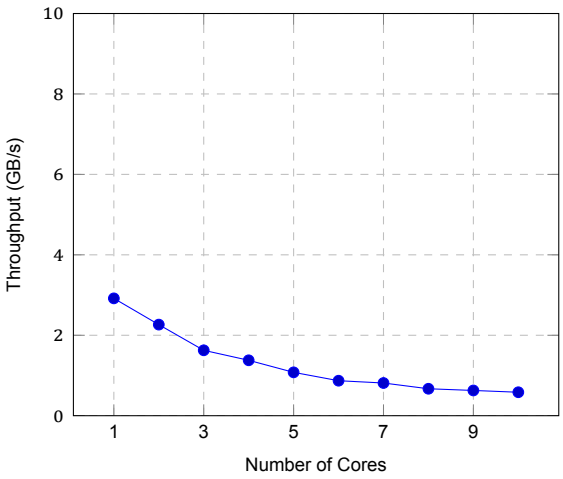
The setup in Architecture in Section 7.2 achieves the maximum throughput of 2.1 GB/s. The Architecture 7.3 achieves 3 GB/s as illustrated in Figure 7.8a. The throughput of FPGA decreases as the Spark core count increases as depicted in Figure 7.8b.

7.3.2. Single Parquet File

The discussions done in the Section 7.2.2 can also be drawn in this section. Figure 7.9a shows that the kernel speed-up is 3.96x while application speed-up is 1.55x. The throughput is roughly 2.02 GB/s for each case, as seen in Figure 7.9b. Moreover, for larger parquet file seen in Figure 7.10a, can reach up to 2x kernel speed-up whereas 1x application speed-up (no speed-up). The corresponding throughput



(a) Runtimes



(b) Throughput per accelerated task

Figure 7.6: Total Size of 21 GB

Module	CLB LUTs(Used)	BRAM Tiles(Used)
oc_func	37.22%(485243)	81.70%(1647)
fw_afu	37.19%(484812)	81.70%(1647)
fletcher_axi_top	34.11%(444714)	79.91%(1611)
extended_col(64 bit)	1.10%(14282)	6.60%(133)
shipdate_col(32 bit)	0.89%(11583)	3.57%(72)
predicate_instance	6.49%(84607)	3.27%(66)

Table 7.1: Utilization Report of query pushdown parquet reader implementation with epc=4

is in average 2.82 GB/s as shown in Figure 7.10b.

7.3.3. Multiple Parquet Files

Figure 7.11a shows the execution times of accelerated workers for a directory with 6.9 GB of size. The peak FPGA throughput of roughly 3.8 GB/s is observed in this experiment as visualized in Figure 7.11b. It appears that the throughput does not change much in this setup. The previous experiments reported that the throughput decreases once the Spark core count increases.

Once the directory size increases, the execution times get larger for Vanilla Spark. As represented in Figure 7.12a, the application speed-up we achieve the most is 1.357x for TPC-H Query 6, while kernel speed-up is roughly 30.8x. The maximum throughput is less than 3 GB/s as seen in Figure 7.12b.

The largest directory size of 21 GB is benchmarked with the current design. Figure 7.13a illustrates that the end-to-end speed-up of 2.5x is the highest achieved speed-up. The kernel speed-up of such design achieves 13.18x. The highest speed-up occurs on single-core. Moreover, as the file size increases, FPGA starts to perform better than the multi-threaded Spark application. The overall execution of accelerated worker is faster than of the vanilla spark until core size equals seven. With six CPU threads, the FPGA can achieve 1.13x end-to-end application speed-up and a kernel speed-up of 13.19x.

The total FPGA execution time(red) is always smaller than vanilla spark runtime. In terms of throughput, Figure 7.13b draws the average throughput numbers, which has a peak of 2.3 GB/s.

7.4. Large Tables

In this section, parquet files with approximately 720, 540, and 360 million rows are tested on a single core, while parquet files reside in the disk. This experiment aims to see the individual parts of the runtime contributing to the FPGA execution and parquet reading, etc. Figure 7.14 plots the runtimes for each row count on single core. The highest achieved end-to-end speed-up is 3.88x. At the same

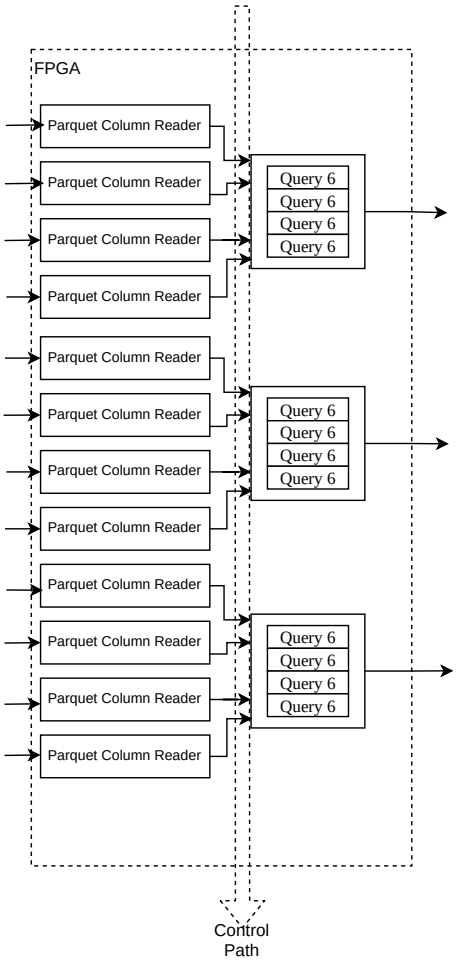


Figure 7.7: Architecture of kernel which can process 4 elements per cycle

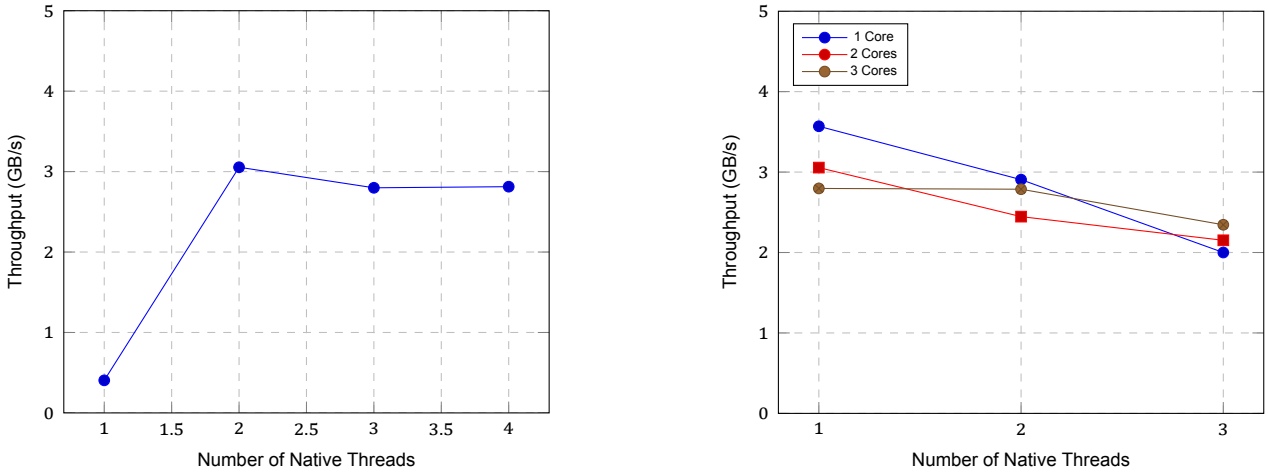
time, the kernel speed-up is 7.24x.

7.5. Discussions

There are several conclusions of this part worth mentioning. Firstly, we have observed that once the number of cores increases, throughput for the FPGA calculation decreases. This can be explained in several ways. One way is regarding the synchronization of a scheduling operation. Scheduling as described in Section 4 is resource-constrained scheduling with only a single FPGA instance. Once every thread tries to write to the same register, there will be spinlock and, they will wait for resources to be available. Moreover, for parquet files with a smaller number of row groups, increasing the core count will result in much higher wait times. The reason is that we parallelize the row groups with *Native ColumnScheduler* and files with *SparkContext*. If we initiate 3 threads in *Native ColumnScheduler* but have 1 row group, the other 2 threads will be idle and waiting for *SparkContext* to push new files to the *Native ThreadPool*.

As seen from the profiling results, pushing projection to reading operation will result in up to 7.3x speed-up (kernel). However, the utilization of such kernel is roughly 37% as depicted in Table 6.2. Therefore, one can add more query pushdown parquet readers or increase the processed elements per cycle to utilize more bandwidth. This thesis proves that adding more parallel query processing modules increases the throughput. As a result, experiment Setup A has the maximum throughput of 3 GB/s. At the same time, Experiment Setup B has 3.8 GB/s.

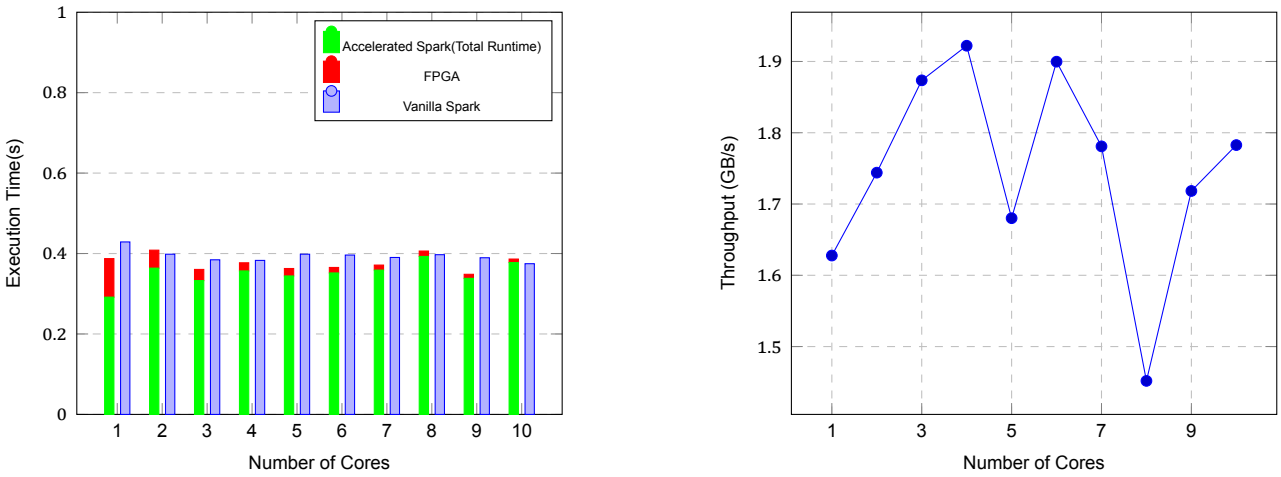
Another conclusion derived from these results is the scaling of FPGA resources compared to the CPU. More specifically, how well the FPGA tasks(accelerated worker) perform once the parallelism level increases. The reader can observe that the performance of each implementation increases as



(a) Single file with 1.4 GB

(b) 10 files each having 1.4 GB size

Figure 7.8: Throughput calculation for different native thread counts



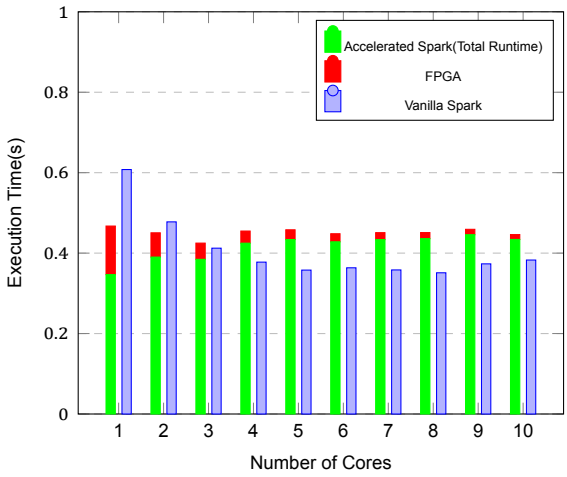
(a) Runtimes

(b) Throughput per accelerated task

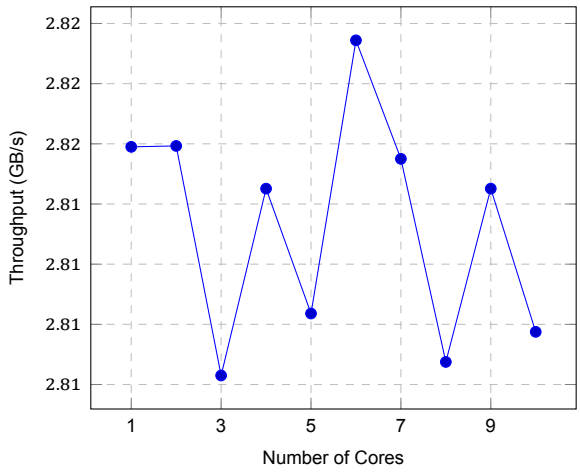
Figure 7.9: Total Size of 0.686 GB

the number of Spark cores increases until some point. CPU performs better than FPGA for large core counts. The reason for this result is explained at the beginning. The performance of CPU overtakes of FPGA for smaller parquet files. For instance, for smaller parquet files, we observe that the CPU can perform better than FPGA once the core count reaches two. Nevertheless, for the total size of 21 GB, seven parallel cores are needed for the CPU to perform better. The FPGA runtimes(red) are smaller than vanilla Spark runtimes for all core counts in both designs. Once the overheads are resolved, multi-threaded FPGA applications can achieve high speed-ups. The highest speed-up observed in experiments occurs in Figure 7.13a.

One other purpose of this thesis was to examine the boundaries of FPGA utilization. We observed that the throughput numbers reach 3.8GB/s, the highest. This is far below the practical boundaries of the platform used in this experiment. The experiment is run on OpenCAPI interconnect with up to 21GB/s practical bandwidth. As stated in the experiment setup, the throughput numbers are calculated on average per task. This means that each Spark task's utilization of the FPGA resources is not ideal. One can overcome this problem by parallelizing row groups on the Spark API calls. In this way, Spark can launch more parallelizable tasks that will keep the FPGA as busy as possible.

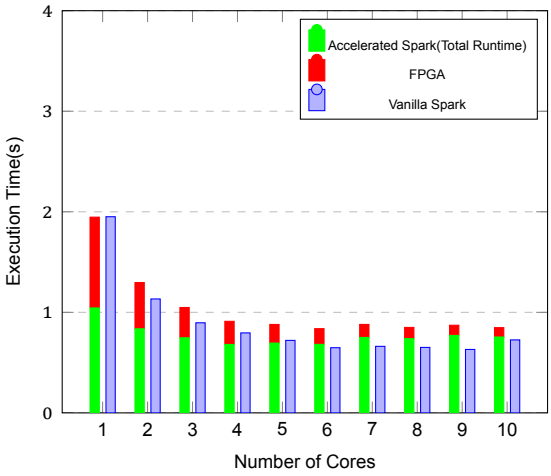


(a) Runtimes

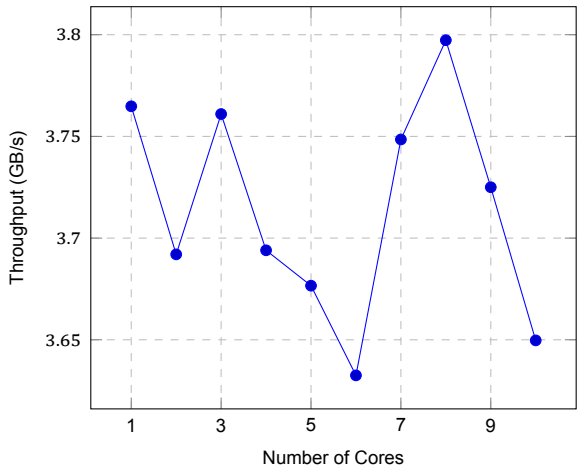


(b) Throughput per accelerated task

Figure 7.10: Total Size of 1.4 GB

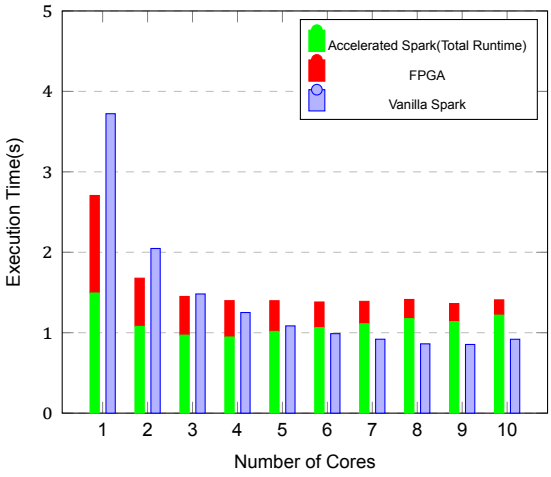


(a) Runtimes

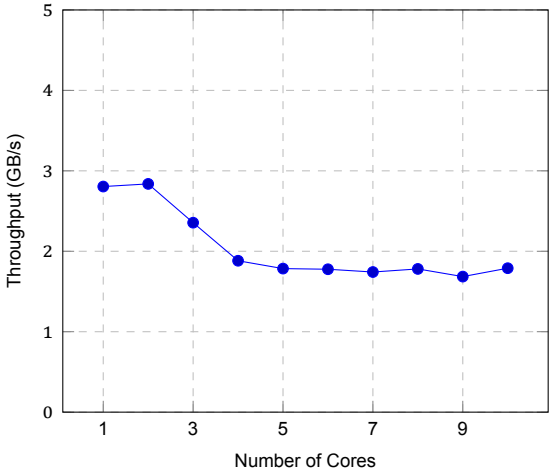


(b) Throughput per accelerated task

Figure 7.11: Total Size of 6.9 GB

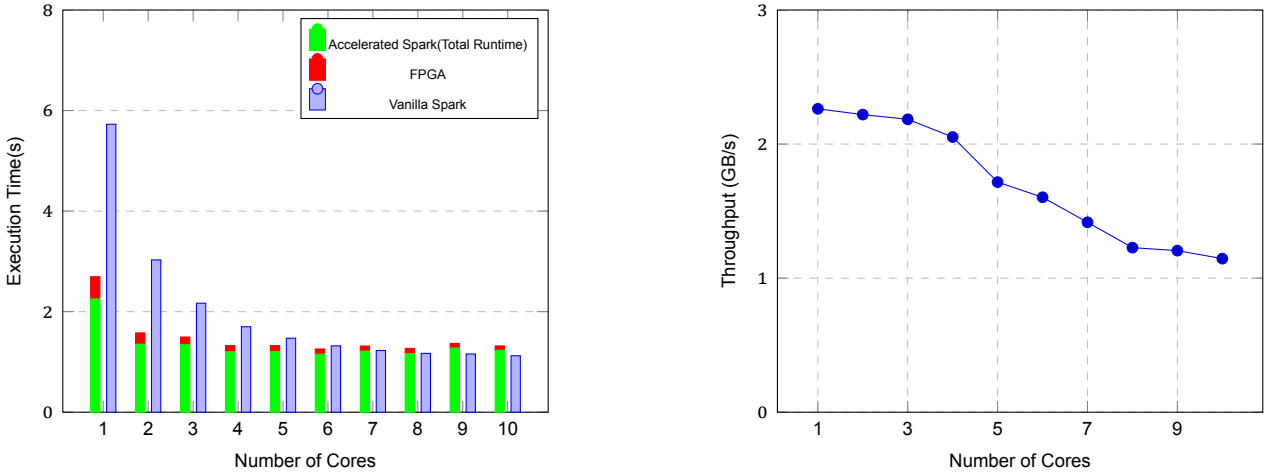


(a) Runtimes



(b) Throughput per accelerated task

Figure 7.12: Total Size of 14 GB



(a) Runtimes

(b) Throughput per accelerated task

Figure 7.13: Total Size of 21 GB

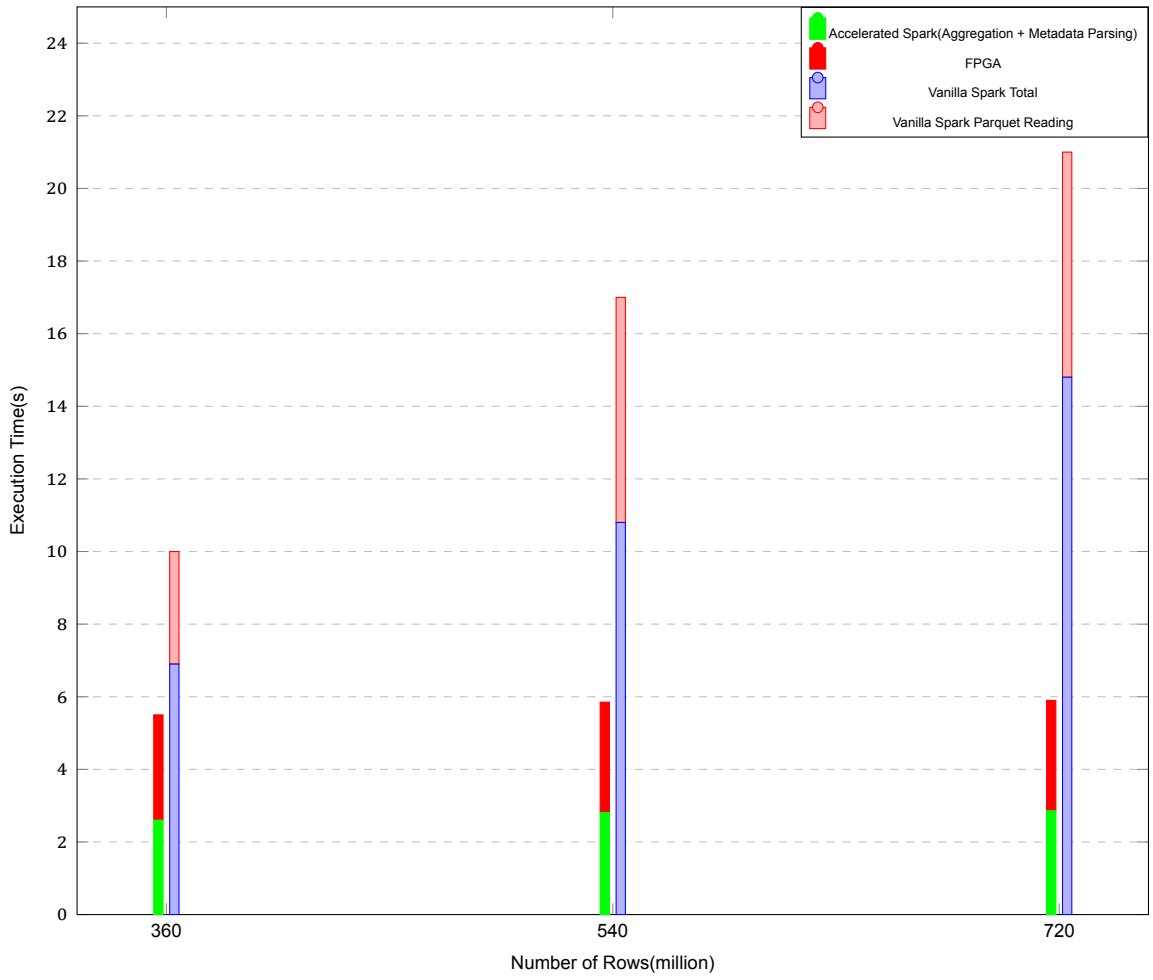


Figure 7.14: Runtimes

8

Conclusion and Future Research

8.1. Conclusion

This thesis aimed to investigate the limitations of query pushdown and propose a hardware and software design to enable this from end-to-end applications. An extended roofline model for analyzing single hardware parquet to arrow conversion operators and TPC-H Query 6 design is used to discover the limitations of current designs for different interconnect technologies. We propose extensions to existing parquet reading hardware designs to enable query pushdown operations on hardware. Apache Spark integration of hardware query pushdown parquet reading operator is shown. Possible different accelerated cluster configurations and limitations of such configurations were evaluated. This thesis enhances our understanding of the integration of FPGAs in the context of parallelized big data frameworks.

The acceleration of Spark SQL workloads on FPGAs creates a significant challenge in terms of cluster configurations. The challenge arises from whether the acceleration is performed on a single FPGA setup or with multiple FPGAs. This thesis investigated the acceleration opportunities and bottlenecks for a single FPGA setup in a multi-threaded environment. This thesis introduces an Apache Spark integration by using the basic building blocks of the Apache Spark File Scan operator. To concurrently access a single FPGA instance with multiple threads, a concurrent native task scheduler is implemented.

One remaining challenge is that a single FPGA instance creates bottlenecks in terms of synchronization. Multiple Spark jobs, accessing a single FPGA, require multi-level task management. The findings in this thesis suggest that such synchronization can create an extra overhead due to the proposed synchronization method. This is because multiple tasks busy-wait on each other to release the FPGA control and status registers. Moreover, this busy-wait results in an increase in CPU load, and thus the power usage also increases. These kinds of limitations create a challenge for future research directions. These overheads resulted in decreasing application throughput. However, TPC-H Query 6 performs better than CPU with a single CPU thread by achieving a 3.88x application speed-up on single core setup for parquet file containing 720 million rows. Moreover, a multi-threaded FPGA implementation can perform better than a CPU, while the number of parallel threads is less than seven. This implies that if the synchronization bottlenecks are resolved and investigated further, FPGA can overtake CPU for even more parallel threads.

The roofline analysis has significant implications for understanding the bottlenecks of query pushdown operations. It has been explored that these operations are always I/O-bound for TPC-H Query 6, a simple ad-hoc query with a single aggregation. Once the hardware design gets more complex than the single aggregation, the application becomes compute-bound for high bandwidth interconnects by including decompressors. The current system designs implement PCI-E Gen. 3 as their interconnect type. The roofline model shows that both the parquet reader and the query pushdown parquet reader hardware designs are I/O bound. Nevertheless, interconnect types such as OpenCAPI can yield high-performance gains for more complex designs while achieving high scalability.

The thesis answered the research questions set out in the introduction of the thesis as follows.

How can query pushdowns be integrated into the state-of-art parquet decoder in dataflow

hardware designs? Chapter 5 proposes an integration solution to push TPC-H Query 6 down parquet reading stage. The architecture investigated in this thesis is bundling parquet column readers, and query to stream read rows into query. Moreover, one can simply extend the query as a separate entity with its controller to have the flexibility of setting the control registers.

How much of a query can we accelerate on FPGA? What is the bounding factor for query pushdown? For simple queries with low computational intensity, the acceleration of a complete query design makes more sense as it results in the highest reduction in memory access. Using the system proposed in Chapter 6, one can decode parquet files in Arrow Columnar formats which are held in on-chip memory on FPGA and then run individual complex operators. There is no direct answer for partial acceleration as the performance will depend heavily on the type of operator being accelerated. However, one can map the computational intensity of the accelerated part in the Roofline Model in Chapter 6 and obtain an attainable performance estimation for the accelerated part.

Concerning the bounding factor for query pushdown, the attainable performance of parquet decoding operations is bound by the I/O for parquet files with fewer columns. Therefore, using an interconnect capable of having a large bandwidth will increase the attainable performance. Unfortunately, many storage-attached accelerators utilize PCI-E gen. 3, which has lower bandwidth compared to other next-generation interfaces. Thus, the first bounding factor is interconnect bandwidth. Secondly, attainable peak performance decreases when the number of instances that can fit on the FPGA decreases (scalability), as described in Chapter 6. Therefore, scalability can be defined as the second limitation for query pushdown.

8.2. Future Research

This study has shown that the computation can be seamlessly pushed down to a parquet reader on a (possibly storage-attached) FPGA without introducing extra host memory accesses. Throughout the thesis, we have discussed several different limitations which curb the performance of our designs. The future research opportunities can be summarized as follows.

- Many ad-hoc queries create good acceleration opportunities. This thesis investigates the TPC-H Query 6. In addition, we implemented and did initial measurements on a TPC-H Query 1 design with a higher compute roof. As roofline analysis shows, there are opportunities to challenge the compute roof with newer queries and make the design I/O bound for larger datasets.
- This thesis proposed a system design in Chapter 6. The profiling results in Chapter 7 are obtained on the Power9 server. In the future, a system that puts the FPGAs in the datapath between storage and host memory (storage-attached) can be helpful to investigate and prove the claims in this thesis.
- Roofline Analysis is conducted only on FPGAs. It would reveal more information if CPU throughputs are also included. However, this is not trivial as the CI of hardware design is constructed by byte operations. Whereas, CI of CPU is calculated by floating-point operations.
- The proposed designs in this thesis do not use additional FPGA memory resources such as DRAM and HBM. This makes the designs very suitable for SmartSSDs (that may use relatively small FPGAs compared to datacenter accelerator cards) but limits the achievable acceleration for full queries that include complex operations such as Join and GroupBy.
- Full query designs on hardware are usually desired once the query is as simple as TPC-H Query 6. Nevertheless, the compute roof for filter designs is simple and easily parallelizable. Therefore, one can integrate a single filter design in the modules proposed in Figure 5.4.
- The proposed concurrent thread scheduler has limitations due to the synchronization of a single FPGA instance. Other types of synchronization primitives such as semaphores or critical sections can be implemented to schedule FPGA tasks. However, parallelizing workloads on a single FPGA is only the ”tip of the iceberg.” This thesis successfully achieves to discovers a design that can parallelize tasks for a single FPGA (single worker, multiple threads). This can be extended for multiple FPGAs in single *Spark Context*. Moreover, one can also implement an architecture such that each *Spark Context* accesses its FPGA instance, and the cluster manager distributes each FPGA to each context (multiple workers, multiple threads).

Bibliography

[1] Swarm 64postgresql benchmarks. URL https://swarm64.com/benchmarks/.

[2] URL https://parquet.apache.org/.

[3] Q&ce servers. URL http://qce-it-infra.ewi.tudelft.nl/qce_servers.html.

[4] 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 '16, page 1–9, New York, NY, USA, 6 2016. Association for Computing Machinery. ISBN 9781450343190. doi: 10.1145/2933349.2933360. URL https://doi.org/10.1145/2933349.2933360.

[5] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, November 2002. ISSN 1066-8888. doi: 10.1007/s00778-002-0074-9. URL https://doi-org.tudelft.idm.oclc.org/10.1007/s00778-002-0074-9.

[6] Amazon. Amazon ec2 f1 instances. URL https://aws.amazon.com/ec2/instance-types/f1/.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: relational data processing in spark. 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 1383–1394. ACM, 2015. doi: 10.1145/2723372.2742797.

[8] Apache Arrow. Apache arrow. URL https://arrow.apache.org/docs/.

[9] Azure. What is azure stack edge pro fpga?what is azure stack edge pro fpga? URL https://docs.microsoft.com/en-us/azure/databox-online/azure-stack-edge-overview.

[10] C. W. Bachman and S. B. Williams. A general purpose programming system for random access memories, 1964.

[11] Charles W. Bachman. Data structure diagrams. 1:4–10, 1969. ISSN 0095-0033. doi: 10.1145/1017466.1017467.

[12] Lekshmi Beena Gopalakrishnan Nair, Andreas Becher, Klaus Meyer-Wegener, Stefan Wildermann, and Jürgen Teich. SQL Query Processing Using an Integrated FPGA-based Near-Data Accelerator in ReProVide. In *Proceedings of EDBT*, page 4, 2020.

[13] Kristi L Berg, Tom Seymour, Richa Goel, et al. History of databases. *International Journal of Management & Information Systems (IJMIS)*, 17(1):29–36, 2013.

[14] Uri Berman, Carl Chamberlin, Don Lundberg, Larry Morgan, Ed Morris, and Vern Watts. https://www.ibm.com/ibm/history/ibm100/us/en/icons/ibmims/. URL https://www.ibm.com/ibm/history/ibm100/us/en/icons/ibmims/.

[15] Weiting Chen Calvin Hung. Accelerating spark sql workloads to 50x performance with apache arrow-based fpga accelerators, 2020. URL https://databricks.com/session_na20/accelerating-spark-sql-workloads-to-50x-performance-with-apache-arrow-based-fpga-acce

[16] Allan Cantle. Opencapi enabled fpgas the perfect bridge to a data centric world. October 2018.

[17] E. F. Codd. A relational model of data for large shared data banks. 13:377–387, 1970. ISSN 0001-0782. doi: 10.1145/362384.362685.

[18] E.F Codd, S.B Codd, C.T Salley, F Codd, S. Codd, and C. Salley. Providing olap to user-analysts: An it mandate. 1993. URL https://www.scienceopen.com/document?vid=9bd0d512-0ec5-4cb7-864c-6c02890637be.

[19] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, SIGMOD '85, page 268–279, New York, NY, USA, 1985. Association for Computing Machinery. ISBN 0897911601. doi: 10.1145/318898.318923. URL https://doi-org.tudelft.idm.oclc.org/10.1145/318898.318923.

[20] TPC Transaction Processing Performance Council. *TPC Benchmark^{TM} DS Standard Specification Revision*. TPC - Transaction Processing Performance Council, February 2021. Rev. 3.0.0.

[21] TPC Transaction Processing Performance Council. *TPC Benchmark^{TM} H Standard Specification Revision*. TPC - Transaction Processing Performance Council, February 2021. Rev. 3.0.0.

[22] Bruno da Silva, An Braeken, Erik H. D'Hollander, and Abdellah Touhafi. Performance modeling for FPGAs: Extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing*, 2013:1–10, 2013. doi: 10.1155/2013/428078.

[23] Florent de Dinechin and Bogdan Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

[24] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 1221–1230, New York, NY, USA, 6 2013. Association for Computing Machinery. ISBN 9781450320375. doi: 10.1145/2463676.2465295. URL https://doi.org/10.1145/2463676.2465295.

[25] IBM Cloud Education. Olap. 2020. URL https://www.ibm.com/cloud/learn/olap.

[26] IBM Cloud Education. Oltp. 2020. URL https://www.ibm.com/cloud/learn/oltp.

[27] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling, 2011.

[28] Jian Fang, Jianyu Chen, Jinho Lee, Zaid Al-Ars, and H.Peter Hofstee. Refine and recycle: A method to increase decompression parallelism. volume 2160-052X, pages 272–280, New York, NY, USA, 2019. IEEE. ISBN 978-1-7281-1602-0. doi: 10.1109/ASAP.2019.00017.

[29] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. In-memory database acceleration on fpgas: a survey. *The VLDB Journal*, 29(1):33–59, oct 2019. ISSN 0949-877X. doi: 10.1007/s00778-019-00581-w. URL https://doi.org/10.1007/s00778-019-00581-w.

[30] M. Franklin, R. Chamberlain, M. Henrichs, B. Shands, and J. White. An architecture for fast processing of large unstructured data sets.

[31] PK Gupta. Xeon+fpga platform for the data center. URL https://research.ece.cmu.edu/calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf.

[32] A. Hadnagy. Dataflow hardware design for big data acceleration using typed interfaces. Master's thesis, TU Delft. URL http://resolver.tudelft.nl/uuid:38d9b35e-2d75-4cab-b6d1-723c2849badb.

[33] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. 15: 287–317, 1983. ISSN 0360-0300. doi: 10.1145/289.291.

[34] Joost Hoozemans, Johan Peltenburg, Fabian Nonnemacher, Akos Hadnagy, Zaid Al-Ars, and H. Peter Hofstee. Fpga acceleration for big data analytics: Challenges and opportunities. *IEEE Circuits and Systems Magazine*, 21(2):30–47, 2021. doi: 10.1109/MCAS.2021.3071608.

[35] IBM. Ibm netezza performance server. *IBM Netezza Performance Server*, . URL https://www.ibm.com/docs/en/netezza.

[36] IBM. *IBM Power System AC922*, . URL https://www.ibm.com/downloads/cas/6PRDKRJ0.

[37] IBM. Ramac the first magnetic hard disk, . URL https://www.ibm.com/ibm/history/ibm100/us/en/icons/ramac/.

[38] Intel. Opae. URL https://01.org/opae.

[39] Kaan Kara and Gustavo Alonso. Pipearch: Generic and context-switch capable data processing on fpgas. *ACM Trans. Reconfigurable Technol. Syst.*, 14(1), November 2020. ISSN 1936-7406. doi: 10.1145/3418465. URL https://doi-org.tudelft.idm.oclc.org/10.1145/3418465.

[40] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. High bandwidth memory on fpgas: A data analytics perspective. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 1–8, 2020. doi: 10.1109/FPL50879.2020.00013.

[41] Jason Lawley. Understanding performance of pci express systems white paper (wp350) - wp350. URL https://www.xilinx.com/support/documentation/white_papers/wp350.pdf.

[42] Bob Luppes. Fpgas in big data. Master's thesis, TU Delft. URL http://resolver.tudelft.nl/uuid:e358f322-3632-45d4-ad9b-6ed5691c87ab.

[43] Sally A. McKee. Reflections on the memory wall, 2004.

[44] MongoDB. What is an object-oriented database? URL https://www.mongodb.com/databases/what-is-an-object-oriented-database.

[45] Raghunath Othayoth Nambiar and Meikel Poess. The making of TPC-DS. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 1049–1058. ACM, 2006. URL http://dl.acm.org/citation.cfm?id=1164217.

[46] F.M. Nonnenmacher. Transparently accelerating spark sql code on computing hardware. Master's thesis, TU Delft. URL http://resolver.tudelft.nl/uuid:f588ca1d-e4ae-4bf4-96ed-221d483b559d.

[47] *Multithreaded Programming Guide*. Oracle, March 2019. URL https://docs.oracle.com/cd/E53394_01/pdf/E54803.pdf.

[48] Muhsen Owaida, David Sidler, Kaan Kara, and Gustavo Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218, 2017. doi: 10.1109/FCCM.2017.37.

[49] Johan Peltenburg, Jeroen van Straten, Lars Wijtemans, Lars van Leeuwen, Zaid Al-Ars, and Peter Hofstee. Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow. pages 270–277, Barcelona, Spain, 2019. IEEE. ISBN 978-1-7281-4885-4. doi: 10.1109/FPL.2019.00051.

[50] Johan Peltenburg, Lars T. J. van Leeuwen, Joost Hoozemans, Jian Fang, Zaid Al-Ars, and H. Peter Hofstee. Battling the cpu bottleneck in apache parquet to arrow conversion using fpga, 2020.

[51] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., USA, 3 edition, 2002. ISBN 0072465638.

[52] David Sidler, Zsolt Istvan, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. Doppiodb: A hardware accelerated database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1659–1662, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3058746. URL https://doi-org.tudelft.idm.oclc.org/10.1145/3035918.3058746.

[53] Xuan Sun, Chun Jason Xue, Jinghuan Yu, Tei-Wei Kuo, and Xue Liu. Accelerating data filtering for database using fpga. *Journal of Systems Architecture*, 114:101908, 2021. ISSN 1383-7621. doi: https://doi.org/10.1016/j.sysarc.2020.101908. URL https://www.sciencedirect.com/science/article/pii/S1383762120301740.

[54] TPC. Tpc benchmarks overview. URL http://tpc.org/information/benchmarks5.asp.

[55] Laurentius Theodorus Johannes van Leeuwen. High-throughput big data analytics through accelerated parquet to arrow conversion. Master's thesis, TU Delft. URL http://resolver.tudelft.nl/uuid:e64b56b7-ecdc-4f47-8aed-3dfbf7e269ac.

[56] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Publications, Special Sales Department Manning Publications Co. 20 Baldwin Road PO Box 261 Shelter Island, NY 11964 Email: orders@manning.com, February 2012.

[57] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proc. VLDB Endow.*, 7(11):963–974, July 2014. ISSN 2150-8097. doi: 10.14778/2732967.2732972. URL https://doi-org.tudelft.idm.oclc.org/10.14778/2732967.2732972.

[58] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall. 23:20–24, 1995. ISSN 0163-5964. doi: 10.1145/216585.216588.

[59] *SmartSSD Computational Storage Drive*. Xilinx. URL https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/xilinx-smartssd-computational-storage-drive-product-brief.pdf.

[60] *Floating-Point Operator v7.1*. Xilinx, December 2020. URL https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_1/pg060-floating-point.pdf.

[61] Serif Yesil, Muhammet Mustafa Ozdal, Taemin Kim, Andrey Ayupov, Steven Burns, and Ozcan Ozturk. Hardware accelerator design for data centers, 2015.

[62] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[63] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016. doi: 10.1145/2934664.

[64] Meghan Zea. Pci-sig® announces upcoming pci express® 6.0 specification to reach 64 gt/s, June 2019. URL https://www.businesswire.com/news/home/20190618005945/en/PCI-SIG®-Announces-Upcoming-PCI-Express-6.0-Specification.

[65] Xianwei Zeng. Fpga-based high throughput merge sorter. Master's thesis, TU Delft. URL http://resolver.tudelft.nl/uuid:95080fd6-bb20-48bb-b3ff-8a115621f4af.

[66] Daniel Ziener, Florian Bauer, Andreas Becher, Christopher Dennl, Klaus Meyer-Wegener, Ute Schürfeld, Jürgen Teich, Jörg-Stephan Vogt, and Helmut Weber. Fpga-based dynamically reconfigurable sql query processing. page 1–24, 8 2016. doi: 10.1145/2845087. URL https://doi.org/10.1145/2845087.

A

Summary Metrics for Different Cluster Setups

In order to prove launching multiple workers should not be in the future plans of Apache Spark, the experiment on two different local cluster modes has been employed. The input is a directory with 10 different files each having 1.6 GB size. The metrics show the summary of first stage with file reading operation. We can clearly see that garbage collection (GC) time and task deserialization time in Figure A.1 is far less than in multiple worker setup in Figure A.2.

Metric	**Min**	**25th percentile**	**Median**	**75th percentile**	**Max**
Duration	11.0 ms	15.0 ms	19.0 ms	77.0 ms	3 s
GC Time	0.0 ms	0.0 ms	0.0 ms	9.0 ms	0.5 s
Input Size / Records	4.6 KiB / 0	4.6 KiB / 0	4.6 KiB / 0	4.6 KiB / 0	160.3 MiB / 6001215
Shuffle Write Size / Records	57 B / 1	57 B / 1	57 B / 1	57 B / 1	59 B / 1
Task Deserialization Time	3.0 ms	4.0 ms	7.0 ms	10.0 ms	2 s
Scheduler Delay	7.0 ms	10.0 ms	13.0 ms	17.0 ms	86.0 ms

Table A.1: Summary Metrics for local cluster setup which has 1 worker node with 12 cores

Metric	**Min**	**25th percentile**	**Median**	**75th percentile**	**Max**
Duration	18.0 ms	27.0 ms	51.0 ms	2 s	4 s
GC Time	0.0 ms	0.0 ms	3.0 ms	94.0 ms	0.7 s
Input Size / Records	4.6 KiB / 0	4.6 KiB / 0	4.6 KiB / 0	4.6 KiB / 0	160.3 MiB / 6001215
Shuffle Write Size / Records	57 B / 1	57 B / 1	57 B / 1	57 B / 1	59 B / 1
Task Deserialization Time	5.0 ms	11.0 ms	18.0 ms	27.0 ms	6 s
Scheduler Delay	11.0 ms	16.0 ms	19.0 ms	23.0 ms	91.0 ms

Table A.2: Summary Metrics for local cluster setup which has 3 worker node with 4 executors

B

TPC-H/DS Analysis Results

The analysis in Figure B.1, Figure B.3 shows the existance of order by operations on TPC-H and TPC-DS benchmarks respectively. TPC-H Query 6,14,17,19 do not implement order by operation. In addition, TPC-DS Query 9,28,32,38,48 do not have any order by operations The analysis in Figure B.2, Figure B.4 shows the existance of order by operations on TPC-H and TPC-DS benchmarks respectively. TPC-H Query 1,6 have no join operators. TPC-DS Query 18 and 41 implements no join operation. These results are important in a sense that the queries with no sorting or join operators will reveal easiest design effort.

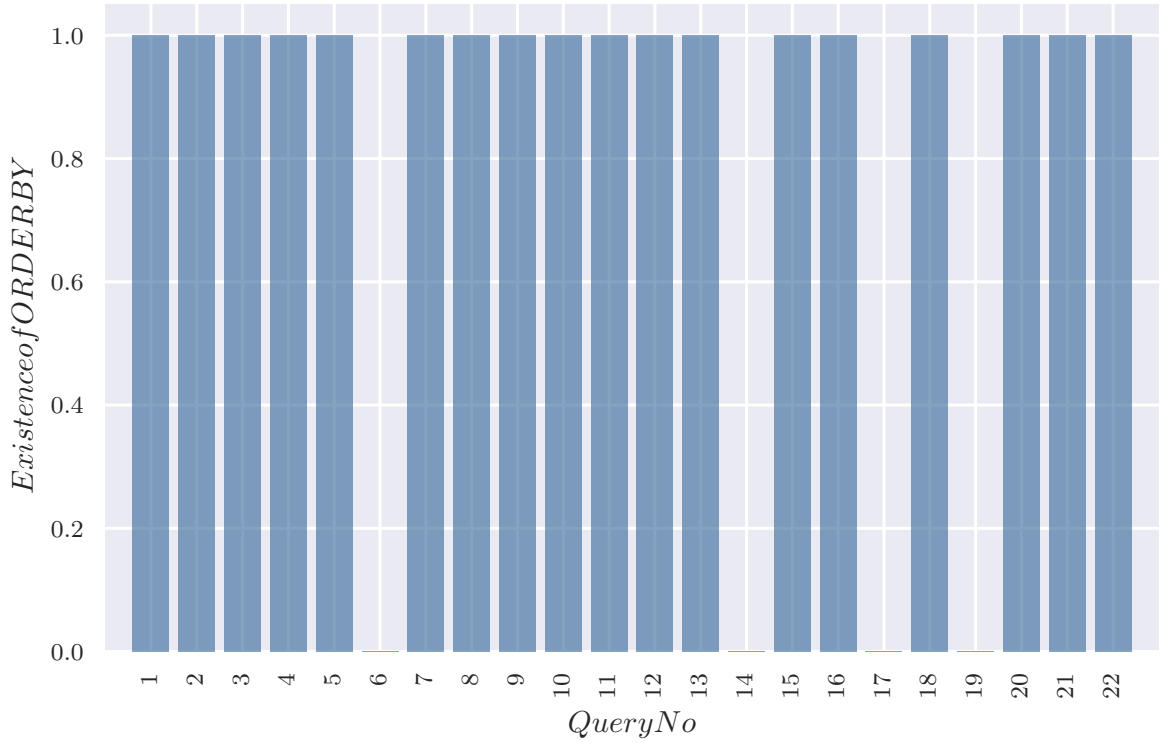


Figure B.1: TPC-H Queries with ORDER BY Operation

Figure B.2: TPC-H Queries with JOIN operation

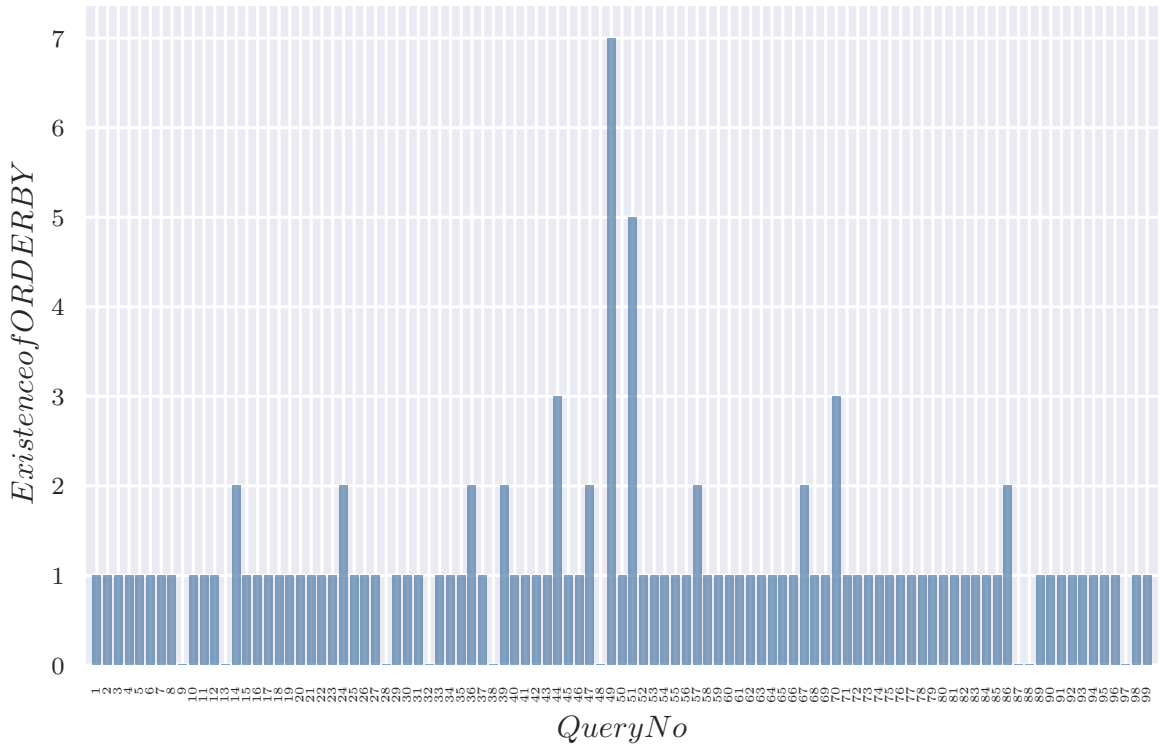


Figure B.3: TPC-DS Queries with ORDER BY Operation

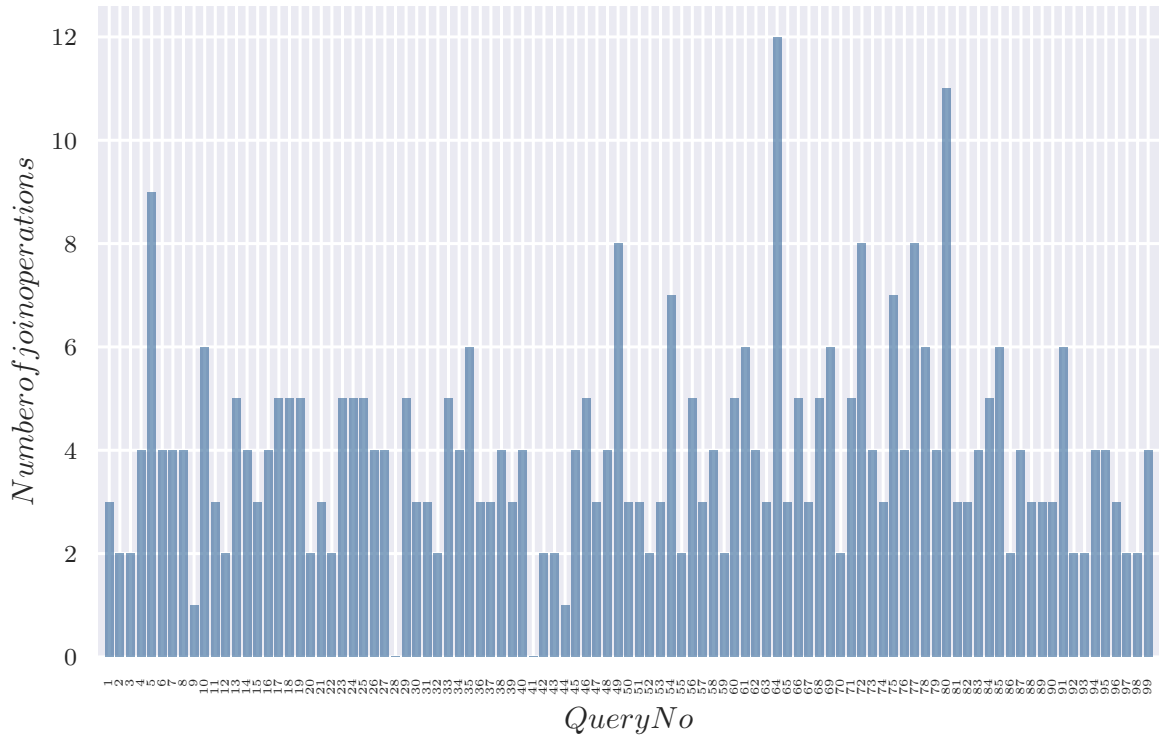


Figure B.4: TPC-DS Queries with JOIN operation