Adaptive Lightweight Compression Acceleration on Hybrid CPU-FPGA System

Dissertation

submitted October 12, 2020

by **M.Sc. Nusrat Jahan Lisa**
born June 7, 1985 in Dhaka, Bangladesh

at Technische Universität Dresden
and Aalborg University

Supervisors:
Prof. Dr.-Ing. Wolfgang Lehner
Prof. Torben Bach Pedersen

THESIS DETAILS

Thesis Title:	Adaptive Lightweight Compression Acceleration on Hybrid CPU-FPGA System
Ph.D. Student:	Nusrat Jahan Lisa
Supervisors:	Prof. Dr.-Ing. Wolfgang Lehner, Technische Universität Dresden Prof. Torben Bach Pedersen, Aalborg University

This thesis consists of the following peer-reviewed and published papers.

1. Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Tuan D. A. Nguyen, and Akash Kumar. Column scan acceleration in hybrid CPU-FPGA systems. In Rajesh Bordawekar and Tirthankar Lahiri, editors, *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018*, pages 22–33, 2018. *[Relates to Chapter 3]*
2. Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Tuan D. A. Nguyen, Akash Kumar, and Wolfgang Lehner. Column scan optimization by increasing intra-instruction parallelism. In *DATA*, pages 344–353. SciTePress, 2018. *[Relates to Chapter 3]*
3. Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Tuan Duy Anh Nguyen, and Akash Kumar. FPGA vs. SIMD: comparison for main memory-based fast column scan. In Christoph Quix and Jorge Bernardino, editors, *Data Management Technologies and Applications - 7th International Conference, DATA 2018, Porto, Portugal, July 26-28, 2018, Revised Selected Papers*, volume 862 of *Communications in Computer and Information Science*, pages 116–140. Springer, 2018. *[Relates to Chapter 3]*
4. Nusrat Jahan Lisa, Tuan Duy Anh Nguyen, Dirk Habich, Akash Kumar, and Wolfgang Lehner. High-throughput bitpacking compression. In *22nd Euromicro Conference on Digital System Design, DSD 2019, Kallithea, Greece, August 28-30, 2019*, pages 643–646. IEEE, 2019. *[Relates to Chapter 4]*
5. Nusrat Jahan Lisa, Dirk Habich, Wolfgang Lehner, Akash Kumar, and Torben Bach Pedersen. Adaptive Lightweight Integer Compression on FPGA. In preparation for submission to the journal *IEEE Embedded Systems Letters*. *[Relates to Chapter 4]*

This thesis has been submitted for assessment in partial fulfillment of the joint Ph.D. degree. The thesis is based on the published scientific papers which are listed above. As part of the assessment, co-author statements have been made available to the assessment committee and are also available at the Technical Doctoral School of IT and Design at Aalborg University and the Faculty of Computer Science at Technische Universität Dresden.

ABSTRACT

With an increasingly large amount of data being collected in numerous application areas, the importance of online analytical processing (OLAP) workloads increases constantly. OLAP queries typically access only a small number of columns but a high number of rows and are, thus, most efficiently executed by column-stores. With the significant developments in the main memory domain even large datasets can be entirely held in the main memory. Thus, main memory column-stores have been established as state-of-the-art for OLAP scenarios. In these systems, all values of every column are encoded as a sequence of integer values and, thus, query processing is completely done on these integer sequences. To improve query processing, vectorization based the Single Instruction Multiple Data (SIMD) parallel paradigm is a state-of-the-art technique. Aside from vectorization, lightweight integer compression algorithms also play an important role to reduce the necessary memory space. Unfortunately, there is no single-best lightweight integer compression algorithm, and the algorithm selection decision depends most importantly on the data characteristics. Nevertheless, vectorization and integer compression complement each other, and the combined usage improves the query performance. Unfortunately, the benefits of vectorization are limited on modern x86-processors due to predefined and fixed SIMD instruction set extensions. Nowadays, the Field Programmable Gate Array (FPGA) offers a novel opportunity with regard to hardware reconfigurable capability. For example, we can use an arbitrary length of processor word in FPGA leading to a higher performance, we can prepare proper pipeline-based custom-made database accelerators, and we can develop embedded systems through utilizing such accelerators. Moreover, modern hybrid CPU-FPGA systems have a direct data communication channel between the main memory and FPGA which is useful for throughput acceleration. Based on these advantages, this thesis examines the utilization of FPGA for main memory column-stores. This examination is two-fold. First, we investigate the column *scan* on compressed data as important operation and second, we systematically look at lightweight integer *compression*. These two aspects are considered from the hardware perspective to guarantee a certain level of query performance acceleration. In particular, this thesis explores different embedded design options and proposes an adaptive lightweight integer compression system. Based on a comprehensive evaluation, we find out the optimal design constraint as per implementation mechanism for column *scan* and lightweight integer *compression*. Finally, we conclude this thesis by mentioning our upcoming research activities.

KURZZUSAMMENFASSUNG (ABSTRACT IN GERMAN)

Die Bedeutung von analytischen Anfragen im Sinne des Online Analytical Processing (OLAP) im Datenbankumfeld nimmt stetig zu, da in zahlreichen Anwendungsbereichen von der Wissenschaft bis zur Industrie immer größere Datenmengen gesammelt werden. Um aus diesen immensen Datenmengen die notwendigen Informationen zu extrahieren, ist es notwendig, eine Vielzahl komplexer analytischer Aufgaben auszuführen. Derartige komplexe Analysen speisen sich normalerweise aus einer zwar kleinen Anzahl von Attributen (Spalten) aber einer hohen Anzahl von Zeilen und werden daher am effizientesten mit Hilfe von spaltenorientierten Datenbanksystemen verwaltet und verarbeitet. Durch die signifikante Vergrößerung der Hauptspeicherkapazitäten im Laufe des letzten Jahrzehnts können mittlerweile große Datensätze vollständig im Hauptspeicher gespeichert werden. Daher sind hauptspeicherzentrische spaltenorientierte Datenbanksysteme Stand der Technik für OLAP-Anwendungen. In diesen Systemen werden alle Werte jeder Spalte als eine Folge von ganzzahligen Werten im Hauptspeicher codiert und die Abfrageverarbeitung erfolgt vollständig auf diesen ganzzahligen Folgen. Um die Effizienz der Abfrageverarbeitung zu verbessern, ist Vektorisierung auf der Basis des parallelen Paradigmas SIMD (Single Instruction Multiple Data) in diesen Systemen von entscheidender Bedeutung. Neben der Vektorisierung spielen auch einfache Ganzzahlkomprimierungsalgorithmen zur Reduzierung des erforderlichen Speicherplatzes eine wichtige Rolle. Leider gibt es keinen optimalen Komprimierungsalgorithmus für Ganzzahlen und die Entscheidung für einen Algorithmus hängt primär von den Dateneigenschaften ab. Trotzdem ergänzen sich Vektorisierung und Ganzzahlkomprimierung, so dass die gezielte Verwendung die Effizienz der Anfrageverarbeitung verbessert. Leider sind die Vorteile der Vektorisierung bei modernen x86-Prozessoren aufgrund vordefinierter und fester Befehlssatzerweiterungen von SIMD begrenzt. Heutzutage bietet aber das Field Programmable Gate Array (FPGA) eine neuartige Möglichkeit hinsichtlich der Hardware-Rekonfigurierbarkeit an. Zum Beispiel können beliebige Längen von Prozessorwörtern in FPGA realisiert werden, was zu einer höheren Leistung führt. Darauf aufbauend können spezifische Pipeline-basierte Datenbankbeschleuniger eingebettet werden. Darüber hinaus verfügen moderne hybride CPU-FPGA-Systeme über einen direkten Datenkommunikationskanal zwischen dem Hauptspeicher der CPU und dem FPGA, was für die Durchsatzbeschleunigung wichtig ist. Hierauf aufbauend untersucht diese Arbeit die Verwendung von FPGA für hauptspeicherzentrische spaltenorientierte Datenbanksysteme und konzentriert sich dabei auf zwei Schwerpunkte. Erstens untersucht die Arbeit die Verarbeitungsoperation Spaltenscan (Column-Scan) komprimierte und zweitens wird die Ganzzahlkomprimierung systematisch betrachtet. Diese beiden Aspekte werden aus Hardware-Sicht erörtert, um ein gewisses Maß an Beschleunigung der Anfrageverarbeitung zu gewährleisten. Dazu werden in dieser Arbeit verschiedene eingebettete Entwurfsoptionen untersucht und ein adaptives Ganzzahlkomprimierungssystem vorgestellt. Basierend auf einer umfassenden Evaluierung werden optimale Entwurfsbeschränkungen ermittelt. Die Arbeit schließt mit einer Diskussion weiterführender Forschungsaktivitäten in diesem Umfeld.

DANSK ABSTRAKT (ABSTRACT IN DANISH)

I takt med at der i mange sektorer indsamles stadigt større mængder data, stiger behovet for effektivt af kunne udføre On-Line Analytical Processing (OLAP) forespørgsler. OLAP-forespørgsler tilgår typisk kun et lille antal søjler, men et stort antal rækker, og kan derfor udføres mest effektivt på data arrangeret i søjler. Samtidig gør betydelige fremskidt inden for systemhukommelse at selv store datasæt kan lagres helt i hukommelsen. Systemer der arrangerer data i søjler og lagrer komplette datasæt i hukommelsen er derfor de førende til udførsel af OLAP-forespørgsler. Disse systemer omskriver alle værdierne i hver søjle til sekvenser af heltal, og alle forespørgsler bliver udført på disse heltalssekvenser. For at udføre forespørgsler mere effektivt kan de paralleliseres ved brug af vektorisering baseret på Single Instruction Multiple Data (SIMD). Udover vektorisering spiller effektive letvægtsheltalskomprimeringsalgoritmer også en vigtig rolle da de reducerer hukommelsesforbruget. Desværre afhænger det af et datasæts karakteristika hvilken letvægtsheltalskomprimeringsalgoritme der giver det bedste resultat. Ikke desto mindre komplementerer vektorisering og heltalskomprimering hinanden, og når de kombineres, kan forespørgsler udføres mere effektivt. Desværre er effekten af vektorisering begrænset for moderne x86-processorer da deres SIMD-instruktioner er foruddefinerede og uforanderlige. Field Programmable Gate Arrays (FPGA)s åbner nye mulighed eftersom deres hardware kan ændres løbende efter behov. For eksempel kan en FPGA konfigureres som en processor med en arbitrær ordlængde hvilket fører til en højere ydeevne. Derudover kan specialiserede pipeline-baserede database acceleratorer udvikles, og indlejrede systemer kan blive udviklet som gør brug af sådanne acceleratorer. Desuden har moderne hybrid CPU-FPGA-systemer en direkte datakommunikationskanal imellem hukommelsen og FPGA'en som er nyttig til at accelerere udførslen af forespørgsler. På bagrund af disse fordele ved FPGAer, analyserer denne afhandling brugen af FPGAer i systemer der arrangerer data i søjler og lagrer komplette datasæt i hukommelsen. Analysen består af to dele. Først undersøges skanning af søjler der lagrer komprimeret data, og herefter analyseres letvægtsheltalskomprimeringsalgoritmer systematisk. Disse to aspekter bliver betragtet ud fra et hardwareperspektiv for at garantere et bestemt niveau af acceleration for forespørgsler. Specifikt udforsker denne afhandling forskellige designmuligheder for indlejrede systemer og foreslår et adaptivt letvægtskomprimeringssystem. Baseret på en omfattende evaluering, finder vi de optimale designbegrænsninger for en implementering af skanning af søjler og letvægtsheltalskomprimering. Endelig afslutter vi denne afhandling med at nævne vores fremtidige forskningsaktiviteter.

CONTENTS

INTRODUCTION

In the data-driven world, modern business technology is constantly taking care of a vast variety of customer demands. To satisfy customer requirements business technology requires to analyze and process information, whereby information is nothing but a collection of data. This leads the business organizations to deal with data from a variety of sources, including any kind of business transaction, industrial equipment, videos, sensors, social media, smart Internet of Things devices, and more. That means the amount of data is growing drastically. This gigantic amount of data arise the term "Big Data" [ES16]. Big data refers to the amount of data which is huge in volume, complex for analytics, and needs to be processed very fast [JWCW15, FLZ15]. This defines insights analytics is very crucial for big data. Moreover, big data even outnumbers Moore's law of digital circuit complexity [Sch97]. As a consequence, database researchers are continuously facing challenges to tackle big data. Big data require fast analysis in the huge amount of complex analytical queries. Fast analysis demands high-performance with high-throughput and low-latency. Unfortunately, traditional database methods are not good enough to manage analytical complex database queries more efficiently along with high throughput and low latency. This situation leads to create fast database architecture. As a result, database systems are constantly adapting novel features to satisfy high performance with low latency demands.

1.1 ANALYTICAL DATA SYSTEMS

Analytical queries are required to quickly analyze massive amounts of data. That means analytical queries are reading lots of rows but only a few columns of the database tables, as all columns are not required for processing. Thus, the organizational pattern of relational tables in the main memory influences the analytical query performance [SAB+05]. We can store a relational table either row-wise or column-wise. However, data can be more precisely accessed by storing in columns rather than by rows. In other words, column-wise storage allows us to ignore all the data that does not apply for a particular query because we can retrieve the required information from the particular columns. In contrast, the query of a row-oriented database system accesses each record along with all of its fields in the database to get the information, which involves a lot of unnecessary reads. As a result, query performance deteriorates drastically. In the case of analytical queries, row-oriented query processing is not effective as it is dealing with a large amount of data. Processing unnecessary information among the massive amounts of data is not performance-efficient. Therefore, column-wise data stores are beneficial for efficient and scalable processing of analytical queries.

Recently, the database community has been focusing on the main memory based system to efficiently exploit the ever-increasing capacities of main memory [BKM08, OBL+17, BATO13]. The main reason is, the price of semiconductor devices, e.g., transistors, becomes cheaper and the memory chip densities increase with more capacity. This enables the main memory to store larger datasets. To efficiently use this feature, database systems are shifting from disk to main memory [BKM08, OBL+17]. The two most important correlated reasons for this shifting are, i) main memory based sequential or random read and write are significantly faster over a disk, ii) fast data communication between main memory and processor words speed up the query processing speed tremendously compared to disk or flash drive based systems [MS14]. Therefore, to speed up the performance of database systems shifting from disk to main memory is fruitful. As a consequence, main memory database systems became the state-of-the-art for analytical workloads [SAB+05, BATO13]. It is important for a specific system that the main memory has a sufficient amount of space for the database. However, the space of main memories is still multi-gigabyte, while disk can be multi-terabyte.

One of the key primitives of a column-store main memory database system is scan, as analytical queries compute aggregations over full or large parts of single columns [LP13, LUH+18c]. The column-store main memory system increases the opportunity for compactness and the ability to process multiple columnar values at once. Thus, the optimization of the scan primitive is very crucial, in particular, for integer type data [LP13, LUH+18c]. Analytical queries usually involve CPU intensive operations, such as aggregation evaluation, column projection, sorting, searching, joins, and more [BATO13]. Using an efficient optimization scheme in modern hardware to process such query operations not only improves the query response time, but also minimizes the impact on query analysis by offloading the expensive queries out of massive data. However, it is highly questionable whether it is possible to hold and to process large operational data sets in main memory, as the space of main memories is still a bottleneck for large databases. Moreover, working with a giant dataset in the main memory increases the hardware maintenance cost and failure rates. To deal with such problems, data compression plays an important role in the main memory for analytical query processing. Thus, another key primitive for the main memory based database system is data compression. Data compression is a well-known optimization technique for database systems [ZHNB06, AMF06]. Generally, analytical data systems have a common approach to encode each data element as an integer value. Therefore, each column is represented as a sequence of integer values. Dictionary encoding is used for non-integer data like strings. Afterward, each sequence of integer data is compressed using a lightweight integer compression algorithm. Based on this, each column is represented as a sequence of compressed column codes. Thus, for the main memory based database system, lightweight compression algorithms are used to optimize main memory processing in terms of evaluating a query directly in the compressed form of integer data [HHDL16, AMF06]. Processing analytical queries on column-store main memory based systems accelerates performance while data are compressed through lightweight compression algorithms. In this case, the processing is completely done on the integer type of data. Any type of operator consumes integer columns for processing and provides outputs of integer columns. During processing analytical queries on compressed datasets in main memory, late materialization occurred to decrease latency, whereby tuples are reconstructed on demand for each operator in the query plan. Lightweight compression algorithms are handy as operators are working directly on compressed data which accelerates the performance drastically. However, in the large corpus of lightweight integer compression algorithms, there is no single best compression algorithm and the choice depends on data characteristics [DUH+19].

1.2 QUERY ACCELERATION

Modern database systems are constantly exposed to a broad variety of novel requirements to improve query performance, ranging from analytical, transactional, or hybrid workload based applications to operators (relational model, graph processing, etc.) and various data characteristics. To satisfy these diverse requirements, there are several modern hardware opportunities available, that use the properties and possibilities of modern hardware appropriately.

There are several hardware possibilities, the common one is a general purpose modern CPU. To speed up CPU performance, CPU manufacturers usually increase the clock speed of single-core processors. After increasing the clock speed of single-core processors to a certain level it became increasingly difficult to increase further due to technological limitations. Hence, CPU manufacturers move towards two directions to speed up CPU performance, i) developing a system with either 2 to 8 processors oriented multicore or hundred to thousand processors oriented manycore architectures, ii) developing

the idea of instruction-level parallelism, which is known as *vectorization*. The length of general processor words in modern CPUs is usually fixed to 64-bit, which limits the performance of query operation in particular for column scan. However, modern column scan methods exploit the intra-instruction parallelism to increase performance [WPB+09, LUH+18b]. To explore such a modern scan method, the *vectorization* concept is used. The Single Instruction Multiple Data (SIMD) instruction set extensions such as Intel's SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) have been available in modern Intel processors for *vectorization* [WWT+14]. SIMD instructions apply one operation to multiple data elements of so-called vector registers at once, which reduces the instruction calls. The size of the vector registers ranges from 128-bit (Intel SSE 4.2) to 512-bit (Intel AVX-512). These registers are used instead of regular processor words to improve performance [LUH+18b]. Therefore, SIMD vectorized implementations are beneficial for analytical queries execution [PRR15, ZR02]. By utilizing SIMD fundamental vector operations and good vectorization principles, several database primitives such as selection scans, hash tables, sorting, join, compression, have been implemented [PRR15]. SIMD based database system is moderate in manufacturing cost as well as performance.

Another modern hardware opportunity to accelerate query performance is GPU, whereby GPU acts as a co-processor to accelerate CPU based query processing. Typically, there is a large number of simple cores residing in a GPU for massive parallelism through thousands of threads of computation at a time. GPU cores are usually organized in blocks of 32 cores that execute the exact instruction at a given time parallelly, similar to the SIMD architecture of CPU. However, the main advantage of using GPU over CPU for query acceleration is bandwidth, e.g., a modern GPU NVIDIA TITAN Xp has 547.7GB/s memory bandwidth. In contrast, a modern CPU Intel Core i7-7700K has about 50GB/s. GPU-accelerated database system is perfect for the column-store due to its connected memory access pattern on the GPU. GPUs are not only suitable for column-store but also other database operations such as relational operations, compression, XML filtering, scan, etc. Although in the modern system, the CPU is transferring data to the GPU through a powerful PCI express bus. Transferring large chunks of memory from CPU to GPU is still a bigger challenge. However, query acceleration either through GPU or by using *vectorization* of CPU is always implemented on the software-level. To guarantee a certain expected level of performance on query processing also requires real-time hardware configuration. Therefore, modern database systems consider efficient ways for query processing not only through software-level implementation but also by hardware-based configuration. This leads database researchers towards the other modern hardware possibilities.

Nowadays, database researchers move towards the development of application-specific integrated circuit (ASIC) based column-store database systems [BKG+18]. The ever increasing demand for more and more performance-based database systems, also becoming power-hungry. Hence, to balance performance and power efficiency, the query processing engine can be implemented by developing the software design together with the hardware architecture. However, it takes a year or more to design ASIC based database systems, as such implementation requires several steps like fabrication, testing, verifications [AIR+17, AHF+14]. Moreover, ASIC based implementation is post-order configuration bound. This defines, after the fabrication process system acts as a permanent circuitry, which can not go through reconfiguration phases to satisfy additional requirements or modifications. As a consequence, such system development is time consuming as well as not cost-effective. All these constraints of ASIC based systems show the database researcher the way to the Field Programmable Gate Array (FPGA) based system.

FPGAs are accumulated with reprogrammable integrated circuits, whereby integrated circuits contain an array of programmable logic blocks. The high parallelism characteristic ensures hardware together with software system development on FPGA. In

other words, an FPGA is an innovative platform, whereby application-specific hardware can be implemented on the fly. Thus, FPGAs not only reduce the energy footprint but also provide software specific hardware-level implementation flexibility. All these features indicate that FPGA is fruitful for the query performance acceleration. However, working with FPGA requires comprehensive skills from high-level programming language skills to low-level circuit implementation knowledge. Previously, the business world has lost interest in FPGA based database acceleration, especially for main memory based database systems as FPGAs have integrated with insufficient bandwidth based memory interfaces which are limiting their performance [FMH+19]. Recently, the new generation FPGAs, e.g. Xilinx® UltraScale+™, have overcome such limitations. Nevertheless, the key feature of FPGA hardware is reconfiguration [Teu17]. Platform with reconfigurable feature is always available to satisfy system-level requirements. Therefore, FPGA is perfect for main memory based database systems regarding many aspects: performance, manufacturing time as well as cost in terms of resource consumption.

1.3 THESIS CONTRIBUTIONS

A Column-store data organizational pattern in main memory based database systems is necessary to accelerate the analytical query processing performance. Among others, the column scan is one of the very crucial operators in the main memory based database system. It has become obvious that the amount of data is ever-increasing in the modern world. As a consequence, main memory based database systems are required to deal with two major obstacles on top of the growing amount of datasets: i) to fit large datasets within few gigabytes of main memory, ii) to speed up the important database operations, e.g., the column scan operator. To deal with such obstacles, on the one side data needs to be compressed, and on the other side, optimization is necessary for the operators to work directly on the compressed form of data. Modern performance-efficient column scan mechanisms make it possible to perform queries directly on compressed data [LP13]. Additionally, lightweight compression algorithms are ideal for reducing the data footprint by reducing the gap between processor speed and main memory bandwidth [LB15]. Thus, constructing a system that producing compressed data through lightweight compressions for direct query evaluation require software-level specifications together with hardware-level attention. Recently, FPGA has become one of the most promising approaches for query acceleration. The flexibility of reconfiguration as per individual database operation implementation requirement makes FPGA an interesting platform. In this situation, considering FPGA for database operators implementation, such as scan or compression, for main memory based database system implies an optimistic choice. Hence, we investigate FPGA regarding two aspects:

- Column scan operation on compressed data.
- Adaptive lightweight integer compression.

Therefore, this thesis investigates the main memory based hardware-level FPGA-oriented implementations, specifically for column scan mechanisms and lightweight data compression algorithms.

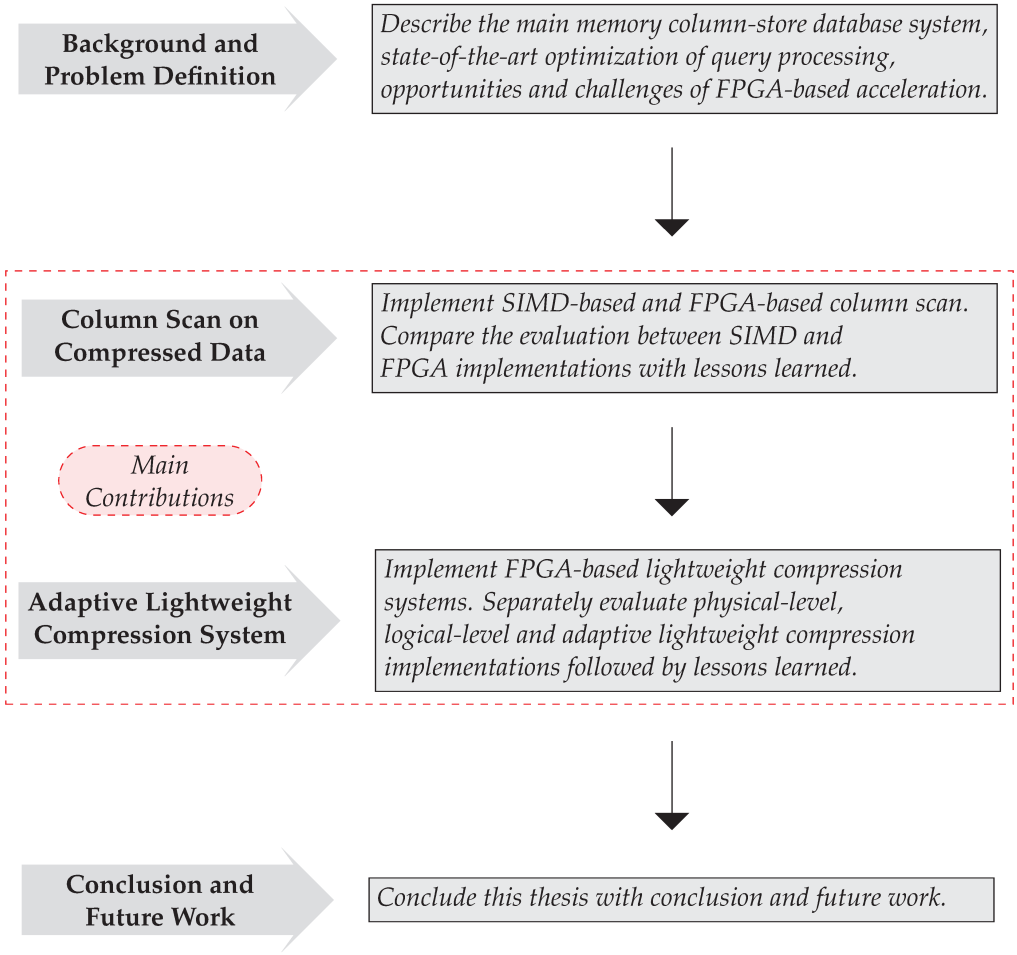

Figure 1.1: The Top-Down Structure of This Thesis.

Based on that, the top-down structure of this thesis is shown in Figure 1.1. Each highlighted arrow represents a chapter of this thesis. In *Chapter 2*, titled *Background and Problem Definition*, this thesis starts with a background study of the main memory column-store database systems as it is the software-side foundation of this thesis, and the hybrid CPU-FPGA architecture as it is the hardware-side foundation of this thesis. Then, we discuss recent research on FPGA-based acceleration to show that this novel class of accelerator is beneficial for query performance, and end with defining the research challenges that this thesis strives to resolve. Next, in *Chapter 3*, titled *Column Scan on Compressed Data*, this thesis gives an elaborate overview regarding state-of-the-art column scan techniques. Here, this thesis focuses on a naive scan technique [Lam75] as well as on a modern intra-instruction parallelism based technique called *Bitweaving* [LP13]. Then, we show the efficient ways of state-of-the-art column scan techniques implementation details using SIMD extensions and FPGA, comparison evaluations between SIMD and FPGA based implementation, and end this chapter with lessons learned. Afterward, in *Chapter 4*, titled *Adaptive LightWeight Compression System*, this thesis describes the possible implementation opportunities on FPGA for lightweight integer compression algorithms. Thus, this thesis implements the hardware-based physical-level, logical-level, and cascaded lightweight integer compression algorithm based systems on FPGA. Moreover, an efficient implementation detail for adaptive lightweight integer compression systems on FPGA is proposed. Then, an exhaustive experimental comparative evaluation of all the implemented compression systems is performed. This chapter ends with the lessons learned from the evaluation. We conclude this thesis with a summary and list of potential future work in *Chapter 5*.

BACKGROUND AND PROBLEM DEFINITION

This chapter presents an elaborative description of main memory column-store database systems along with selected related work. Thereafter, the state-of-the-art optimization opportunities for query processing acceleration are illustrated. Then, the modern hybrid CPU-FPGA architecture is discussed as the novel class of accelerator hardware including recent database domain-related works on FPGAs. Finally, the research challenges and proposed solutions of this thesis are defined.

2.1 MAIN MEMORY COLUMN-STORE DATABASE SYSTEMS

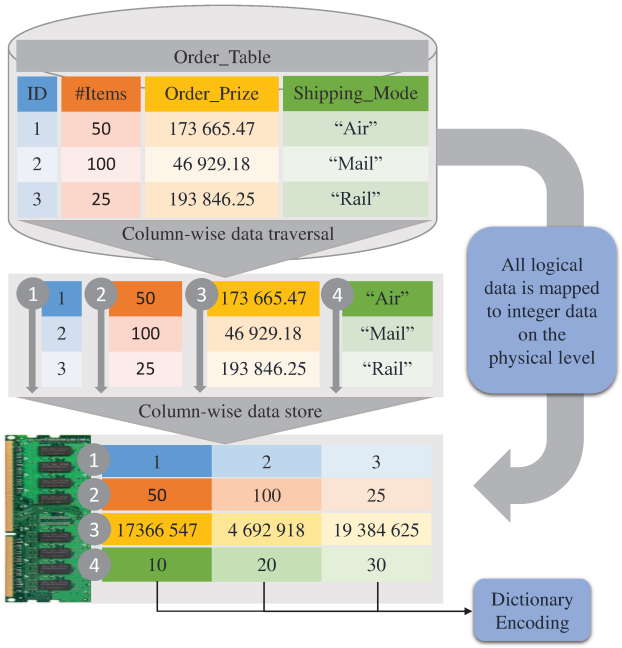

Figure 2.1: Illustration of Main Memory Column-Store Database System.

For decades, the main research goal of the database community has been performance. To meet this goal, the database community has explored possible alternatives including indexing, materialized views, vertical/horizontal partitioning, etc. Recently, the so-called column-store database systems have become state-of-the-art regarding query performance for certain workloads. Column-store database systems organize records or tuples or data tables in a columnar fashion. Unlike traditional systems, column-store databases are responsible to store and perform different types of queries. In various tools or software, column-store databases act as the backbone of the system to extract, transform, load, or for the visualization of data. However, storing data in columns benefits the database systems, because they are accessing the particular data they need to perform a query, but do not require scanning and discarding of unwanted data. The query performance of the database system is explicitly correlated with the efficiency of data movement between storage and CPU registers for processing. Usually, the storage database systems are disk-oriented. A typical disk has an average read/write latency of 5 milliseconds while the main memory has 50 nanoseconds, which is nearly 100,000 times as fast [OBL+17]. Undoubtedly, the disk-oriented data access is the major bottleneck of the database systems. However, the characteristic of column-store databases increases performance as less amount of read operation is required on disk. Additionally, shifting column-store database systems from disk to main memory benefit even more regarding performance. Figure 2.1 illustrates the main memory column-store database system in an exemplary way. In this figure, the table called `Order_Table`, which has four

columns, namely `ID`, `#Items`, `Order_Price` and `Shipping_Mode`. The `ID` and `#Items` columns consist of integer values, the `Order_Price` and `Shipping_Mode` columns consist of float and string values, respectively. In column-store organization, all data traversal happens column-wise one after another (see Figure 2.1). All logical data represented in `Order_Table` are mapped to integer data on the physical level in terms of main memory. Thus, the string values of the `Shipping_Mode` column are encoded to integer values using the dictionary encoding technique. Similarly, the float values of the `Order_Price` column are stored as integer and keep the precisions as metadata. As data are stored column-wise at the physical level, it is easy to access a particular column at once for query processing which accelerates the performance. Subsequently, the era of main memory column-store database systems arises which creates several research dimensions for optimization.

For instance, Stonebraker et al. [SAB+05] invented the extended version of column-store database system as C-Store. Usually, an entry sequence order is maintained to insert data in columns. This is efficient for new data insertion, either batch-wise or transactionally, at the end of the column. However, this situation becomes very difficult and expensive for column update operations without an entry sequence of orders, which is a common problem. C-Store overcomes such issues through a read-optimized column-store and an update/insert-oriented writeable store, connected by a tuple mover. C-Store is implemented on top of several components. It has, i) a small *Writeable Store (WS)* component to support high-performance inserts and updates, ii) the *Read-optimized Store (RS)* is capable of supporting very large amounts of information, iii) additionally, *RS* is optimized for reading and supports a restricted form of insert, for instance, the batch movement of records from *WS* to *RS* is performed by the tuple mover [SAB+05]. Stonebraker *et al.* evaluated C-Store on the TPC-H benchmark and the result showed that C-Store is on average 21 times faster than the native column-store system [SAB+05]. The design of C-Store emphasized radical departure from the traditional database systems. C-Store targeted for the read-intensive database market. C-Store is a column-store representation connected to a query execution engine [SAB+05]. C-Store focused on economizing the storage representation on disk by coding data values and dense-packing the data [SAB+05]. C-Store supports distributed transactions without a redo log or two-phase commit and efficient snapshot isolation [SAB+05].

In contrast, MonetDB is the first database system that accumulates the main memories for column-store database systems effectively and efficiently for query processing [BZN05, IGN+12]. To increase performance, MonetDB mainly focuses on analytical workloads that are read-intensive, whereby updates are appending new data in terms of large chunks to the database. MonetDB reconsiders all aspects of the traditional database systems in terms of design, architecture, and implementation. However, technology-wise MonetDB effectively exploits the potentials of modern hardware. For example, Boncz et al. [BZN05] investigated the inefficiencies of modern CPUs. On one side, they triggered out huge amount of query execution introduces interpretation overhead which prevents the compiler from using the most performance-critical optimization techniques, such as loop pipelining. On the other side, they analyzed that the main memory database system MonetDB applies a column-at-a-time materialization policy, which makes it memory bandwidth bound. Therefore, Boncz et al. [BZN05] proposed pipelined operators that pass to each other small, cache-resident, vertical data fragments called vectors and they called this new query engine MonetDB/X100. This work [BZN05] evaluated their system on the TPC-H benchmark for 100 GB data, showing that MonetDB/X100 can be up to two orders of magnitude faster than the existing system. Internally, MonetDB does not follow the exact column-oriented data storage logic. MonetDB came up with a whole new design tailored for columnar execution along with cache-conscious data structures and algorithms to provide optimal use of hierarchical memory systems [BZN05, IGN+12].

Thus, we can state that the main memory column-store database systems rely on two key factors,

(a) Columnar Storage Organization.
(b) Columnar Processing.

(a) Columnar Storage Organization: In columnar storage organization, each column is encoded as a sequence of integer values. Hence, it is very important to reduce the column size to fit in the main memory. To obtain this, the columns are usually compressed with a common approach: (i) Encode the values of each column as a sequence of integers using some kind of dictionary encoding and (ii) Apply lightweight lossless integer compression to each sequence of integers resulting in a sequence of compressed column codes. Generally, there are two main types of compression: (i) Heavyweight compression, (ii) Lightweight compression. Usually, lightweight compression algorithms work for main memory oriented datasets and heavyweight compression algorithms are for disk-based datasets. However, lightweight compression has two major benefits compared to heavyweight compression. Firstly, the data transmission rate of lightweight compression is higher than the heavyweight compression, as the main memory stays closer to the CPU than the disk. Secondly, the computational effort of the lightweight compression algorithm, especially for the (de)compression, is much lower than for heavyweight compression algorithms. To achieve these benefits, there is a large corpus of compression schemes that have been developed in the domain of lightweight lossless integer compression. Every individual lightweight compression scheme depends on a unique data property, such as value distributions, run lengths, sorting, or the number of distinct data elements. Hence, in this large corpus of lightweight compression schemes, there is no single-best scheme suitable for all datasets [DUH+19].

(b) Columnar Processing: It is very crucial to determine when the projection of columns should occur to accelerate the performance during processing columnar queries. In column-store, attributes are usually stored in a distributed manner on the storage medium. Information of a single entity could be distributed over several columns. As a consequence, query processing over several attributes is required to access a single entity. Hence, during query processing, multiple columns have to be reassembled at some point to create tuples. This tuple reconstruction process is called *Materialization*. In the query plan design context, there are two types of *Materialization*, (i) *Early Materialization*, (ii) *Late Materialization*. In *Early Materialization*, tuple reconstruction happens early in the query plan. This means, during query processing, it selects all relevance columns and reconstructs tuples from their component attributes first. Afterward, it executes standard row-store operations on the resulting tuples. Thus, implementation-wise *Early Materialization* is easy but generally performance-wise is poor. In contrast, *Late Materialization* fetches columns on demand for each operator in the query plan. In *Late Materialization*, tuple reconstruction happens at the end of the query plan. This makes it necessary to keep efficiently the intermediate join operations results that have been conducted on individual columns. These intermediate results are represented as a set of ranges of positions. These positions can be intersected to extract the values of interest which finally join into the final projection. Thus, implementation-wise *Late Materialization* is complex but generally performance-wise is better. Additionally, in the context of analytical columnar query processing, all query operators are working on integer data and usually, the tuples are only reconstructed at the end for the query result. Thus, the analytical columnar query processing applies the *Late Materialization* concept to achieve good performance. For instance, MonetDB defined all operators using the Monet Interpreter Language (MIL) which is an algebraic query language including a good join order through emphasizing the *Late Materialization* concept [BK99]. Column-store is suitable for the compression scheme. However, this advantage is not available in

row-store since a tuple contains data from multiple different attributes. Column-store does not guarantee to give better compression ratios but the column-store has the potential to be highly compressible. Different factors are taken into account, such as column cardinality, data types, and sorting, which help to decide about the compression scheme. In this context, *Late Materialization* becomes more effective since it processes the column values before forming tuples that helps to work directly on the compressed columnar data. Therefore, the major advantage of lightweight integer compression in the main memory based column-store databases is that some query operators can process the compressed data directly, without decompression.

The goal of the above discussion was to show the importance of the main memory column-store database systems. Main memory column-store database systems keep and process data column-wise, which is perfect for analytical queries, because each query is read-intensive on a limited set of columns. Another key aspect of the main memory column-store database system is data compression. Nevertheless, data compression is a well-known optimization technique for database systems. In particular, for the main memory column-store database systems, lightweight compression algorithms are used to optimize query processing by evaluating the query directly in the compressed form of integer data. However, the most important aspect of the main memory column-store database system is query performance. Hence, the software-side foundation of this thesis is based on the main memory column-store database systems.

2.2 STATE-OF-THE-ART OPTIMIZATION OF QUERY PROCESSING

Main memory based analytical databases are continuously exploring a variety of optimization opportunities with respect to the underlying hardware. Among several modern hardware systems, SIMD-vectorization and GPU-based accelerator became the state-of-the-art optimization opportunities for query processing. Therefore, this section concentrates on some recent related works regarding SIMD-vectorization and GPU-based database systems including the limitations of such optimization opportunities.

2.2.1 Optimization using SIMD-Vectorization

To accelerate performance, three different sources of parallelism are possible on the hardware side: i) Thread parallelism, ii) Instruction level parallelism, iii) Data parallelism. Among these parallelisms, data parallelism can be achieved through SIMD instructions. Single instruction multiple data (SIMD) performs the same operation on multiple data simultaneously. SIMD operations utilize a set of vector registers, whereby one operation is applied to all elements of the vector at the same time. In other words, a single instruction can be applied to multiple data elements in parallel. Several processors such as Intel's Xeon Phi, Intel Sandy Bridge, Haswell/Broadwell, AMD Bulldozer, ARM, and PowerPC implement SIMD instructions, although the instruction details may vary. Since hardware-specific SIMD instructions involved operations depend on vector operands. SIMD operations are also referred to as vectorizations. Therefore, many database systems continuously utilize SIMD-vectorization as an optimization opportunity to accelerate performance. For instance, recent related works [PRR15], [PR19], and [BRT+18] utilized SIMD-vectorization to implement the fundamental database operators. In the following, we will detail on related works [PRR15], [PR19], and [BRT+18].

Polychroniou et al. [PRR15] presented main memory based analytical database execution using generic SIMD vectorized implementations. Selection scans, hash tables, sorting, and join operators were implemented through the fundamental SIMD-vector operations. For instance, SIMD-based fundamental vector operations were implemented that are required to implement vectorized database operators. Among several, two operations are termed as selective loads and selective stores. Selective stores write a specific subset of the extension/vector lanes to a memory location contiguously based on a mask. Selective loads are the symmetric operation that involves loading from a memory location contiguously to a subset of extension/vector lanes based on a mask. Afterward, selective store operation is utilized in the implementation of a selection scan database operator, whereas the selection scan is a very useful operator for main memory query execution. A selection scan is a conditional scan operation that uses selective store operation (which Polychroniou et al. [PRR15] implemented through SIMD provided instructions) to store the extensions or vector lanes that satisfy selection predicates. Polychroniou et al. [PRR15] evaluated the performance of selection scans on Xeon Phi and Haswell systems for the selection condition $k_{min} \leq k \leq k_{max}$, where k_{min} and k_{max} are query constants. Polychroniou et al. [PRR15] varied the selectivity and measured the throughput for six different implementation versions of the selection scan to show the efficiency of vectorized implementation over non-vectorized implementation. Polychroniou et al. [PRR15] implemented two scalar variants with and without branching of selection scan. The rest variants implemented using SIMD vectorization. On both systems: Xeon Phi and Haswell, all vector codes outperform scalar codes. Besides selection scan, other vectorized implementations, e.g., hashing, partitioning, sorting, and join, were evaluated against scalar and vector code on Haswell and Xeon Phi systems. This work [PRR15] utilized SIMD-vectorization in the context of main memory database operators, and focused on the impact of vectorization on algorithmic designs, as well as the architectural designs.

Some years later, the same database research group introduced an analytical query engine implemented entirely by using SIMD instructions namely VIP [PR19]. The VIP engine supports all fundamental database operators, such as selections, hash joins, group-by aggregations, multiple data types, compression, and complex predicates or expressions. The latest AVX-512 SIMD instructions are utilized on VIP, which supports 512-bit width based vector registers and additional functionality, e.g. conflict detection or prefetching. Polychroniou et al. [PR19] implemented one of the key lightweight physical-level compression called *Bitpacking* using only 5 AVX-512 SIMD instructions. During the evaluation of the TPC-H workload, VIP outperforms query-specific hand-optimized scalar code.

Hashing is one of the important database operator that is very useful for hash-based joins and aggregations. Hence, Behrens et al. [BRT+18] focused on the efficient ways of hashing implementation. However, Behrens et al. [BRT+18] noticed that vectorized database operators improve performance but require processor-specific APIs. Therefore, Behrens et al. [BRT+18] vectorized the essential primitives, such as gather, scatter, selective load, and selective store in OpenCL to reduce code complexity and to ensure portability. This work [BRT+18] used Intel Core i7-6700K CPU and Xeon Phi processors for evaluations. During evaluation, SIMD intrinsics based vectorized hashing outperforms OpenCL-based hashing. However, on Intel Core i7-6700K CPU, OpenCL-based vectorized hashing outperforms scalar hashing for moderately sized hash tables that fit into the L2 cache. In this case, Behrens et al. [BRT+18] OpenCL-based hashing scheme is competitive to intrinsics-based hashing.

Although the above-mentioned works explain that the SIMD-optimization became one of the popular opportunities for database researchers, it is still not used in all cases. For instance, a heavy flow control-oriented operator like parsing is not doable from SIMD. However, SIMD-vectorize comparisons based parsing is possible using a large

cache that can store more intermediate states. But it requires exclusive SIMD intrinsics based cache control features which are not available in SIMD. Additionally, the extended vector register increases power consumption and chip area of the system which increases the system cost. Moreover, gathering data into SIMD vector registers and storing it to the correct destination locations requires extra efforts and sometimes it is inefficient, which implies data alignment is a big obstacle in SIMD. Finally, SIMD instructions are hardware architecture-specific. Different processors have different sets of SIMD instructions which implies that database researchers must provide multiple vectorized database operator implementations to operate optimally on any given processor. Hence, SIMD is a hardware-tight opportunity and it is not suitable for all types of operator implementation. As a consequence, it provides limited system performance. All these shortcomings show that SIMD-vectorization is not a flexible optimization opportunity in the database domain.

2.2.2 Optimization using GPU-Accelerator

The massive parallel capabilities with thousands of cores for analyzing volumes of data to solve difficult computational problems with great speed makes the GPU-accelerator an interesting platform for data-intensive applications. As a consequence, database researchers adopted the key feature *perform parallel operations on multiple sets of data* of GPU (Graphics Processing Units) to accelerate performance. Initially, GPU was used in databases to enhance operations by adding compute workloads to the graphics shields. When the NVIDIA introduced GPU along with CUDA [BS10], the application-specific general-purpose logic implementation became more convenient on GPU. Architecture and processing paradigm wise GPU is completely different from CPU. GPU is advantageous as it is keeping up with today's big data demands. Hence, GPU based databases can provide speedups over the CPU-based system. Some related works are discussed below.

The recent trend of database systems are changing hardware platform from homogeneous CPU systems towards heterogeneous CPU-GPU based systems including different computing units (CUs). The computing architecture in a heterogeneous system is different from a homogeneous system. Therefore, database management systems (DBMS) are continuously adapting to this hardware trend to efficiently utilize the given opportunities. Traditional database systems require query execution plans (QEP) to compute query results. To determine the most efficient QEP, the query optimizer is used for logical and physical optimizations. In a heterogeneous CPU-GPU based database system, before query execution, performing a placement optimization is essential, because it assigns physical operators of the most efficient QEP to the ideal computing units. Hence, Karnagel et al. [KHL17] proposed a novel adaptive placement approach for query processing on a heterogeneous CPU-GPU system. This work [KHL17] approaches a physical query execution plan as input and divides the plan into disjoint execution islands at compile-time. The execution islands are determined in a way that the cardinalities of intermediate results within each island are known or can be precisely calculated. The placement optimization and execution are performed separately per island at query runtime. The processing of the execution islands takes place successively following data dependencies. Karnagel et al. [KHL17] also proposed two additional improvements: (i) a fine-grained runtime estimation technique, (ii) a placement-friendly data transfer technique. This work [KHL17] implemented the techniques as a virtualization layer called HERO (HEterogeneous Resource Optimizer), which can be used by any OpenCL-based database system. This work's [KHL17] evaluation is based on OpenCL driver implementation with the OpenCL database systems gpuDB2 and Ocelot3. Karnagel et al. [KHL17] used a highly heterogeneous

hardware setup consisting of a CPU and three different GPUs (AMD Radeon R7, Nvidia Tesla K20, Nvidia GT 640), because of their general availability and their support for OpenCL. In the evaluation, this work [KHL17] showed that their approach improves the performance up to 50x over gpuDB and Ocelot system by choosing good heterogeneous placements.

In the last few years, many approaches utilized GPUs for the database system acceleration. Breß et al. [BHS+14] theoretically explored the design space of GPU-aware database systems. Breß et al. [BHS+14] argued that a GPU-aware DBMS should be a main memory, column-oriented DBMS using the block-at-a-time processing model, possibly extended by a just-in-time-compilation component. The system should have a query optimizer that is aware of co-processors and data-locality, and can distribute a workload across all available (co-)processors. Breß et al. [BHS+14] validated these findings by surveying the implementation details of eight existing GPU-aware DBMSs and classifying them according to the mentioned dimensions. Additionally, this work [BHS+14] summarized common optimizations implemented in GPU-based DBMSs and inferred a reference architecture for GPU-aware DBMSs, which acts as a starting point in integrating GPU-acceleration in popular main memory DBMSs.

Despite having massive parallel capabilities, the GPU can only be programmed using its available static vectored instruction set. Typically, a GPU is connected as a co-processor of CPU, so the GPU-based programming model is quite restricted under CPU. However, modern NVIDIA GPU overcomes a certain level of restrictions. However, the transfer of data to the GPU increases the latency as it needs to go through the PCIe channel. Therefore, the PCIe bus is still a bottleneck of GPU-based query processing. Moreover, on GPU the low-level instructions are not implementable, which makes GPU a hardware-tight platform. Additionally, GPU is not suited for all database algorithms. Mostly, the database operations are reformulated to take advantage of the parallelism of GPU, which increases the operation overheads. Finally, GPU is an instructional computation device that needs to run various computing units to perform a particular operation, which makes GPU a power-hungry device. All these limitations corroborate that GPU is not perfectly suited for all types of database operators.

2.2.3 Summary

Recent related works show that SIMD optimization is heavily applied for different query operators as well as lightweight integer compression [PRR15, PR19, BRT+18]. Additionally, recent works also show that columnar processing is suited for GPU co-processing [KHL17, BHS+14]. In the database research domain, SIMD optimization and GPU optimization are heavily used, but their benefits are limited to fixed hardware. Fixed hardware defines that SIMD and GPU based on some fixed set of instructions. Database researchers are bound to work within this fixed set of instructions. That means researchers are neither able to modify any instruction nor implement any new instruction to accelerate performance. This limits the implementation-level flexibility to accelerate query performance. To overcome such limitations, a new class of accelerator hardware arises the so-called FPGA. FPGA is not bound to fixed hardware. It is a flexible platform where different types of operators can be implemented on-the-fly as per system requirements. Hence, this thesis targets this new class of hardware to accelerate query performance.

2.3 OPPORTUNITIES AND CHALLENGES OF FPGA-BASED ACCELERATION

In this section, the benefits and opportunities of the new class of accelerator hardware, namely the hybrid CPU-FPGA architecture is defined. The hybrid CPU-FPGA architecture is the hardware-end backbone of this thesis. Afterward, some exclusive database related works with regards to FPGAs are discussed as an application of a new class of accelerator hardware. In the end, we conclude this section by defining the research challenges of this thesis.

2.3.1 Hybrid CPU-FPGA Architecture

In the early 1980s, the digital circuit community envision "*reconfigurable after being manufactured*" type technology to gear up the scalability and flexibility regarding the development process of integrated devices. As a continuation, the programmable logic array (PLA) device was introduced for fixed architecture logic devices with programmable AND gates followed by programmable OR gates. Manufacturing-wise PLA was efficient while it collapsed during scaling as the programmable instances in the array grew with the square of the number of inputs. Therefore, PLA became impractical for integrated circuitry regarding power efficiency and performance. Later, other devices like CPLD, EEPROM came onto the digital market, but none of them was able to completely satisfy the "*reconfigurable after being manufactured*" vision. Later as a successor of PLA in 1984, Xilinx® first introduced the concept of Field Programmable Gate Array (FPGA), which fulfills the vision of reconfiguration [FMH+19]. In Field Programmable Gate Array (FPGA), a group of reconfigurable logic array memory cells is distributed along with interconnecting switching nodes and wiring. The reconfigurable logic array memory cells of FPGA increase the scalability in terms of power efficiency and performance compared to programmable AND-OR-based PLA structure.

Recently, FPGA vendors like Intel® or Xilinx® incorporate an FPGA and a CPU into a common system leading to System-on-Chip (SoC) architecture. Thus, the modern generation hardware era of hybrid CPU-FPGA hardware systems arises. This thesis is evaluated on a hybrid CPU-FPGA hardware system from Xilinx® called Zynq UltraScale+™ [Xil19b] as hardware foundation. The architecture of this hybrid system is depicted in Figure 2.2. As shown in this figure, the target system is divided into two major top-level blocks, *i) Processing System (PS), ii) Programmable Logic (PL)*, while *the AXI Communication Protocol* is used as a bridge to connect these two blocks.

Processing System (PS) is based on so-called multiple processor system-on-chip (MPSoC) architecture. Therefore, PS integrates 64-bit quad ARM® Cortex-A53 based Application Processing Unit (APU), dual-core ARM® Cortex-R5 based Real-Time Processing Unit (RPU), ARM® Mali-400 based Graphics Processing Unit (GPU) in a single chip. The maximum frequency of APU is 1.5 GHz. Although the PS part features RPU and GPU cores, both are not considered further in this thesis. The PS part also includes on-chip memory and a rich set of peripheral connectivity interfaces. Moreover, the Zynq UltraScale+™ has two DDR4 main memories. While the size of one DDR4 is 4 GB, the size of other one is 512 MB. Additionally, the platform management unit controls the power of the overall system and the system control unit is occupied by various categories of clock buffers and some pre-fixed controllers, such as direct memory access (DMA).

Programmable Logic (PL) part is responsible for designing all programmable custom-made hardware. Internally, *PL* is composed of configurable logic blocks (CLBs), a

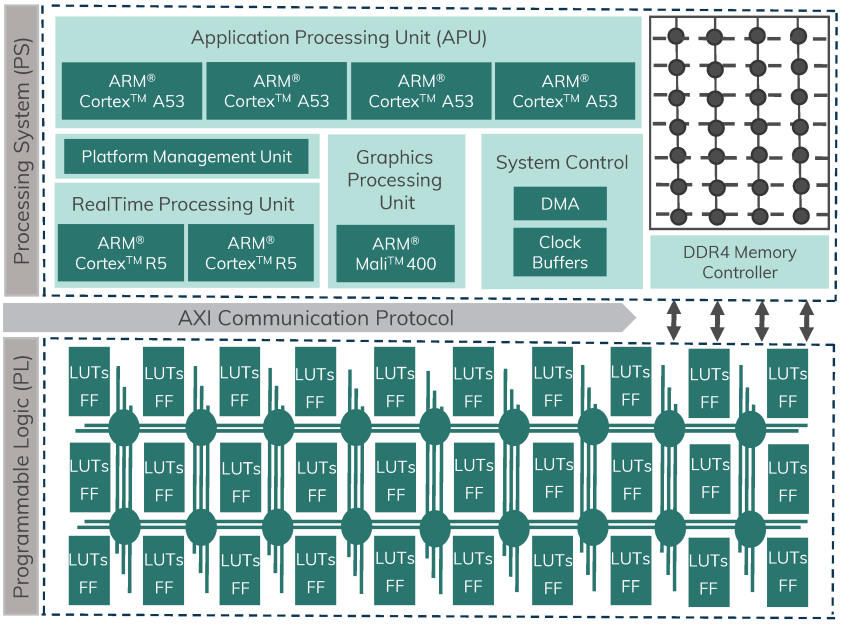

Figure 2.2: Hybrid CPU-FPGA Architecture of Zynq UltraScale+™[Xil19b].

collection of block and distributed memories (BRAM, LUTRAM), and arithmetic units (DSPs) [MTA09, TW13]. CLBs plays a significant role in the reconfiguration context, whereby it occupied with look-up tables (LUTs). LUT is reflected as static RAM (SRAM), which holds a custom-made truth table while the power is turned on. Basically, a LUT circuit is occupied with SRAM, multiplexer (MUX), and flip-flops. The SRAM cells hold the outputs of a custom-made truth table which is connected with a MUX and the inputs of MUX act as the address lines for a corresponding one-bit-wide SRAM cell. When a specific boolean function is configured, the SRAM cells are loaded with their corresponding truth table. Therefore, instead of wiring lots of AND-OR-XOR-based logic gates, LUTs just simulates this with SRAMs. A 2-input LUT circuit requires (4X1)-bit SRAM as shown in Figure 2.3, whereby it can configure any 2-input boolean expression *F(A, B)*. Likewise 2-input LUT, 3-, 4-, 5-, as well as up to 6-input LUTs are available in modern generation FPGAs. In CLBs, LUTs are cascaded to implement *n*-ary boolean functions. Therefore, any type of application-specific custom-made hardware is realized by cascading a bunch of CLBs in terms of various sizes of LUTs.

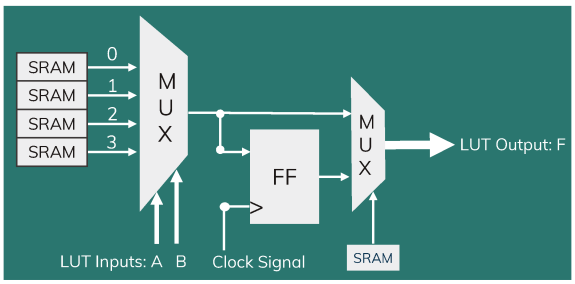

Figure 2.3: Internal Circuit of 2-Input LUT.

In contrast to previous FPGAs offered by Xilinx®, the *AXI Communication Protocol* between the *PS* and the *PL* on the Zynq UltraScale+™ is more powerful. Concretely, it has several high performance AXI interfaces between *PL* and *PS* providing a data bus width of 32-bit/64-bit/128-bit. More precisely, the configurable hardware on the *PL* part has direct access to the main memory on the *PS* part, so that *PS* and *PL* can work on the same data elements as shown in Figure 2.2.

To describe the operation mode of a specific application logic on FPGAs, a hardware description language, such as Verilog or VHDL is used. This description is then synthesized to RTL (register transfer logic) through several steps towards implementation along with bitstream for the FPGAs. Typically, FPGAs offer a higher performance while maintaining a lower power dissipation than the CPUs. To be competitive with common CPUs, the configurable hardware for a specific application logic has to be well-designed, since current FPGAs usually run at very low clock-rates around $(200 - 400)$ MHz. To enable that, the most challenging issue is to create efficient processing pipelines due to the proximity of logic and memories [MTA09, TW13].

The above-mentioned architecture defines that the hybrid CPU-FPGA system is extremely beneficial for the main memory column-store database systems, in particular for two reasons, i) the direct access to the main memory, ii) a large variety of custom-made design implementation possibilities. To utilize such advantages for query performance, the hardware-side foundation of this thesis is based on hybrid CPU-FPGA architecture.

Moreover, Xilinx® have very recently offered Alveo™ U50 data center accelerator cards to optimize acceleration. The key features of such a card are 8 GB HBM memory with data transfer rates up to 2 GT/s and 32 *AXI* channel access [Xil19a]. It clarifies that the modern trends of FPGA is significantly useful for the main memory column-store database systems regarding query performance.

2.3.2 Related Works on FPGA-based Acceleration

The application of FPGA in the database domain regarding acceleration is very common nowadays. The database community continuously utilizes the high parallelism feature of FPGA for the acceleration of distinct database operation. This thesis analyzes some related works to show that FPGA is used to implement database operators, in particular, XML parsing [TWN13], regular expression [SIOA17], near-data processing [ISA17], and binary packing compression [MMFB20]. In the following, we detail on related works.

In the database system, a parser is responsible to break down the data information into its corresponding specified grammar components leading to high parallel activities. Thus, Teubner et al. [TWN13] utilized the high parallelism feature of FPGAs for XML parsing processing through reducing the computation and main memory overhead. They implemented an FPGA-based XML processor as so-called *XLynx*, whereby design perspective-wise a skeleton automata is used for data-intensive hardware circuits that offer high expressiveness and quick reconfiguration at the same time. However, skeleton automata is a subset of finite state automata that provides the generic implementation. *XLynx* is a microbenchmark-based engine for projection. The throughput of *XLynx* in-network filtering is 180 MB/s, whereby in-network filtering with *XLynx* significantly eases the XML parsing burden on the backend XML processor. *XLynx* operates on streaming mode and processes one input character per clock cycle. Thus, the filtering throughput of this system is independent of the query workload. *XLynx* addressed the key limitation in modern system designs through savings up to 94% of electrical power consumption. Finally, in many real-world database systems the main bottleneck is parsing and *XLynx* is improving the overall query execution time in terms of performance by reducing the parsing cost.

In the database domain, a regular expression operator provides a powerful and flexible pattern match that helps to implement efficient search utilities for the systems. In other words, regular expressions enable the search patterns in string type data by utilizing standardized syntax. However, real-world data is occupied with complex information,

and a regular expression operator is able to extract information within a limited number of known formats which makes it compute-bound. Sidler et al. [SIOA17] looked into this compute-bound regular expression matching operator. They exploited the modern generation hybrid CPU-FPGA system for regular expression operator to overcome the compute-bound bottleneck. In particular, they used Intel® provided by a hybrid Xeon-FPGA system, whereby FPGA has coherent access to the main memory through the QPI bus. Operator-wise they focused on two commonly used SQL operators for strings, *LIKE* and *REGEXP_LIKE*. They implemented these regular expression operators on the hybrid Xeon-FPGA system. The overall system can be categorized into three parts, i) a CPU-based regular expression *Hardware User Defined Function (HUDF)* which is integrated with the popular column-store system MonetDB, ii) a *Hardware Operator Abstraction Layer (HAL)* for the interaction between Xeon and FPGA, iii) the FPGA-based hardware part of the *HAL* and four regular expression *Regex* engines. All *Regex* engines can operate independently for different queries and are parameterizable at runtime. The max data processing throughput per *Regex* engine is 6.4 GB/s, while the combined four *Regex* engines throughput is 25.6 GB/s. This meant a significant improvement regarding response time and throughput. Precisely, Sidler et al. [SIOA17] evaluated the following types of complex regular expression queries:

```
SELECT count (*) FROM address_table WHERE REGEXP_LIKE (address_string,
'[0-9]+(USD|EUR|GBP)')
```

The traditional CPU-based MonetDB performs about 5x to 15x times slower than the FPGA-based regular expression system [SIOA17]. In other words, the FPGA-based regular expression system is capable of speeding up complex pattern matching by an order of magnitude in comparison to the database running on a 10-core CPU. In conclusion, Sidler et al. [SIOA17] proposed regular expression *HUDF* not only shows a significant acceleration of query execution in comparison to traditional MonetDB, but also provides predictable performance independent of regular expression complexity.

To achieve high bandwidth, low latency, and high parallelism, the near-data processing system is a very attractive option for the database community nowadays. Therefore, István et al. [ISA17] explored near-data processing database system on FPGA. The proposed system is called *Caribou*, an intelligent distributed storage layer with specialized hardware. Internally, *Caribou* builds on top of distributed LUTs of FPGA. The overall *Caribou* architecture is organized with, i) a so-called *Cuckoo* hash table to handle read and write with constant time lookups, ii) a separate slab-based memory allocator module for efficient memory allocation and this module is also responsible for carrying out scans, iii) a filtering module for near-memory processing, which supports selection on both, structured and less structured data and, iv) a TCP/IP stack-based network interface to communicate both with clients and with other FPGAs. *Caribou* provides scan performance up to 5 GB/s for low selectivity and 1.25 GB/s for high selectivity. Therefore, *Caribou* is a highly efficient and intelligent data store that boosts the performance of database operations as well as reduces power consumption.

Database researchers need to reduce the memory footprint, especially for many column-based databases so that it can fit in multi-gigabyte's main memory for query processing. Thus, database researchers are considering FPGA to implement compression. For instance, Mahmoud et al. [MMFB20] investigated how to achieve a better compression ratio for integer compression using binary packing and prefix suppression offloaded to an FPGA. Mahmoud et al. [MMFB20] presented a general OpenCL-based parallelization approach with a multi-level distributor-collector architecture to scale

FPGA performance until it reaches a PCIe bus limitation. Mahmoud et al. [MMFB20] experiments showed that the OpenCL-based implementation in an FPGA outperforms CPU-based compression in SAP HANA by a factor of 2 in compression throughput along with a 60% compression rate improvement.

Recently, István et al. [Ist20] pointed out the need for more efficient large-scale data management and storage solutions, and proposed FPGA-based high-level solution idea. István et al. [Ist20] found FPGA is very useful because they offer network-bound performance even with small key-value pairs and near-data processing in a fraction of the energy budget of regular servers. On the other side, the consistency guarantees for concurrent client transactions are essential to be ensured. Thus, István et al. [Ist20] presented a high-level view of the typical pipelined architecture of FPGA-based Key Value Stores (KVSs) that most existing designs follow, and show three different ways of implementing transactions, i) through operation batching, ii) through two phase locking (2PL), and iii) through a simplified snapshot isolation model.

The above-mentioned related works prove that FPGA-based acceleration is a novel opportunity in the database domain for optimizing query performance. Previously, database researchers were not much interested in FPGA, especially for big data analysis cases. However, this scenario is changing now. The modern generation FPGA exhibits new features, e.g., high bandwidth-based main memory [Xil19a], powerful AXI communication protocol, etc., leading researchers to move towards the FPGA-based acceleration.

2.3.3 Research Challenges

This thesis investigates in Section 2.1 and Section 2.2 the literature overview regarding database system optimization opportunities either through defining innovative data maintenance mechanisms or utilizing modern hardware technology. Query performance acceleration is always the key issue for any research work in the database domain. However, the effectiveness of the database system does not only rely on high-performance aspects. Several other system dependencies also require major consideration. Among several requirements, one of the key demands is optimizing hardware resource utilization in the database system, which is proportional to the system cost and power consumption. This leads to the following crucial research challenge:

> *RC1: How to make a balanced tradeoff between query performance acceleration and optimal hardware resource utilization?*

FPGAs are a novel and flexible class of hardware technology which offer interesting features. Therefore, to tackle the research challenge *RC1*, this thesis chooses the high-parallelism and power-efficient modern hybrid CPU-FPGA hardware platform as depicted in Section 2.3.1. Such a hardware platform is advantageous to provide system-level design configuration flexibility. This feature of the target hybrid CPU-FPGA system leads to the next research challenge:

> *RC2: What are the certain level of design configuration aspects achievement feasible on top of hybrid CPU-FPGA system for database operators?*

To resolve the research challenge *RC2* this thesis concentrates on one of the crucial database operations which is *scan* as an exemplary way for the main memory column-store database system. Modern *scan* technology integrates with intra-instruction parallelism-aware technique, whereby any database operation execution is possible directly on compressed data [LP13]. Therefore, this thesis exploits modern *scan* technology on top of the hybrid CPU-FPGA system to define possible design configuration aspects. Additionally, such modern *scan* techniques require a compressed form of data to perform query operations, which demands fast compression algorithms. Recently, lightweight integer compression algorithms are applied in the main memory oriented common CPU based system to tackle the gap between processor speed and main memory bandwidth. FPGAs are already applied for database systems but not for the columnar database system in particular for processing compressed data and compression algorithms. Hence, the importance of compression in main memory column-store databases leads to the final research challenge of this thesis:

> *RC3: To what extent are the design configuration aspects on top of the hybrid CPU-FPGA system beneficial for lightweight data compression acceleration?*

This thesis addresses the research challenge *RC3* by proposing an adaptive lightweight integer compression system on top of a hybrid CPU-FPGA system with high compression throughput and optimum resource utilization as FPGA is an ideal hardware approach which allows to design specific hardware components which ease the hardware limitations.

COLUMN SCAN ON COMPRESSED DATA

In this chapter, we present the designs and the evaluations of FPGA-implementations for state-of-the-art column scan techniques. Therefore, we start with a review of two state-of-the-art column scan mechanisms: i) *Naïve*, ii) *BitWeaving* in Section 3.1. We focus on two well established hardware-oriented implementation opportunities for the mentioned column scan mechanisms. Firstly, we discuss the implementations using SIMD vector registers in Section 3.1.3. Secondly, the FPGA-based implementations are described in Section 3.2. Later, an exhaustive evaluation using selective results for both hardware-oriented implementations are compared in Section 3.3. At last, we summarise this chapter with lessons learned in Section 3.4. This chapter presents the experimental contributions of this thesis which is based on our publications [LUH+18b], [LUH+18c], [LUH+18a]. This chapter resolves the first two research challenges as mentioned in Section 2.3.3.

3.1 COLUMN SCAN

Column-oriented database management systems store data column-wise [BKM08]. Such a system is constructive for scan performance, (i) as each column is considered separately, and (ii) as columns always preserve the similar adjacent values. Both advantages offer the opportunity for compactness and the ability to process multiple column values at once which ensures high performance. Thus, the efficient realization of a column scan is an active research topic, whereby the following scan mechanism consists of two components: (i) storage layout for column values and (ii) scan operation (predicate evaluation) on the proposed storage layout.

3.1.1 Naïve

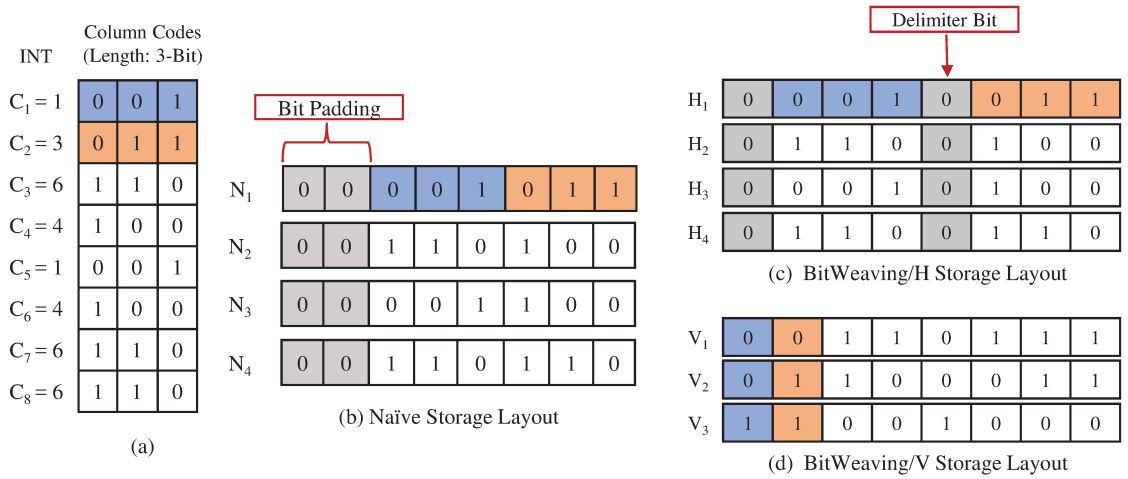

Figure 3.1: Storage layout example with (a) 8 integer values with their 3-bit codes, (b) data representation in *Naïve* layout, (c) data representation in *BitWeaving/H* layout and (d) data representation in *BitWeaving/V* layout (figure taken from [LUH+18b]).

As described in Section 2.1, each column is encoded with a fixed-length order-preserving code (see Figure 3.1(a)) as a base of column scan operations. Traditionally, the types of column values are either numeric or string, which are encoded as unsigned integer codes [BHF09, HHDL17]. The term *column code* refers to the encoded column value.

An intra-value parallelism-based compact storage layout is very useful to improve scan performance while processing multiple column codes in a single processor word. In 1975, Lamport et al. [Lam75] first introduced intra-value parallelism-based column scan mechanism. We call this method as *Naïve* column scan technique. The storage layout of *Naïve* column scan technique is shown in Figure 3.1(b), whereby column codes are continuously stored horizontally in processor words N_i. While storing the fixed-length based codes into the processor words it may have some extra leftover bits as unused bits in the words. These unused bits in the processor word are padded with zeros. In Figure 3.1(b), 8-bit size oriented 4 processor words from N_1 to N_4 are used, whereby two 3-bit column codes fit into one processor word including 2-bit padding per processor word.

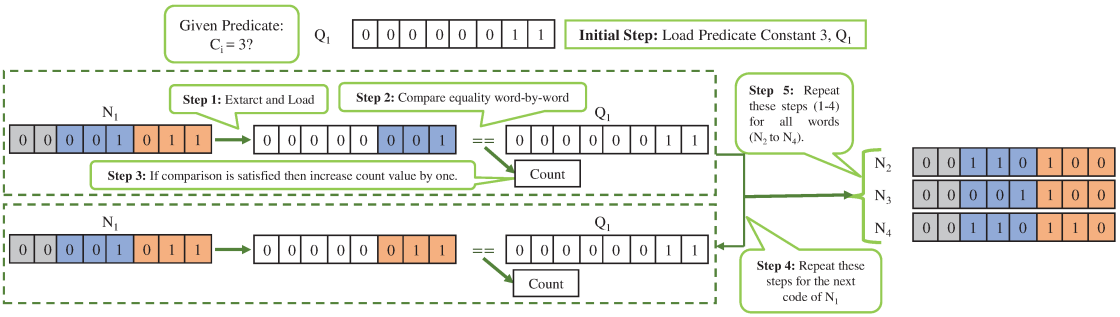

Figure 3.2: Equality predicate evaluation using *Naïve/S* technique with extract-load-compare each column code (figure taken from [LUH+18b]).

During *Predicate Evaluation*, the task of a column scan is to compare each column code with a constant C and to output the number of *Count* indicating how many times the corresponding code satisfies the comparison condition. The predicate evaluation on *Naïve* layout can be done in two ways. *Firstly*, we can evaluate any predicate by simply extracting, loading and evaluating each (single) code with the comparison condition consecutively, without exploiting code-level parallelism. We named this technique as *Naïve/S*. Figure 3.2 describes the *equality* check in an exemplary way. The input from Figure 3.1(b) is tested against the condition $C_i = 3$. The predicate evaluation steps are as follows:

Initially: Load the predicate constant *3* in word Q_1.
Step 1: Extract one code from N_1 and load in a temporary word.
Step 2: Check equality word-wise between Q_1 and temporary word.
Step 3: If comparison satisfies, then increment the value of *Count*.
Step 4: Repeat Steps (1 to 3) for the next column code of N_1.
Step 5: Repeat Steps (1 to 4) for the rest of words N_2 to N_4.

Secondly, we can evaluate any predicate directly on the *Naïve* layout by exploiting code-level parallelism. The main advantage of such technique is that predicate evaluation is done without decoupling the column codes from a word. We named this technique as *Naïve/M*. Figure 3.3 illustrated this technique in an exemplary way for the same input and test condition like *Naïve/S*. The detailed steps are described as follows,

Initially: Load the *Naïve* layout of predicate constant *3* in Q_1.
Step 1: Check the equality bit-wise of each code between N_1 and Q_1 in parallel. There are 1-bit S_i flag registers for each code of *Naïve* word. For this example, each word has two (S_1 and S_2) flag registers (see Figure 3.3). If the condition is satisfied, then set one to S_i flags, otherwise set zero.
Step 2: Perform addition between S_1 and S_2, and store the result in *Count* word.

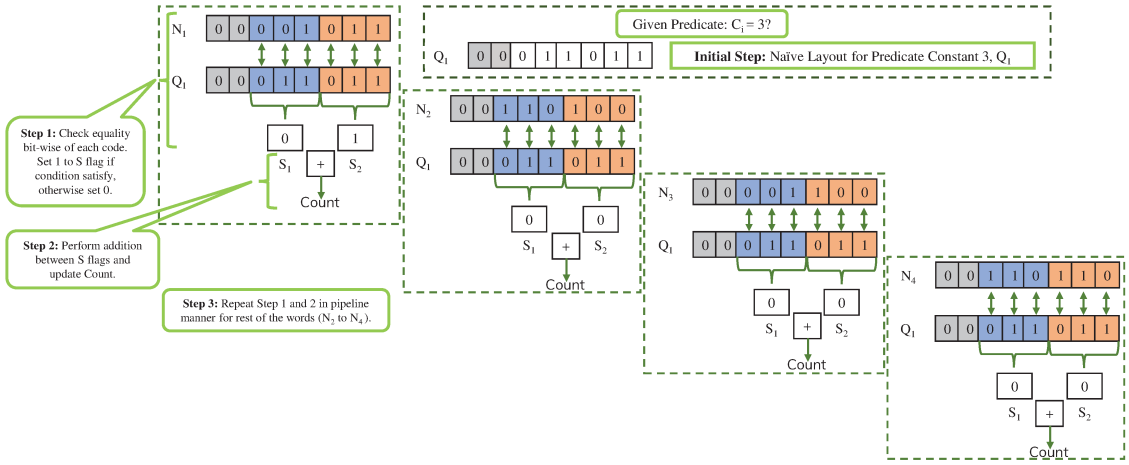

Figure 3.3: Equality predicate evaluation using *Naïve/M* technique with directly evaluate on compact words (figure taken from [LUH+18b]).

Step 3: Repeat Step 1 and Step 2 for the rest of words N_2 to N_4 in pipeline manner by overlapping instructions.

In both examples (Figure 3.2 and Figure 3.3), only the second code (C_2) satisfies the predicate, so the resulting *Count* value is one. In order to accelerate column scan, *Naïve/M* technique is a better choice than *Naïve/S* for two reasons, i) *Naïve/M* technique evaluates predicate directly on the compact word, ii) it is using instruction overlapping mechanism which reduces the number of clock cycles significantly. However, *Naïve/M* technique is difficult to implement on common CPUs using a 64-bit processor word, as common processor words do not support intra-data parallelism. More precisely, the common CPU provided instruction sets do not support multiple data processing at once. Generally, the common CPU is an instruction set architecture (ISA) based abstract model. On the contrary, *Naïve/M* technique is based on multiple data processing on a single cycle. Moreover, *Naïve* technique does not have its arithmetic framework to perform operations. Thus, it is very difficult to implement such a column scan technique on top of the 64-bit processor word of common CPUs.

3.1.2 BitWeaving

To overcome the difficulties of *Naïve* technique, *BitWeaving* [LP13] has been proposed. As illustrated in Figure 3.1(a), *BitWeaving* takes each column separately and encodes the column codes using a fixed-length order-preserving code (lightweight data compression [AMF06, DHHL17]), whereby the types of all values, including numeric and string types, are encoded as an unsigned integer code [LP13]. To accelerate column scans, *BitWeaving* technique introduced two types of storage layouts along with an arithmetic framework instead of comparisons for predicate evaluations: *BitWeaving/H* and *BitWeaving/V* [LP13].

BitWeaving/H

In the storage layout of *BitWeaving/H*, the column codes of each column are presented at the bit-level and the bits are aligned in memory in a way that enables the exploitation of

the intra-cycle (intra-instruction) parallelism for the predicate evaluation. As illustrated in Figure 3.1(c), column codes are continuously stored in processor words H_i, where the most significant bit of every code is used as a delimiter bit between adjacent column codes. In Figure 3.1(c), 8-bit size oriented 4 processor words form H_1 to H_4 are presented, whereby two 3-bit column codes fit into one processor word including one delimiter bit per code. The delimiter bit is used later to store the result of a predicate evaluation.

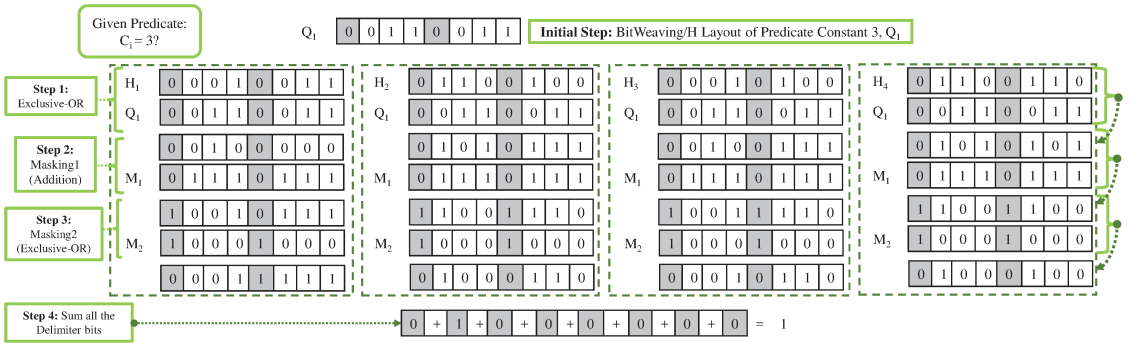

Figure 3.4: Equality predicate evaluation using *BitWeaving/H* technique [LP13].

To efficiently perform column scans using the *BitWeaving/H* storage layout, Li et al. [LP13] proposed an arithmetic framework to directly execute predicate evaluations on the compressed data. There are two main advantages: (i) predicate evaluation is done without decompression, (ii) multiple column codes are simultaneously processed within a single processor word using full-word instructions (intra-instruction parallelism) [LP13]. The supported predicate evaluations include equality, inequality, and range checks, whereby each predicate consisting of arithmetical and logical operations are defined [LP13]. Figure 3.4 highlights the *equality* check in an exemplary way. The input from Figure 3.1(c) is tested against the condition $C_i = 3$. Then, the predicate evaluation steps are as follows:

Initially: Load the *BitWeaving/H* layout of predicate constant *3* in Q_1.
Step 1: Exclusive-OR operations between the words (H_1, H_2, H_3, H_4) and Q_1 are performed.
Step 2: Masking1 operation (*Addition*) between the intermediate results of Step 1 and the M_1 mask register (where each bit of M_1 is set to one, except the delimiter bits) is performed.
Step 3: Masking2 operation (Exclusive-OR) between the intermediate results of Step 2 and the M_2 mask register (where only delimiter bits of M_2 is set to one and rest of all bits are set to zero) is performed.
Step 4: Add delimiter bits to achieve the total count (final result).

The output is a result bit vector, with one bit per input code that indicates if the code matches the predicate. In the example of Figure 3.4, only the second code (C_2) satisfies the predicate which is visible in the resulting bit vector.

BitWeaving/V

In *BitWeaving/V*, the codes are stored vertically across several processor words [LP13], such that one word contains one bit of several codes. Figure 3.1(d) shows the *BitWeaving/V* layout oriented presentation of the column codes of Figure 3.1(a). The words V_i are 8-bit long. The bits of the first number C_1 are stored in the first position

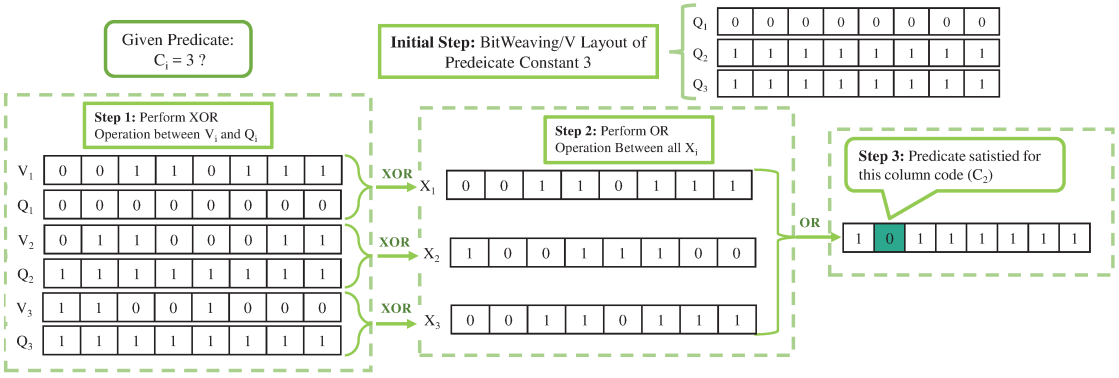

Figure 3.5: Equality predicate evaluation using *BitWeaving/V* technique [LP13].

of each word, the bits of the second number is stored in the second position, and so on. This way, eight 3-bit codes can be stored across three 8-bit words.

To evaluate predicates in this layout, we consider the restriction operation *Equality*. However, any kind of (restriction type) predicate evaluation can be performed in this layout. Figure 3.5 illustrated the *Equality* check predicate evaluation for *BitWeaving/V* in a exemplary way. In the example, we evaluate the column codes C_i for an equality with 3. The necessary steps are:

Initially: Predicate constant *3* is loaded as *BitWeaving/V* layout $(Q1, Q2, Q3)$.
Step 1: XOR operations are performed between *BitWeaving/V* layout based words and predicate constant as follows,

$$X_1 = V_1 \oplus Q_1$$
$$X_2 = V_2 \oplus Q_2$$
$$X_3 = V_3 \oplus Q_3$$

Step 2: Performed bitwise *OR* operations between (X_1, X_2, X_3).
Step 3: In the result word, there is only one position set to *0*. That means, the example condition is satisfied for only one column code and the total count value is one.

3.1.3 SIMD Implementation

We prepared the SIMD-implementations for *Bitweaving/H*, *Bitweaving/V*, and *Naïve/M* as common CPU do not support intra-data parallelism feature for typical 32-bit or 64-bit processor words. Thus, we cannot rely on a typical 32-bit or 64-bit processor word of common CPU to implement an intra-data parallelism oriented column scan technique. It is established that the modern SIMD (Single Instruction Multiple Data) extensions are characterized by extensive data parallelism which is able to gain potential performance. In particular, in SIMD extensions one instruction applies for multiple elements execution at once. Therefore, the SIMD extensions of a common CPU is one of the vital hardware-based opportunity to optimize column scan technique. However, we did not consider to implement the *Naïve/S* (without code-level parallelism based *Naïve* technique) as this equals a SIMD-Scan [WPB+09] when it is extended to SIMD. A comparison between a SIMD-Scan and the original *BitWeaving* variants has already been done by Li et al. [LP13]. A SIMD implementation requires a system with the corresponding vector registers and instructions. Initially, SIMD vector registers were 128-bit in size. In recent years, hardware vendors have introduced new SIMD instruction set extensions operating on wider vector registers. For instance, Intel's Advanced Vector

Extensions 2 (AVX2) operates on 256-bit vector registers and Intel's AVX-512 uses 512-bit for vector registers. The wider the vector registers, the more data elements can be stored and processed in a single vector. Additionally, each new vector extension comes with new instructions, e.g. gather-instructions were first introduced in AVX2. AVX-512 consists of several instruction sets, each providing different functionality, e.g. conflict detection or prefetching. For the evaluation of the SIMD-implementation, we used an Intel Xeon Gold 6130 with DDR4-2666 memory offering SIMD extensions with vector registers of sizes 128-, 256-, and 512-bit (SSE, AVX2, and AVX-512). This system offers the AVX-512 Vector Length Extensions (VL), which provide most AVX-512 intrinsics for 128-bit and 256-bit registers, that would otherwise only work with 512-bit registers. There is a 32 KB L1 cache for instructions and 32 KB L1 for data. The L2 cache is 1 MB and the LLC (Last Level Cache) is 22 MB. The CPU runs at a base frequency of 2.1 GHz. It has 4 sockets, each containing 16 cores with up to two hyperthreads per core. However, in this thesis, we considered the influence of the different vector layouts and sizes, not the influence of multiple memory channels or CPU cores. Thus, all benchmarks are single threaded.

The SIMD-implementation shows different challenges depending on the evaluation algorithm to be applied. For instance, the code-level parallelism based *Naïve* technique *Naïve/M* could be implemented using regular registers. However, this would not be efficient because single bits cannot be addressed. This introduces an overhead to test whether a set of arbitrary bits, which may or may not be aligned within byte boundaries, is set or not. A SIMD implementation has to solve this with a limited number of available instructions. Furthermore, while *BitWeaving/V* is trivially extended from the original approach to vector sizes, *BitWeaving/H* either has to make compromises in the usage of the registers, or work around the fact that the instruction set does not offer a full adder for numbers larger than 64-bit.

Naïve/M

Vector storage layout: The *Naïve* storage layout can easily be adapted to vector registers. Figure 3.6 shows different layouts in an exemplary way for 128-bit registers and 10-bit codes. The *Naïve* layout stores all codes consecutively in a register. Since 10-bit codes can not be fitted evenly over a 128-bit register, some bits of a register remain unused. However, the *Naïve* layout is more compact than the *BitWeaving/H* layouts, because there are no delimiter bits.

Predicate Evaluation: The most simple predicate evaluation, which can directly be performed on data in the *Naïve* layout, is an equality check. For such an evaluation, two tasks have to be solved: (1) a bit-wise equality check between the input data and the predicate, and (2) a check for all code words in the input, whether all bits of the comparison from step 1 are set. While task one can simply be done by applying an exclusive OR and negating the outcome, task two requires an additional bit-mask to filter the bits of each code word and perform the comparison. This is because we cannot explicitly access arbitrary bits of a vector register. The exact procedure for 128-bit is as follows:

1. Load the predicate in *Naïve* layout with `_mm_loadu_si128`.
2. Load data in naïve layout (*input*) with `_mm_loadu_si128`.
3. Perform bitwise XOR on the registers loaded in step 1 and step 2 with `_mm_xor_si128`.
4. Negate the result from step 3. Perform a bit-wise AND with a vector, where the bits at the position of the current code are set to 1 and all other bits set to 0 (*filter*). `_mm_andnot_si128` performs both operations.

5. Compare the result from step 4 with *filter* using `_mm_cmpeq_epi32_mask`. The result is an 8-bit mask with the first four bits set to one if both vectors are equal.
6. Compare the result from step 5 with an 8-bit number where the first four bits are set to one.
7. If all codes in *input* have been processed, repeat from step 2, else repeat from step 4 with the next code in *input*.

This procedure can be ported to 256-bit and 512-bit by simply renaming the intrinsics accordingly.

BitWeaving/H

Vector storage layouts: A straightforward way to implement *BitWeaving/H* using vector extensions is to load several 64-bit values containing the column codes and delimiter bits into a vector register. In this case, the original processor word approach is retained as proposed in *BitWeaving*. This vector layout is shown as *Layout 1* in Figure 3.6. However, this method does not use the register size optimally. For instance, in a 128-bit register, there is space for 11 column codes with a bit width of 10 and their delimiter bits (see Figure 3.6 *Layout 2*), but *Layout 1* can only hold 10 codes. In *Layout 2*, we treat the vector register as a full processor word and arrange the column codes according to the vector register size. Figure 3.7 shows the percentage of unused register space for different register sizes and both layouts, where the dashed line shows the usage for *Layout 1* and the remaining lines for *Layout 2*. As we can see, *Layout 2* makes better use of the vector register.

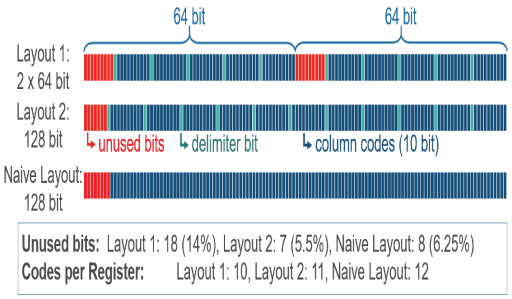

Figure 3.6: Different variants to arrange column codes in a vector register (figure taken from [LUH+18b]).

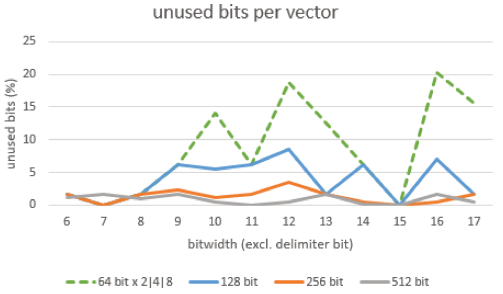

Figure 3.7: Percentage of unused bits per vector register depending on the vector layout (figure taken from [LUH+18b]).

Predicate Evaluation: Like in the original approach, the query evaluation on data in the *BitWeaving/H* layout in vector registers consists of several bit-wise operations and one addition. The exact bit-wise operations and their sequence depends on the comparison operator. For instance, a *smaller than* comparison or an *equality* check requires XOR operations and an addition as shown in Section 3.1.2. For counting the number of results quickly, an AND is also necessary. For 512-bit registers, this is realized by using AVX-512 intrinsics. The following steps are necessary for a *smaller than* comparison if the data is using the vector *Layout 1* (see Figure. 3.6):

1. The predicate and the data in *BitWeaving/H* layout is loaded with `_mm512_loadu_si512`. The filter value need to be loaded once.
2. The bit-wise XOR is performed with `_mm512_xor_si512`.

3. The addition is performed with `_mm512_add_epi64`.
4. Optional: To set only the delimiter bits, an AND between the precomputed inverted bit-mask and the result from step 3 is performed with `_mm512_and_si512`.
5. Optional: For counting the number of set delimiter bits `_mm512_popcnt_epi64` is applied.
6. Optional: The result from step 5 can be further reduced by adding the individual counts with `_mm512_reduce_add_epi64`.
7. Finally, the result is stored with `_mm512_storeu_si512`. If only the number of results is required, this step can be skipped. Afterwards, a new iteration starts at step 1.

Note that the SIMD intrinsics for steps 5 and 6 do not exist for 128-bit and 256-bit registers. In these cases, the result is written back to memory and treated conventionally, i.e. like an array of 64-bit values. These steps work for *Layout 1* but not for *Layout 2*, because, in step 3, a full adder is required. However, this functionality is supported for words containing 16-, 32-, or 64-bit, but not for 128-, 256-, or 512-bit. For *Layout 2* larger than 64-bit adder is required. However, there is no full adder available on recent CPUs larger than 64-bit. Therefore, to realize the addition for 128-, 256-, or 512-bit, more than one 64-bit full adders are used consecutively, whereby the carry at the 64-bit boundaries is determined and added to the subsequent 64-bit value. The detail of this custom-made addition mechanism is described in [LUH+18b].

BitWeaving/V

The implementation of vectorized *BitWeaving/V* is straightforward because all needed functionality is provided by the SIMD intrinsics of SSE, AVX2, and AVX-512. The layout stays the same as described in Section 3.1.2. In our case, the processor words V_i are 128-bit, 256-bit, or 512-bit long. The number of necessary words for a segment equals the number of bits per code word. The evaluation is also done as described in Section 3.1.2. For instance, an equality check using AVX-512 requires the following steps for each segment:

1. Load the first word of the segment and a vector filled with the 1st bit of the predicate with `_mm512_loadu_si512`.
2. Perform a bit-wise XOR on the registers loaded in step 1 with `_mm512_xor_si512`.
3. Invert result from step 1. Perform a bit-wise AND with a 1-vector if it is the first word of the segment, perform bit-wise AND with the result from the last iteration otherwise. The inverting and bit-wise AND are done with `_mm512_andnot_si512`.
4. Repeat from step 1 with next word of the segment and the next bit of the predicate.

3.2 FPGA IMPLEMENTATION

Besides the implementation of column scan technique using SIMD extensions, the second hardware-based implementation possibility is Field Programmable Gate Arrays (FPGAs). As described in Section 2.3.1, FPGAs are integrated circuits, which are reconfigurable after being manufactured. More specifically, a hardware description language, e.g., Verilog, is used to describe the hardware modules. This description is then translated via several steps to an implementation for the FPGAs. From the perspective of intra-code (intra-instruction) parallelism based storage layout, the advantage of FPGAs is that we can use an arbitrary length of processor word in the custom made hardware design.

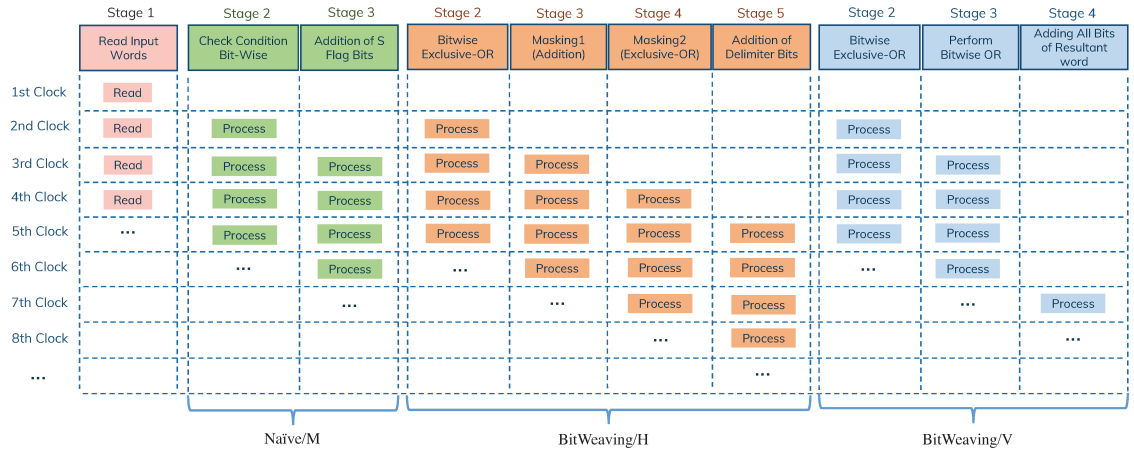

Figure 3.8: Pipeline-based PE for different intra-instruction parallelism based column scan techniques (figure taken from [LUH+18b]).

3.2.1 Processing Element

Inside the PL area of the targeted FPGAs, we developed Processing Element (PE) modules for restriction type of predicates using Configurable Logic Block (CLB) slices, where each CLB slice consists of Look-up Tables (LUTs), Flip-Flops (FFs), and cascading adders [TW13]. As illustrated in Figure. 3.8, the stages of PEs are processing words in pipeline manner through overlapping instructions, whereby we developed 3-stage, 5-stage and 4-stage pipeline-based PE for equality check predicate evaluation on the basis of *Naïve/M*, *BitWeaving/H* and *BitWeaving/V* techniques as introduced in Figure 3.3, Figure. 3.4 and Figure 3.5, respectively. All PEs have a common *Stage 1* of reading data words from main memory (see Figure 3.8). Rest in every stages a specific task is performed as shown in Figure 3.8, whereby the stages for different techniques are grouped by colors. The detail of *Naïve/M* pipeline stages are:

Stage 2: Check equality condition bit-wise and set *S* flag values according to the condition satisfying result,
Stage 3: Perform addition between *S* flags in order to count the matched column codes.

Then, the detail of *BitWeaving/H* pipeline stages are:

Stage 2: Executing bit-wise Exclusive-OR operations,
Stage 3: Masking operations (Addition),
Stage 4: Masking operations (Exclusive-OR) using predefined mask registers to prepare the output word,
Stage 5: Adding delimiter bits of output words in order to count the matched column codes.

Finally, the detail of *BitWeaving/V* based pipelines are:

Stage 2: Executing bit-wise Exclusive-OR operations,
Stage 3: Executing bit-wise OR operations,
Stage 4: Adding all bits of previous stage resultant words in order to count the matched codes (this stage would execute after every w cycles, whereas w is the width of column code).

For all cases, we write only the final output word of *count* to the main memory. This is not shown in Figure 3.8 as it is a non-pipeline stage which executes once only. Therefore, the total number of cycles for *Naïve/M*, *BitWeaving/H* and *BitWeaving/V* is $(n + 3)$, $(n + 5)$ and $(n + 4)$, respectively, where n is the total number of input words.

3.2.2 Basic Architecture

We started with developing 64-bit word based hardware design as Basic Architecture (BASIC_64) and subsequently increased the word width to 128-bit (BASIC_128) (see Figure 3.9). In this architecture, we use Direct Memory Access (DMA) between the main memory and the PE, to reduce the load of the ARM core and to reduce the latency of accessing the main memory. We prepared basic architecture based designs having either *Naïve/M* or *BitWeaving/H* or *BitWeaving/V* technique based PE, whereas each design is processing either 64-bit or 128-bit words. Therefore, Basic Architecture implies a very simple hardware design.

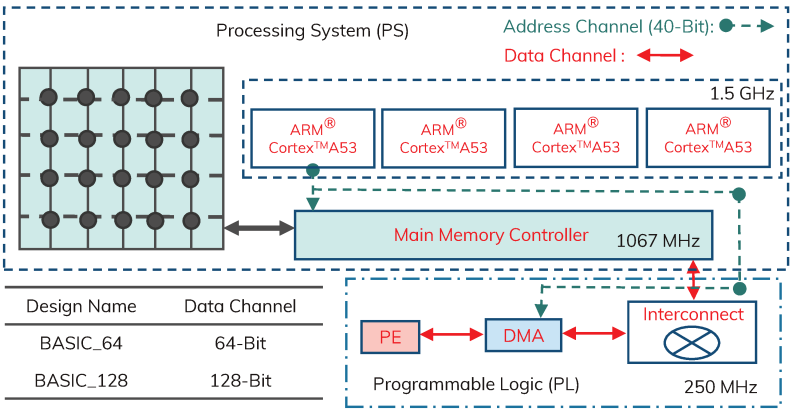

Figure 3.9: Basic Architecture (figure taken from [LUH+18b]).

3.2.3 Hybrid Architecture

Processing more data elements at once may provide benefits for further improvement on the column scan performance. In this case, the main challenge arises when the word to be processed becomes larger than 128-bit, because the width of the data channel of the main memory can only be extended up to 128-bit although the PEs are capable to handle word sizes beyond 128-bit. To tackle this challenge, we developed a hybrid architecture based on multiple DMAs, where each DMA is accessing the main memory via an independent data channel. As a consequence, we replicate our PE and DMA a few times depending on the number of available main memory data channels.

Moreover, two main memory modules are available on our targeted FPGA platform as mentioned in Section 2.3.1, whereby one is connected with the PS, and the other one is connected to the PL. The PS part main memory has four data channels, while the PL part has only one. However, the maximum channel width is 128-bit. So, maximum of five times of 128-bit words can be processed in parallel by using multiple main memory modules. However, having the maximum number of data channels in a design saturates the bandwidth of main memory. Therefore, we can prepare another custom hardware module, whereas 128-bit words can be combined into larger words. Thus, we implemented and replicated a custom combiner (namely Combiner_256) to combine two

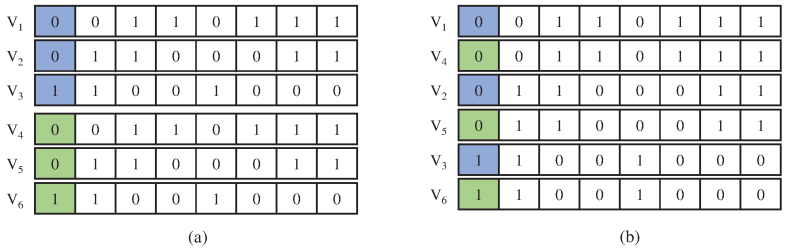

Figure 3.10: *BitWeaving/V* storage layout patterns, (a) for basic and (b) for hybrid architectures (figure taken from [LUH+18b]).

128-bit words to produce a 256-bit word. This introduces another stage in each proposed pipeline design, such that each PE is processing a 256-bit word in each clock cycle. Such a combiner can easily adaptable in *Naïve/M* and *BitWeaving/H* techniques based hardware designs as they stored codes in words horizontally rather than vertically like *BitWeaving/V*. Therefore, for *BitWeaving/V*, the input words are stored alternately rather than sequentially as illustrated in Figure 3.10(b) for 3-bit column codes, so that combiner can merge two words perfectly without breaking the sequence of codes. However, we keep the as usual storage pattern of *BitWeaving/V* for basic architecture as described in Section 3.1.2 (see Figure 3.10(a)).

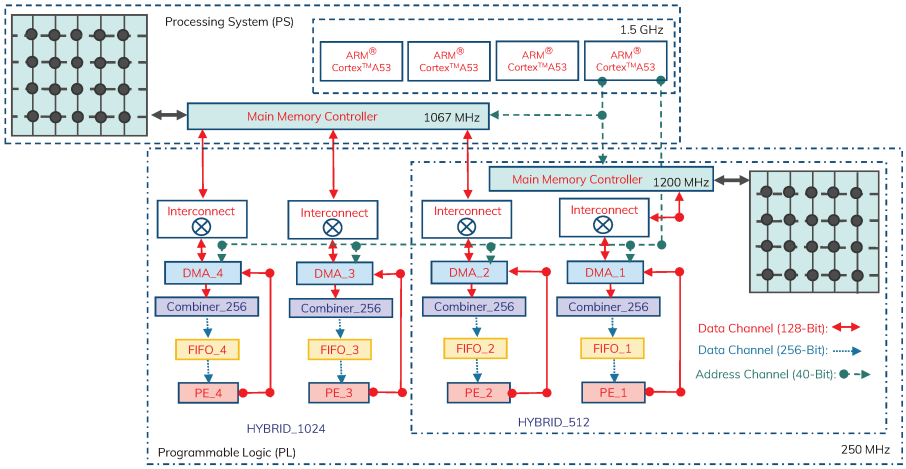

Figure 3.11: Hybrid Architecture (figure taken from [LUH+18b]).

Moreover, we use appropriate depth based FIFO between the combiners and the PEs to synchronize IO transmission between PEs and main memory, whereas stream-based data transmission is used. This avoids an overflow of the buffer. By mixing all the above mentioned concepts, we prepared hybrid architecture based designs as HYBRID_512 and HYBRID_1024, to process two and four times of 256-bit word in parallel to make 512-bit and 1024-bit words, respectively for all techniques (see Figure 3.11). So, Hybrid Architecture implies a complex hardware design.

3.3 COMPARATIVE EVALUATION

This section contains the evaluation results of our presented SIMD and FPGA based column scan mechanisms implementation techniques, whereby we separately evaluate each implementation.

3.3.1 SIMD Evaluation

In the evaluation, we want to observe the influence of the different vector layouts and sizes, not the influence of multiple memory channels or CPU cores. Thus, all benchmarks are single threaded. All measurement values are averaged over ten runs.

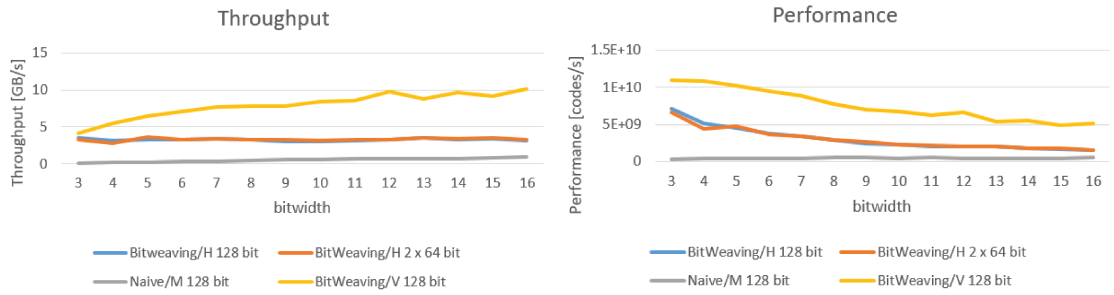

Figure 3.12: Throughput and performance of all presented 128-bit implementations for growing code sizes in terms of bitwidth (figure taken from [LUH+18b]).

Overview

Figure 3.12 shows a comparison of the performance (codes/s) and the throughput (GB/s) of all implementations using 128-bit. As expected, the *Naïve* layout provides the lowest throughput and performance. The time needed for evaluating every code in a register individually cannot make up for the slightly better usage of the available bits. The two layouts of *BitWeaving/H* do not show any significant differences but perform better than the *Naïve/M* approach.

Finally, *BitWeaving/V* shows the highest throughput and performance as could be expected since it is the approach with the least operations, which have to be performed while the input vector layout is more compact than in *BitWeaving/H*. Moreover, it has the smallest output size, resulting in less store operations, i.e. the bits containing the evaluation result are stored consecutively in the result register.

	Naïve/M	*BitWeaving/H*							*BitWeaving/V*		
Register size [bit]	*128*	*64*	*128*	*2 x 64*	*256*	*4 x 64*	*512*	*8 x 64*	*128*	*256*	*512*
Throughput-wise	☒	○	○	○	○	○	○	○	☑	☑	☑
Performance-wise	☒	○	○	○	○	○	○	○	☑	☑	☑

Figure 3.13: Comparison of the implemented column scan techniques (figure taken from [LUH+18b]).

BitWeaving/V is the only implementation, where the throughput increases when the code size increases, while *Naïve/M* and *BitWeaving/H* show a constant throughput. A reason for this behaviour is, in *BitWeaving/V*, the number of result bits per input bit decreases when the code size increases because more data is needed to compute a result. This leads to less store operations for the same amount of input data. For instance, if the bit width of the codes is 3, one segment consists of 3 processor words. Thus, the result, i.e. one processor word, is written back after these 3 processor words have been evaluated. But if the bit width is 15, there are 15 processor words, which are evaluated before one processor word is written back to memory. At the same time, the performance decreases for all *BitWeaving*

approaches while the size of the codes increases. In *BitWeaving/H*, this is because less codes fit into one processor word when the code size increases. In *BitWeaving/V*, it takes more operations before a result is computed as explained before. Before going into detail, Figure 3.13 shows an overview of all implementations. While *BitWeaving/V* stays clearly on top of the other approaches, it also shows some variation between the different vector sizes. *BitWeaving/H* is less influenced by the vector size.

BitWeaving/H

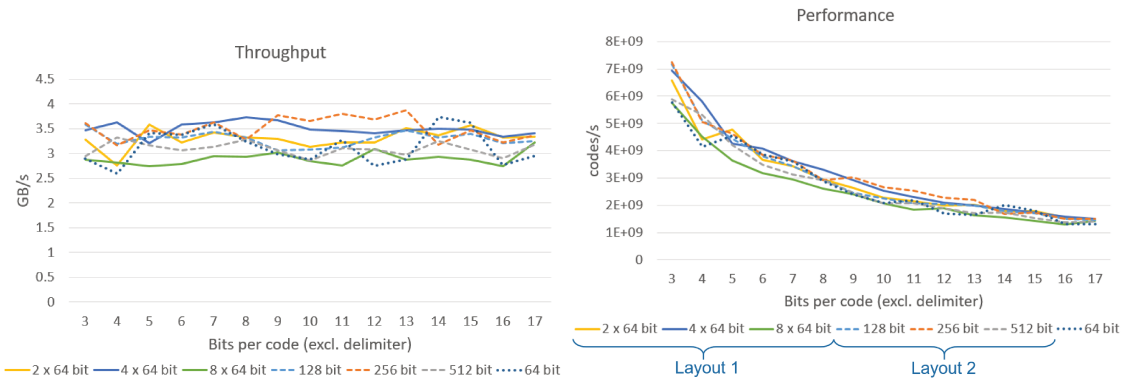

Figure 3.14: Throughput and performance for *BitWeaving/H* (figure taken from [LUH+18b]).

For codes containing 3-bit and a delimiter bit, the non-optimized 64-bit implementation achieves a throughput of 2.9 GB/s, which equals a performance of almost $5.8e9$ codes per second. The results for 3-bit column codes for all different horizontal vector layouts are shown in Table 3.1. All values are averaged over 10 runs. The results show, that there is a performance gain when using the vectorized approach, but it is not as significant as expected. For instance, we would expect a 100% speed-up when changing from 64 to 128 bits since we can process twice the data at once. Unfortunately, the throughput and the performance increase only by 14%. Moreover, it even decreases when changing from 256 to 512 bits for both vector layouts. However, these numbers can only provide a rough estimation since the throughput varies by up to 0.5 GB/s between the individual runs. Figure 3.14 shows the throughput and performance for all implemented *BitWeaving/H* versions and different code bitwidths. For comparison, we also implemented a scalar 64-bit *BitWeaving/H* version without any further optimization for special cases, such that the predicate evaluation is always executed in the same way.

The differences between the vectorized implementations and the scalar implementation becomes even smaller when the code size increases while the throughput oscillates between 2.5 GB/s and 4 GB/s for all versions (see Figure 3.14). There is a mere tendency of the 256-bit implementations to provide the best performance on average and for the 512-bit versions to provide the least performance. Nevertheless, the insignificance of the differences cannot be explained with the query evaluation itself. To find the bottleneck, we deleted the evaluation completely, such that only the vectorized load and store instructions were left. Then, we measured the throughput again and received results between 3 GB/s and 4 GB/s. A simple `memcopy` had a stable performance around 4.5 GB/s. Hence, in contrast to the naive implementation, the vectorized implementations are bound by the performance of loading and storing data, while the peak throughput cannot become larger than 4.5 GB/s.

Table 3.1: Evaluation Results on Intel Xeon Gold 6130, 3 Bits Per Code, Average over 10 Runs (table taken from [LUH+18b]).

Vector Layout	Throughput (GB/s)	Performance (Codes/s)
none (64-bit) (baseline)	2.9	5.8e9
2X64-bit (Layout 1)	3.3	6.6e9
4X64-bit (Layout 1)	3.5	6.9e9
8X64-bit (Layout 1)	2.9	5.8e9
128-bit (Layout 2)	3.6	7.2e9
256-bit (Layout 2)	3.6	7.2e9
512-bit (Layout 2)	2.9	5.9e9

BitWeaving/V

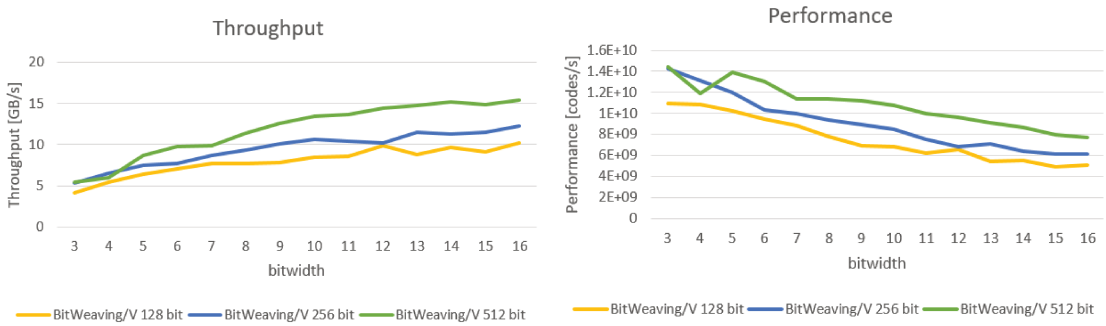

Figure 3.15: Throughput and performance for *BitWeaving/V* (figure taken from [LUH+18b]).

The performance and throughput of all implemented *BitWeaving/V* versions for different code sizes are shown in Figure 3.15. Contrary to *BitWeaving/H*, there is a clear increase in performance and throughput when the register size increases. A reason for this is the already mentioned smaller output size. Unlike in *BitWeaving/V*, in the horizontal approach, there is a padding between the result bits, which is as wide as a code word. To get these result bits, the whole vector word has to be extracted to several regular registers, where the bits can be shifted together, or even written back to memory completely if there are not enough registers. Since it is common for CPU cores to have only 16 general purpose registers, this worst-case is the usual case. However, *BitWeaving/V* does not have such padding, which makes the output more compact and reduces store operations. This relaxes the memory bandwidth bottleneck to a certain degree. This is especially obvious in the throughput for larger code sizes, where there are more input registers processed before the output register is written back. The performance decrease for 512-bit at a code size of 4-bit is reproducible. It comes with a throughput, which is not increased as much as expected. We did not find an explanation for this in the algorithm itself, especially because it only occurs for 512-bit. A possible reason is the fail of the optimizer during compilation. To test this theory, we compiled the same source code with icc, whereas we were using gcc before. The results did not show the decrease at 4-bit. Instead, there is a peak at 10-bit and the overall increase is less steady. Thus, it is safe to assume that these outliers are caused by the compiler rather than the implementation or the hardware.

3.3.2 FPGA Evaluation

Experiments are evaluated using two main metrics: throughput (GB/s) and performance (Codes/s). In these evaluations, we did not show energy consumption due to having the same behavior as performance as it depends on codes. However, in our work [LUH+18c], we evaluated the energy consumption metric as estimated energy and actual energy for codes per joule on *BitWeaving/H* scan. These evaluations are targeted to analyze the behaviour between *Naïve/M*, *BitWeaving/H*, *BitWeaving/V* column scan techniques for basic and hybrid architectures. We evaluated with BASIC_64, BASIC_128, HYBRID_512 and HYBRID_1024 designs for *Naïve/M*, *BitWeaving/H* and *BitWeaving/V* scan techniques, whereby Figure 3.16 shows the results for 3-bit column codes (excluding delimiter bit for *BitWeaving/H* scan) with equality check predicate.

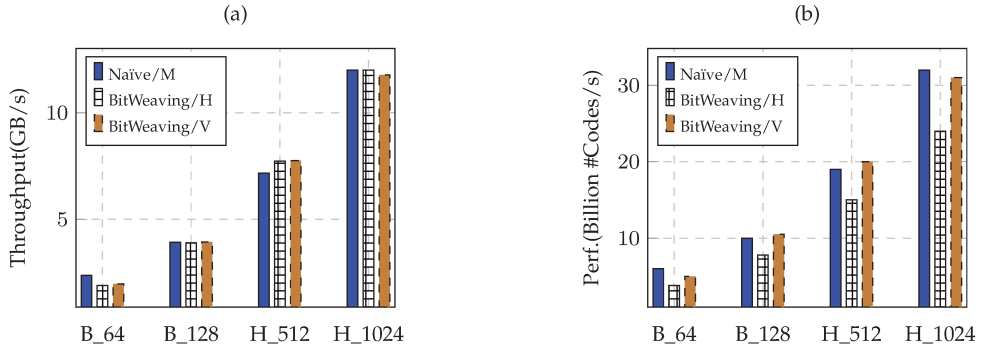

Figure 3.16: Analysis on (a) Throughput-wise, (b) Performance-wise for basic and hybrid architectures using different column scan techniques for 3-Bit Column Code (figure taken from [LUH+18b]).

We started with BASIC_64 design based evaluations and found that, *Naïve/M* provides higher throughput than *BitWeaving* techniques (see Figure 3.16(a)). Because it is able to execute on 300MHz frequency due to having simple logic instruction based technique, whereas others execute on 250MHz. However, this scenario changed for BASIC_128, HYBRID_512 and HYBRID_1024 based designs, where we achieved approximately the same throughput for all techniques (see Figure 3.16(a)) as the frequency of these designs are identical. Moreover, the different numbers of total clock cycles of PEs for different techniques as shown in Section 3.2.1 do not affect the throughput due to its pipeline mechanism. In the hybrid architectures-based designs data words are uniformly distributed among the PEs. In addition, the hybrid architecture based designs are processing beyond 256-bit width based data words through multiple main memory data channels and also flexible to use additional hardware (i.e., Combiner_256, FIFO), which is not available on BASIC_64 and BASIC_128 designs. As a consequence, for all techniques, HYBRID_1024 gives a peak throughput of approx. 12GB/s, whereas three data channels from PS part main memory and one data channel from PL part main memory, are used. Although the PS part main memory has maximum of four data channels. But using the maximum number of channels in parallel saturates the bandwidth of PS part main memory. So, in HYBRID_1024 we used multiple main memories in order to have four individual data channels.

Performance-wise evaluation varies between different techniques. *BitWeaving/H* provides always less performance in terms of codes per second among all techniques (see Figure 3.16(b)). In BASIC_64 design, *Naïve/M* provides the highest performance (see Figure. 3.16(b)). However, rest in all designs the performance become marginal between *Naïve/M* and *BitWeaving/V* (see Figure 3.16(b)). There are two reasons. On the one side, the number of *bit padding* increases in *Naïve/M* technique based BASIC_128,

HYBRID_512 and HYBRID_1024 designs exponentially than BASIC_64 as the word size increases. As mentioned earlier, hybrid architecture merged two 128-bit words to make one 256-bit word. So, there are 2-bit *bit padding* in one 128-bit word for 3-bit column code. It extends to 4-bit *bit padding* for 256-bit word and so on. As a consequence, we are losing number of codes per word as the word size increases which effects the performance. On the other side, there is no chance of losing codes in *BitWeaving/V* as each bit of a code is stored vertically per word (see Figure 3.1(d)). This makes the marginal balance of processing codes per second between *Naïve/M* and *BitWeaving/V*. Therefore, performance-wise *Naïve/M* and *BitWeaving/V* both win over *BitWeaving/H*.

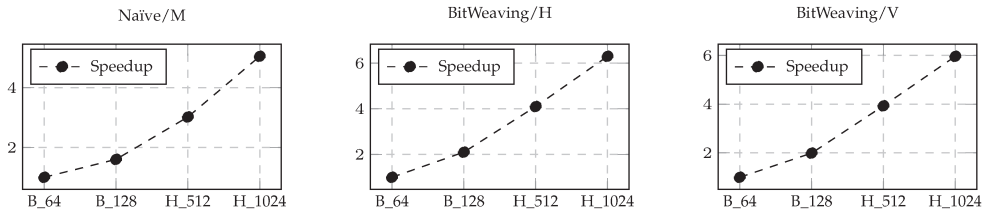

Figure 3.17: Analysis in terms of Speedup between basic and hybrid architectures for all column scan techniques (figure taken from [LUH+18b]).

Technique-wise the behavior of throughput and performance are identical among the basic and hybrid architectures (see Figure 3.16). Therefore, the speedup for main memory based intra-value parallelism based scan techniques among the basic and hybrid architectures on the targeted FPGA platform is linear (see Figure 3.17), whereas the BASIC_64 design is the baseline. This defines, that the HYBRID_1024 design is best for all mentioned column scan techniques on FPGAs.

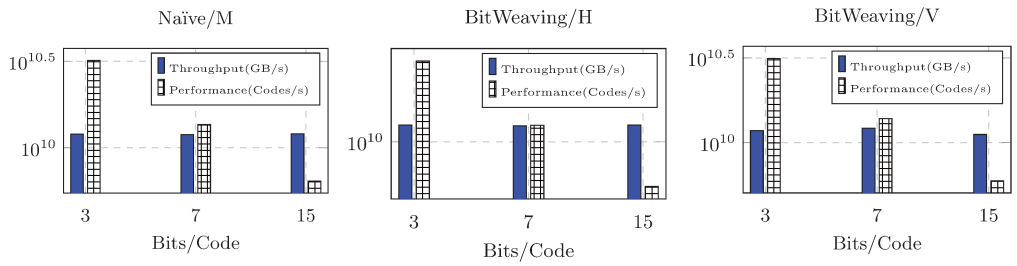

Figure 3.18: Analysis on HYBRID_1024 design using different column scan techniques for different number of bits per code (figure taken from [LUH+18b]).

We also evaluated different numbers of bits per (column) code for three mentioned techniques using the best design: HYBRID_1024 (see Figure 3.18). In this case, symmetrical behavior was found between all techniques, whereby a linearly decreasing behavior was found for performance as the bits per code increases except the throughput. The reason is, increasing the code size decreases the number of codes per word which negatively affects the performance which is evaluated on the basis of the number of codes as expected, whereas throughput evaluation is independent of codes.

Table 3.2 illustrated the overall resource utilization in terms of LUTs(%) and FFs(%) for the best design HYBRID_1024 among all techniques using Xilinx® resource analyzer, whereby *Naïve/M* technique requires least resources due to its straight-forward predicate evaluation mechanism. After all kind of evaluations we found that, throughput-wise all techniques showed identical behaviour, performance-wise *Naïve/M* and *BitWeaving/V* techniques are better than *BitWeaving/H*, but resource utilization-wise *Naïve/M* technique is the most optimum one. Finally, these leads us to conclude that, *Naïve/M* technique is the best technique for FPGAs (see Figure. 3.19).

Table 3.2: Resource Utilization for Hybrid_1024 Designs (table taken from [LUH+18b]).

Scan Tech.	LUTs(%)	FFs(%)
Naïve/M	12.89	8.64
BitWeaving/H	13.68	9.5
BitWeaving/V	13.99	9.15

	Naïve/M	*BitWeaving/H*	*BitWeaving/V*
Throughput-wise	☑	☑	☑
Performance-wise	☑	☒	☑
Resource Utilization-wise	☑	☒	☒

Figure 3.19: Evaluation matrix-wise analysis on the column scan techniques (figure taken from [LUH+18b]).

3.4 LESSONS LEARNED AND SUMMARY

Section 3.1.3 and Section 3.2 showed that *Naïve* and *BitWeaving* column scan mechanism is not only implementable using common CPU based wider SIMD vector registers but also using new class of accelerator namely FPGA. It is also noticeable that the hardware-based optimizations of *Naïve* and *BitWeaving* employing SIMD or FPGA is feasible.

In SIMD-implementation, exhibiting different varieties of storage layouts for the wider vector registers (i.e., SSE, AVX2, AVX-512) to increase intra-data parallelism provides slightly increased performance but not as much as expected. Two vital reasons are highlighted for this less performance gain: i) the main memory bandwidth is already fully utilized for low vector register size, as a consequence utilizing wider vector registers become pointless for performance gain; ii) SIMD intrinsics available instructions are often limited, it requires very good understanding of data parallelism feature. In particular, SIMD programming is highly complicated. Therefore, neither wider vector registers nor different categories of vector storage layout overcome the performance penalties for column scan acceleration using SIMD.

However, the evaluation results proved that, FPGAs is a reliable opportunity in the context of performance gain for column-scan operator. The key advantage of such hardware-based optimization is, we can design a custom-made system as per requirement. FPGA provides flexibility to develop pipeline based custom-made processing element for any type of column scan technique. Pipelineing is one of the effective ways to improve the performance. The architectural approach of pipeline allows the simultaneous execution of several instructions. In other words, pipeline exploits parallelism at the instruction level by overlapping the execution of several instructions. Proper pipeline oriented processing elements as well as design implementation is also helpful to reduce the usage of resources of FPGAs, whereby reduce resource utilization improves cost-efficiency. However, unlike SIMD, FPGA is not bound with limited instructions as the key feature of FPGA is customizability. Additionally, working on FPGA flawlessly two perspectives are considered: i) custom-made hardware implementation, e.g., processing element; ii) pre-build hardware utilization, i.e., DMA, ARM® core. Because implementing any type of hardware accelerator oriented design on FPGA is doable, but such design also requires a control mechanism on top of it to reduce system complexity and resource utilization. Therefore, we targeted the CPU-FPGA

based hybrid system to accelerate performance as well as reduce resource utilization, whereby FPGA is used for developing the column-scan accelerator and ARM® processor is used for initiating the address and control signals to the overall custom-made column scan system. More importantly, the high bandwidth based direct data communication channels to main memory of the targeted hybrid CPU-FPGA system play a significant role regarding performance gain. Finally, the FPGA optimization is superior to SIMD optimization from the performance as well as throughput perspective. However, 512-bit SIMD vector registers with a *BitWeaving/V* deliver the best performance on our test SIMD intrinsics oriented hardware. In contrast to SIMD, the FPGA optimization brings a significant increase in performance. In this case, a data width of 1024-bit based hybrid architecture delivers the best performance. However, hybrid architecture utilizes more resources than basic architecture. Thus, the first and second research challenges of this thesis are resolved through hybrid architecture, whereby it defines the feasible level of design configuration aspects including a balanced tradeoff between performance enlargement and optimum hardware resource utilization.

In this chapter, we explored two hardware-based implementation opportunities for column scan optimization using SIMD extensions and using hybrid CPU-FPGA system. In particular, we analyzed the behavioral differences between *Naïve* and *BitWeaving* scan mechanisms as per hardware-based implementation. With both implementations, we can improve the scan performance, whereas the FPGA is best for *Naïve* technique and *BitWeaving* is perfect for SIMD. Therefore, improving scan performance through FPGA does not require any fancy scan mechanism like *BitWeaving* due to its high parallelism, adaptability, reconfigurable criteria which makes it easier to prepare any technique as per requirements.

ADAPTIVE LIGHTWEIGHT COMPRESSION SYSTEM

So far, we concentrated on column scan on compressed data through utilizing the compact columnar storage layout for performance acceleration, not on the construction of the compressed layout. In this context, lossless lightweight integer compression schemes are crucial to keep the memory storage capacity as low as possible and to speed up the column scan as presented in Chapter 3. Hence, this chapter gives an overview of lossless lightweight integer compression. We focus on physical-level as well as logical-level lightweight integer compression algorithms. Moreover, we discuss the FPGA-based implementations for physical-level, logical-level, and cascaded compression algorithms as well as develop the adaptive lightweight integer compression system. Later, an exhaustive experimental evaluation for all the hardware-oriented implementations is shown. At last, we summarize this chapter with lessons learned. The physical-level lightweight integer compression implementation is based on our publication [LNH+19]. This chapter resolves the third research challenge mentioned in Section 2.3.3.

4.1 LIGHTWEIGHT INTEGER COMPRESSION

As mentioned in Chapter 2, the database architecture shifted from a disk-centric to a column-store main memory-centric architecture to efficiently exploit the ever-increasing capacities of main memory. To reduce the gap between computing power of the CPU and main memory bandwidth, to minimize the necessary storage capacity, and to increase the query performance, main memory stored columnar integer values are usually compressed using a lightweight integer compression algorithm [AMF06, DHHL17, HHDL16, LMF+16]. Some computational efforts in terms of lightweight integer compression algorithm always include to reduce the main memory space. As a consequence, the result is a compressed columnar representation, whereby the input sequence is represented with as few bits as possible on the physical level. Therefore, this section is structured as follows:

(1) We give an overview of available integer compression schemes including a classifcation.
(2) We introduce state-of-the-art implementation concepts.
(3) We discuss advantages and disadvantages of the current solutions.

4.1.1 Overview and Classification

The lightweight integer compression algorithms can be differentiated into two levels. Firstly, compressing integer values physically by reducing the number of bits per value. Secondly, compressing integer values logically by mapping large values to small values. Hence, the large corpus of lossless lightweight integer compression algorithms is categorized in three types, i) Physical-Level, ii) Logical-Level, and iii) Cascades of logical-level and physical-level, whereby physical-level compression algorithms reduce values at bit-level, logical-level compression algorithms reduce values at the value-level, and cascades reduce values at value-level as well as bit-level, respectively. Four well-known and frequently used lightweight integer compression algorithms are, *Null Suppression (NS)*, *Delta*, *Frame of Reference (FOR)* and *Run Length Encoding (RLE)*, whereby *NS* is the physical-level and the other three algorithms are the logical-level compression algorithms as illustrated in Figure 4.1.

Null Suppression (NS). *NS* is one of the most studied physical-level compression schemes in this domain [AMF06, LMF+16]. The basic idea of *NS* is to discard the leading unused zeros in the bit representation of integer values. For instance, in Figure 4.1, for a given

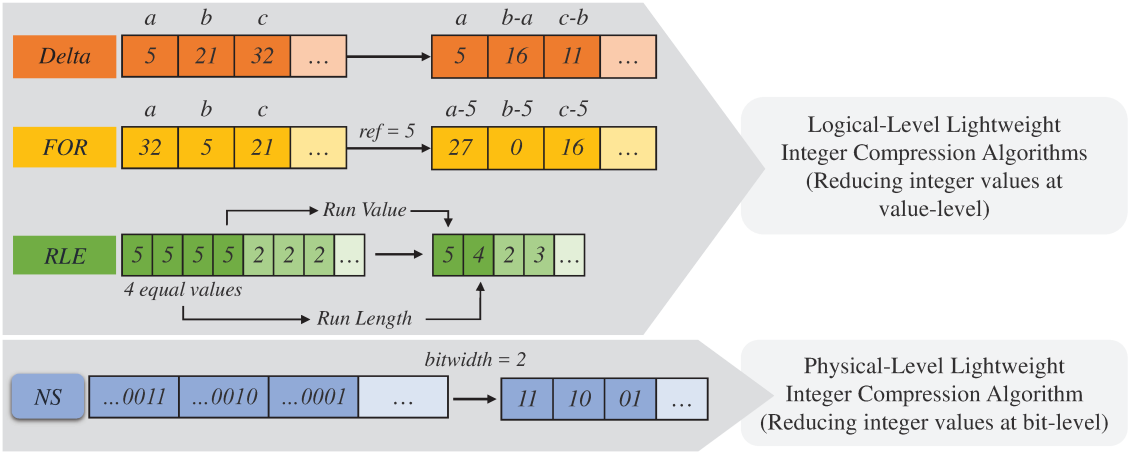

Figure 4.1: Illustration of Lightweight Integer Compression Algorithms.

dataset all the leading unused zeros are discarded in each value and keep the reduced bitwidth based values after performing *NS* compression.

Delta. *Delta* compresses integer values through preserving the consecutive values differences. That means *Delta* reduces a value by keeping the difference from the preceding position value. Therefore, two requirements need to be satisfied for *Delta* compression. Firstly, *Delta* compression is suitable for a sorted dataset, so that it can avoid the negative value occurrences. Secondly, the first value of a dataset is always required to store for decompression. For instance, in Figure 4.1, for a given sorted dataset the first (a), second (b), and third (c) values are 5, 21, and 32, respectively. Thus, after *Delta* compression, the first three consecutive differences are 5, 16, and 11, whereby it keeps the first value as it is.

Frame of Reference (FOR). Two steps are required in *FOR* compression. Firstly, *FOR* detects the smallest value as a reference number. Secondly, *FOR* keeps the differences between integer values and the smallest value. As all values deduct the smallest value, there is no chance for negative value occurrences. Hence, *FOR* is suitable for sorted and unsorted datasets. For instance, in Figure 4.1, for a given dataset the first (a), second (b), and third (c) values are 32, 5, and 21, respectively, and the reference value is 5. Thus, after *FOR* compression, the first three differences are 27, 0, and 16.

Run Length Encoding (RLE). *RLE* counts the successive subsequence of equal values occurrences. Hence, it reserves integer values, and its corresponding counts of equal values occurrences as run lengths. For instance, in Figure 4.1, for a given dataset the successive equal value of 5 is 4. Thus after *RLE*, the run length of the corresponding run value 5 is 4.

However, *Delta* and *FOR* always have a $(1 : 1)$ mapping between uncompressed input and compressed output values. But *RLE* has a $(N : 1)$ mapping instead of a $(1 : 1)$ mapping between uncompressed input and compressed output values. It happens because *RLE* encodes a successive subsequence of equal values in the output as a pair of a value and run length. Therefore, in *RLE*, it is not always the case that every input value is mapped necessarily to an encoded output value.

4.1.2 State-of-the-art Implementation Concepts

Database researchers are moving their research direction more and more towards lightweight integer compression algorithms due to its potential benefits. Thus, there are lightweight integer compression algorithms research works available, not only in the direction of a CPU-based implementation [DUH+19], but also for GPU [RB17, PK12, FHL10]. Some selected research works regarding CPU or GPU-based lightweight integer compression implementation concepts are discussed below.

A dataset for compression is usually partitioned with a group of a fixed number of values called *block*, as most of all lightweight integer compression algorithms implementational concepts depend on *block*. The size of a *block* should be power of 2, such as $2^7 = 128$, $2^8 = 256$, $2^9 = 512$, etc. For instance, a dataset can be partitioned with a group of 128 integer values. That means each *block* consists of 128 integer values.

Work by Damme et al. (DUH+19)

In recent years, CPU-based vectorized implementation of lightweight integer compression algorithms using SIMD (Single Instruction Multiple Data) instructions has attracted a lot of attention as SIMD reduces the computational effort. For instance, Damme et al. [DUH+19] implemented several lightweight integer compression algorithms as well as cascades of basic techniques utilizing CPU-based vectorization for implementations. In particular, Damme et al. [DUH+19] used the 128-bit vector registers of the Intel SIMD instruction set extension SSE, which can fit four uncompressed 32-bit integers. In the following, we introduce some SIMD-based compression algorithms implementation concepts that are based on Damme et al. [DUH+19] work.

Bitpacking (BP) is one of the most frequently applied *NS* algorithms which shows a very good behavior for different data properties [DHHL17]. *BP* compresses integer values by omitting leading unused zeros that physically reduce the values at bit-level. *BP* partitions uncompressed dataset with a fixed number of values as *blocks* and detects the largest value bitwidth per *block*. Afterward, it packs the values per *block*, whereby the bitwidth of all the values in a *block* is dependent on its corresponding largest value bitwidth. Damme et al. [DUH+19] obtained the implementation of *BP* as SIMD-BP128 from the FastPFOR-library.6. The *BP* implementation uses vectorized shift and mask operations as well as a dedicated optimized packing and unpacking routine for each of the 32 possible bitwidths of a vector register.

The vectorized implementation of *Delta* represents each input element as the difference to its fourth predecessor [DUH+19]. Thus, the processing of four integers happens at once through vectorization. In the vectorized *Delta* implementation, the first four elements are always copied from the input to the output during compression. Damme et al. [DUH+19] calculated the differences using `_mm_sub_epi32()`.

Damme et al. [DUH+19] implemented the *FOR* compression as a vectorized two-pass algorithm. Firstly, the vectorized *FOR* implementation iterates over the input and determines the reference value of the minimum using `_mm_min_epu32()`. Then, the vectorized *FOR* implementation copies this minimum into all four elements of one vector register. Secondly, the vectorized *FOR* implementation iterates over the input again and subtracts this vector register from four input elements at a time using `_mm_sub_epi32()`. At the end of the vectorized *FOR* implementation, the reference value is appended to the output.

Damme et al. [DUH+19] proposed a four-step based vectorized *RLE* implementation. In the first step, the vectorized *RLE* implementation loads one 128-bit vector register with four copies of the current input element. Secondly, the next four input elements are loaded. Thirdly, a parallel comparison by intrinsic `_mm_cmpeq_epi32()` is employed, and the result is stored in a vector register. Fourthly, the vectorized *RLE* implementation obtains a 4-bit comparison mask using `_mm_movemask_ps()`. Each bit in the mask indicates the (non-)equality of two corresponding vector elements. The number of trailing 1-bits in this mask is the number of elements for which the run continues. If this number is 4, then the run's do not end yet and continue from the second step. Otherwise, the run's has reached the end, the implementation appended the run value and run length to the output, and continue from step one with the next element. The implementation is repeated the above steps until the end of the dataset is reached.

Finally, Damme et al. [DUH+19] mention that many lightweight integer compression algorithms can be ported to the SIMD extensions of AVX2 (256-bit vector registers) or AVX-512 (512-bit vector registers), but for some algorithms, this is not possible. For instance, large *block*-based null suppression algorithms increase the vulnerability of these algorithms to outliers in the data, which affects both the compression rate as well as the performance negatively. Basically, the algorithms quickly become memory-bound when the computations are accelerated through wider vector registers processing more data elements at once.

Work by Fang et al. (FHL10)

Fang et al. [FHL10] propose nine lightweight integer compression schemes (*null suppression with a fixed length, null suppression with variable length, dictionary encoding, bitmap, run-length encoding, frame of reference, delta, separate, scale*) on the GPU and the combinations of these schemes for a better compression ratio. Based on this work [FHL10], some of the GPU-based compression algorithms implementation concepts are highlighted below.

Fang et al. [FHL10] implement the GPU-based *RLE* in four steps. First, the implementation identifies boundaries between runs. A boundary is represented by a 1 between 0's. Fang et al. [FHL10] called this array of 0's and 1's boundary array. Second, the implementation gets the write positions for output data by applying an exclusive prefix sum on the boundary array. Third, given the write positions, the implementation scatters both the values and the boundary positions. Finally, the implementation computes each run length by subtracting the corresponding boundary position from the next boundary position. The compression process of Fang et al. [FHL10] is lock-free, which well exploits the massive parallelism of the GPU.

In the *FOR* implementation, Fang et al. [FHL10] transform each value in a column into an offset from a base value, whereby the base value of a column represents the smallest value. The *FOR* implementation uses the map primitive to subtract the base value from each array element.

The *Delta* implementation of Fang et al. [FHL10] encodes a value in a column as the difference from the value at the preceding position. The first value in the column is stored in the database catalog. The differences are usually within a small value domain for a sorted column. The *Delta* implementation uses the map primitive to perform compression and applies the inclusive prefix sum for decompression.

Fang et al. [FHL10] designed a compression planner to find the optimal combination. This GPU-based compression and decompression achieved a processing speed of 45

and 56 GB/s, respectively. This implementation also utilized the partial decompression concept to improve GPU-based query co-processing performance. Fang et al. [FHL10] included their GPU-based compression into MonetDB, an open-source column-oriented DBMS, and demonstrated the feasibility of offloading compression and decompression to the GPU.

Work by Przymus et al. (PK12)

Przymus et al. [PK12] focus on the acceleration of the GPU-based decompression performance by the utilization of shared memory. Przymus et al. [PK12] developed optimized GPU parallel decompression methods using global and shared memories. This work [PK12] uses lightweight compression for time series and applies it to a data-intensive pattern matching mechanism. This work [PK12] shows that data decompression in GPU shared memory generally improves the performance of other data-intensive applications due to ultrafast parallel decompression procedure. This work [PK12] analyzes the relationship between global and shared memory decompression showing that although shared memory may limit the number of possible applications due to lack of global synchronization mechanism, it significantly improves performance. Przymus et al. [PK12] achieve speed up improvement from 2% with decompression rate 2 up to 10% with decompression rate 6. However, this work's current implementation version does not have a general-purpose library that could be used in a similar way to the CUDA library. Moreover, the implementation uses a common iterator pattern that has a possibility of overlapping memory blocks which could break the barrier of inter-block threads communication.

Work by Rozenberg et al. (RB17)

Recently, Rozenberg et al. [RB17] have focused on decompressing columns into GPU memory. Rozenberg et al. [RB17] combine two prominent approaches, i) materializing small chunks into a cache between query plan operations, ii) just-in-time compiled pipelined execution of multiple fused operators, where tuple data is passed through CPU registers. In these approaches, data is required to be decompressed into registers before filtering and aggregation. For the cascaded compression schemes, it is decompressed into the *block*-shared memory, whereby *blocks* are performing operations on decompressed chunks. In both of these cases, the decompression implementations serve well as they try to get rid of the bottleneck of having written much than read into global GPU memory. This work [RB17] also includes an initial exploration of GPU oriented patched compression schemes. In the patched compression schemes, a query plan compiler schedules three different work paths for the underlying-scheme decompressed column. In the first work path, no patches are applied. In the second work path, the naive patch data of a small column is represented in sparse. In the third work path, the query plan compiler schedules a different decompressor that patches locally, whereby queries are operating on the smallest byte-addressable compressed form of data directly. The third path points at the possibility of queries operating on the smallest byte-addressable compressed form of data directly. If data are compressed using sub-byte bitwidths, then partial decompression to the smallest whole byte-width could be used, which has the disadvantage of decompressing to the full SQL type the schema demands.

4.1.3 Discussion

As presented in the preceding subsection there are a lot of algorithms and implementations available. Damme et al. [DUH+19] conducted an exhaustive experimental survey by evaluating several lightweight integer compression algorithms as well as cascades of basic techniques. Damme et al. [DUH+19] considered three different hardware platforms for their evaluation: *Haswell*, *Xeon Phi* and *Skylake*. During the survey evaluation Damme et al. [DUH+19] observed some general trends. For instance, null suppression usually performs better when the values are lower, SIMD-BP128 is one of the good choices among other null suppression techniques, *RLE* works well for long runs, the logical-level techniques can improve the data distributions significantly in favor of null suppression. Therefore, cascades of logical-level and physical-level techniques can achieve very good compression rates, which are depending on the data characteristics. Damme et al. [DUH+19] concluded their survey with two statements, i) there is no single-best lightweight integer compression algorithm suitable for all situations, ii) the best algorithm regarding the compression rate is often not the fastest. Damme et al. [DUH+19] proposed a cost-based model for the selection strategy and proved its ability to select a suitable lightweight integer compression algorithm for a given dataset. However, the selection strategy has to be done on the algorithm level during runtime which restricts the flexibility. Moreover, a *block*-based selection strategy on the algorithm level during runtime would be more advantageous.

Generally, compression is an additional processing step that should not come with additional cost during runtime. Thus, compression should be provided *transparently* without compromising the overall system performance. To achieve that, advances in hardware are always an interesting opportunity, but the utilization aspects are also a major challenge. At the moment, hardware systems are more and more moving from homogeneous CPU systems towards hybrid systems with different computing units. In particular, hybrid hardware systems incorporating a Field Programmable Gate Array (FPGA) and a CPU are emerging, being very interesting from a performance perspective. Generally, FPGA is an integrated circuit, which is reconfigurable after being manufactured. Thus, FPGA works as a hardware extension to the database systems where some specialized function is implementable efficiently. Additionally, modern FPGAs come with more flexibility for an efficient implementation that has direct access to the main memory as well as offloading specialized function to the FPGA.

We are looking for an adaptive technique out of some compression artifacts depends on a *block*-based selection strategy that is orchestrating at runtime. Based on this, we focus on lightweight integer compression acceleration by offloading such functionality to Field Programmable Gate Array (FPGA) in our target hybrid CPU-FPGA hardware system. Moreover, to the best of our knowledge, none of the existing works investigate, neither the domain of FPGA-based hardware-level implementation of a lightweight integer compression algorithm nor exploring the different categories of hardware designs on FPGA to achieve high-throughput regarding compression.

4.2 FPGA-BASED IMPLEMENTATION OF LIGHTWEIGHT INTEGER COMPRESSION ALGORITHMS

Using FPGA resources, such as LookUp-Tables (LUTs), FlipFlops (FFs), etc. we can implement any type of application-specific custom-made hardware. Therefore, in this section, we describe the implementation details of custom-made physical-level and logical-level lightweight compression accelerators. We explored different design templates for physical-level and logical-level lightweight compression by incorporating multiple accelerators for the best possible compression throughput.

4.2.1 Recap FPGA-based Architecture

In Chapter 3, we described the single (see Subsection 3.2.2) and multiple (see Subsection 3.2.3) DMA-based FPGA architecture for *column scan*. The lesson learned of such a type of FPGA architecture is that the independent access between the main memory and the custom-made hardware through a DMA is beneficial regarding throughput. Thus, a DMA is acting as a bridge between the main memory and the *PL* part custom-made hardware. We define such a type of design template having a single DMA as `DMA_1`, which is illustrated in Figure 4.2. As described in Subsection 3.2.3 we can replicate the design template `DMA_1` multiple times to prepare multiple DMA-based design templates to accelerate the throughput. The number of DMA replication in an FPGA architecture depends on the number of available AXI data channels of the main memory controller. In our targeted Hybrid CPU-FPGA system, the *PS* part main memory controller has 4 AXI data channels, whereby each data channel is 128-bit wide. That means we can replicate the design template `DMA_1` from 2 to 4 times, whereby each DMA is accessing the main memory through an independent AXI data channel to avoid the data collision. We named such templates `DMA_2`, `DMA_3`, and `DMA_4` (see Figure 4.2).

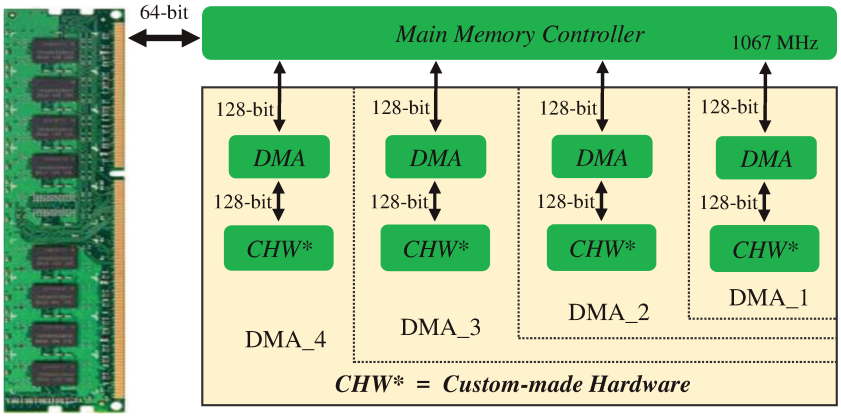

Figure 4.2: Different #DMA Oriented Hardware Design Templates.

4.2.2 Custom-made Compression HW Implementation

This section describes the proposed custom-made physical-level and logical-level lightweight compression hardware implementation in detail.

Physical-Level Compression

In this section, we describe our implementation detail of the FPGA-aware physical-level compression algorithm *BitPacking (BP)*.

As mentioned in Subsection 4.1.2, *BP* partitions a sequence of integer values into *blocks*, determines the bitwidth of the largest value per *block*, and compresses each value as per largest value bitwidth in a *block*. Thus, two similar read operations are required, (i) one for determining the bitwidth of the largest value in a block, and (ii) one for packing the values in a block. Accessing the main memory twice just to read the same set of values is inefficient. As a consequence, an effective FPGA option is to use internal buffers with a depth which equals the *block* size to temporarily store the integer values *block*-wise. As

mentioned earlier, the *block* size has always the power of 2, such as $2^7 = 128$, $2^8 = 256$, $2^9 = 512$, etc. Thus, the depth of a buffer could be 128, 256, 512, etc. Additionally, this buffer option helps filling up the pipeline stages in an FPGA-aware *BP* implementation.

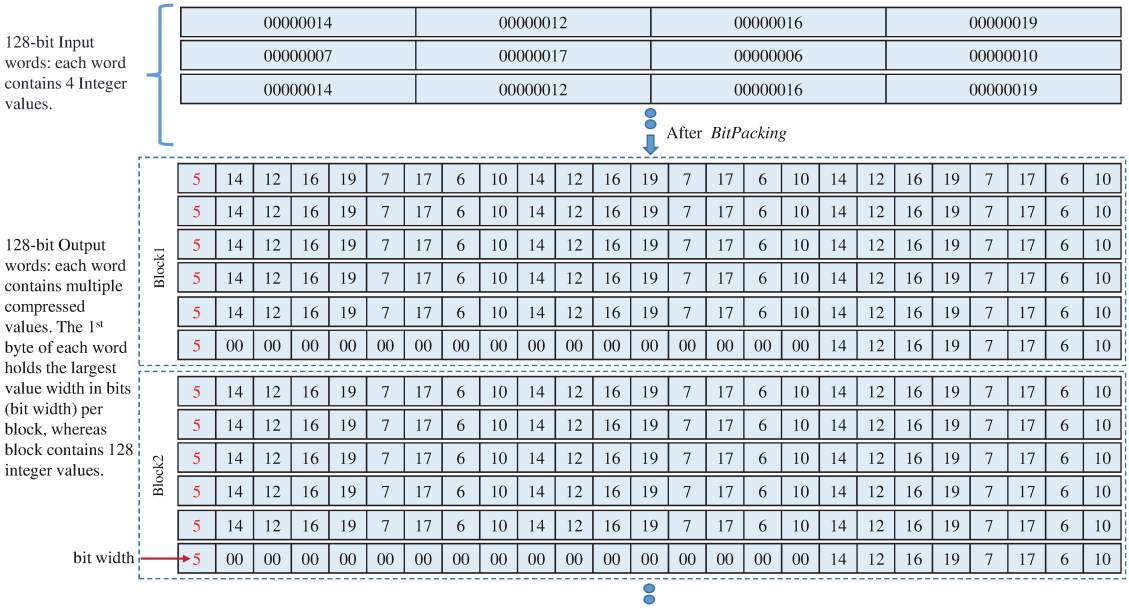

Figure 4.3: Illustration of *Bitpacking* Compression.

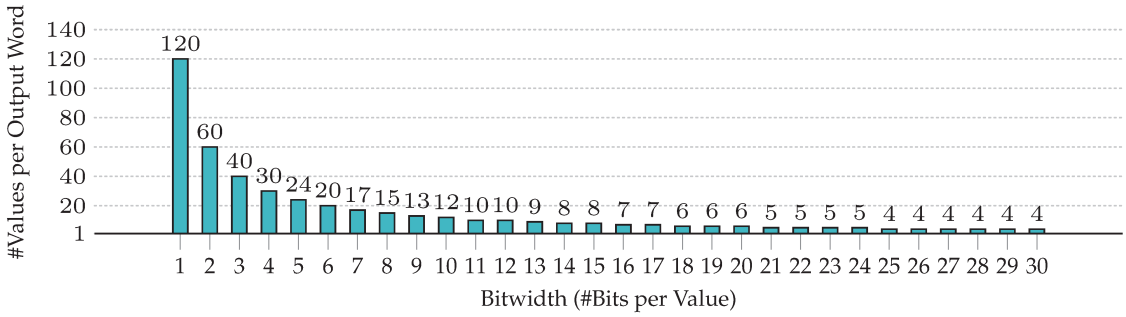

Figure 4.4: Analysis in terms of Bitwidth and #Values Packing Per Output Word.

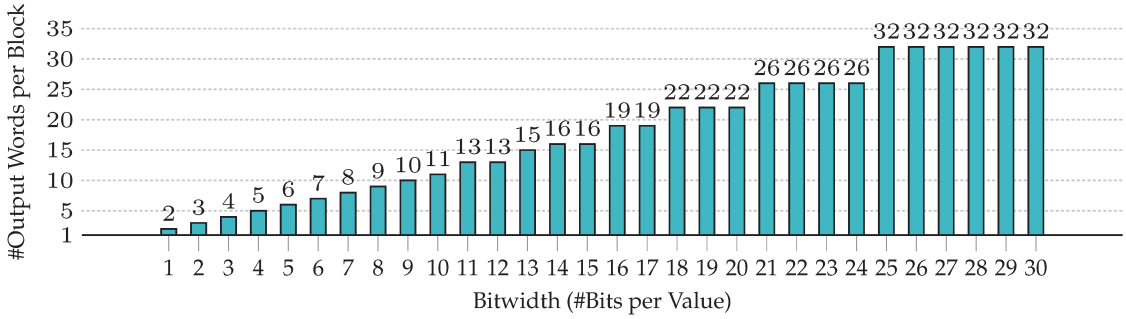

Figure 4.5: Analysis in terms of Bitwidth and #Output Words in a *Block* for *Block* Size 128.

As per our target system maximum *I/O* port bitwidth configuration, the input and output words bitwidth in the *BP* implementation are 128-bit. Therefore, an input word contains four 32-bit integer values, and the number of values packed in an output word depends on the largest value bitwidth in a *block*. Figure 4.3 depicts an exemplary way of *BP*,

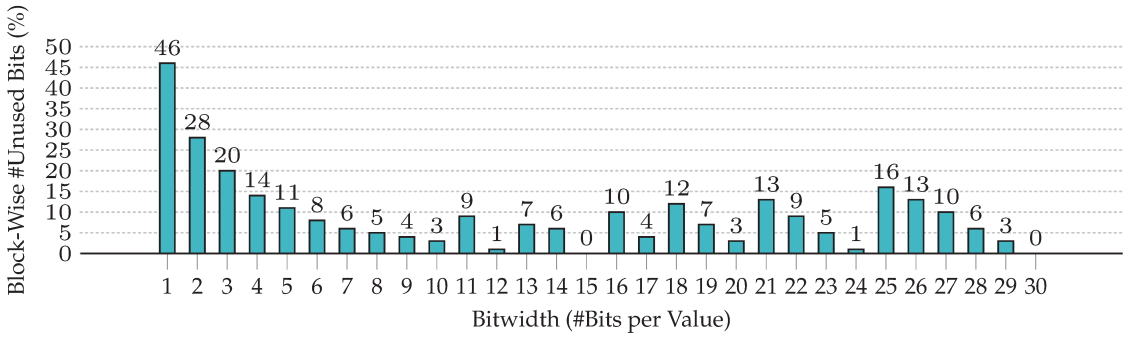

Figure 4.6: Analysis in terms of #Unused bits (%) in a *Block* After Packing for *Block* Size 128.

whereby a *block* is occupied with 128 integer values. In Figure 4.3, input and output values are shown in hexadecimal format, whereby the bitwidth of each value after *bitpacking* is 5-bit. The most significant 8-bit of the output words contains the bitwidth of the largest value per *block*, and the rest 120-bit are used for packing values (see Figure 4.3). It seems redundant to append the bitwidth of the largest value in each output word per *block*, but it is necessary for decompression. In Figure 4.3, after packing each output word contains 24 values as the largest value bitwidth per *block* is 5-bit. Figure 4.4 and Figure 4.5 illustrate the possible number of values per output word for packing, and the possible number of output words per *block* for a given largest value bitwidth ranging from 1 to 30, respectively. The number of output words per *block* depends on the *block* size and the largest value bitwidth. For instance, the largest value bitwidth of 5-bit in a given *block* contains 128 integer values can be packed into $\lceil \frac{128}{24} \rceil = 6$ output words, where each output word contains 24 integer values (see Figure 4.5).

It is worth noting that such implementation is not able to utilize every possible bit per output word to pack values due to a fixed length of encoding values as well as a fixed *block* size. For instance, in Figure 4.3, the first 5 output words in each *block* contain 24 values, but the last output word contains the remaining value which is 8. In this case, there are 11% unused bits per *block* (see Figure 4.6). Thus, the implementation of *bitpacking* requires attention to the following three aspects, i) the number of values per output word (see Figure 4.4), ii) the number of output words per *block* (see Figure 4.5), iii) the number of unused bits after packing in a *block* (see Figure 4.6). However, despite having some overhead in storing the data after packing, this method simplifies the hardware implementation. As a consequence, it reduces FPGA resource consumption. Furthermore, it eases the implementation of the decompression technique. Although we do not focus on decompression in this work as most queries can be processed directly on compressed data. But we keep decompression implementation flexibility so that we can extend this work in the future for unusual query cases that can not be processed on compressed data.

Based on this, Figure 4.7 illustrates the proposed pipeline-based FPGA accelerator for *BP* called *CBP* (*Custom-made BitPacking*), whereby the width of each input/output word is 128-bit. In this case, a *block* is occupied by 128 integer values, hence the *block* size is 128. The *CBP* works as follows:

① Read a 128-bit input word per clock cycle, whereby each input word contains four 32-bit integer values.
② Store the input word into the buffer. In parallel, to detect the bitwidth of the largest value, perform bit-wise *OR*-operations between input words per *block* to create a combined word. Afterward, determine the width of the combined word by finding the left-most bit of which value is 1. This operation is done by using predefined mask registers to achieve a constant one clock cycle latency.

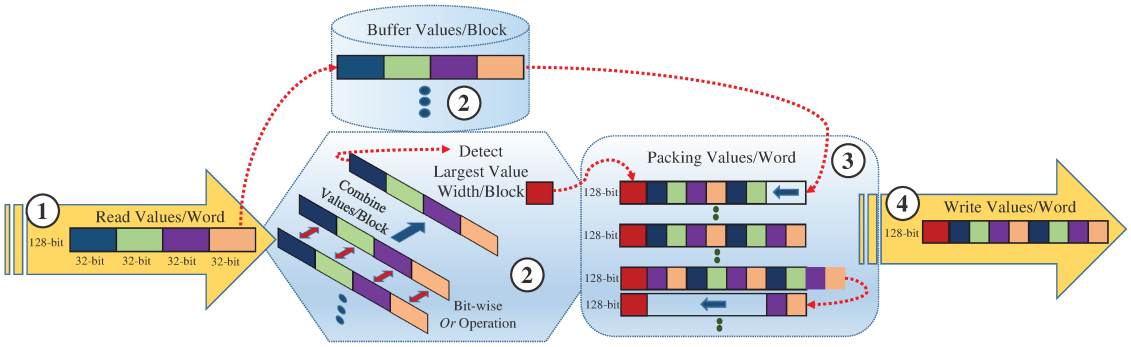

Figure 4.7: *CBP* Overview—Flows for Offloading Values per Pipeline Stage (figure taken from [LNH+19]).

③ Pack buffer values into the output words, while each buffer value is compressed with the largest value bitwidth per *block* by performing bit-wise *Right-Shift* operation. During packing, the most significant 8-bits of each 128-bit output word contains the bitwidth of the largest value per *block* and the remaining 120-bit are used for packing values.
④ Write the 128-bit output word, while one output word is fully packed with compressed values.

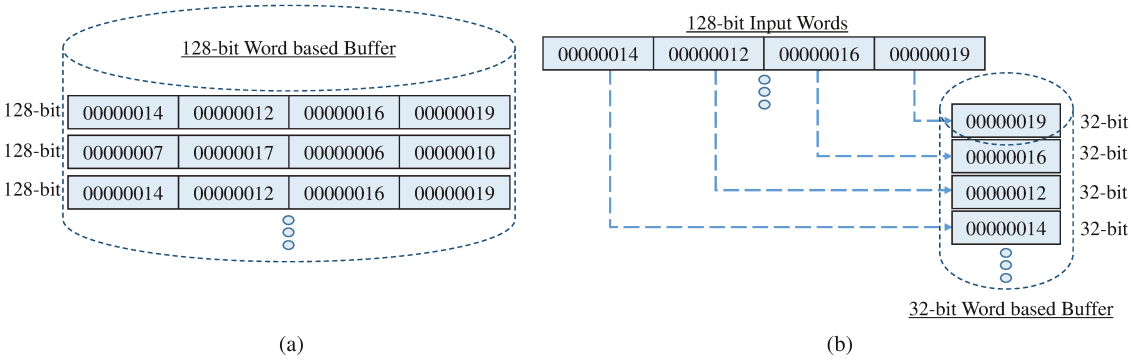

Figure 4.8: (a) 4-Value Based Buffer Words, (b) 1-Value Based Buffer Words.

In the implementation, we can use either 128-bit or 32-bitwidth based buffer words. A 128-bit width-based buffer stores 4-value per word as illustrated in Figure 4.8(a). A 32-bit width-based buffer stores 1-value per word in little-endian ordering format, whereby the least significant value per input word is stored first as shown in Figure 4.8(b).

Although in *CBP* the input and output words width are 128-bit, but for packing with the maximum possible number of values per word for a given largest value bitwidth as mentioned in Figure 4.4, the buffering word width need to be 32-bit instead of 128-bit containing a 1-value instead of a 4-value. The reason is that packing by a 128-bit word based buffer creates a misalignment problem for some cases, because the number of values per output word is not always divisible by 4 as mentioned in Figure 4.4.

For instance, for the largest value bitwidth 17-bit, the number of values per word for packing is 7 which is not divisible by 4. In such cases, a 1-value per word buffer is required, and every cycle it packs one value, which increases the latency exponentially. For example, as illustrated in Figure 4.9(a), for the largest value bitwidth 5-bit, it takes 24 cycles to finish packing a word by using 1-value per word buffer. On the contrary, using 4-value per word buffer takes only 6 cycles to finish packing a word (see Figure 4.9(b)). In

Figure 4.8 and Figure 4.9, the input and output words values are shown in hexadecimal format. Thus, to reduce the latency as well as to avoid misalignment problems, we prepared the *CBP* using a 4-value buffer concept as illustrated in Figure 4.8(a) with some modification in the number of values packed per output words for some of the largest value bitwidth cases.

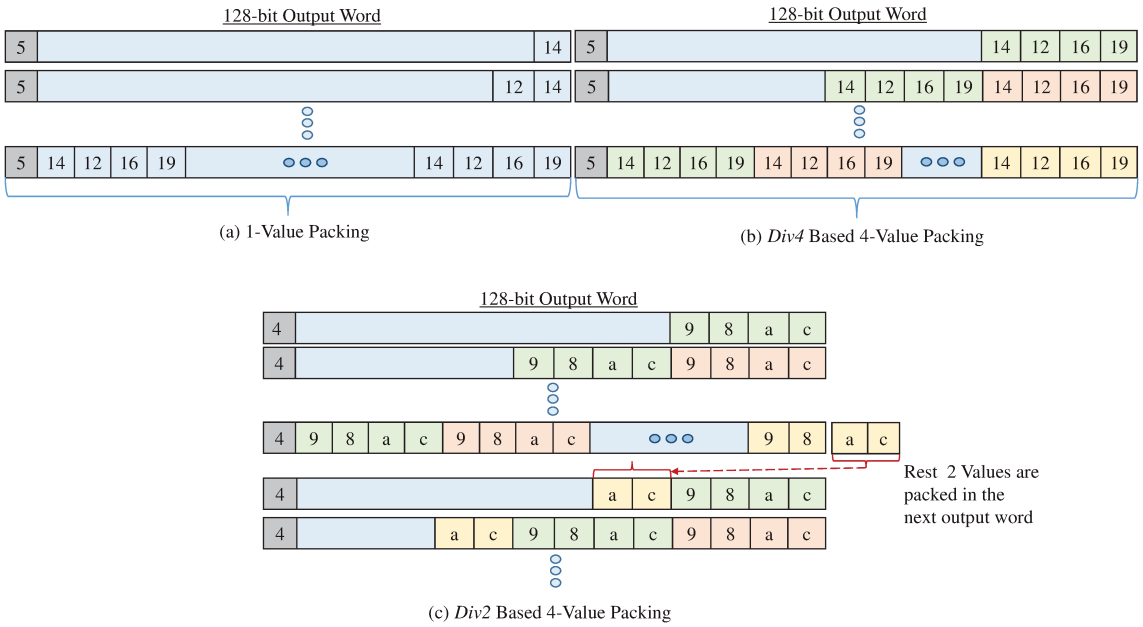

Figure 4.9: (a) 1-Value Based Buffer Words Packing, (b) 4-Value Based Buffer Words Packing using `Div4`, (c) 4-Value Based Buffer Words Packing using `Div2`.

Table 4.1: Iteration of Packing Values.

#Clock Cycles	*#Values Packing using Div4*	*#Values Packing using Div2*
n^{th}	4	4
$(n+1)^{th}$	$4+4$	$4+4$
$(n+m)^{th}$	$4+4+...+4$	$4+4+...+2$
$(n+m+1)^{th}$	4	$2+4$
$(n+m+2)^{th}$	$4+4$	$2+4+4$
$(n+m+p)^{th}$	$4+4+...+4$	$2+4+4+...+4$

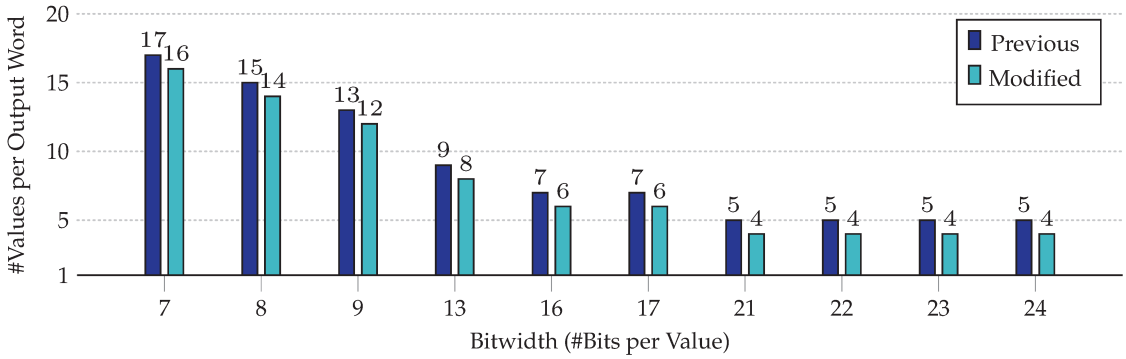

Figure 4.10: Rearrangement of the #Values per Output Word for Some Specific Largest Value Bitwidth Cases.

Thus, we categorized the number of values packed per output word in two parts, (i) `Div4`: the number of values packing per output word is divisible by 4, (ii) `Div2`: the number of values packing per output word is divisible by 2. As a result, we rearranged the number of values packed per output word for the largest value bitwidth of 7-, 8-, 9-, 13-, 16-, 17-, 21-, 22-, 23- and 24-bit which are neither divisible by 4 nor by 2. It is done by reassigning the nearest smaller number which is either divisible by 4 or by 2 (see Figure 4.10). For instance, the number of values packed per output word for the largest value bitwidth of 7-bit is $\left\lfloor \frac{120}{7} \right\rfloor = 17$, which is neither divisible by 4 nor by 2. The nearest smaller number of 17 which is either divisible by 4 or by 2 is 16. Thus, the new number of values packed per output word for this example is 16.

The number of values packing per iteration for `Div4` and `Div2` packing categories is illustrated in Table 4.1. For instance, in the n^{th} clock cycle both packing categories start packing with a new 4-value buffer word. In the $(n+m)^{th}$ clock cycle packing one output word is finished in both cases. In this clock cycle, `Div4` category packed a complete 4-value buffer word, but `Div2` category packed only 2-value of the current buffer word, and keeps the rest 2-value of the current buffer word for the next iteration packing. Hence, in the next $(n+m+1)^{th}$ clock cycle, `Div4` category start with a new 4-value buffer word for packing. On the other side, `Div2` category starts packing with the last cycle leftover 2-value including a new 4-value buffer word. Therefore, in every clock cycle, a new 4-value buffer word is packed in `Div4` category packing. However, this scenario is not similar to `Div2` category packing. In `Div2`, in some iterations, while one-word packing is finished, there are two values leftover in every alternate output word which is packed in the next output word, and it continues packing as described in Figure 4.9(c). In Figure 4.9(c), the largest value bitwidth for a given block is 4-bit, and the number of values per output word for the bitwidth of 4-bit is 30 which is divisible by 2 not by 4. Thus, a bitwidth of 4-bit satisfies the criteria of `Div2` packing.

Logical-Level Compression

In this section, we describe our implementation detail of FPGA-aware logical-level compression algorithms, and we consider three frequently used logical-level algorithms: *Delta*, *Frame of Reference (FOR)*, *Run Length Encoding (RLE)*.

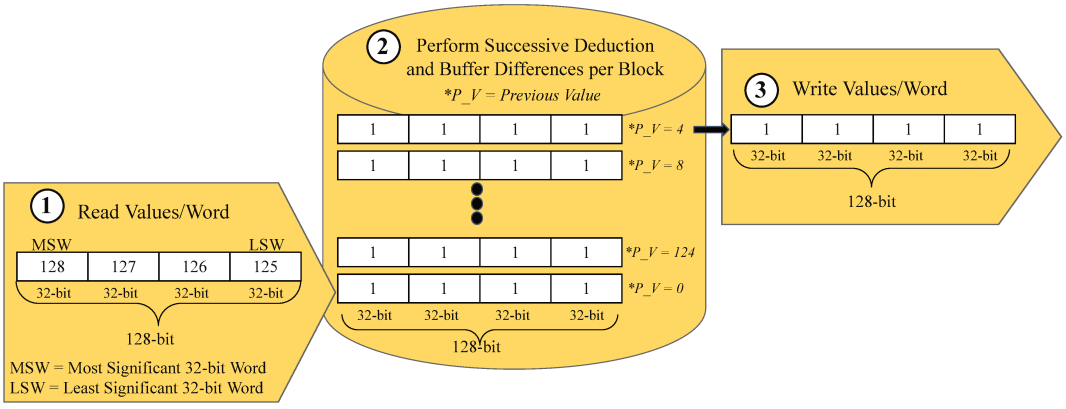

Figure 4.11: *CDelta* Overview—Flows for Offloading Values per Pipeline Stage with Example.

Delta. *Delta* represents each value as the difference to its predecessor value. In other words, *Delta* performs the successive deduction between consecutive values to reduce values. The first values per *block* are required to store for decompression purposes. Usually, *Delta* compression algorithm is suitable for sorted data, so that it can avoid generating the negative integer values. Figure 4.11 illustrates the proposed pipeline-based FPGA accelerator for *Delta* called *CDelta* (*Custom-made Delta*) in an exemplary way. The input and output words size in the implementation are 128-bit like *CBP*. Hence, input and output words contain four 32-bit integer values. However, output words contain four reduced 32-bit integer values. The pipeline flows of *CDelta* are as follows:

① Read a 128-bit input word per clock cycle, whereby each input word contains four 32-bit integer values. For instance, a block is occupied with 128 sorted integer values (i.e., 1, 2, 3, ... , 126, 127, 128). So, there are 32 input words in a block as each input word contains four 32-bit integer values. Hence, the last input word contains ({128}, {127}, {126}, {125}) integer values (see ① in Figure 4.11).
② Perform successive deduction between the consecutive values of a word, and buffer the reduced values per clock cycle. The buffer size depends on the size of a *block*. In our implementation, each *block* is occupied by 128 integer values. Hence, the buffer is 128-bit wide and the depth of buffer is 32. In an input word, values are stored in descending order. Each cycle, the most significant 32-bit word of the current input word is stored in a 32-bit register called *P_V* (*Previous Value*) to maintain the consecutive deduction constraint for the least significant 32-bit word of the next input word. For instance, a block is occupied with sorted 128 integer values (i.e., 1, 2, 3, ... , 126, 127, 128). There are 32 input words per *block*, whereby in an input word, four integer values are stored in descending order. Such as the most and the least significant 32-bit of the last input word of the *block* contain 128 and 125 integer values, respectively. Hence, the last input word of the *block* contains 128, 127, 126, 125 integer values (see ① in Figure 4.11). In this case, the value of *P_V* is 124, as it was the most significant 32-bit word of the previous input word. So, after performing successive deduction ({128-127}, {127-126}, {126-125}, {125-124}) the differences ({1}, {1}, {1}, {1}) are buffered. In this cycle, the *P_V* is updated with zero, as all values of a block are reduced (see ② in Figure 4.11).
③ Write each word of the buffer to the 128-bit output word per clock cycle, while the buffer of pipeline stage ② is full.

It is noteworthy to mention that pipeline stages ① and ② have one clock cycle latency. The pipeline stage ③ starts writing the buffer words to the output words, while the buffer is full. It defines the stage ③ is required to wait until the buffer is not full, and stage ② is required to wait until the buffer is not empty. It extends the latency. Moreover, value overwrite problem may occur unless these stages are not waiting for each other. Hence, to overcome such problems we keep two buffers (namely *buffer1* and *buffer2*) of same size in stage ②. As a consequence, while stage ③ is writing from *buffer1*, stage ② is buffering the reduced values in *buffer2*, and vice versa. In this way, we overcome the latency and value overwrite problems.

Frame of Reference (FOR). *FOR* represents each integer value as the difference to a certain given reference value. More precisely, *FOR* requires two steps to be performed: i) detect the smallest values per *block* as the reference value, ii) subtract the values of a *block* to the smallest value. The smallest values per *block* are required to store for decompression purposes. *FOR* compression algorithm is perfect for both sorted and unsorted data, as there is no chance to generate negative integer values. Figure 4.12 illustrates the proposed pipeline-based FPGA accelerator for *FOR* called *CFOR* (*Custom-made FOR*) through an example. The pattern of input and output words are similar to *CDelta*. Below, the pipeline flows of *CFOR* are depicted:

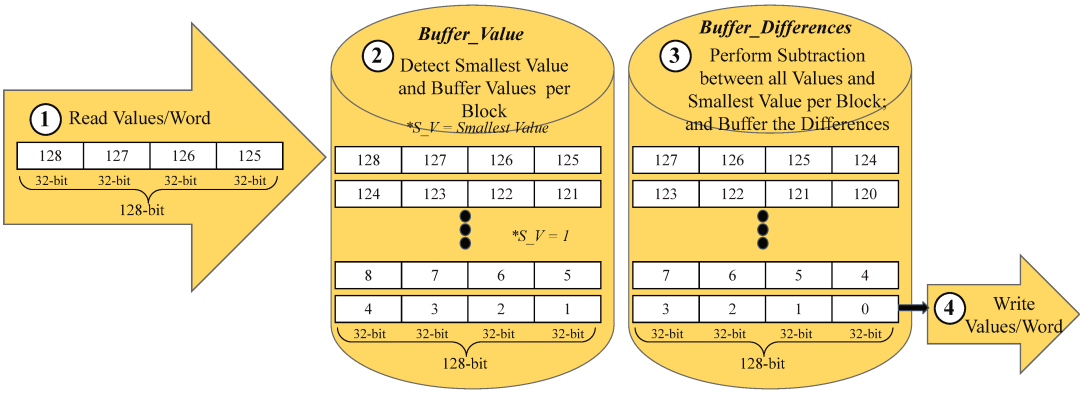

Figure 4.12: *CFOR* Overview—Flows for Offloading Values per Pipeline Stage with Example.

① Read a 128-bit input word per clock cycle, whereby each input word contains four 32-bit integer values. For instance, a *block* is occupied with 128 integer values (i.e., 1, 2, 3, ... , 126, 127, 128). So, there are 32 input words in a *block* as each input word contains four 32-bit integer values. Hence, the last input word contains ({128}, {127}, {126}, {125}) integer values (see ① in Figure 4.12).

② Detect the smallest value of a *block*, and store the smallest value in a 32-bit register called *S_V*(*Smallest Value*). Store the input words of a *block* in a buffer called *Buffer_Value*. The buffer pattern is similar to *CDelta*. For instance, a block is occupied with 128 integer values (i.e., 1, 2, 3, ... , 126, 127, 128). Each cycle is buffering an input word having four integer values in *Buffer_Value* as well as detecting the smallest value among these four integer values and the *S_V* register. As a consequence, each cycle it is updating the *S_V* register with the smallest value. Once all the input words of a *block* are buffered, the *S_V* register contains the final smallest value and in this case it is 1 (see ② in Figure 4.12).

③ Perform subtraction from all the values of *Buffer_Value* to the final smallest value of *Buffer_Value*, and store the differences in an another buffer called *Buffer_Differences*. As illustrated in Figure 4.12, once the *Buffer_Value* of the pipeline stage ② is full, the pipeline stage ③ starts subtraction from the values per word of the *Buffer_Value* to the *S_V* register in each cycle, and store the differences per word in *Buffer_Differences*. The ③ in Figure 4.12 showed the outcome of *Buffer_Differences* for the same example.

④ Write the words of *Buffer_Differences* to the 128-bit output words per clock cycle, while *Buffer_Differences* is full.

In addition, we keep two buffers in stage ② called *Buffer_Value1*, *Buffer_Value2*, two buffers in stage ③ called *Buffer_Differences1*, *Buffer_Differences2* instead of one buffer to overcome the increased latency and value overwrite problems likewise *CDelta*.

Run Length Encoding (RLE). *RLE* encodes the uninterrupted sequences of the same integer values occurrences as so-called *runs*. In *RLE*, each *run* contains its corresponding *value* and *length*. Therefore, the compressed data in *RLE* is a sequence of *{value, run-length}* pairs. Figure 4.13 illustrates the proposed pipeline-based FPGA accelerator for *RLE* called *CRLE* (*Custom-made RLE*) through an example. In *CRLE*, the maximum *run-length* for encoding the uninterrupted sequences of the same integer values occurrences in an uncompressed data depends on *block* size. However, in the proposed implementation, a *block* contains 128 integer values. The input and output words size in the implementation are 128-bit, whereby the input word contains four 32-bit integer values and the output words organization is different than the other two logical-level algorithm

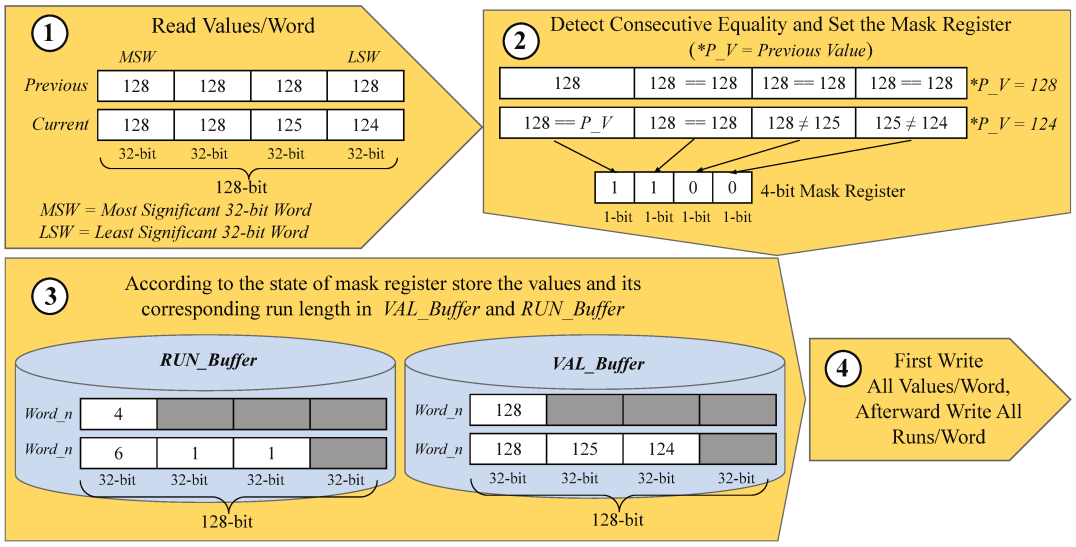

Figure 4.13: *CRLE* Overview—Flows for Offloading Values per Pipeline Stage with Example.

implementations. In this case, output words require a $(1 : 1)$ mapping between *values* and their corresponding *run-lengths*. Thus, in the implementation two buffers are used to store *values* and their corresponding *run-lengths* to maintain $(1 : 1)$ mappings. The pipeline flows for *CFOR* implementation are described below,

① Read a 128-bit input word per clock cycle, whereby each input word contains four 32-bit integer values. For instance, in ① of Figure 4.13 the *Previous* and *Current* input words contain ({128}, {128}, {128}, {128}) and ({128}, {128}, {125}, {124}) integer values, respectively.

② Perform consecutive equality checks per input words to encode the uninterrupted sequences of the same integer values occurrence, and preserve the outcome in a 4-bit mask register. The least significant 32-bit word of an input word is required to store in a register called *P_V* (*Previous Value*) to maintain the uninterrupted sequences checking, and every cycle *P_V* is required to update. The example in ② of Figure 4.13, the values of *Current* input word ({128}, {128}, {125}, {124}) are checked for equality consecutively, and the results are kept in the corresponding bit position of mask register. In this case, the *P_V* is 128. Thus, the first two equality conditions are satisfied, and remaining two conditions are not satisfied. Hence, the 4-bit mask register value is $(\{1\}, \{1\}, \{0\}, \{0\})$. In this cycle, the *P_V* is updated to 124 as the value of the least significant word of *Current* input word is 124 (see ② in Figure 4.13).

③ This pipeline stage maintains two buffers to store *values* and its corresponding *run-lengths* called *VAL_Buffer* and *RUN_Buffer*, respectively. These buffers update according to the state of mask register. These buffers are 128-bit wide and the depth is 32. Hence, both buffers can store four *values* and its corresponding four *run-lengths* per words. In both buffers, *values* and *run-lengths* are stored from left-most to right-least significant word order. The count of *run-lengths* depends on the mask register, whereby a bit position of the mask register occupied by 1 means equality, and 0 means non-equality. Such as the mask value $(\{1\}, \{1\}, \{0\}, \{0\})$ means, among the four values, two values are equal and two values are not equal. As illustrated in ① of Figure 4.13, the *Previous* and *Current* input words contain ({128}, {128}, {128}, {128}) and ({128}, {128}, {125}, {124}), respectively. The mask value of *Previous* input word is $(\{1\}, \{1\}, \{1\}, \{1\})$. This mask value defines all four integer values of this word are the same. Hence, *run-length* is 4. Thus, the most significant 32-bit word

of *Word_n* in *VAL_Buffer* stores 128, and the most significant 32-bit word of *Word_n* in *RUN_Buffer* stores 4. Now, for the *Current* input word ({128}, {128}, {125}, {124}), the mask value is $(\{1\}, \{1\}, \{0\}, \{0\})$. That means the first two values are equal to the previous value, and last two values are not equal to each other. Thus, the most significant 32-bit word of *Word_n* in *RUN_Buffer* increases to 6 (4+2), and the next two 32-bit words are occupied with 1 and 1, respectively. Similarly, the most significant 32-bit word of *Word_n* in *VAL_Buffer* is already filled up with 128, and the next two 32-bit words are occupied with 125 and 124, respectively (see ③ in Figure 4.13). In this way, $(1 : 1)$ mapping between *values* and *run-lengths* is maintained through *VAL_Buffer* and *RUN_Buffer*. In Figure 4.13, the empty 32-bit words of the both buffers in ③ are highlighted with dark color. Once four 32-bit words of a 128-bit buffer word is full then it starts buffering in the next word.

④ Write the words of *VAL_Buffer* and *RUN_Buffer* one after another to the 128-bit output words per clock cycle, while *VAL_Buffer* and *RUN_Buffer* are full.

Additionally, we keep four buffers in stage ③ called *VAL_Buffer1*, *VAL_Buffer2*, *RUN_Buffer1*, *RUN_Buffer2* instead of two for similar reasons like *CDelta* and *CFOR*.

4.2.3 Lightweight Integer Compression System Implementation

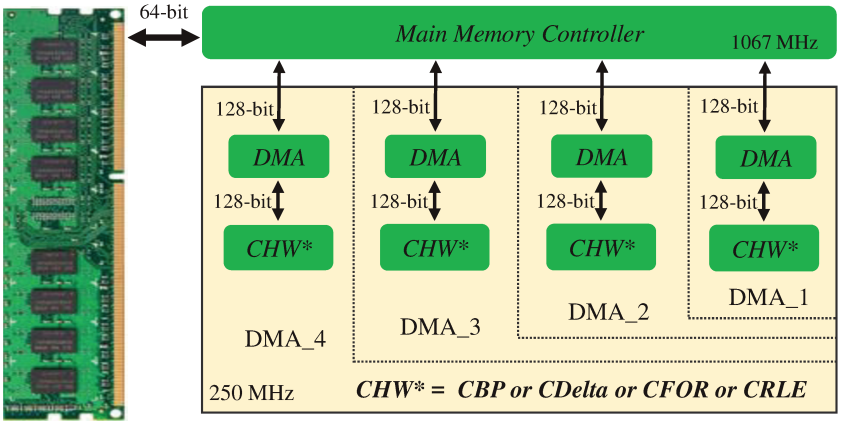

Figure 4.14: Lightweight Integer Compression System based on Different Custom-made Compression Hardware using Different Design Templates.

After the implementation of custom-made compression accelerators as described in Subsection 4.2.2, we start with the development of lightweight integer compression system based on design template `DMA_1` for each custom-made (i.e., *CBP*, *CDelta*, *CFOR*, *CRLE*) compression accelerator. Afterward, we developed multiple DMA-based design templates `DMA_2`, `DMA_3`, and `DMA_4` oriented lightweight integer compression systems for each custom-made compression accelerator (see Figure 4.14). All these designs with single or multiple DMA work perfectly with 250MHz *PL* part frequency (see Figure 4.14). That means all these designs meet the timing with 250MHz. However, in the multiple DMA-oriented designs, we distribute data among the multiple DMA evenly. That means we virtually partitioned the dataset in main memory, and distribute them consecutively one after another DMA. For instance, compressing values using `DMA_4` design, the dataset is virtually partitioned into 4 in main memory, whereby the first partition dataset transfers to DMA1, the second partition dataset transfers to DMA2, and so on. Hence, in the multiple DMA-based designs, the virtually partitioned dataset in the main memory is evenly distributed among the multiple DMA consecutively.

4.2.4 Discussion

The proposed implementation mechanism of *CBP*, *CDelta*, *CFOR* and *CRLE* on FPGA is highly effective due to the hardware-level custom-made implementation flexibility of FPGA. At this point, it is important to mention that usually lightweight compression algorithm implementation is highly read-intensive. For example, the vectorized implementation of *FOR* in [DUH+19] reads each input data element two times from the main memory: i) for detecting the reference values, ii) for detecting the differences from the input values to the reference values. Reading the same data elements two times from the main memory affects the performance negatively. This problem is easily overcome on our FPGA based algorithm implementations through introducing the input buffer concept. In our FPGA-based implementation, each data is read from the main memory once, and while processing values it buffers the input values on block RAMs as so-called BRAMs [TW13] of FPGA in parallel for future use. By using BRAMs, we avoid the redundant reads from the main memory. Moreover, we explore four design templates for each of the proposed custom-made compression accelerators from `DMA_1` to `DMA_4`. These designs prove that proposed custom-made compression accelerators are highly adaptive to other hardware. However, the `DMA_4` requires the maximum resource of FPGA compared to other designs as it is utilizing all the available direct data channels of main memory. It is important to distribute input data evenly among the data channels. Because the main memory controller chooses to serve a particular *DMA* through a data channel in a round robin fashion as physically there is only one data channel connected between the main memory and the main memory controller (see Figure 4.14). However, it is not possible to distribute data evenly in `DMA_3` as it has an odd number of data channels, and some custom-made compression accelerator such as *CBP* compresses data elements *block*-wise, whereby each *block* contains an even number of data elements. Hence, in such case `DMA_3` may affect the performance negatively.

4.3 ADAPTIVE COMPRESSION SYSTEMS

In the previous section, we proposed custom-made physical-level as well as logical-level lightweight compression accelerators. Moreover, we implemented different template oriented hardware design architectures based on proposed custom-made lightweight compression accelerators. All these designs are compressing the whole dataset using a single compression algorithm. For instance, the `DMA_4` design of *CDelta* compression system compresses the whole dataset using *Delta* algorithm, the `DMA_4` design of *CBP* compression system compresses the whole dataset using *Bitpacking* algorithm and so on. However, to guarantee a certain level of compression throughput acceleration, the *block*-wise compression is necessary as each *block* may have different data properties. Thus, this thesis concentrates on the *block*-based lightweight integer compression system implementation, whereby different *blocks* are compressed with different custom-made compression accelerators. Therefore, in this section, we describe different types of implementations for the FPGA-aware adaptive lightweight compression system. We explore different implementation opportunities in the target CPU-FPGA system, by utilizing our custom-made physical-level and logical-level compression accelerators, with taking care of minimum resource utilization, and for the best possible compression throughput. FPGA-aware adaptive implementation is based on a two-level compression accelerator for integer values. In the first level, reducing values by custom-made logical-level lightweight compression accelerator. In the second level, logically reduced values are compressed by a custom-made physical-level lightweight compression accelerator. Thus, selecting the appropriate logical-level accelerator demands appropriate design-level specifications based on data properties. To satisfy such demand, we start our implementation with a straightforward *User-Specified Adaptive System*, and later we introduce other types of *Adaptive Systems*.

4.3.1 User-Specified Adaptive System

The *User-Specified Adaptive System* is a straightforward design, whereby during runtime a user can select a lightweight compression algorithm to compress integer values. A user has the flexibility to compress integer values, either logically at value-level as well as physically at bit-level, or physically at bit-level only without any logical-level compression. Therefore, we prepared a custom-made scheduler called *CScheduler* to select a user-specified compression algorithm. Figure 4.15 illustrates a single DMA based *User-Specified Adaptive System*. In the *User-Specified Adaptive System*, the uncompressed data including the choice of user-specified compression algorithm are passed through DMA from the main memory controller to the *CScheduler*. *CScheduler* is responsible to choose a specific custom-made compression accelerator as per user choice. Therefore, *CScheduler* has data channel connections to all the logical-level compression accelerators (*CDelta*, *CFOR*, *CRLE*). Conceptually, *CScheduler* can transfer data *block*-wise into different custom-made compression accelerators as per user-specified *block*-wise choices. However, we do not consider such scenario in this implementation.

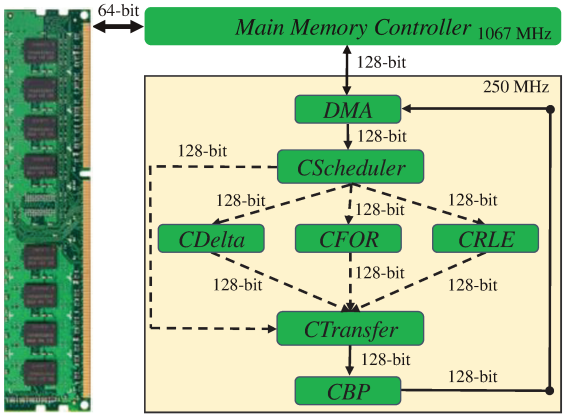

Figure 4.15: *User-Specified Adaptive System* for Lightweight Integer Data Compression.

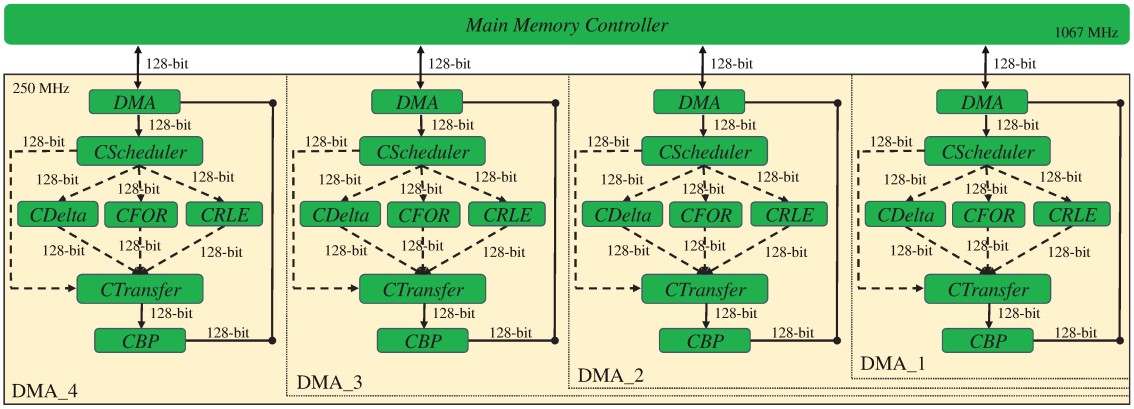

Figure 4.16: Different #DMA Based Hardware Designs for *User-Specified Adaptive System*.

Moreover, during compression *CBP* accelerator is always required as it is reducing data at bit-level. That means all logical-level compression accelerators including *CScheduler* are required to access this *CBP*. But *CBP* is designed with a single pair of input-output ports. Thus, we prepared a custom-made hardware called *CTransfer*. *CTransfer* is responsible to pass data from the active data channel of either *CScheduler* or *CDelta* or *CFOR* or *CRLE* to the input port of *CBP*. Basically, *CTransfer* is working like a $(4:1)$ input switch for *CBP*

(see Figure 4.15). Moreover, there is a direct data channel that comes from the *CScheduler* to the *CTransfer* as the *User-Specified Adaptive System* also supports compressing values only using the physical-level compression accelerator *CBP* without cascading with any logical-level compression accelerator (see Figure 4.15).

Afterward, we developed single and multiple DMA-based hardware designs of the *User-Specified Adaptive System* as `DMA_1`, `DMA_2`, `DMA_3`, `DMA_4`, where each DMA is accessing the main memory controller via an independent data channel (see Figure 4.16). Hence, we replicated the *User-Specified Adaptive System* up to 4 times as the number of available data channels in the main memory controller of the targeted system is 4. Likewise, the *CBP* designs, the single and multiple DMA-based *User-Specified Adaptive Systems* meet the timing with 250MHz (see Figure 4.16).

4.3.2 HW-Specified Adaptive Systems

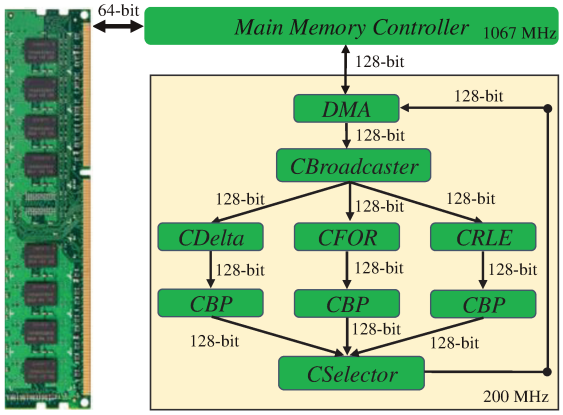

Figure 4.17: *HW-Specified Adaptive System* for Lightweight Integer Data Compression.

The *User-Specified Adaptive System* is a naive implementation as it is a user-dependent platform, which is not effective for real-life purposes. *User-Specified Adaptive System* is not autonomous as the user is responsible to choose a compression algorithm. The goal of this thesis is to prepare an autonomous system for lightweight integer data compression, whereby the hardware is responsible for choosing the appropriate algorithm as per data properties. Thus, we introduced the *HW-Specified Adaptive System*, whereby the hardware makes the decision for choosing the appropriate algorithm as per data properties.

Damme et al. [DHHL17] already mentioned that all lightweight integer compression algorithms are highly data properties dependent. Therefore, an appropriate algorithm selection requires in-depth data properties analysis. Preparing a compression system which first analyzes the properties of data to select an appropriate algorithm for compression and later compresses the same data with its corresponding algorithm selection increases the read overhead drastically. To avoid such constraint, we implement the compression system in a speculative way, where a dataset compresses through system available different compression algorithms in parallel, and later a sophisticated custom-made hardware selects the compressed values from the appropriate algorithm.

As illustrated in Figure 4.17, the *HW-Specified Adaptive System* is based on the post-order decision making to choose an appropriate compression algorithm as per data properties instead of making the pre-order decision. In the implementation, a custom-made broadcaster called *CBroadcaster* is used to broadcast uncompressed data from a DMA to all the logical-level compression accelerators (*CDelta*, *CFOR*, *CRLE*). Each logical-level

compression accelerator has its own *CBP* accelerator to compress values at the bit-level. Finally, three different algorithm based compressed values go to a custom-made hardware called *CSelector*. *CSelector* is responsible to identify the most compressed data among the three different compression algorithm based compressed data. In this case, the mechanism of *CSelector* is very simple. *CSelector* works on the basis of first come first out. This defines *CSelector* output that algorithm-based compressed data which came as input to it at first. For example, for a given uncompressed dataset, if the (*CDelta* + *CBP*) accelerator generates faster compressed values-based output words than the other compression accelerators, then *CSelector* accepts the (*CDelta* + *CBP*) accelerator generated output words as input words, and ignores the other compression accelerator generated inputs. In this adaptive system implementation, we do not consider to compress data using *CBP* accelerator only. Moreover, this adaptive design meets the timing with 200MHz instead of 250MHz (see Figure 4.17).

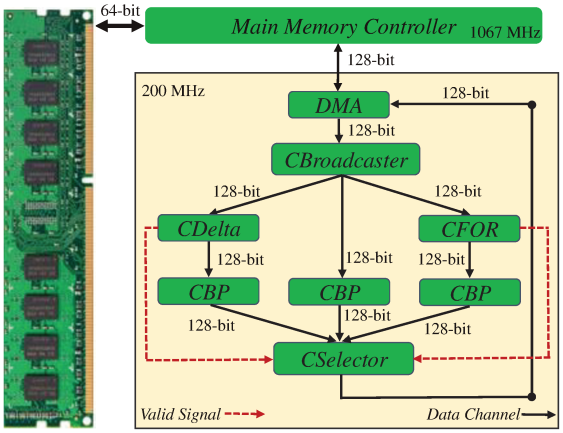

Figure 4.18: *HW-Specified Pre-BitPacking Adaptive System* for Lightweight Integer Data Compression.

It was mentioned earlier that there is no single best lightweight compression algorithms [DHHL17]. All lightweight compression algorithms are data properties dependent. For instance, *Delta* algorithm is suitable for sorted data. However, the *HW-Specified Adaptive System* is compressing the whole uncompressed dataset using one of the logical-level compression accelerators that produces faster-compressed output words among the others, which may not always provide the most optimum compression throughput. In the real world, a single dataset consists of different properties. Thus, if a single dataset is partitioned into a fixed number of *blocks* then different types of properties can be distinguished. As a consequence, in this *block*-based partitioned dataset, we can apply different compression algorithms in different *blocks* as per *block*-based data properties which may provide optimum compression throughput. Therefore, compressing data *block*-wise has a possibility to accelerate compression throughput. Thus, instead of compressing the whole dataset using a logical-level compression algorithm, compressing data *block*-based using different logical-level compression algorithms as per *block*-based data properties is more ideal. Therefore, we modified the *HW-Specified Adaptive System* and developed the adaptive design for lightweight integer data compression called *HW-Specified Pre-BitPacking Adaptive System* as depicted in Figure 4.18, whereby values are compressed *block*-wise. As a consequence, the *CSelector*, *CDelta* and *CFOR* are modified, whereby *CDelta* and *CFOR* are generating 1-bit valid signals to *CSelector*. These valid signals define whether a particular *block* is suitable for a particular logical-level compression algorithm or not. In *CDelta*, the value of the valid signal depends on whether a particular *block* of integer values is sorted or not. If the values of a *block* are sorted then the valid signal is set to active-high signal, otherwise it is set to active-low signal. In *CFOR*, the value of the valid signal depends on the value of the smallest value

per *block*. If the value of the smallest value per *block* is larger than 1024 then it is set to active-high, otherwise active-low. In *CSelector*, at first it is checked the *CDelta* produces valid signal, and then it is checked *CFOR* produces signal, per *block*. The checking of signals in *CSelector* is based on the following three priority conditions:

Priority 1: If the *CDelta* produced valid signal is active-high, then *CSelector* chooses the (*CDelta* + *CBP*) generated compressed values.

Priority 2: If the *CDelta* produced valid signal is active-low, then *CSelector* checks the *CFOR* produced valid signal. If the *CFOR* produced valid signal is active-high, then *CSelector* chooses the (*CFOR* + *CBP*) generated compressed values.

Priority 3: If the *CFOR* produced valid signal is active-low, then *CSelector* chooses only the *CBP* generated compressed values.

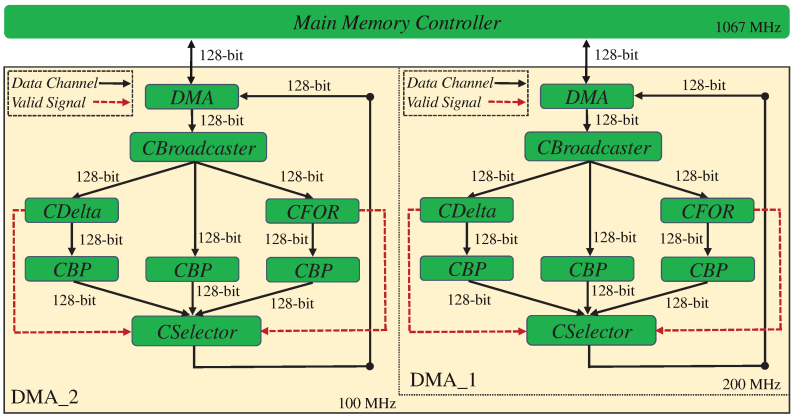

Figure 4.19: Different #DMA Based Hardware Designs for *HW-Specified Pre-BitPacking Adaptive System*.

Moreover, in the *HW-Specified Pre-BitPacking Adaptive System*, it is important that all the logical-level algorithms, compress values without changing the number of *blocks* in a dataset. However, in *RLE* the number of *blocks* is not constant like *Delta* or *FOR*. In the worst-case scenario, in a given uncompressed dataset if there are no uninterrupted sequences of the same integer values occurrences, then *RLE* may double the number of *blocks*. On the contrary, in the best-case scenario, for a given uncompressed dataset, if all the values are the same integer values then it may have only one *block*. As a consequence, a $(1:1)$ *block* mapping is not possible for *CRLE* like the other compression accelerators. Hence, in *HW-Specified Pre-BitPacking Adaptive System*, we do not consider *RLE* algorithm. Although the one DMA-based *HW-Specified Pre-BitPacking Adaptive System* meets the timing at 200MHz, it goes down to 100MHz for two DMA-based design (see Figure 4.19). Hence, we do not go for further DMA-based design implementation for *HW-Specified Pre-BitPacking Adaptive System*.

However, *HW-Specified Pre-BitPacking Adaptive System* is using *CBP* accelerator three times, that increases the hardware resource utilization. To reduce hardware resource utilization, we modified the *HW-Specified Pre-BitPacking Adaptive System*, and prepared another design for lightweight integer data compression called *HW-Specified Post-BitPacking Adaptive System* (see Figure 4.20). In the *HW-Specified Post-BitPacking Adaptive System*, we placed the *CSelector* hardware in between the logical-level and physical-level compression accelerator. Hence, the logically compressed values per *block* of *CDelta* and *CFOR* go to *CSelector* directly, and *CSelector* sends the selected *block* of values to the *CBP*, whereby the selection of a *block* in *CSelector* is based on the values of valid signals. Thus, in *HW-Specified Post-BitPacking Adaptive Design*, there is only one *CBP* accelerator. In addition, there is a direct data channel connected between *CBroadcaster* and *CSelector*, so that *CSelector* can have the uncompressed *block* of data for *CBP* to

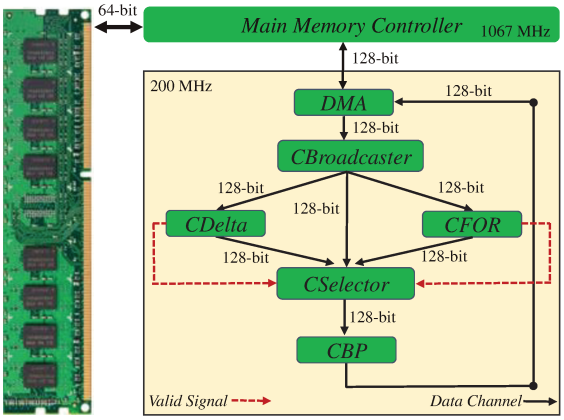

Figure 4.20: *HW-Specified Post-BitPacking Adaptive System* for Lightweight Integer Data Compression.

satisfy the priority condition 3 of *CSelector*. In this case, the one and two DMA-based *HW-Specified Post-BitPacking Adaptive System* meets the timing at 200MHz, it goes down to 100MHz for the three DMA-based design (see Figure 4.21). Hence, we do not go for four DMA-based design implementation for the *HW-Specified Post-BitPacking Adaptive System*.

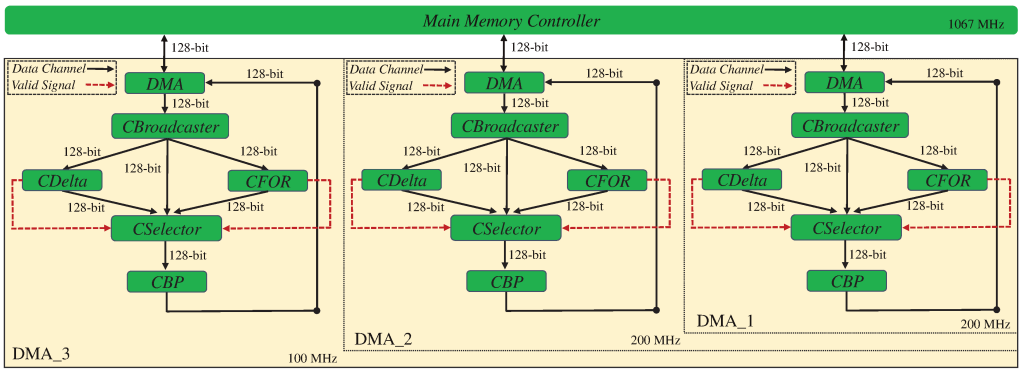

Figure 4.21: Different #DMA Based Hardware Designs for *HW-Specified Post-BitPacking Adaptive System*.

4.4 EXPERIMENTAL EVALUATION

In this section, we experimentally analyze the behavior regarding *compression throughput(GB/s)*, *resource consumption(%)* and *power dissipation(W)* based on a combination of three-dimensional instances on the targeted hybrid CPU-FPGA system. More precisely, most of the evaluations are based on Zynq UltraScale+™ hybrid CPU-FPGA system. The combination of three-dimensional instances shown in Figure 4.22. The three-dimensional instances are based on data characteristics, compression system implementation categories, and data channels. That means we have evaluated different data characteristics oriented synthetic datasets on different categories of compression systems implementation based on the different number of data channels. We have categorized the compression system implementation based on lightweight integer algorithms into four types: physical-level, logical-level, cascaded, and adaptive. Each of these implementation categories has sub-category implementations based on a different number of available data channels of the main memory controller of our targeted CPU-FPGA system. Each data channel is accessing a DMA to take control over the CPU on main memory, and transfer data between main memory and *PL* part custom-made compression system. Thus, in the remainder of this section, we call such implementations `DMA_1`, `DMA_2`, `DMA_3`, `DMA_4`, whereby `DMA_1`, `DMA_2`, `DMA_3`, `DMA_4` systems are accessing 1, 2, 3, 4 data channels of the main memory controller, respectively. Therefore, this section is subdivided as follows,

- In Subsection 4.4.1, we define different data characteristics oriented synthetic datasets.
- In Subsection 4.4.2, we evaluate the physical-level compression systems based on different numbers of DMAs for the synthetic datasets.
- In Subsection 4.4.3, we evaluate the logical-level compression systems based on different numbers of DMAs for the synthetic datasets.
- In Subsection 4.4.4, we evaluate the cascaded compression systems based on different numbers of DMAs for the synthetic datasets.
- In Subsection 4.4.5, we evaluate different types of proposed adaptive compression systems based on different numbers of DMAs for the synthetic datasets.

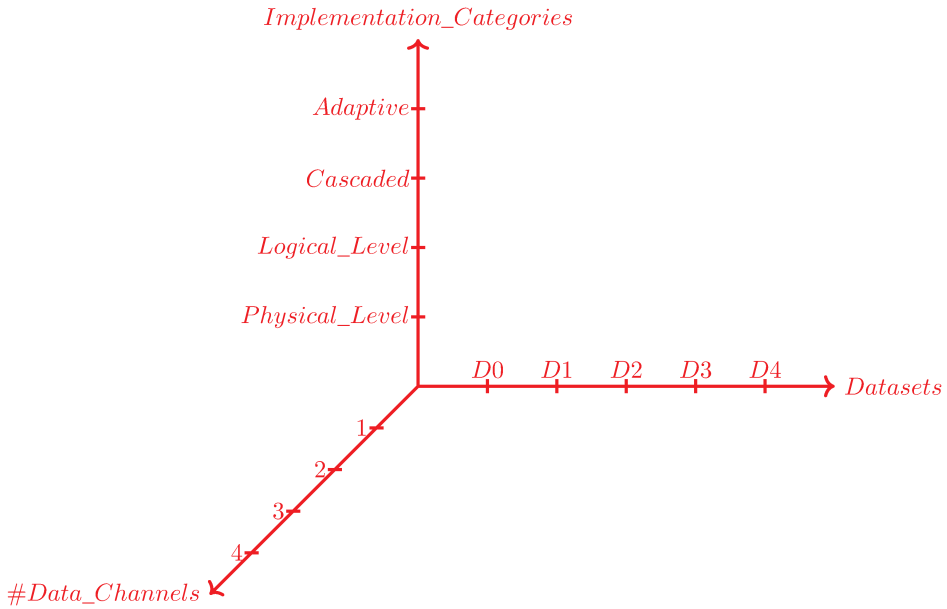

Figure 4.22: The Combination of Three-Dimensional Instances for Experimental Analysis.

4.4.1 Data Properties Definition

We have prepared different characteristics of synthetic datasets for our exhaustive experimental evaluations to observe the influence of various data properties over the proposed different compression designs. We categorize synthetic datasets into five. The data features description of the different datasets is shown in Table 4.2. The first category dataset called *D0* is based on the exact bitwidth of integer values. However, in Subsection 4.2.2, we have mentioned that in the implementation of *bitpacking* the most significant 8-bit of each output word is preserved for keeping the bitwidth of the largest value of a *block* and the remaining 120-bit is used for keeping the compressed integer values, whereby each output word is 128-bit wide. Thus, in our implementation, a maximum of 30-bit wide integer values can be processed. Therefore, we have prepared 60 exact bitwidth based datasets from 1-bit to 30-bit. Among 60 datasets 30 datasets are sorted and 30 are unsorted called *D0_sorted* and *D0_unsorted*, respectively (see Table 4.2).

The next category dataset is called *D1*, which consists of sorted data, and the data is distributed by different numbers of distinct consecutive integer values. This kind of dataset is suitable for *RLE* algorithm. For instance, a *D1* category dataset called Dist_15 defines that the dataset is distributed by 15-bit bitwidth based distinct consecutive integer values.

Afterward, we prepared ratio-based different bitwidth oriented datasets called *D2* and *D3*, whereby *D2* dataset is distributed by 90% small and 10% big integer values, and *D3* dataset contains 50% small and 50% big integer values. In Subsection 4.4.2, we have defined the small and big integer values based on our evaluation results. In *D2* and *D3* datasets, we have considered sorted and unsorted categories. Therefore, *D2* and *D3* have the sub-categories *D2_sorted*, *D2_unsorted*, *D3_sorted* and *D3_unsorted* (see Table 4.2).

Finally, we have prepared a *block*-wise distributed sorted and unsorted category dataset called *D4*. This category dataset has mixed bitwidth based integer values. For instance, a *D4* category based dataset called Block_128 means 128 sorted, and 128 unsorted mixed bitwidth based integer values are distributed alternately. Similarly, a *D4* category based dataset called Block_512 means 512 sorted, and 512 unsorted mixed bitwidth based integer values are distributed alternately (see Table 4.2).

The size of each dataset is 128MB consists of $2^{25} = 33,554,432$ (in words: thirty-three million, five hundred fifty-four thousand, four hundred thirty-two) uncompressed 32-bit integer values.

Table 4.2: Different Categories Synthetic Datasets.

Datasets	Sorted	Data Properties
D0_sorted	Yes	common exact bitwidth based integer values
D0_unsorted	No	
D1	Yes	#distinct consecutive integer values
D2_sorted	Yes	90% small integer values and 10% big integer values
D2_unsorted	No	
D3_sorted	Yes	50% small integer values and 50% big integer values
D3_unsorted	No	
D4	Both	*block*-wise sorted and unsorted mixed bitwidth based integer values

4.4.2 Physical-Level Compression

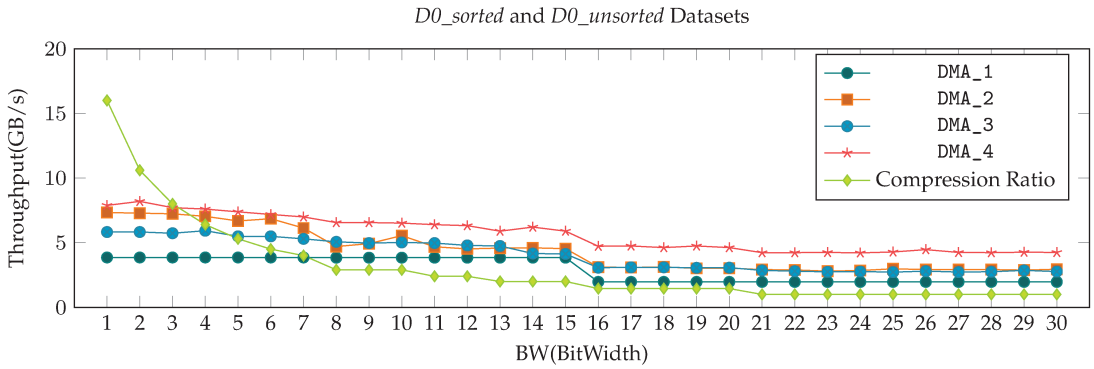

Figure 4.23: Throughput Analysis for *CBP* Compression Systems based on Different #DMAs (Average over 5 runs).

We have started our evaluation with the physical-level compression systems based on different numbers of DMAs for *D0* category datasets as illustrated in Figure 4.23. As *BitPacking* is compressing data at bit-level, it does not matter whether the data is sorted or not sorted. Therefore, same throughput effects were found on *D0_sorted* and *D0_unsorted* in all *CBP* compression systems (see Figure 4.23). In Figure 4.23, a symmetric throughput of 3.8 GB/s (for the largest value bitwidth of 1- to 15-bit) and 1.9 GB/s (for the largest value bitwidth of 16- to 30-bit) has found at `DMA_1`, whereby these are the read and read-write throughput of any custom-made AXI channel based hardware, respectively. It defines three points:

① The latency of the *CBP* hardware depends on read/write-operations not on compressing the values.
② As the bitwidth increases, the compression ratio decreases, and after bitwidth 15-bit the compression ratio is $\approx$1 (see the Compression Ratio curve of `DMA_1` in Figure 4.23).
③ As the throughput decreases above 15-bit, we have defined ($\leq$15)-bit bitwidth based integer values as small values and ($>$15)-bit bitwidth based integer values as big values.

Point number ② defines, after bitwidth of 15, the compression rate is approximately 1 as both input and output words contain almost equal number of values. The other hardware designs have achieved improved throughput compared to `DMA_1` as the values are evenly distributed among the multiple *CBPs* for parallel compression which increases the throughput, except `DMA_3` which provides mostly the same throughput as `DMA_2`. The reason is, *BitPacking* compresses values per *block* basis, whereby the *block* size is always even and divisible by the power of 2. In `DMA_3`, data for compression is not evenly distributed, as it has an odd number of data channels. However, the throughput behavior in the other designs per bitwidth is not symmetrical as `DMA_1`. The reason is, multiple DMAs interact with the main memory controller in a round robin fashion. That means while one DMA is ready to interact, the main memory controller may be busy with others, which affects the throughput [LUH+18a].

Besides throughput, the other evaluation metric is hardware resource consumption. In this case, we have considered two types of Zynq UltraScale+™ FPGAs, i) ZCU102, ii) Ultra96. Ultra96 is a smaller version FPGA board in terms of resources than ZCU102. We have chosen both FPGAs to see the effects regarding resource consumption, power dissipation, and timing (see Table 4.3). In both FPGAs, resource consumption and power

Table 4.3: Resource Utilization and *PL* Part Frequency Analysis for *CBP* Compression Systems based on Different #DMAs.

ZCU102				
#DMA	LUTs(%)	FFs(%)	Power(W)	Freq(MHz)
DMA_1	4.48	2.09	3.709	250
DMA_2	8.79	4.06	3.915	250
DMA_3	12.93	5.96	4.110	250
DMA_4	17.09	7.85	4.331	250

Ultra96				
#DMA	LUTs(%)	FFs(%)	Power(W)	Freq(MHz)
DMA_1	17.42	8.12	2.724	250
DMA_2	34.10	15.79	2.829	200
DMA_3	50.13	23.15	2.882	150
DMA_4	66.25	30.48	2.848	100

dissipation increase as the number of DMA increases. Ultra96-based implementations have less power dissipation and more resource consumption than ZCU102-based implementations. It happens because ZCU102 has four times more hardware resources in terms of transistors per logic function than Ultra96. As a result, for a specific logic function implementation on both FPGAs, on average, Ultra96 requires more hardware resources compared to ZCU102. The power dissipation analysis, especially based on the specific design resource usage on top of overall resources. Thus, as Ultra96 has less number of overall hardware resources than ZCU102, Ultra96 consumes per design less power dissipation compared to ZCU102. However, unlike ZCU102, Ultra96-based implementations do not meet the timing at 250 MHz except DMA_1. Therefore, working on Ultra96 is productive, as all designs fit on it resource-wise, and it consumes less energy than ZCU102. However, Ultra96 can not meet the timing at 250 MHz in all design cases, which reduces the throughput drastically. Moreover, optimum throughput is the main focus of this thesis. Thus, in the rest of our evaluation, we do not consider Ultra96.

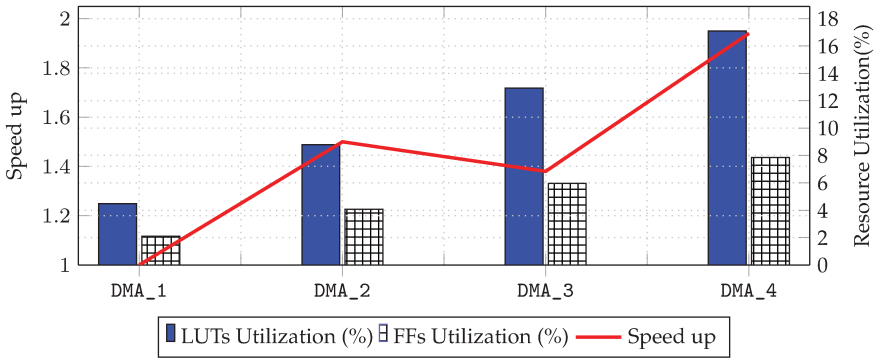

Figure 4.24: Comparison Between Speed up and Resource Utilization for *CBP* Compression Systems based on Different #DMAs.

In Figure 4.24, we show the trade-off between resource consumption and speed up of the *CBP* designs. Based on that our observations are as follows,

- The resource consumptions are linearly increasing DMA-wise.
- The speed up is linearly increasing with a break down in DMA_3 compared to DMA_2 due to non-even data distribution.
- DMA_4 speeds up to 1.9x. Hence, DMA_4 is the best design for the physical-level compression system.
- The best design of the physical-level compression system provides the maximum throughput with maximum hardware resources consumption.
- The throughput acceleration of the physical-level compression system depends on bitwidth of integer values and different #DMAs in terms of #data channels of the main memory controller.

4.4.3 Logical-Level Compression

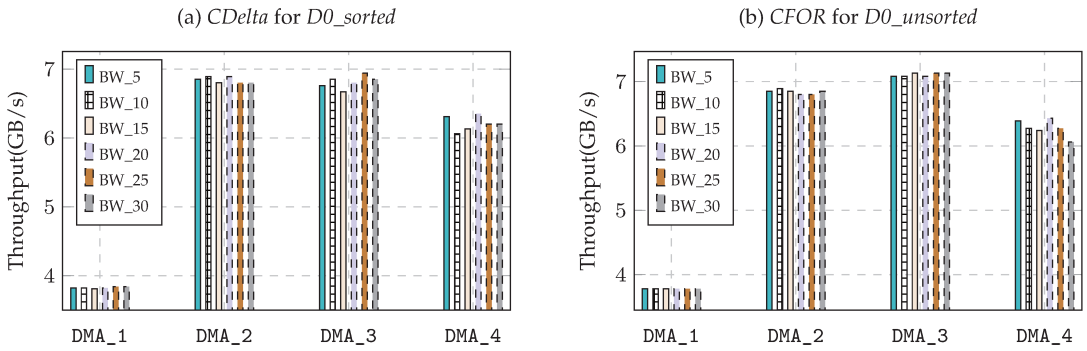

Figure 4.25: Throughput Analysis for (a) *CDelta*, (b) *CFOR* Compression Systems based on Different #DMAs (Average over 5 runs).

CDelta and *CFOR* Compression Systems

In the evaluation of the logical-level compression system, we have started with the different numbers of DMA-based *CDelta* and *CFOR* compression systems for *D0_sorted* and *D0_unsorted* category datasets as illustrated in Figure 4.25, respectively. In this case, we have considered the 5-, 10-, 15-, 20-, 25- and 30-bit bitwidth based *D0_sorted* category datasets for *CDelta* and *D0_unsorted* category datasets for *CFOR*. During evaluation, bitwidth-wise almost equal throughput was found in all design cases. The reason is, *CDelta* and *CFOR* generate an equal number of input and output words as *Delta* and *FOR* are compressing values in value-level. As a consequence, there is almost equal throughput on different bitwidth based datasets. DMA-wise throughput improvement has been achieved in both cases. However, in both cases, `DMA_2` and `DMA_3` have given almost equal throughput and, in `DMA_4` throughput goes down, as expected.

Table 4.4 shows the resource consumption for the different numbers of DMA-based *CDelta* and *CFOR* compression systems. As the logic of *Delta* and *FOR* is very simple compared to *Bitpacking*, they consume very little resource and power compared to *CBP*. However, all designs meet the timing at 250 MHz.

Table 4.4: Resource Utilization and *PL* Part Frequency Analysis for (a) *CDelta*, (b) *CFOR* Compression Systems based on Different #DMAs.

(a) *CDelta*				
#DMA	**LUTs(%)**	**FFs(%)**	**Power(W)**	**Freq(MHz)**
`DMA_1`	2.28	1.76	3.678	250
`DMA_2`	4.30	3.40	3.843	250
`DMA_3`	6.20	4.96	3.977	250
`DMA_4`	8.12	6.53	4.183	250

(b) *CFOR*				
#DMA	**LUTs(%)**	**FFs(%)**	**Power(W)**	**Freq(MHz)**
`DMA_1`	2.92	1.79	3.703	250
`DMA_2`	5.58	3.47	3.887	250
`DMA_3`	8.12	5.07	4.067	250
`DMA_4`	10.67	6.66	4.252	250

In Figure 4.26, we show the trade-off between resource consumption and speed up of the *CDelta* and *CFOR* designs. Based on that our observations are as follows,

- In both cases, the resource consumptions are linearly increasing DMA-wise.
- *CDelta* speeds up to 1.78x at `DMA_2` (see Figure 4.26(a)), whereas, *CFOR* speeds up to 1.87x at `DMA_3` (see Figure 4.26(b)).

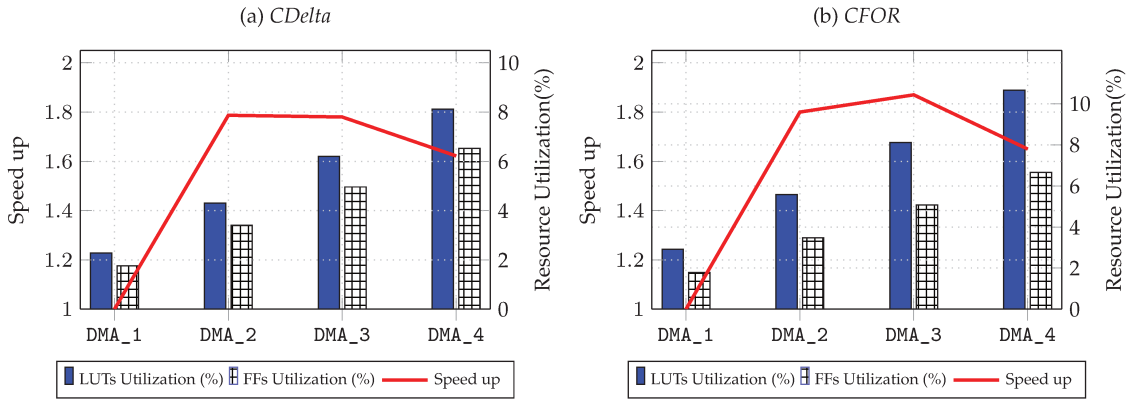

Figure 4.26: Comparison Between Speed up and Resource Utilization for (a) *CDelta*, (b) *CFOR* Compression Systems based on Different #DMAs.

- `DMA_2` is the best design for the *CDelta* compression system. On the other side, `DMA_3` is the best design for the *CFOR* compression system.
- In both cases, the best design does not consume maximum hardware resources.
- Throughput acceleration of *CDelta* and *CFOR* compression systems are bitwidth independent, but they depend on the different #DMAs.

CRLE Compression Systems

In Figure 4.27, we have considered two different categories of datasets for *CRLE* compression systems throughput analysis, i) *RLE* algorithm compatible with *D1* dataset and ii) *RLE* algorithm not compatible with *D0_unsorted* category dataset. During evaluation, the *RLE* compatible dataset *D1* has accelerated 6.9 GB/s on average throughput at `DMA_3`. On the contrary, the *D0_unsorted* dataset which is not *RLE* algorithm compatible has accelerated 4 GB/s on average throughput at `DMA_4`. In both category datasets, bitwidth-wise throughput effects are almost the same as *RLE* does not compress values at bit-level. Table 4.5 shows the resource consumption for the different numbers of DMA-based *CRLE* compression systems, whereby all designs meet the timing at 250 MHz.

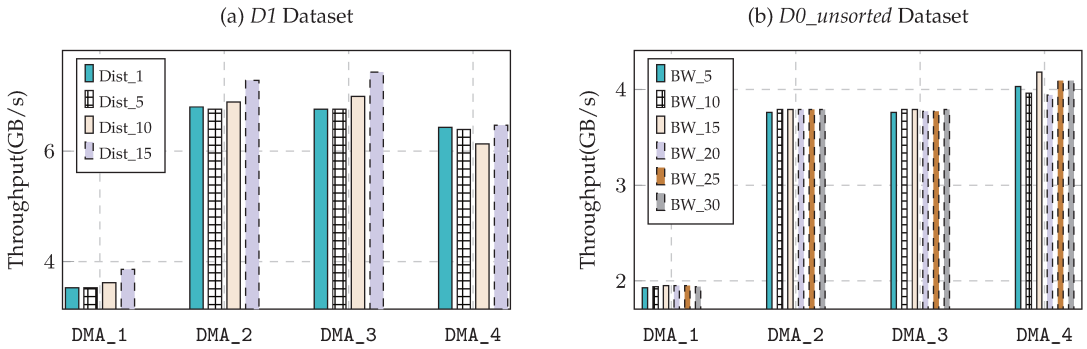

Figure 4.27: Throughput Analysis for *CRLE* Compression Systems based on Different #DMAs (Average over 5 runs).

In Figure 4.28, we show the trade-off between resource consumption and speed up of the *CRLE* designs. Based on that our observations are as follows,

Table 4.5: Resource Utilization and *PL* part Frequency Analysis for *CRLE* Compression Systems based on Different #DMAs.

#DMA	LUTs(%)	FFs(%)	Power(W)	Frequency(MHz)
DMA_1	2.95	2.02	3.696	250
DMA_2	5.63	3.92	3.900	250
DMA_3	8.20	5.74	4.074	250
DMA_4	10.76	7.56	4.236	250

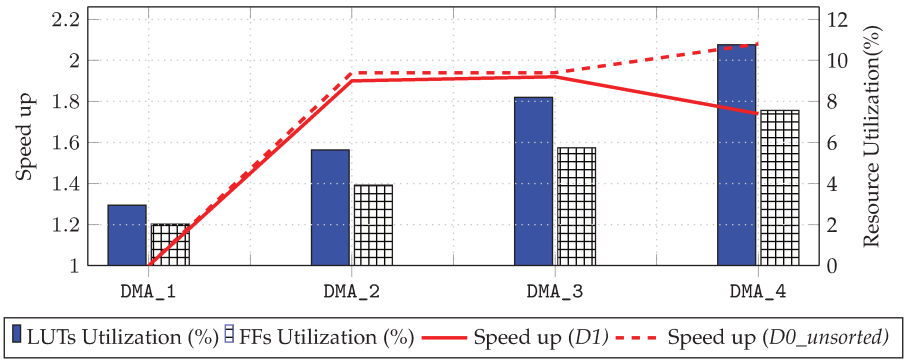

Figure 4.28: Comparison Between Speed up and Resource Utilization for *CRLE* Compression Systems based on Different #DMAs.

- The resource consumptions are linearly increasing DMA-wise.
- The *RLE* suited dataset *D1* speeds up ≈2x at DMA_2, and the *RLE* not suited dataset *D0_unsorted* speeds up 2x at DMA_4. This proves that *RLE* throughput acceleration depends on the *RLE* suited dataset.
- The best design depends on the #distinct consecutive values based dataset and on the different #DMAs.

4.4.4 Cascaded Compression

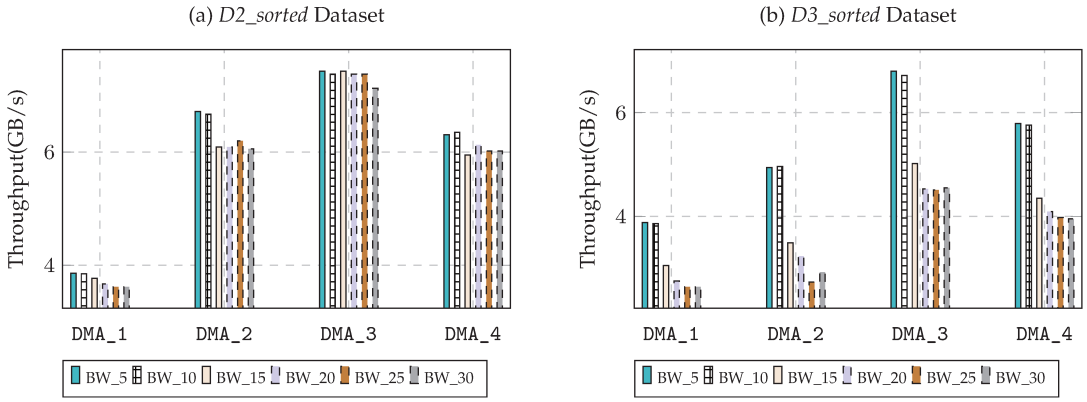

Figure 4.29: Throughput Analysis for Cascaded (*CDelta+CBP*) Compression Systems based on Different #DMAs (Average over 5 runs).

(*CDelta+CBP*) Compression Systems

In the cascaded compression system evaluation, we have used the *D2* and *D3* category datasets. We have evaluated throughput for different #DMAs based cascaded (*CDelta+CBP*) compression systems using *D2_sorted* and *D3_sorted* datasets (see Figure 4.29). In this case, bitwidth-wise different numbers of throughput have been generated. For instance, in Figure 4.29, the throughput of *BW_5* and *BW_10* is larger than others in all designs. The reason is, *D2_sorted* category based *BW_5* dataset consists of 90% 5-bit and 10% (>15)-bit bitwidth based integer values, and *D3_sorted* category based *BW_5* dataset consists of 50% 5-bit and 50% (>15)-bit bitwidth based integer values. Similarly, *D2_sorted* category based *BW_10* dataset consists of 90% 10-bit and 10% (>15)-bit bitwidth based integer values, and *D3_sorted* category based *BW_10* dataset consists of 50% 10-bit and 50% (>15)-bit bitwidth based integer values and so on. Therefore, *BW_5* and *BW_10* consists with more small integer values than other datasets. It has been proven from our *CBP* compression systems experiments that small values based data always provides better throughput than the big values (see Figure 4.23). DMA-wise both cases accelerate throughput up to `DMA_3`. Table 4.6 shows the resource consumption for the different numbers of DMA-based cascaded (*CDelta+CBP*) compression systems, whereby all designs meet the timing at 250 MHz.

Table 4.6: Resource Utilization and *PL* Part Fequency Analysis for Cascaded (*CDelta+CBP*) Compression Systems based on Different #DMAs.

#DMA	LUTs(%)	FFs(%)	Power(W)	Frequency(MHz)
`DMA_1`	4.69	2.26	3.773	250
`DMA_2`	9.12	4.40	4.042	250
`DMA_3`	13.42	6.47	4.307	250
`DMA_4`	17.73	8.53	4.468	250

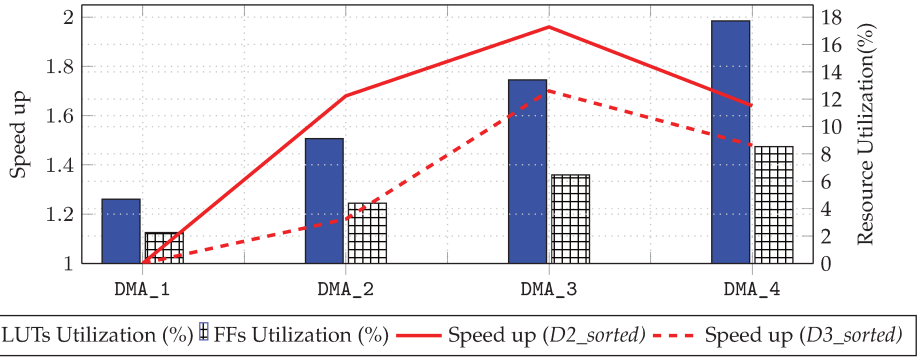

Figure 4.30: Comparison Between Speed up and Resource Utilization for Cascaded (*CDelta+CBP*) Compression Systems based on Different #DMAs.

In Figure 4.30, we show the comparison between resource consumption and speed up of the cascaded (*CDelta+CBP*) compression systems. Based on that our observations are as follows,

- The resource consumptions are linearly increasing DMA-wise.
- `DMA_3` gives the maximum throughput.
- Achieved speed up ≈2x for *D2_sorted* and 1.7x for *D3_sorted* dataset at `DMA_3`.
- `DMA_3` is the best design for (*CDelta+CBP*) compression system.
- The best design does not consume maximum hardware resources.
- Throughput acceleration depends on bitwidth of integer values including on the distribution of small and big values, and on the different #DMAs.

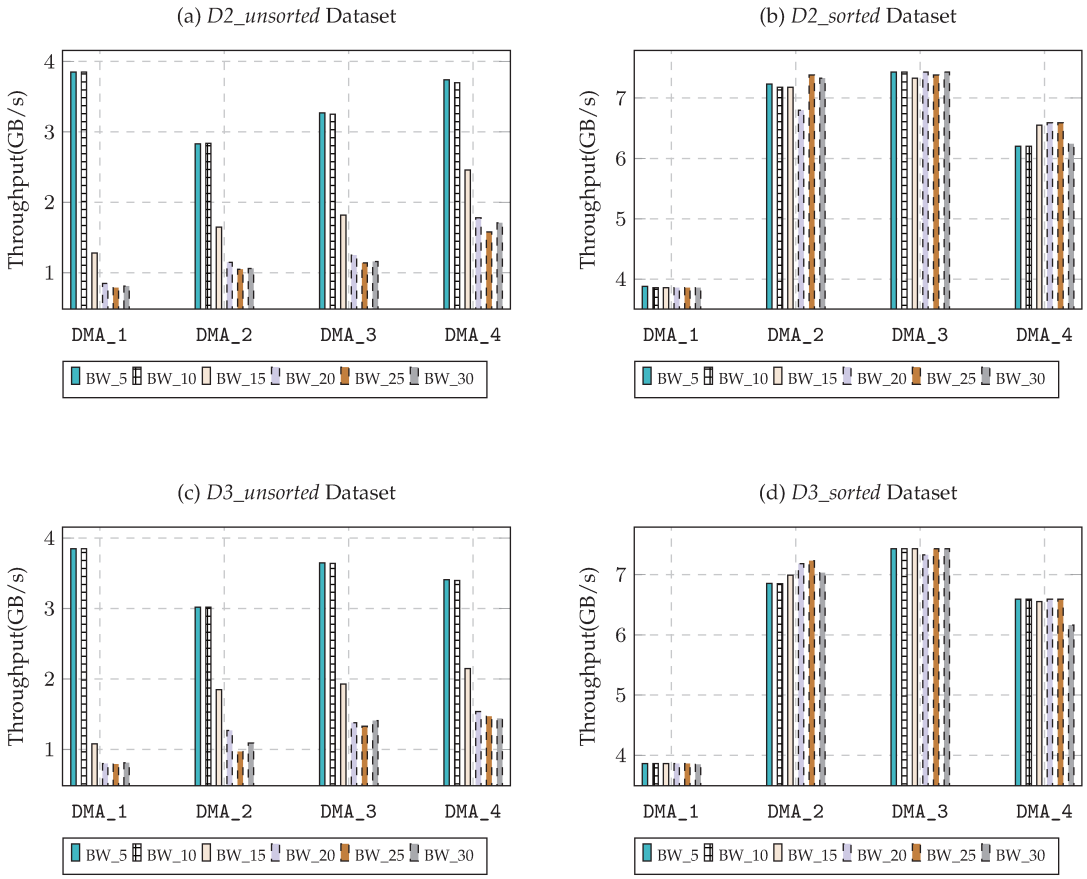

Figure 4.31: Throughput Analysis for Cascaded (*CFOR+CBP*) Compression Systems based on Different #DMAs (Average over 5 runs).

(*CFOR+CBP*) Compression Systems

After the evaluation of cascaded (*CDelta+CBP*), we have evaluated cascaded (*CFOR+CBP*) compression systems. Here, we have considered both types of sorted and unsorted based *D2* and *D3* category datasets as *FOR* algorithm is compatible with both sorted and unsorted datasets. Different types of behavior in terms of throughput have been noticed in this case. In unsorted dataset cases, bitwidth-wise small values-based datasets have accelerated throughput more than big values and DMA-wise it could not accelerate throughput after `DMA_1`, e.g., *BW_5* and *BW_10* have accelerated the throughput at `DMA_1` (see Figure 4.31(a) and (c)). The reason is that while values are distributed in an unsorted manner including more (>15)-bit bitwidth based values than small values, there are very low chances to produce lower bitwidth based compressed values using *FOR* algorithm. More precisely, the distance between a value and the smallest value of a *block* increased that produces larger bitwidth based compressed values. Thus, the compression rate decreases which affects the throughput negatively especially in the scenario of more than one DMA-based design. However, this behavior is not similar in sorted dataset cases. In sorted dataset cases, bitwidth-wise throughput is almost equal and DMA-wise throughput linearly increases up to `DMA_3` (see Figure 4.31(b) and (d)). It happens because values are distributed in a sorted manner which produces small and almost equal distance-based compressed values *block*-wise for all bitwidth based sorted datasets. Therefore, in sorted datasets, we have achieved better throughput than unsorted datasets. Table 4.7 shows the resource consumption for the different numbers

Table 4.7: Resource Utilization and *PL* Part Frequency Analysis for Cascaded (*CFOR+CBP*) Compression Systems based on Different #DMAs.

#DMA	LUTs(%)	FFs(%)	Power(W)	Frequency(MHz)
DMA_1	5.33	2.29	3.768	250
DMA_2	10.39	4.47	3.999	250
DMA_3	15.33	6.57	4.243	250
DMA_4	20.28	8.67	4.462	250

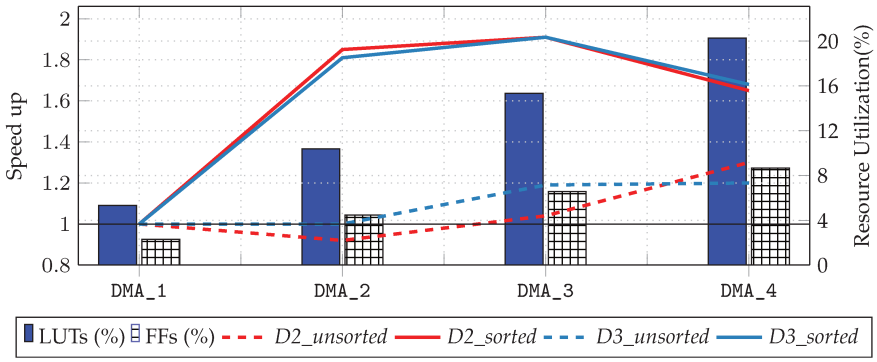

Figure 4.32: Comparison Between Speed up and Resource Utilization for Cascaded (*CFOR+CBP*) Compression Systems based on Different #DMAs.

of DMA-based cascaded (*CFOR+CBP*) compression systems, whereby all designs meet the timing at 250 MHz.

In Figure 4.32, we show the comparison between resource consumption and speed up of the cascaded (*CFOR+CBP*) compression systems. Based on that our observations are as follows,

- The resource consumptions are linearly increasing DMA-wise.
- Sorted and unsorted datasets show different behavior regarding speed up. The sorted dataset speed up at DMA_3. In contrast, the *D2_unsorted* dataset slow down below baseline in DMA_2, afterward it speed up again at DMA_4, and the *D3_unsorted* dataset provides a speed up at DMA_3.
- Throughput acceleration depends on bitwidth of integer values including the distribution of small and big values, and the different #DMAs.

(*CRLE+CBP*) Compression Systems

In Figure 4.33, we have considered two different categories of datasets for cascaded (*CRLE+CBP*) compression systems throughput analysis like *CRLE*, i) *RLE* algorithm compatible *D1* dataset and ii) *RLE* algorithm not compatible *D0_unsorted* category dataset. During evaluation, *RLE* compatible dataset *D1* has accelerated 7.45 GB/s on average throughput at DMA_3 which is higher than the *CRLE* system. On the contrary, *RLE* not compatible dataset *D0_unsorted* has accelerated 2.55 GB/s on average throughput at DMA_4 which is lower than the *CRLE* system. Bitwidth-wise *D1* category datasets have almost an equal throughput. The throughput gradually decreases bitwidth-wise for *D0_unsorted* datasets. This scenario proves that cascading *CRLE* with *CBP* is useful for *RLE* algorithm compatible datasets. Table 4.5 shows the resource consumption for the

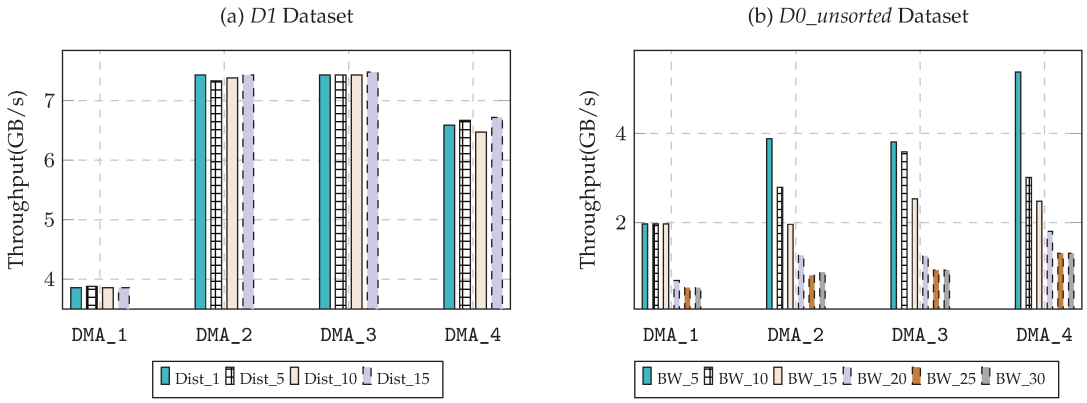

Figure 4.33: Throughput Analysis for Cascaded (*CRLE+CBP*) Compression Systems on Different #DMA (Average over 5 runs).

Table 4.8: Resource Utilization and *PL* Part Frequency Analysis for Cascaded (*CRLE+CBP*) Compression Systems based on Different #DMAs.

#DMA	LUTs(%)	FFs(%)	Power(W)	Frequency(MHz)
DMA_1	5.36	2.52	3.760	250
DMA_2	10.44	4.92	4.032	250
DMA_3	15.41	7.24	4.247	250
DMA_4	20.39	9.56	4.555	250

different numbers of DMAs-based *CRLE* compression systems, whereby all designs meet the timing at 250 MHz.

In Figure 4.34, we show the comparison between resource consumption and speed up of the cascaded (*CRLE+CBP*) compression systems. Based on that our observations are as follows,

- The resource consumptions are linearly increasing DMA-wise.
- The *RLE* algorithm suitable dataset *D1* speeds up ≈1.92x at DMA_2. In contrast, the *D0_unsorted* dataset which is not suitable for *RLE* algorithm speeds up linearly up to 1.98x at DMA_4.
- The best design depends on the #distinct consecutive values based dataset and on the different #DMAs.

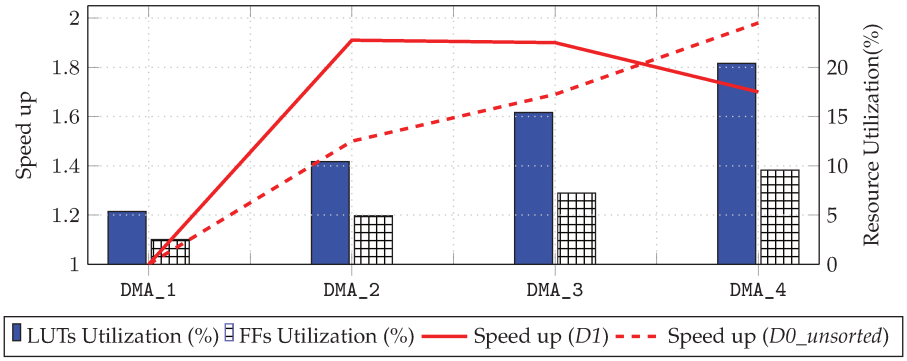

Figure 4.34: Comparison Between Speed up and Resource Utilization for Cascaded (*CRLE+CBP*) Compression Systems based on Different #DMAs.

4.4.5 Adaptive Compression

User-Specified Adaptive Systems

In *User-Specified Adaptive* systems, we have considered the *D0_sorted* category datasets for throughput analysis (see Figure 4.35). During evaluation, at first *CBP* and later (*CDelta+CBP*) have been chosen for throughput analysis. In this case, *CBP* has accelerated throughput less than (*CDelta+CBP*) bitwidth-wise as well as DMA-wise as expected. For instance, *CBP* and (*CDelta+CBP*) have generated 6.8 GB/s and 7.3 GB/s throughput at `DMA_2` for *BW_15* (see Figure 4.35). On the contrary, the physical-level compression system has accelerated throughput at `DMA_4` with 5.89 GB/s for *BW_15* based *D0_sorted* category dataset (see Figure 4.23). However, the throughput is almost the same for *CBP* and (*CDelta+CBP*) of *User-Specified Adaptive* system at `DMA_3` and it goes down at `DMA_4`. Table 4.9 shows the resource consumption for the different numbers of DMA-based *User-Specified Adaptive* compression systems. Here, all designs meet the timing at 250 MHz. Thus, this evaluation indicates that adapting physical-level and all the cascaded compression systems together is fruitful in terms of throughput as well as resource consumption. The reason is that we have achieved maximum throughput in the *User-Specified Adaptive* system of `DMA_2` with 12.96% of LUTs, 6.49% of FFs usage and in the physical-level compression system of `DMA_4` with 17.09% of LUTs, 7.85% of FFs consumption on Zynq UltraScale+™ (see Table 4.9 and Table 4.3).

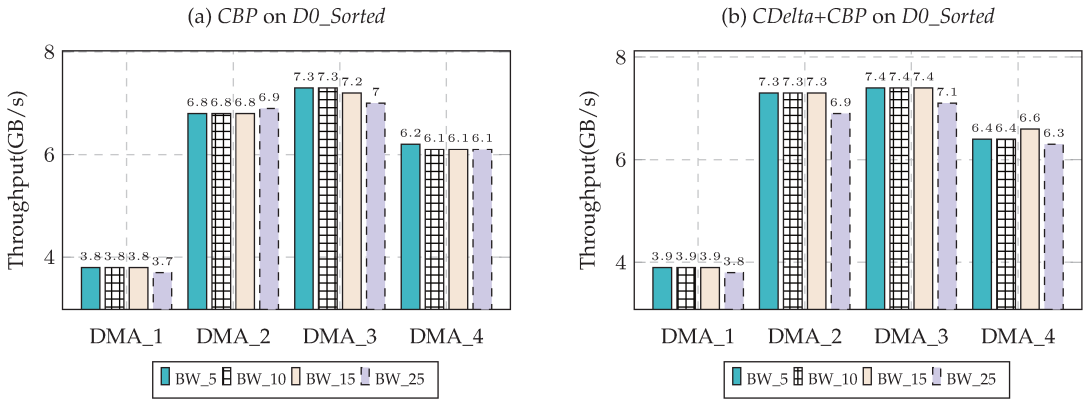

Figure 4.35: Throughput Analysis for *User-Specified Adaptive* Compression Systems based on Different #DMAs (Average over 5 runs).

Table 4.9: Resource Utilization and *PL* Part Frequency Analysis for *User-Specified Adaptive* Compression Systems based on Different #DMAs.

#DMA	LUTs(%)	FFs(%)	Power(W)	Frequency(MHz)
`DMA_1`	6.61	3.30	3.719	250
`DMA_2`	12.96	6.49	3.926	250
`DMA_3`	19.21	9.59	4.354	250
`DMA_4`	25.44	12.70	4.591	250

In Figure 4.36, we show the comparison between resource consumption and speed up of the *User-Specified Adaptive* compression systems. Based on that our observations are as follows,

- The resource consumptions are linearly increasing DMA-wise.

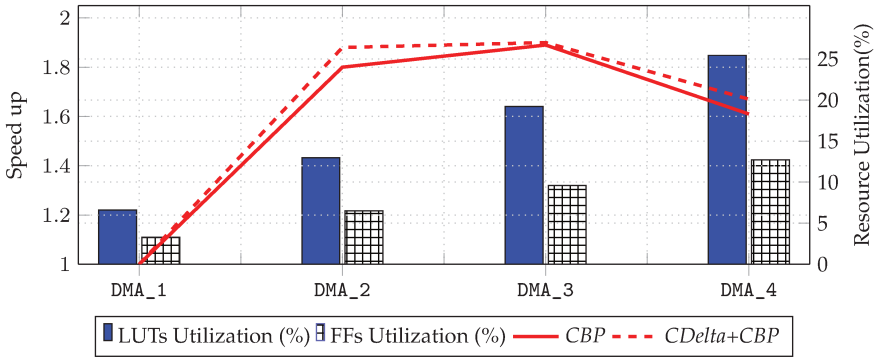

Figure 4.36: Comparison Between Speed up on *D0_Sorted* Dataset and Resource Utilization for *User-Specified Adaptive* Compression Systems based on Different #DMAs.

- Cascaded compression hardware is faster than physical-level compression in *User-Specified Adaptive* system.
- `DMA_3` gives the maximum speed up in a *User-Specified Adaptive* compression system, although the speed up between `DMA_2` and `DMA_3` is almost equal. Additionally, `DMA_3` consumes more resources than `DMA_2`. Hence, to make a trade-off between throughput and resource consumption, `DMA_2` is the best design.

HW-Specified Adaptive Systems

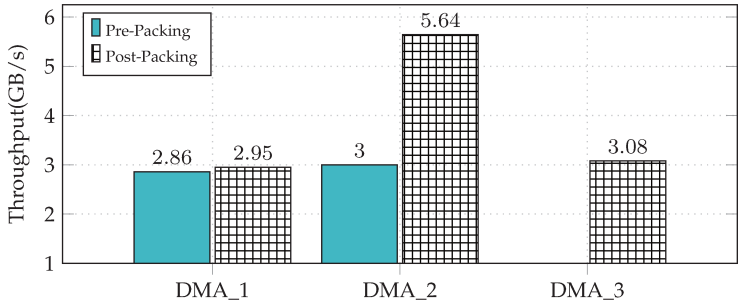

Figure 4.37: Throughput Analysis Comparison Between *Adaptive Pre-Packing* and *Post-Packing* Compression Systems based on Different #DMAs for *D4* Dataset (Average over 5 runs).

Table 4.10: Resource Utilization and *PL* Part Frequency Comparison Between *Adaptive Pre-Packing* and *Adaptive Post-Packing* Compression Systems based on Different #DMAs.

#DMA	*Adaptive Pre-Packing*				*Adaptive Post-Packing*			
	LUTs(%)	FFs(%)	Power(W)	Freq(MHz)	LUTs(%)	FFs(%)	Power(W)	Freq(MHz)
`DMA_1`	23.00	23.00	4.398	200	14.92	15.58	4.138	200
`DMA_2`	45.89	45.84	4.275	100	29.64	31.06	4.775	200
`DMA_3`	–	–	–	–	44.36	46.14	4.395	100

In this subsection, the *HW-Specified Adaptive* systems evaluation are shown. We have started with the comparative evaluation between *Adaptive Pre-Packing* and *Adaptive Post-Packing* compression systems. Here, we have considered the *D4* category dataset that is compatible with such types of compression systems. Table 4.9 shows the resource consumption for the different numbers of DMA-based *HW-Specified Adaptive* compression systems. The evaluation of the *Adaptive Pre-Packing* system is considered for

DMA_1 and DMA_2 designs as the frequency is gradually decreased from DMA_1 to DMA_2 at 200 MHz to 100 MHz, respectively (see Table 4.10). On the contrary, the evaluation of an *Adaptive Post-Packing* system is considered up to a DMA_3 design. The reason is that the timing does not meet at 200 MHz after DMA_2. However, the timing of *HW-Specified Adaptive* system does not meet at 250 MHz, likewise *User-Specified Adaptive* system. This happens becasue the *HW-Specified Adaptive* systems have a complicated logic based on a custom-made hardware called *CSelector* which reduces the timing of the system. In Figure 4.37, DMA-wise the *Adaptive Post-Packing* compression systems have generated always larger throughput than the *Adaptive Pre-Packing* systems. This indicates *Adaptive Post-Packing* is better than *Adaptive Pre-Packing* compression systems. In Figure 4.38, we have evaluated *Adaptive Post-Packing* systems using *D0_sorted* and *D0_unsorted* datasets for more investigation. And we have found that bitwidth-wise all designs have given the same throughput in both datasets. As expected, DMA-wise DMA_2 have generated larger throughput than the others.

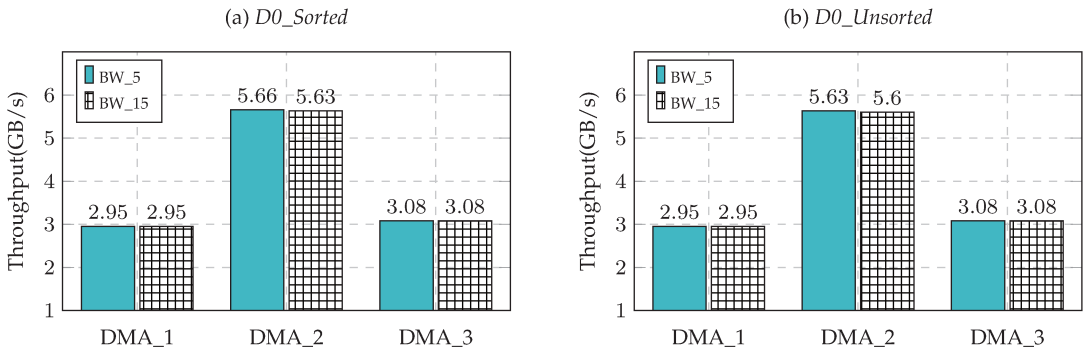

Figure 4.38: Throughput Analysis for *Adaptive Post-Packing* Compression Systems based on Different #DMAs (Average over 5 runs).

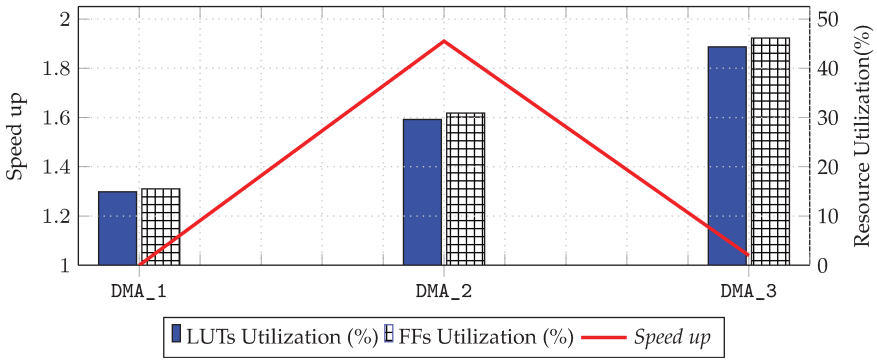

Figure 4.39: Comparison Between Speed up and Resource Utilization for *Adaptive Post-Packing* Compression Systems based on Different #DMAs.

In Figure 4.39, we show the comparison between resource consumption and speed up of the *Adaptive Post-Packing* compression systems. Based on the evaluations of *HW-Specified Adaptive* systems our observations are as follows,

- The resource consumptions are linearly increasing DMA-wise in all designs.
- *Adaptive Post-Packing* compression systems are more beneficial than *Adaptive Pre-Packing* systems throughput-wise as well as resource consumption-wise, whereby post-order bitpacking is performed after *CSelector* hardware. Hence, *Adaptive Post-Packing* compression system is the best.
- DMA_2 of the *Adaptive Post-Packing* compression system provides the maximum throughput.

- The achieved speed up is 1.91x at DMA_2 of *Adaptive Post-Packing* compression system.
- DMA_2 is the best design for the *Adaptive Post-Packing* compression system.
- The best design does not consume maximum hardware resources.
- Throughput acceleration is data property independent, but it depends on the different #DMA.

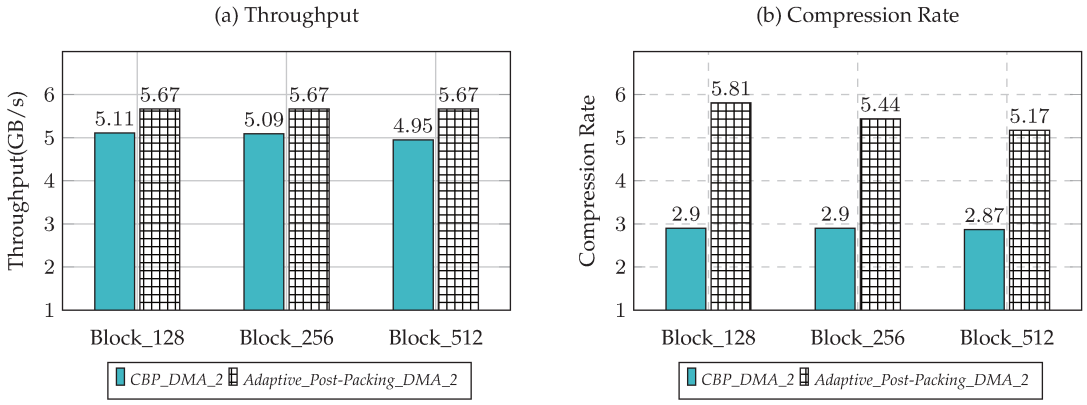

Figure 4.40: (a) Throughput and (b) Compression Rate Analysis Between *CBP_DMA_2* (250MHz) and *Adaptive_Post-Packing_DMA_2* (200MHz) Compression Systems on Different Block Distribution Based Dataset.

In the end, to show the efficiency of the adaptive system, we have performed a comparative evaluation between *Adaptive_Post-Packing* and physical-level *CBP* systems. Here, we have considered three *D4* category (see Table 4.2) dataset called *Block_128*, *Block_256* and *Block_512* having the *block*-size of 128, 256, and 512, respectively. For fair comparison, we have considered the DMA_2 design of *Adaptive_Post-Packing* and *CBP* systems. During evaluation, we have found that among all datasets the *Adaptive_Post-Packing* system has generated larger throughput than *CBP* (see Figure 4.40(a)). Hence, the *Adaptive_Post-Packing* system is faster than *CBP* despite having less frequency than the physical-level compression system.

Last but not the least, one of the key primitives for any compression technique is the compression rate. The compression rate for all the proposed FPGA-based compression implementations entirely depends on the ratio of input and output words. The number of input words depends on the total number of integer values for a given dataset and the number of values per input word is shown in equation 4.4.1. The number of output words depends on the number of uncompressed values per *block* and the number of compressed values per output word is shown in equation 4.4.2, whereby the number of compressed values per output word calculation is shown in 4.4.3. It was mentioned earlier that in our implementation, the output word bitwidth for packing values is 120-bit, whereby the most significant 8-bit per output word is used for packing the largest value bitwidth per *block*. Finally, the compression rate calculation depends on the uncompressed and compressed number of values in terms of the number of input and output words as shown in equation 4.4.4.

$$\#Input\_Words = \frac{\#Integer\_Values}{\#Values\_per\_Input\_Word} \tag{4.4.1}$$

$$\#Output\_Words = \sum_{n=1}^{\#Blocks} \frac{\#Uncompressed\_Values\_per\_Block}{\#Compressed\_Values\_per\_Output\_Word} \tag{4.4.2}$$

$$\#Compressed\_Values\_per\_Output\_Word = \frac{Output\_Word\_Bitwidth}{Largest\_Value\_Bitwidth} \quad (4.4.3)$$

$$Compression\_Rate = \frac{\#Input\_Words}{\#Output\_Words} \quad (4.4.4)$$

Based on the above equations we have calculated the compression rates for *Adaptive_Post-Packing* and *CBP* systems using *Block_128*, *Block_256* and *Block_512* datasets (see Figure 4.40(b)). Hence, the *Adaptive_Post-Packing* system provides 49%, 53% and 55% improved compression rates over *CBP* on *Block_128*, *Block_256* and *Block_512* datasets, respectively.

4.5 LESSONS LEARNED AND SUMMARY

The subsection 4.2 shows that the lightweight integer compression algorithms are perfectly implementable on FPGA. Moreover, it is noticeable from the subsection 4.3 that the hardware-based adaptive lightweight compression system employing FPGA is feasible. The evaluation results of subsection 4.4 prove that FPGA is a reliable opportunity in the context of compression throughput acceleration as well as compression rate improvement for lightweight integer compression.

In all design cases, the resource consumption increases linearly as the number of DMAs increases as expected. However, some compression systems do not get the benefit of increasing the number of DMAs regarding compression throughput acceleration. For instance, none of the *HW-Specified Adaptive* systems are effective for `DMA_4` design as the circuit became too complex to meet the timing at the range from 150 MHz to 250 MHz. While implementing multiple DMA-based systems on FPGA, among several setups one of the important setups is the burst ratio of DMA for AXI streaming data communication with the main memory. Anomaly may occur due to an inconsistent setup of the burst ratio, especially for multiple DMA-based cases. DMA burst defines the number of input and output words transferred in one transaction between DMA and main memory. As mentioned earlier, logically main memory has the maximum 4 data channels. However, physically it has only one data channel. The main memory is serving each logical data channel in a round robin fashion, whereby each data channel is accessing a DMA. In Figure 4.41, we show three different cases for the interaction of two DMAs with the main memory which described below:

Case1: The input and output bursts in both DMAs are set to 32. That means 32 input and output words transfer in each cycle between a DMA and main memory. For instance, in n^{th} clock, DMA1 reads the first 32 words from main memory. In $(n+1)^{th}$ clock, DMA1 writes the first 32 words to main memory, and DMA2 reads the first 32 words from main memory, in parallel. In the next following clock, DMA1 reads the second 32 words from main memory, and DMA2 writes the first 32 words to main memory, in parallel, and so on. Thus, Case1 is the best-case scenario as no anomaly occurs.

Case2: Case2 is similar to Case1 regarding the number of input and output bursts of DMAs, which is 32. In the case of AXI streaming parallel read/write, it is noticeable that one of the DMAs may not ready to write the words to the main memory in it's defined clock. For instance, in $(n+1)^{th}$ clock, DMA1 is not ready to write the first 32 words to the main memory, but DMA2 reads the first 32 words from the main memory. Then, in $(n+2)^{th}$ clock, DMA1 reads the second 32 words from

main memory, and DMA2 writes the first 32 words to main memory, in parallel. Afterward, in $(n+3)^{th}$ clock, DMA1 writes the second 32 words instead of the first 32 words to main memory, and DMA2 reads the second 32 words from main memory, in parallel. As a consequence, inconsistent occurs as output data is missing. Thus, Case2 is the worst-case scenario due to anomaly occurrence.

Case3: To avoid the anomaly occurrence of Case2, in this case, the input and output bursts in both DMAs are set to 32 and 16, respectively. That means 32 input and 16 output words transfer in each cycle between a DMA and main memory. For instance, in n^{th} clock, DMA1 reads the first 32 words from main memory. In $(n+1)^{th}$ clock, DMA1 writes the first 16 words to main memory and DMA2 reads the first 32 words from main memory, in parallel. In the next following clock, DMA1 reads the second 32 words from main memory, and DMA2 writes the first 16 words to main memory, in parallel, and so on. In this way, the output data missing anomaly is avoidable as the read burst size is bigger than the write burst. Usually, a DMA does not respond to the main memory for writing a burst until its write buffer is full, or it gets a AXI_TLast signal. For proper data communication between multiple DMAs and main memory, the input and output burst size of the DMA should not be equal in the AXI streaming parallel read/write scenario. Thus, Case3 is the base-case scenario that can generate optimal output.

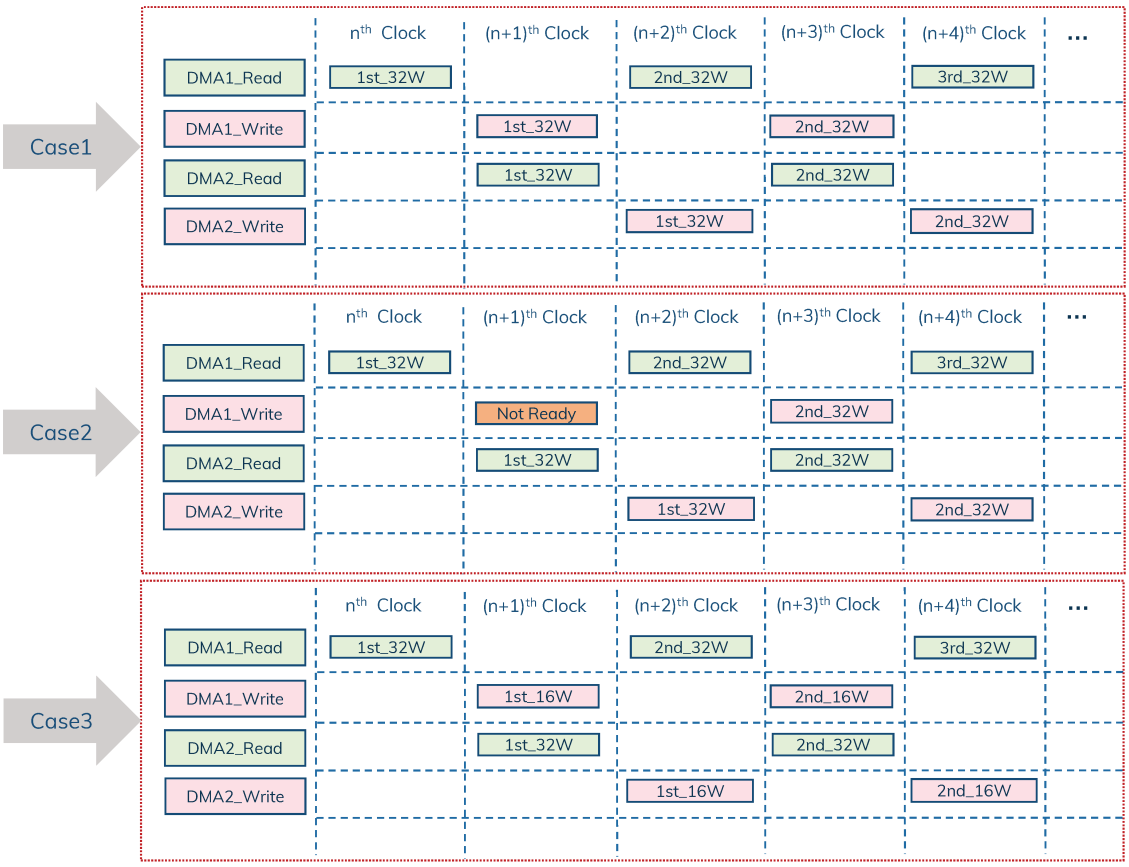

Figure 4.41: Streaming Data Communication Between Multiple DMAs and Main Memory.

Therefore, we set up the burst ratio on *HW-Specified Adaptive* systems as per Case3 that helps to prevent the output data missing problem. Based on these, the following two key points always need to be taken care of while implementing AXI streaming parallel read and write based system on FPGA,

- The DMA burst read/write ratio.
- Adequate setup the *PL* part frequency.

In this chapter, we explored hardware-based implementation opportunities for lightweight integer compression using a hybrid CPU-FPGA system. In particular, we analyzed the behavioral differences of physical-level, logical-level, cascaded, and adaptive compression systems as per hardware-based implementation. In adaptive compression implementations, we improved the compression rate compared to others. This proves, FPGA is perfect for lightweight integer compression. System reliability highly depends on the DMA burst ratio, proper frequency setup, and different numbers of DMA usage. Therefore, improving the compression rate through a hybrid CPU-FPGA system, besides the proper pipeline-based hardware implementation, it requires a balanced setup regarding frequency, number of DMAs and DMA burst ratio as per requirements.

CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

The core service of every business and scientific application is data management. The ever-increasing amount of data requires an adequate understanding of efficient management. From the modeling perspective creating a conventional database schema through large datasets is a challenging task. On the other side, from the system architectural perspective, the main requirement is flexibly storing and processing large datasets. To satisfy these requirements, the architecture of database systems is constantly evolving [FKL+17], [Leh17]. For instance, the architectural concepts of database systems shifted from a disk-centric to a main memory-centric architecture to efficiently exploit the ever-increasing capacities of main memory. Thus, the main memory architecture has become state-of-the-art and has characterized by the fact that all relevant data is completely stored and processed in the main memory. However, the gap between the computing power of the CPU and main memory bandwidth continuously increases, which has become the main bottleneck for high-performance query processing.

One out of several proposals to address this bottleneck is a shift of the memory organization of relational tables from a row- to a column-oriented format [FKL+17]. A key primitive in this main memory column-store database systems is *column scan* [FLKX15], [LP13], [WPB+09], because analytical queries usually compute aggregations over full or large parts of columns. Moreover, the increasing demand for main memory based data processing leads to the fast *scan* operation in main memory database systems [FLKX15], [LP13], [WPB+09]. Thus, the optimization of the *scan* primitive is very crucial [FLKX15], [LP13], [WPB+09]. Recently, the *BitWeaving* approach addresses this need by packing multiple compressed columnar data into processor words, and applying full-word instructions for predicate evaluations using a well-defined arithmetic framework [LP13]. To further reduce the necessary memory space and to increase the processing performance, main memory column-store database systems optimize the columnar representations through (i) encoding the values of each column as a sequence of integers using some variant of dictionary encoding, and (ii) applying lightweight lossless integer compression to each sequence of integers resulting in a sequence of compressed column values. Furthermore, the columns are usually compressed in such a specific way that the query engine can directly work on the compressed representation (e.g. *filter*, *scan*). Although there exists a large corpus of lossless lightweight integer compression algorithms, Damme *et al.* [DUH+19] have shown, there is no single-best algorithm, and the decision highly depends on data characteristics. Additionally, to select an appropriate compression algorithm, a trade-off between performance and compression rate must be defined [DUH+19].

To guarantee a certain expected level of performance, developing software through utilizing innovative algorithms (such as lightweight integer compression algorithms) is not enough. Therefore, the semiconductor industry has moved towards parallel computing to accelerate performance. However, common CPU based parallel systems are fixed-hardware bound, and to guarantee a certain expected level of performance in-depth reconfiguration is required. That means we need a flexible way for hardware-level reconfiguration. The era of modern hardware has introduced a novel class of accelerators called Field Programmable Gate Array (FPGA). Thus, FPGA has become an interesting alternative, which can be reconfigurable after being manufactured. Unlike ASIC, FPGA does not require the time-consuming process of fabrication to produce a permanent circuitry. Hence, FPGA has become state-of-the-art due to its hardware reconfigurable capability. One of the major advantages of using FPGA is that we can use an arbitrary length of processor word leading to a higher performance. However, the arbitrary length of processor word depends on the system available hardware resource constraint in terms of the number of LUTs, Flip-flops, etc. Query

processing acceleration using FPGA, especially using a hybrid CPU-FPGA architecture is very promising. Additionally, the hybrid CPU-FPGA system has direct access to the main memory, which reduces the latency of data communication between FPGA and main memory [LNH+19]. Due to the large variety of query processing operations, the most challenging task is to cope with the limited hardware resources of FPGA for individual query processing tasks running concurrently.

This thesis outlined the design aspects to enable an efficient hardware-level adaptability for the development of database operators with the optimal resource consumption. Thus, this thesis presented the following,

- We presented various FPGA design opportunities and optimizations for *scan* mechanisms in a systematic way, whereby each design has its unique properties and accelerated the *scan* performance compared to CPU-based implementation [LUH+18a].

- In detail, we explored two hardware-based implementation opportunities for optimization using SIMD extensions and custom-made architectures on FPGA for different *scan* mechanisms [LUH+18b]. In particular, we analyzed the behavioral differences between *Naïve* [Lam75] and *BitWeaving* [LP13] *scan* mechanisms as per hardware-based implementation. With both implementations, we achieved the improvement regarding *scan* performance, whereby the FPGA is best for *Naïve* technique and *BitWeaving* is perfect for SIMD. Therefore, improving the *scan* performance through FPGA does not require any fancy *scan* mechanism like *BitWeaving* due to its high parallelism criteria, and flexibility to reconfigure hardware as per requirements. For instance, the best performing HYBRID_1024 design does not use all available data channels of the PS part main memory controller, but uses both available main memory modules of the targeted hybrid CPU-FPGA system. That means the optimal design depends on the concrete hybrid system.

- Next, we presented a brief overview of pipeline-oriented hardware implementation for high-throughput *BitPacking* compression on FPGA called *CBP* [LNH+19]. To enable seamless pipelining, we resolved algorithmic dependencies regarding read overheads and nonalignment writes by introducing internal buffering as well as categorized packing mechanism. These changes sacrifice some amount of compression rate, but they enable the implementation to scale up to $\approx$7.5GB/s throughput for 2-bit bitwidth based integer values. We prepared pipeline implementation for *BitPacking* compression in a scalable and resource-efficient way. Moreover, we explored different hardware designs based on the different number of DMAs and embraced resource-throughput trade-off relations. Finally, the custom-made *BitPacking* implementation achieved very high-throughput and resource-efficiency on the targeted hybrid CPU-FPGA system.

- Afterward, we explored the logical-level lightweight compression algorithm implementations in the targeted hybrid CPU-FPGA system, by utilizing proposed custom-made *CDelta*, *CFOR*, *CRLE* hardware. Besides logical-level, we prepared the cascaded compression system implementations based on different logical-level lightweight compression algorithms. In the cascaded implementation, firstly, values are reduced by logical-level lightweight compression hardware at the value-level. Secondly, logically reduced values are compressed by physical-level lightweight compression hardware called *CBP* at the bit-level. All implementations were explored with different number of DMAs. In both cases, throughput acceleration is highly dependent on data properties and the number of DMAs.

- Finally, we proposed the adaptive lightweight compression system implementations in the targeted hybrid CPU-FPGA system, by utilizing the proposed custom-made lightweight algorithm-based compression hardware. The main concern of these adaptive implementations is to achieve the best possible compression throughput with minimum resource utilization. We implemented a naive version called *User-Specified Adpative* as well as a sophisticated version called *HW_Specified Adpative* compression systems. The *HW_Specified Adpative* compression system depends on a sophisticated custom-made hardware called *CSelector*, whereby the *CSelector* is responsible to choose appropriate custom-made lightweight compression hardware as per data property at run-time. The *HW_Specified Adpative* compression system has been categorized by the *Pre-Packing* and *Post-Packing* systems, whereby pre-order and post-order bit packing has been performed before and after the *CSelector* hardware, respectively. The evaluation proved that the *HW_Specified Post-Packing Adpative* system outperforms the *HW_Specified Pre-Packing Adpative* system throughput-wise as well as resource consumption-wise. In the end, the *HW_Specified Post-Packing Adpative* system improved the compression rate on average $\approx$50% compared to the physical-level compression system.

5.2 FUTURE WORK

The proposed *HW-Specified Adaptive* systems for lightweight integer compression required more detailed investigations. Therefore, the list of following are the potential future work,

- The *HW-Specified Pre-Packing* and *Post-Packing Adaptive* systems have considered the *Delta* and *Frame of Reference (FOR)* logical-level algorithms, whereby *Delta* works for only sorted dataset. The *CFOR* custom-made hardware does not consider the exceptions. For instance, in a sequence of the input dataset with one large value, e.g., 5, 10, 125, 7, 1, 15, 4294967295, is no longer compressible using any algorithm as this sequence required 32-bit integer due to having one large value. To avoid such problem, Zukowski *et al.* [ZHNB06] proposed *Patching Frame of Reference (PFOR)*. In *PFOR*, they classify the whole dataset with regular coded values and exceptions. They kept the exceptions in a separate location. In the future, we will implement and accumulate the *PFOR* algorithm in the proposed adaptive systems to deal with such exceptions of a block.

- The proposed *HW-Specified Pre-Packing* and *Post-Packing Adaptive* systems have not considered the *RLE* algorithm, as the proposed designs compressed values block-wise and *RLE* algorithm does not always compress values block-wise. Therefore, in the future, we will update the proposed system architecture in a way where data will examine at first for an uninterrupted sequence of integer values. Hence, the proposed *CBroadcaster* hardware will be modified to identify the uninterrupted sequence of integer values for a predefined block size to verify whether the block is suitable for *CRLE* hardware or not.

- Decompression is an extra workload for database systems as most of all queries can work on compressed datasets. However, decompression may be necessary for some query workload cases. That means some specific query plans may require a decompressed dataset. There are two data decompression strategies available. Firstly, eager decompression that decompresses data when data resides in main memory [IW94]. Secondly, lazy decompression that keeps data compressed during query execution as long as possible, and decompressed some data explicitly when it

is necessary [WKHM00]. The proposed adaptive systems have not considered any decompression strategies. Thus, in the future, the proposed adaptive systems will efficiently accumulate decompression depending on the system-level complexity.

- Join operation plays a significant role in the database domain. It is hard to implement efficiently. Mostly, the join operation implementation is using the hash join algorithm. However, hash join operation consumes lots of query processing time. Thus, for efficient join operation implementation, other relevant aspects need to consider. For instance, the processing time may improve if the join operation operates on compressed data. Compression is not only useful for reducing the memory footprint, but also useful regarding the CPU expenses as CPU requires to join data every time while processing quires. Thus, performing a join operation on a compressed dataset will improve performance. Hence, in the future, we will include join operators in the proposed adaptive lightweight compression systems.

In these contexts, we will investigate in detail and define appropriate hardware design rules for FPGA-accelerated designs on the hybrid CPU-FPGA system for the above-mentioned furure work.

BIBLIOGRAPHY

[AHF+14] Oliver Arnold, Sebastian Haas, Gerhard Fettweis, Benjamin Schlegel, Thomas Kissinger, and Wolfgang Lehner. An application-specific instruction set for accelerating set-oriented database primitives. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD'14, pages 767–778, New York, NY, USA, 2014. Association for Computing Machinery.

[AIR+17] Sandeep R Agrawal, Sam Idicula, Arun Raghavan, Evangelos Vlachos, Venkatraman Govindaraju, Venkatanathan Varadarajan, Cagri Balkesen, Georgios Giannikis, Charlie Roth, Nipun Agarwal, and et al. A many-core architecture for in-memory data processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50'17, pages 245–258, New York, NY, USA, 2017. Association for Computing Machinery.

[AMF06] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[BATO13] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, 2013.

[BHF09] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD'09, pages 283–296, New York, NY, USA, 2009. ACM.

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

[BK99] Peter A. Boncz and Martin L. Kersten. Mil primitives for querying a fragmented world. *The VLDB Journal*, 8(2):101–119, October 1999.

[BKG+18] Cagri Balkesen, Nitin Kunal, Georgios Giannikis, Pit Fender, Seema Sundara, Felix Schmidt, Jarod Wen, Sandeep Agrawal, Arun Raghavan, Venkatanathan Varadarajan, Anand Viswanathan, Balakrishnan Chandrasekaran, Sam Idicula, Nipun Agarwal, and Eric Sedlar. Rapid: In-memory analytical query processing engine with extreme performance per watt. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD'18, pages 1407–1419, New York, NY, USA, 2018. Association for Computing Machinery.

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[BRT+18] Tobias Behrens, Viktor Rosenfeld, Jonas Traub, Sebastian Breß, and Volker Markl. Efficient simd vectorization for hashing in opencl. In *Advances in Database Technology - EDBT 2018. International Conference on Extending Database Technology (EDBT-2018), 21th International Conference on Extending Database Technology, March 26-29, Vienna, Austria*, pages 489–492, Universität Konstanz 78457 Konstanz, Germany, 2018. EDBT Association, OpenProceedings.

[BS10] Peter Bakkum and Kevin Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 94–103, New York, NY, USA, 2010. Association for Computing Machinery.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[DHHL17] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. Lightweight data compression algorithms: An experimental survey (experiments and analyses). In *EDBT*, pages 72–83, 2017.

[DUH+19] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. From a comprehensive experimental survey to a cost-based selection strategy for lightweight integer compression algorithms. *ACM Trans. Database Syst.*, 44(3), June 2019.

[ES16] Isitor Emmanuel and Clare Stanier. Defining big data. In *Proceedings of the International Conference on Big Data and Advanced Wireless Technologies*, BDAW'16, New York, NY, USA, 2016. Association for Computing Machinery.

[FHL10] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3(1-2):670–680, September 2010.

[FKL+17] Franz Faerber, Alfons Kemper, Per-Åke Larson, Justin J. Levandoski, Thomas Neumann, and Andrew Pavlo. Main memory database systems. *Foundations and Trends in Databases*, 8(1-2):1–130, 2017.

[FLKX15] Ziqiang Feng, Eric Lo, Ben Kao, and Wenjian Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *SIGMOD*, pages 31–46, 2015.

[FLZ15] Shaokun Fan, Raymond Y. K. Lau, and J. Leon Zhao. Demystifying big data analytics for business intelligence through the lens of marketing mix. *Big Data Research*, 2(1):28–32, 2015.

[FMH+19] Jian Fang, Yvo Mulder, Jan Hidders, Jinho Lee, and H.P. Hofstee. In-memory database acceleration on fpgas: a survey. *The VLDB Journal*, 10 2019.

[HHDL16] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. Compression-aware in-memory query processing: Vision, system design and beyond. In *ADMS Workshop at VLDB*, pages 40–56, 2016.

[HHDL17] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. Compression-aware in-memory query processing: Vision, system design and beyond. pages 40–56, 03 2017.

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

[ISA17] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent distributed storage. *PVLDB*, 10(11):1202–1213, 2017.

[Ist20] Zsolt István. Let's add transactions to fpga-based key-value stores! In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN'20, New York, NY, USA, 2020. Association for Computing Machinery.

[IW94] Balakrishna R. Iyer and David Wilhite. Data compression support in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB'94, pages 695–704, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[JWCW15] Xiaolong Jin, Benjamin W. Wah, Xueqi Cheng, and Yuanzhuo Wang. Significance and challenges of big data research. *Big Data Research*, 2:59–64, 2015.

[KHL17] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. Adaptive work placement for query processing on heterogeneous computing resources. *Proc. VLDB Endow.*, 10(7):733–744, March 2017.

[Lam75] Leslie Lamport. Multiple byte processing with full-word instructions. *Communications of the ACM*, 18(8):471–475, 1975.

[LB15] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.*, 45(1):1–29, 2015.

[Leh17] Wolfgang Lehner. The data center under your desk - how disruptive is modern hardware for DB system design? *PVLDB*, 10(12):2018–2019, 2017.

[LMF+16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. Data blocks: Hybrid oltp and olap on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 311–326, New York, NY, USA, 2016. ACM.

[LNH+19] Nusrat Jahan Lisa, Tuan Duy Anh Nguyen, Dirk Habich, Akash Kumar, and Wolfgang Lehner. High-throughput bitpacking compression. In *22nd Euromicro Conference on Digital System Design, DSD 2019, Kallithea, Greece, August 28-30, 2019*, pages 643–646. IEEE, 2019.

[LP13] Yinan Li and Jignesh M. Patel. Bitweaving: Fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

[LUH+18a] Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Tuan D. A. Nguyen, and Akash Kumar. Column scan acceleration in hybrid CPU-FPGA systems. In Rajesh Bordawekar and Tirthankar Lahiri, editors, *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2018, Rio de Janeiro, Brazil, August 27, 2018*, pages 22–33, 2018.

[LUH+18b] Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Wolfgang Lehner, Tuan Duy Anh Nguyen, and Akash Kumar. FPGA vs. SIMD: comparison for main memory-based fast column scan. In Christoph Quix and Jorge Bernardino, editors, *Data Management Technologies and Applications - 7th International Conference, DATA 2018, Porto, Portugal, July 26-28, 2018, Revised Selected Papers*, volume 862 of *Communications in Computer and Information Science*, pages 116–140. Springer, 2018.

[LUH+18c] Nusrat Jahan Lisa, Annett Ungethüm, Dirk Habich, Tuan D. A. Nguyen, Akash Kumar, and Wolfgang Lehner. Column scan optimization by increasing intra-instruction parallelism. In *DATA*, pages 344–353. SciTePress, 2018.

[MMFB20] Mahmoud Mohsen, Norman May, Christian Färber, and David Broneske. Fpga-accelerated compression of integer vectors. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN'20, New York, NY, USA, 2020. Association for Computing Machinery.

[MS14] Onur Mutlu and Lavanya Subramanian. Research problems and opportunities in memory systems. *Supercomput. Front. Innov.: Int. J.*, 1(3):19–55, October 2014.

[MTA09] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, August 2009.

[OBL+17] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. Memory management techniques for large-scale persistent-main-memory systems. *Proc. VLDB Endow.*, 10(11):1166–1177, August 2017.

[PK12] Piotr Przymus and Krzysztof Kaczmarski. Improving efficiency of data intensive applications on gpu using lightweight compression. In Pilar Herrero, Hervé Panetto, Robert Meersman, and Tharam Dillon, editors, *On the Move to Meaningful Internet Systems: OTM 2012 Workshops*, pages 3–12, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[PR19] Orestis Polychroniou and Kenneth A. Ross. Towards practical vectorized analytical query engines. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, New York, NY, USA, 2019. Association for Computing Machinery.

[PRR15] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD'15, pages 1493–1508, New York, NY, USA, 2015. Association for Computing Machinery.

[RB17] Eyal Rozenberg and Peter Boncz. Faster across the pcie bus: A gpu library for lightweight decompression: Including support for patched compression schemes. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, DAMON'17, New York, NY, USA, 2017. Association for Computing Machinery.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[Sch97] Robert R. Schaller. Moore's law: past, present and future. 1997.

[SIOA17] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *SIGMOD*, pages 403–415, 2017.

[Teu17] Jens Teubner. Fpgas for data processing: Current state. *it - Information Technology*, 59(3):125, 2017.

[TW13] Jens Teubner and Louis Woods. *Data Processing on FPGAs*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[TWN13] Jens Teubner, Louis Woods, and Chongling Nie. XLynx—an FPGA-based XML filter for hybrid XQuery processing. *ACM Transactions on Database Systems (TODS)*, 38(4):23, nov 2013.

[WKHM00] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. The implementation and performance of compressed databases. *SIGMOD Rec.*, 29(3):55–67, September 2000.

[WPB+09] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units. *VLDB*, 2(1):385–394, August 2009.

[WWT+14] Haichuan Wang, Peng Wu, Ilie Gabriel Tanase, Mauricio J. Serrano, and José E. Moreira. Simple, portable and fast simd intrinsic programming: Generic simd library. In *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, WPMVP'14, pages 9–16, New York, NY, USA, 2014. Association for Computing Machinery.

[Xil19a] Xilinx, Inc. *Alveo U50 Data Center Accelerator Card*, 2019.

[Xil19b] Xilinx, Inc. *Zynq UltraScale+ MPSoC Data Sheet: Overview*, 2019.

[ZHNB06] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 59. IEEE Computer Society, 2006.

[ZR02] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

LIST OF FIGURES

LIST OF TABLES

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, October 12, 2020