BSc Thesis Task Description

Vladyslav Moisieienkov
candidate for BSc degree in Electrical Engineering

FPGA-based compactions for LSM-tree Key-Value databases

Field-Programmable Gate Arrays (FPGA) devices can enhance the performance and efficiency of several computationally intensive systems, including databases. These accelerators leverage FPGAs, which are reconfigurable integrated circuits, to accelerate database operations and queries. By offloading specific tasks from traditional CPUs to FPGAs, these accelerators offer several advantages for database management.

The aim of this thesis is to design and implement an accelerator module for log-structured merge-tree (LSM-tree) databases. Since one of the most critical parts of these databases is the compaction process, the thesis aims to implement an FPGA-based compaction module for this type of database.

Tasks to be performed by the student will include:

- Study and analyze the possible hardware-based compaction approaches for the database.
- Implement the FPGA-based compaction module for the LSM-tree database.
- Verify the compaction module using simulation.
- Test the compaction module using a simple database files.
- Evaluate the results and the performance of the FPGA-based acceleration.

Supervisor at the department: Dr. Ákos Nagy, senior lecturer

Budapest, 2 October 2023

Dr. Hassan Charaf
professor
head of department

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

FPGA-based compactions for LSM-tree Key-Value databases

Bachelor's Thesis

Author
Vladyslav Moisieienkov

Advisor
dr. Ákos Nagy

December 12, 2023

Contents

STUDENT DECLARATION

I, *Vladyslav Moisieienkov*, the undersigned, hereby declare that the present Bachelor thesis work has been prepared by myself and without any unauthorized help or assistance. Only the specified sources (references, tools, etc.) were used. All parts taken from other sources word by word, or after rephrasing but with identical meaning, were unambiguously identified with explicit reference to the sources utilized.

I authorize the Faculty of Electrical Engineering and Informatics of the Budapest University of Technology and Economics to publish the principal data of the thesis work (author's name, title, abstracts in English and in a second language, year of preparation, supervisor's name, etc.) in a searchable, public, electronic and online database and to publish the full text of the thesis work on the internal network of the university (this may include access by authenticated outside users). I declare that the submitted hardcopy of the thesis work and its electronic version are identical.

Full text of thesis works classified upon the decision of the Dean will be published after a period of three years.

Budapest, December 12, 2023

Vladyslav Moisieienkov

Kivonat

Különböző iparágak széleskörűen használják a Log-structured Merge Tree (LSM-fa) kulcs-érték típusú adatbázisokat. Az LSM-fa alapú tárolók számára meghatározó a háttérben zajló tömörítési művelet. Az adatokkal való hatékony gazdálkodás által tudják ezek a műveletek fenntartani az LSM-fa alapú struktúrák előnyeit. A háttérben zajló tömörítés meglehetősen CPU-intenzív folyamat, amely akadályozza az adatbázison folyó olvasási és írási műveleteket. Ráadásul az adatbázis teljesítménye lecsökken a tömörítés hatására, mivel az erőforrások megoszlanak a tömörítés és az olvasási és írási műveletek között. A dolgozat bemutatja az első nyílt forráskódú, FPGA-alapú tömörítés-gyorsítót LSM-Compactron3000 néven, amely felgyorsítja a tömörítési folyamatot és ezáltal tehermentesíti a processzort. A gyorsítót egy Xilinx FPGA alapú, Zybo Zynq-7000 fejlesztőkártya segítségével teszteltük. A gyorsító teljesítményét összehasonlítottuk egy csak CPU-t használó implementációval is. A dolgozatban ezenkívül bemutatjuk a megoldás hátrányait, és megoldásokat javaslunk az áthidalásukra.

Abstract

Log-structured Merge Tree (LSM-tree) key-value (KV) databases have been widely adopted in various industries. The critical mechanism within the LSM-tree databases is the implementation of background compaction operations. These operations ensure efficient housekeeping and sustain the advantages of LSM-tree structures. The compaction, which is performed in the background, is a CPU-intensive process that interferes with the read/write operations of the database. It degrades database performance by introducing a trade-off between allocating resources for compaction vs. read/write operations. This thesis introduces LSM-Compactron3000, the first open-source FPGA-based compaction accelerator, to speed up the compaction process and offload the CPU. We describe the hardware architecture and compare the accelerator's performance to the CPU-only baseline using a Zybo Zynq-7000 board with Xilinx 7-series FPGA logic. In addition, we analyze current bottlenecks and propose solutions to overcome them.

Chapter 1

Introduction

The key-value (KV) databases based on the Log Structure Merge (LSM) tree [1] are widely used in the industry due to their high write throughput and low write latency. Some of the popular LSM-tree-based KV stores include Google BigTable [2], Google LevelDB [3], and RocksDB [4]. The databases can achieve high write performance by converting random writes into sequential writes, which is greatly favored by modern storage devices. While serving write and read requests, the database accumulates many files that require periodic maintenance. This process is called *compaction* and is discussed in detail in Section 2.2.

As discovered by Zhang et al. [5], the compaction process, which is performed in the background, interferes with the read and write operations of the database by taking away CPU and I/O resources. It introduces a problematic trade-off between allocating resources for compaction and write/read operations, leading to the database's performance degradation. The solution proposed by Zhang et al. [5] is to offload the background compaction process to an FPGA-based hardware accelerator. Their implementation accelerated compactions by 2-5 times, improved the database throughput by up to 23%, and increased the number of transactions per watt by up to 31.7%, compared with the CPU-only baseline. Those are significant improvements. However, the paper only provided high-level hardware and software architecture without giving any source code. This is likely due to the cooperation of the authors with Alibaba and the use of their solution in proprietary Alibaba's X-Engine database [6].

Similar work on FPGA-based compactions has been done by Sun et al. [7], achieving a 92x speedup in compaction and 6.2x in the throughput for their solution. The paper thoroughly explains their design and integration with LevelDB [3]. However, no source code was provided.

The results from previous works are promising. However, the lack of source code makes reproducing and building upon the results difficult. In this thesis, we propose an FPGA-based compaction accelerator called *LSM-Compactron3000*. The project's primary focus is to build a simplified version of the Zhang et al. [5] hardware architecture. We implement the hardware design and software to control it.

The LSM-Compactron3000 is released under a permissive MIT open-source license on GitHub. It is the *first open-source* project of its kind. The repository contains a source code and development documentation. The open-source software is more accessible and promotes transparency and reproducibility.

Our solution is not intended for production use cases. However, LSM-Compactron3000 and its open-source nature can be a good starting point for future research and development in LSM-tree database hardware accelerators.

The following chapters of the thesis are:

- **Background**, provides information about work related to hardware optimizations in LSM-tree based databases and then explains typical operations of LSM-tree based KV databases. The chapter will provide a good foundation to understand the rest of the thesis;
- **Design and implementation**, provides an overview of Zhang et al. [5] work that inspired this thesis, then goes into details of hardware implementation in LSM-Compactron3000 project with a brief explanation of how software controls the hardware;
- **Evaluation**, provides results and observations from running our hardware implementation on the Zynq FPGA board;
- **Conclusion** discusses the results, issues, and directions for the project's future development.

Chapter 2

Background

This chapter will discuss the work done to optimize the compaction process using hardware and work principles of LSM-tree-based key-value (KV) databases.

2.1 Related work

The LSM-tree key-value stores are a subset of key-value stores, and the compaction process described in the Section 2.2 is specific only to them. The hardware optimizations for LSM-tree compactions are not an extensively covered topic, with only a few papers published. We will try to summarize the relevant work that has been done in the following paragraphs.

According to Zhang et al. [8], a lot of existing studies are focusing on decreasing the compaction frequency, reducing IOs, or confining compactions on hot data key ranges, but they do not consider the computational cost that can take up to 60% of the total compaction time. The above observations align with Sun et al. [9] comments that current optimizations mainly focus on reducing the overload of compaction in the host and rarely consider parallelism with other hardware.

Zhang et al. [5], Sun et al. [7], Ding et al. [10], Xu et al. [11] found that with an increase in the number of writes, LSM-tree compactions consume a significant portion of the CPU power. With more writes, the background compaction process is triggered more often. If you increase the amount of CPU (cores) dedicated to background compaction, it will negatively affect the write throughput of the database. This prompts a difficult question of resource management or requires a novel solution.

Zhang et al. [5], Sun et al. [7], Ding et al. [10], Xu et al. [11], Sun et al. [9, 12, 13, 14] propose to offload the compaction process to an external device. However, their approaches differ in the type of devices they use.

For example, Xu et al. [11] offloaded the compaction process to a GPU device. It achieved 2x higher throughput and 2x data processing speed with more stable 99th percentile latencies than LevelDB and RocksDB.

Sun et al. [9, 12, 13, 14] implemented a compaction process using near data processing (NDP) model. In this model, the compaction process is offloaded to a storage device with computational capabilities. It reduces the data transfer between the host and the storage device. Their results showed that the compaction process can be accelerated several times.

Ding et al. [10] based their compaction offloading on data processing units (DPUs). A DPU is a new class of programmable processors, a system on a chip that combines three key elements: multi-core CPUs, high-performance network interfaces, and programmable acceleration engines. The results show that compaction performance is accelerated by 2.86 to 4.03 times, system write and read throughput is improved by up to 3.2 times and 1.4 times, respectively, and host CPU contention is reduced.

Zhang et al. [5], Sun et al. [7] proposed and implemented FPGA hardware accelerators to offload the compaction process. They both got significant results in improving the performance of the compaction process and the overall throughput and power efficiency of the database.

Zhang et al. [5] preferred FPGA for the offloading due to the pipelined nature of the compaction process, which is more suitable compared to GPUs that employ the SIMD (Single Instruction Multiple Data) paradigm.

Overall, we can see that the compaction process is one of the main bottlenecks of LSM-tree-based databases, and offloading it to an external device improves the compaction process and the overall performance of the database.

Unfortunately, none of those mentioned above papers have published their source code, which makes it difficult to reproduce their results and advance the research in this area.

For example, Sun et al. [7] and Zhang et al. [5] have similarities in the architecture of their FPGA accelerators. Sun et al. [7] applies optimization of splitting key and value into separate streams. But, deducing from their diagrams and description, Zhang et al. [5] does not employ this strategy. Because of the unavailability of the source code, there is no possibility to conduct tests to compare proposed implementations.

The Dann et al. [15] provides an extensive overview of FPGA usage in various types of non-relational database systems such as graph, document, *key-value*, wide-column, etc. FPGAs are reconfigurable computing platforms that can implement massive custom parallel hardware design architectures. They can be placed in various system parts, for example, between host and network, host and disk, or as a separate co-processor. FPGA optimizations in non-relational databases are mainly researched academically with few commercial applications.

According to the survey by Dann et al. [15], the reviewed literature shows two primary motivations for using FPGAs in key-value stores. Either a single critical function of a key-value store is accelerated (e.g., insertion, hashing, or compaction) because the CPU does not meet the performance requirements. Or a whole key-value system is built inside FPGA to eliminate CPU processing time that increases roundtrip network latencies. They also mentioned that while wide-column stores are not represented in the literature, solutions described for the key-value stores can be applicable. Dann et al. [15, page 255:29] also found that usage of FPGAs in key-value systems is better researched and represented commercially than in other types of non-relational databases.

2.2 How LSM-tree based key-value databases work

This section will briefly explain the main processes of LSM-tree-based key-value (KV) databases. The explanation should be enough to understand the rest of the thesis.

The typical LSM-tree-based database performs two basic operations:

- **write**, insert a new KV pair into the database;
- **read**, retrieve the value of a specified key;

In the following sections, we will cover each of these operations. It is important to note that each database implementation may have variations in how certain operations are performed but the general principles remain the same.

2.2.1 Write

When a new KV pair is inserted into the database (e.g., via a network request), it is first stored in the memory in a sorted data structure called *memtable* [2, page 11]. After a certain amount of data is stored in the memtable, it is flushed to the disk to a file called **Sorted String Table** (SSTable) [16, page 76] [2, page 7]. The SSTable is immutable, and all KV pairs are sorted by their keys in ascending (alphabetical) order. The most recent SSTable files contain up-to-date information. The visualization of the process can be found in the Figure 2.1.

The writes are very fast because there is no need to save to disk immediately. In addition, a database usually keeps an append-only log for the memtable called a Write-Ahead Log (WAL). Each new KV pair is also appended to the end of the WAL file. In case of a crash, the database can restore its memtable from the WAL file. For the sake of this thesis, we are not interested in WAL. Our focus is on the SSTables.

The delete operation is considered a special insert operation. When a KV pair needs to be deleted, an insert request that contains a special value called **tombstone** [16, page 74] is sent to the database. In the Figure 2.1, we use *DEL* to mark a tombstone.

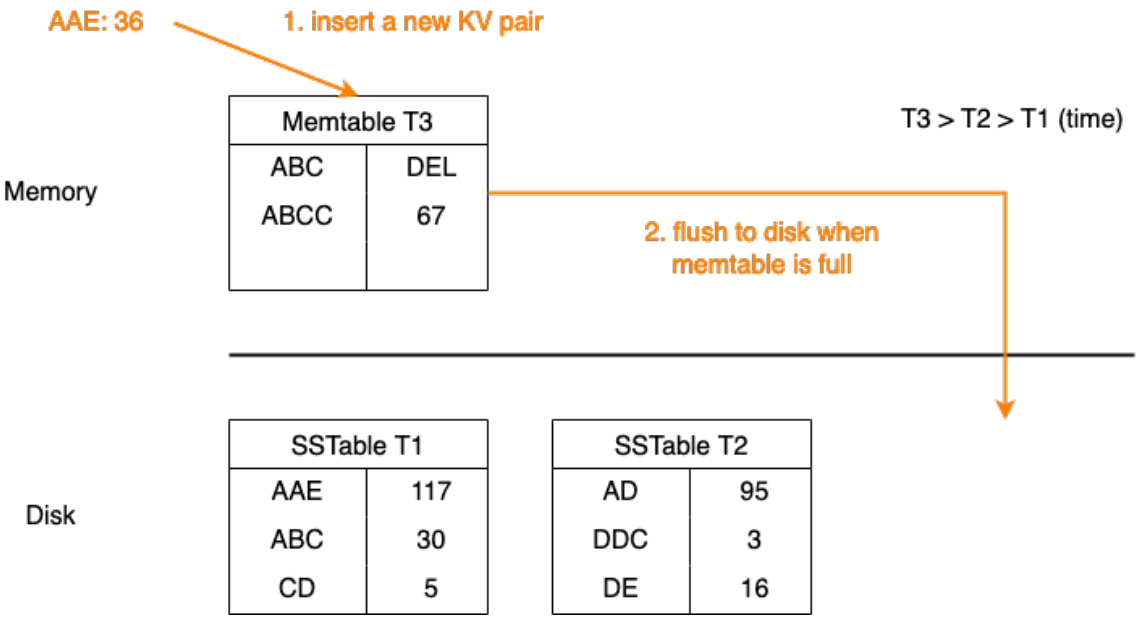

Figure 2.1: A typical write path of a LSM-tree based KV database.

2.2.2 Read

Now that we know how writes work let's see what happens when a database receives a read request.

When a read request arrives, it contains a key that *may* exist in the database. First, a database will check the memtable. If the key is not found, it will look at SSTable files, from the most recent to the oldest. The key can be quickly located because each SSTable is sorted. If the key cannot be found in memtable or SSTables, a database will return that key does not exist.

But this method has one issue. With more writes, more files will be created. The same key can be updated frequently in a short time, leaving a lot of outdated KV pairs in SSTables. It means the database needs to scan more files over time, leading to slower reads and a waste of storage space. The issue becomes more apparent when the database is used for a long time and under heavy load. A process called **compaction** [16, pages 76-77] [2, page 12] is used to fix the issue.

2.2.3 Compaction

A compaction process removes outdated KV pairs and reduces the number of SSTables by *merging* them into bigger SSTables. The process is performed in the background. The frequency depends on the database settings.

The compaction steps are similar to how *mergesort* algorithm works. The database starts reading the lowest keys (according to sort order) from input files side by side. The keys are compared against each other, and the KV pair corresponding to the lowest key is written to a new file. This process continues for all keys in the files. If the lowest keys are the same, the KV pair from the most recent SSTable file is selected. At the end, a new file is also sorted by a key. The visualization of the process can be found in the Figure 2.2.

The compaction process can occur between smaller SSTables recently flushed from the memtable or bigger SSTables that were compacted before. The process is repeated, forming bigger and bigger SSTables that will be shaped like a pyramid. There are several strategies in how the pyramid is formed [16, page 79] [17], but the core of merging SSTables remains the same.

For the compaction of tombstones, the approach can depend on the particular database implementation. For example, in Google BigTable [2, page 12], tombstone entries are removed only when major compaction occurs, meaning input SSTables are compacted to exactly one SSTable.

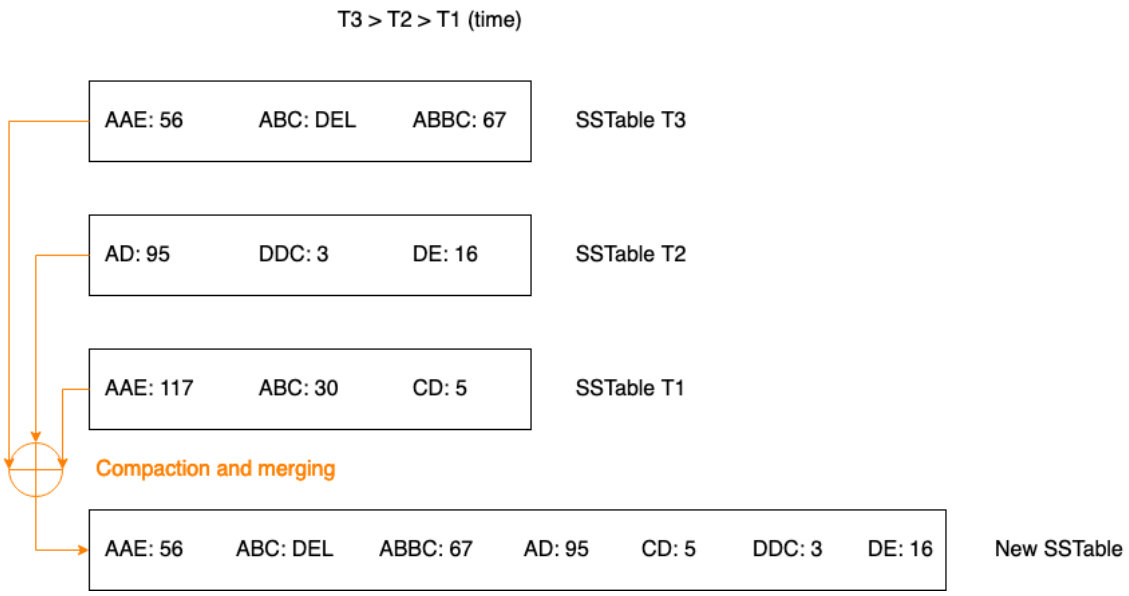

Figure 2.2: An example of compaction process.

Chapter 3

Design and implementation

In this chapter, we will describe the design and implementation of the LSM-Compactron3000. We will start with a high-level overview of the system to give the reader a general idea of the components of the system and how they interconnect. Then, we will go into details of each component and describe it in more detail.

3.1 High-level overview

The LSM-Compactron3000 project consists of hardware design for FPGA and software to communicate with hardware. It is primarily inspired by Zhang et al. [5] implementation. We will discuss details of their hardware implementation and how it differs from ours in Section 3.3.

The base module for performing compactions is called *Compaction Unit* (CU). On Figure 3.1, you can find a high-level overview of how CU communicates with external components. CU uses the AXI Stream interface to input and output the SSTable files. The number of AXI Stream input interfaces is configurable and depends on how many ways (SSTable files) we want to merge. A single output represents merged KV pairs. AXI Stream interfaces are connected to multiple Direct Memory Access (DMA) controllers. The AXI Stream interface was chosen for its simplicity which great speeded up the integration with the FPGA board.

An AXI Lite interface is used by the software to read and write registers in the CU to control its behavior.

The hardware in LSM-Compactron3000 was designed using the Chisel3 library, which will be discussed in Section 3.2, and synthesized using Xilinx Vivado soft-

ware. In the Figure 3.1, you can find a diagram view of CU in Xilinx Vivado. Inputs and outputs from the diagram are related to the labels on Figure 3.1.

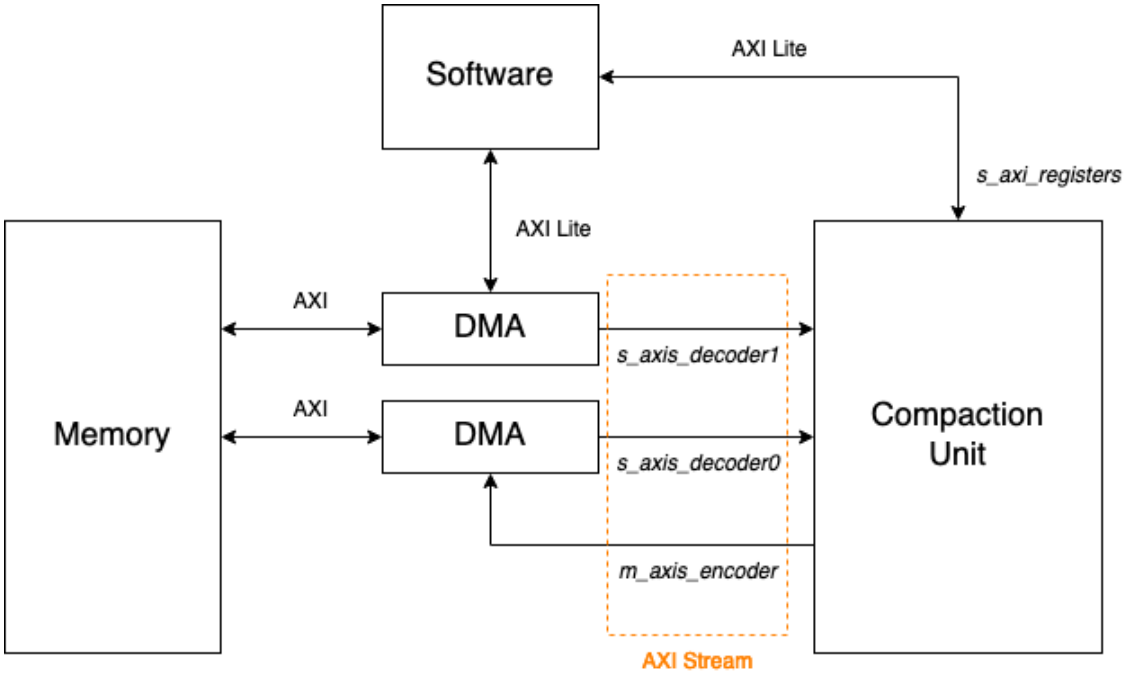

Figure 3.1: A high-level example of 2-way Compaction Unit from LSM-Compactron3000.

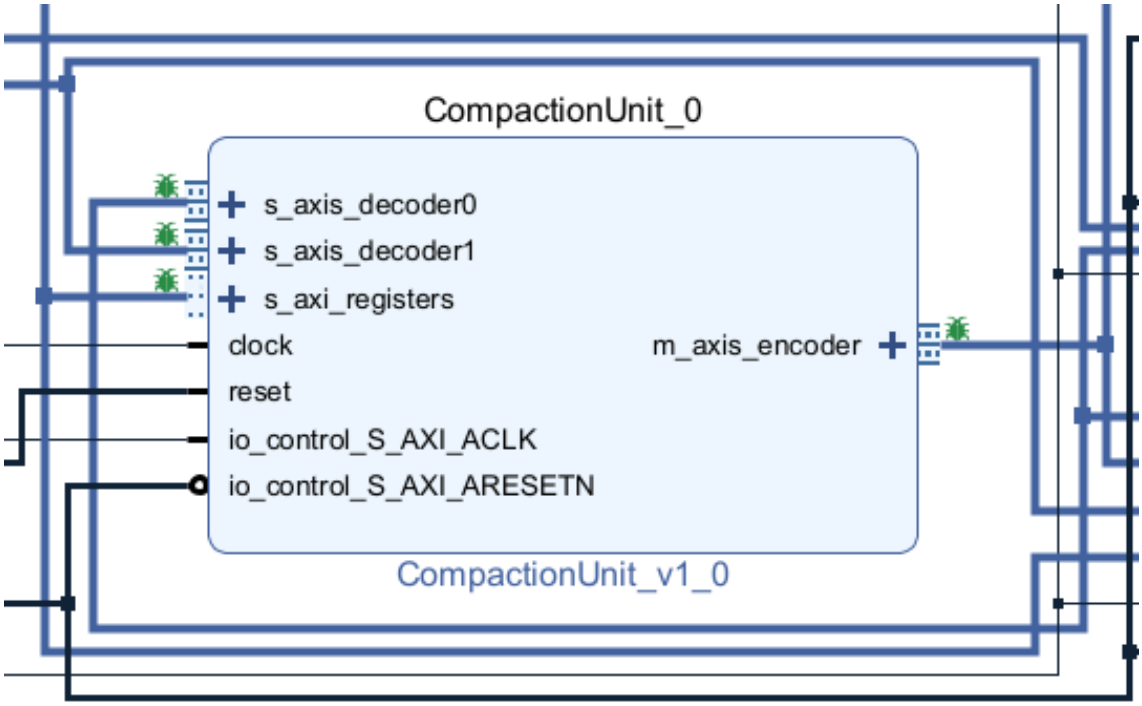

Figure 3.2: A screenshot of Compaction Unit IP in Vivado.

3.2 Chisel3

Most of the hardware design nowadays is described using a hardware description language (HDL) such as Verilog or VHDL. These languages are very low-level, require a lot of code to describe even simple circuits, and lack modern programming language features. To speed up the development process and facilitate future re-use, a new HDL called Chisel3 [18] (a third version of Chisel) was used in this project.

Chisel is an open-sourced domain-specific language (DSL) embedded in the general-purpose programming language called Scala [19].

Scala is a statically typed programming language that runs on Java Virtual Machine (JVM) and is interoperable with Java. It contains elements of object-oriented and functional programming. Scala allows hardware designers to use modern programming language features with Chisel.

Chisel was initially developed at the University of California, Berkeley, with a first release in 2012 and is actively developed to this day. It has been used in some advanced designs such as Rocket Chip [20] and Google Edge TPU [21], and is used by some companies such as SiFive [22]. But besides that, its adoption in the industry is still minimal.

Chisel focuses on the hardware generators, reusability of the code, and productivity of the hardware designers. To integrate with the existing toolchain, Chisel generates Verilog code that the standard industry tools can use. It also provides a testing framework [23] that allows the designer to write unit tests with the possibility of using Verilator [24] as a simulator backend. The formal verification tools are not yet officially available for Chisel3, but there is an ongoing effort to add them [25].

Lennon and Gahan [26] found that Chisel3 provides a productivity benefit for designers who work with a lot of repetitive code and want to focus on architecture design. They also concluded that while SystemVerilog can increase productivity over Verilog, Chisel's ability to use Scala's modern features makes it a better choice for complex system-on-chip (SoC) designs.

Käyrä and Hämäläinen [27] came to similar conclusions in their survey that confirm that Chisel language is expressive and can be used to create very complex designs. They also concluded that Chisel increases designer productivity and design reuse without negatively impacting physical design results.

In Listing 3.1, you can see an example of a combinational circuit in Chisel3 called *NextIndexSelector*. It is a standard Scala class that extends the Module class provided by Chisel. The *NextIndexSelector* module aims to select the next index based

on the input mask and current index. For example, if the current index is *1* and mask is *1010b* (a least significant bit on the right), the module will output *3* as the next index because we have *1* in the last position in the mask. You can see more examples of inputs and outputs in the test class called *NextIndexSelectorSpec*.

You can see that the *NextIndexSelector* module is parametrized over *n*, which indicates how many indexes we need (e.g., four for the previous example). You can write a regular Scala code to generate Chisel3 modules. The for-loop in the example is used to create *when/otherwise* statements that are hardware definitions. Only hardware definitions will be converted to Verilog, not every line of Scala code. Later, you can see *PriorityEncoder* module being used. It is provided as part of the Chisel3 default library. You can distribute your Chisel3 modules the same way you distribute Scala packages.

What is not mentioned in the above papers but is important is that recent versions of Chisel have transitioned to CIRCT [28] hardware compiler infrastructure to generate Verilog. CIRCT is an experimental project that aims to use modern compiler technologies to improve the hardware design process.

CIRCT is based on MLIR [29], and both projects are part of LLVM Foundation. MLIR is a compiler infrastructure used in many famous projects and programming languages such as TensorFlow [30] and Mojo [31]. The advanced compile techniques can be used to generate better hardware designs that are tailored to the specific use cases. In addition, by building higher-level abstractions in the hardware, a powerful compiler can optimize the design for the specific hardware platform.

The downsides of Chisel are a steep learning curve for the designers that are not familiar with Scala and relying on Chisel-to-Verilog translation, which makes the jobs of the hardware verification engineers harder [21].

```
1  class NextIndexSelector(n: Int) extends Module {
2      val io = IO(new Bundle {
3          val mask = Input(UInt(n.W))
4          val currentIndex = Input(UInt(log2Ceil(n).W))
5
6          val nextIndex = Output(UInt(log2Ceil(n).W))
7          val overflow = Output(Bool())
8      })
9      val modifiedMask = Wire(Vec(n, Bool()))
10
11     for (i <- 0 until n) {
12         when (i.U <= io.currentIndex) {
```

```
13              modifiedMask(i) := false.B
14          }.otherwise {
15              modifiedMask(i) := io.mask(i)
16          }
17      }
18
19      when (Cat(modifiedMask).asUInt === 0.U) {
20          io.nextIndex := PriorityEncoder(io.mask)
21          io.overflow := true.B
22      }.otherwise {
23          io.nextIndex := PriorityEncoder(modifiedMask.asUInt)
24          io.overflow := false.B
25      }
26  }
27
28  class NextIndexSelectorSpec extends AnyFreeSpec with
        ChiselScalatestTester {
29      "Different inputs produce correct outputs" in {
30          test(new NextIndexSelector(n =
        4)).withAnnotations(Seq(WriteVcdAnnotation)) { dut =>
31              val testCases = Seq(
32                  ("b1111", 0, 1, false),
33                  ("b1111", 1, 2, false),
34                  ("b1111", 2, 3, false),
35                  ("b1111", 3, 0, true),
36
37                  ("b1001", 0, 3, false),
38                  ("b0100", 2, 2, true),
39                  ("b1000", 3, 3, true),
40                  ("b0110", 2, 1, true),
41                  ("b0110", 1, 2, false),
42
43                  ("b0000", 0, 3, true),
44                  ("b0000", 1, 3, true),
45                  ("b0000", 2, 3, true),
46                  ("b0000", 3, 3, true),
47              )
48
49              for ((mask, currentIndex, nextIndex, overflow) <- testCases) {
50                  dut.io.mask.poke(mask.U)
51                  dut.io.currentIndex.poke(currentIndex.U)
```

```
52                  dut.io.nextIndex.expect(nextIndex.U)
53                  dut.io.overflow.expect(overflow.B)
54              }
55          }
56      }
57  }
```

Listing 3.1: An example of Chisel3 code from LSM-Compactron3000 project

3.3 The reference implementation

As mentioned above, this thesis is heavily influenced by Zhang et al. [5]. They proposed and implemented FPGA hardware and a driver that integrates with a proprietary database X-Engine. The Figure 3.3 showcases the high-level architecture of their solution.

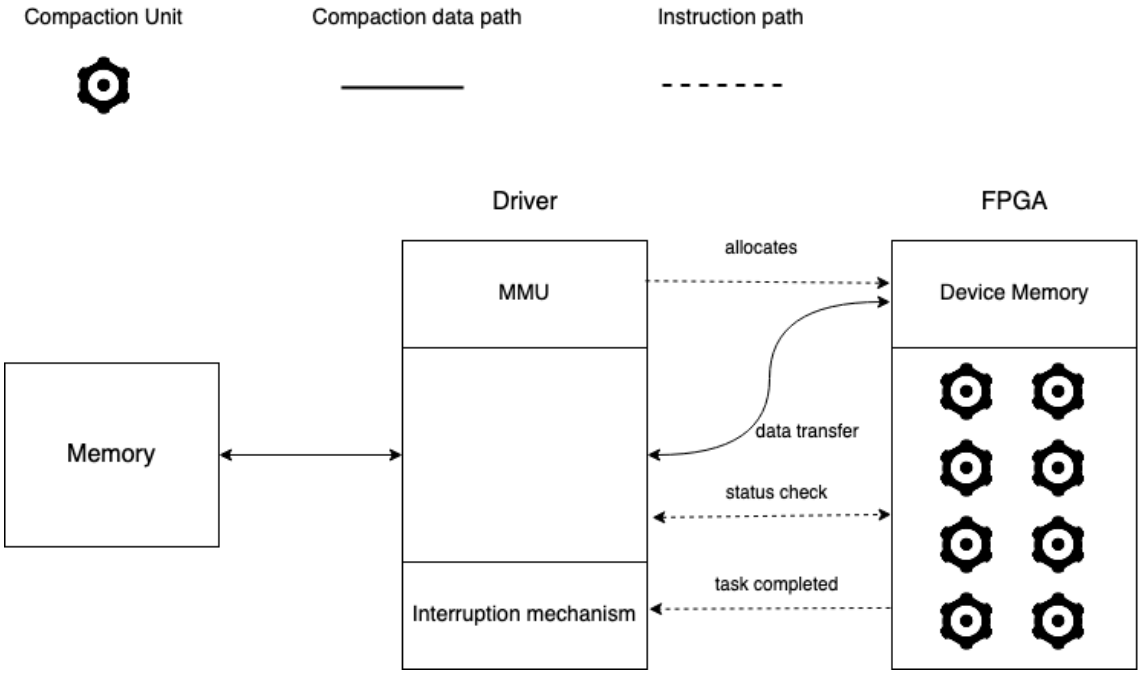

Figure 3.3: A diagram of [5] accelerator.

The scope of their project is enormous, so we decided to focus on the part of the hardware implementation - *Compaction Unit* (both of our projects use the same name). The CU is a single unit that can perform 4-way (up to 4 files) compactions. Zhang et al. [5] use a high-end FPGA device that allows to place up to 8 CUs at a time. The compaction tasks are distributed between available CUs.

The Figure 3.4 demonstrates various modules within a CU. As we will see in the Section 3.4, the high-level design of a CU unit is similar to the design of CU in LSM-Compactron3000. The modules of CU are the following:

- The **Decoder** is responsible for decompressing SSTables that reside in the device memory and outputting KV pairs to the KV Ring Buffer. Because databases can have different SSTable formats, the Decoder needs to be designed for a specific database.
- The **KV Ring Buffer** is a temporary storage for KV pairs. It allows the decoupling of the Decoder from the KV Transfer module.

- The **KV Transfer** has two responsibilities. First, copy keys from KV Ring Buffers to Key Buffer for comparison. Second, move selected KV pairs from the KV Ring Buffer to the Output Buffer.
- The **Merger** reads keys from Key Buffer and compares them. The result of the comparison is signaled to the Controller.
- The **KV Output Buffer** is effectively the same as the KV Ring Buffer. It decouples KV Transfer from the Encoder.
- The **Encoder** performs the reverse operation of the Decoder and is also specific to a database. It outputs merged KV pairs in an SSTable format of the database to the device memory.
- The **Controller** acts as a coordinator. It aims to match speeds between the modules for the best performance. For example, the Merger can only compare the keys when KV Ring Buffers are half full.

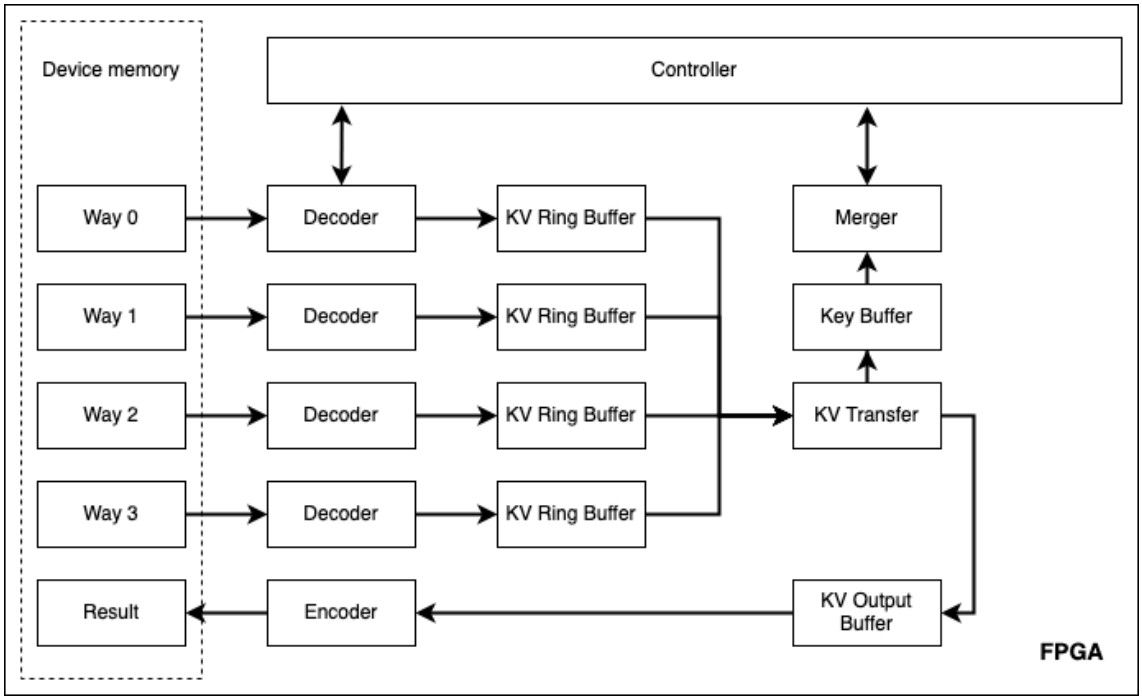

Figure 3.4: A diagram of a CU unit.

3.4 The hardware design

In this section, we will discuss the details of the hardware components in the LSM-Compactron3000 project. The CU module consists of the same high-level components discussed in Section 3.3. We can describe our architecture using the same Figure 3.4.

The CU can compact in 2, 4, or more ways. All of the components of CU besides the Decoder and Encoder are parametrized. You can configure bus width, maximum key and value sizes, number and capacity of buffers. This allows us to adapt design for different needs.

3.4.1 Decoder and Encoder

The Decoder reads SSTables from AXI Stream and outputs decoded KV pairs to the corresponding KV Ring Buffer. The Encoder reads merged KV pairs from the KV Output Buffer, encodes and transfers them via AXI Stream to the DMA controller that subsequently writes the data into the memory.

Both the Decoder and Encoder implementation can differ from database to database. Because SSTable formats used in production systems like RocksDB or X-Engine involve complex operations such as compression or prefix-length encoding, for this thesis, we decided to create a simplified custom SSTable format. The format is not designed for production use cases. The Figure 3.5 showcases how data is structured in our SSTables. A single KV pair consists of three parts:

- **Metadata**, a 32-bit number that contains how many chunks key and value part has, and if KV pair is the last one in SSTable;
- **Key chunks**, one or more 32-bit chunks that together constitute a key part of KV pair;
- **Value chunks**, one or more 32-bit chunks that together constitute a value part of KV pair.

The Decoder reads metadata, determines the length of the key and value streams, and loads them correctly. An LSB bit in the metadata indicates if current KV pair is the last. In the Figure 3.6, you can find details on how metadata field is structured. When the Decoder finishes writing the last KV pair to the KV Ring Buffer, it will set a flag to notify the Controller that it has seen the last pair.

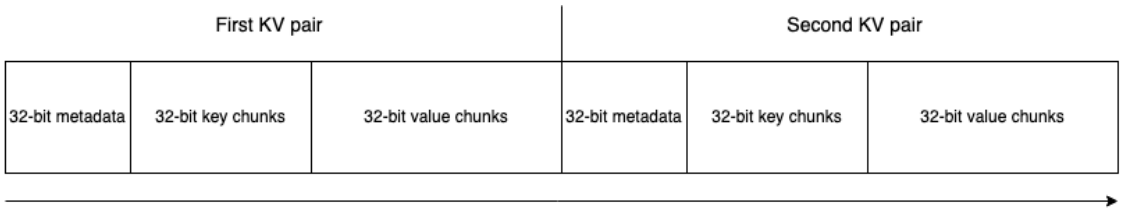

Figure 3.5: A diagram of a custom SSTable format used in LSM-Compactron3000.

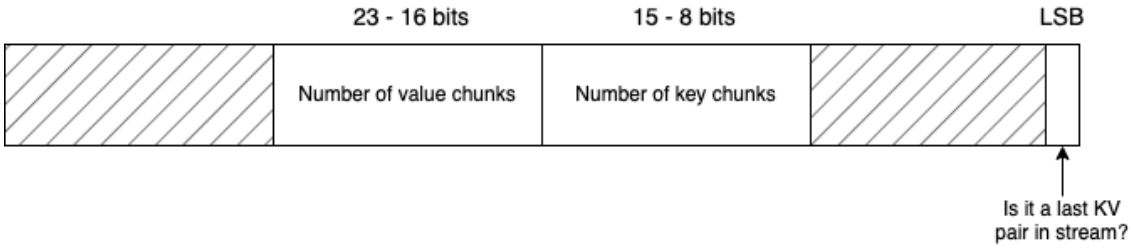

Figure 3.6: A diagram of fields inside the 32-bit metadata part of KV pair.

3.4.2 KV Ring Buffer and KV Output Buffer

A KV Ring Buffer and KV Output Buffer are temporary storage for KV pairs placed between Decoder-KV Transfer and KV Transfer-Encoder modules. They decouple components and help effectively pipeline data through the CU.

KV Ring Buffer and KV Output Buffer have the exact underlying implementation in LSM-Compactron3000. For the rest of this section, we will call the underlying implementation *the buffer*.

The buffer is a standard implementation of a circular ring buffer that uses a ready/-valid interface to communicate with other modules. The interface has several adjustments to signal when the key or value and last chunk are transferred.

Internally, the buffer uses synchronous memory to store data. The write operation has zero cycle delay, but the read operation has one cycle delay, so we needed to account for that in our design. In the Listing 3.2, you can see an example from the buffer implementation of how to instantiate the memory in Chisel3.

The buffer maintains two main pointers: write and read. In addition, it has other pointers responsible for writing or reading certain parts within the KV pair.

The write pointer is used to specify where in memory a KV pair data should be saved and is incremented when new data is successfully written into the buffer.

The read pointer behaves differently for the KV Ring Buffer and the KV Output Buffer. It is controlled by a parameter that can be specified when generating Verilog

code. For the KV Ring Buffer, the Controller increments the read pointer when the KV pair is transferred to the KV Output Buffer. The read pointer in the KV Output Buffer is incremented automatically when the Encoder successfully transfers the KV pair via AXI Stream. The buffer is considered full when the write pointer approaches the read pointer. The diagram of the buffer can be found in the Figure 3.7.

In addition, the KV Ring Buffer outputs three status signals: *empty*, *half full*, and *full*. The Controller uses the signals to determine when to start or stop the comparison of keys in the Merger.

Each KV pair stored in the buffer memory has four parts. The below list describes those parts for 32-bit bus width:

- **Key length**, a 32-bit metadata number that indicates how many 32-bit chunks of a key part are stored;
- **Value length**, a 32-bit metadata number that indicates how many 32-bit chunks of a value part are stored;
- **Key chunks**, one or more 32-bit chunks that together constitute a key part of KV pair;
- **Value chunks**, one or more 32-bit chunks that together constitute a value part of KV pair;

Note that key and value lengths are stored in separate 32-bit values in the memory. It differs from the metadata field described in the Section 3.4.1. This is due to legacy development, which is something to address in the design for future improvements.

For the production system, metadata fields will need to be extended. For example, to support tombstone records described in Section 2.2.1.

```
1 val memSize = depth * (maxKeySize + maxValueSize + maxMetadataSize)
2 val mem = SyncReadMem(memSize, UInt(busWidth.W))
```

Listing 3.2: An example of instantiating synchronous memory in Chisel3.

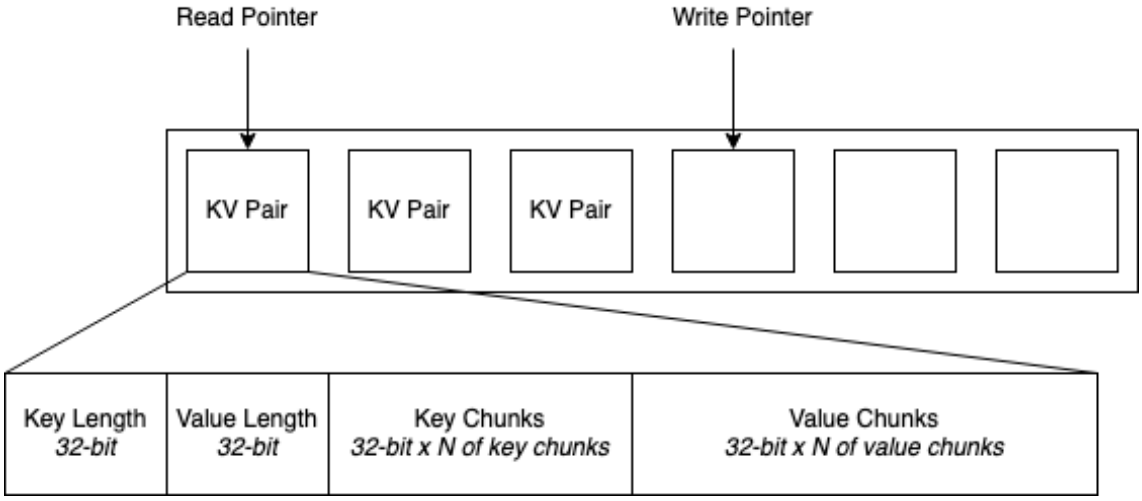

Figure 3.7: A diagram of a KV Ring Buffer in LSM-Compactron3000.

3.4.3 KV Transfer

The KV Transfer module performs two functions:

- **Transferring keys** from KV Ring Buffers to Key Buffer so the Merger can compare them;
- **Transferring KV pair** from selected KV Ring Buffer to KV Output Buffer when the Merger identifies the winner.

The module can only transfer keys from a single KV Ring Buffer at a time. This decision was made to match the reference architecture. From their analytical model [5, page 231], we can see that the Merger requires to transfer key data 5 times for a single comparison round. We deduced because they have four buffers, most likely, their KV Transfer implementation supports only loading data from a single buffer at a time. Their diagram supports the same conclusion.

The Controller is responsible for configuring necessary inputs for the KV Transfer. It is also possible to specify precisely using a mask from which buffers the KV Transfer should load key chunks. This is especially useful if only a subset of buffers is used, as it allows skipping unused buffers. This functionality is supported by the *NextIndexSelector* module described in Section 3.2.

3.4.4 Key Buffer

The Key Buffer temporarily stores *only* key chunks for the current comparison round. The Merger reads key chunks from the Key Buffer and compares them. It is similar to the KV Ring Buffer and KV Output Buffer in that it is used to decouple components and effectively pipeline data.

The Key Buffer loads key chunks into *rows*. Each row consist of *number of buffers* key chunks. Therefore, the maximum memory usage is *number of buffers* $*$ *maximum key size*. In the Figure 3.8, you can see an example memory layout in the Key Buffer in case 4 ways/buffers are used.

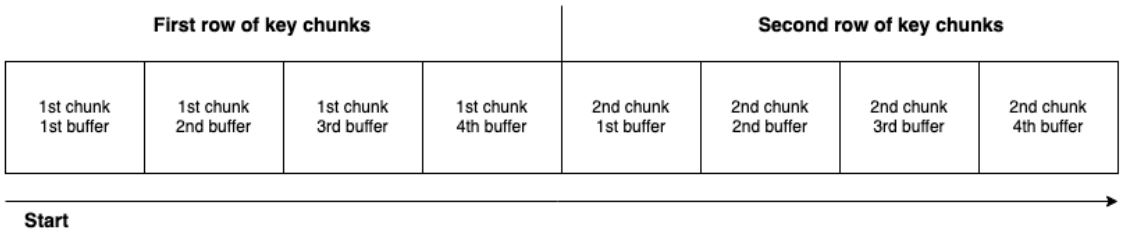

Figure 3.8: A memory layout of a Key Buffer in LSM-Compactron3000.

3.4.5 Merger

The Merger is a core module that performs comparison between keys to determine which KV pair goes next to the KV Output Buffer and eventually to the output SSTable. It consists of two submodules:

- **Key Chunks Comparator**, a *combinational* circuit that accepts key chunks one by one and determines a winner KV pair;
- **Merger**, a *sequential* wrapper around Key Chunks Comparator that loads key chunks from Key Buffer to registers and communicates with the Controller.

The comparison process follows a behaviour described in Section 2.2.3. A single round of comparison consists of comparing rows of key chunks until winner is found. It takes *number of buffers* cycles to load a row into registers inside the Merger. When all valid key chunks are loaded, the Key Chunks Comparator will determine whether there is a winner.

In the Figure 3.9, you can find a flow chart of the decision process happening in Key Chunks Comparator to select a winner key. It is important to note that the buffers

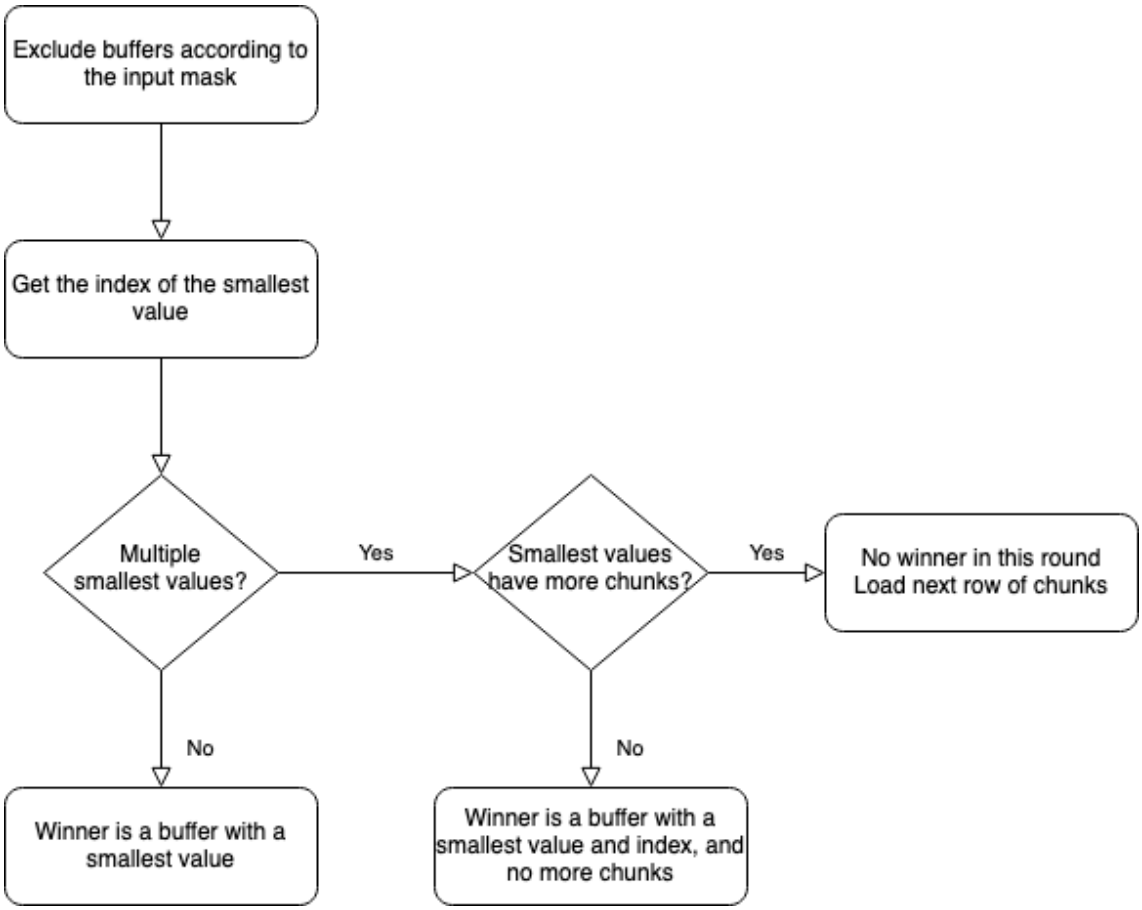

Figure 3.9: A flow chart comparison process in the Key Chunks Comparator.

with a smaller indexes are considered more up-to-date. Therefore, in case two exact same keys are compared the winner will be the buffer with a smallest index. If no winner is found for the current row, the Merger loads the next row of key chunks and repeats the process. The worst case scenario to determine a winner is when we need to compare all rows of key chunks but, often, we can conclude earlier. When the Merger determines a winner, it stops and await until the Controller resets it for the next round of comparison.

In the Listing 3.3, you can see combinational code for the Key Chunks Comparator module. It uses generators to build hardware components depending on the parameters specified.

In some cases, not all buffers must be considered during the comparison. For example, suppose one of the buffers is fully processed, and no more data is available. In that case, the Controller sets a mask at the beginning of the comparison to notify the Merger which buffers are included in the comparison. In the code, at the beginning, *maskIn* is a mask provided to the Merger by the Controller and passed to the Key Chunks Comparator. Together with other inputs, such as a row of key chunks,

the Key Chunks Comparator checks for a winner. After each comparison, even if a winner is not found for this row, the module provides an updated mask, *maskOut*. The updated mask will be used as a new mask in *maskIn* for the next row. The process continues until the winner is found. The Merger guarantees to find a winner. The *maskOut* values are used to determine which buffers read pointers need to be advanced after the comparison round. The *winnerIndex* will supply an index of a buffer from which the KV pair should be transferred to the KV Output Buffer.

In the Listing 3.4, you can see how the Key Chunks Comparator module excludes buffers. If the *maskIn* value for a specific buffer is *0*, the MSB of the key chunk value is extended with *1*. Otherwise, the MSB is extended with *0*. It ensures that any current, previous, or random data for excluded buffers will always be larger and not affect the selection of the smallest index.

```
1  class KeyChunksComparator(busWidth: Int = 4, n: Int = 4) extends Module {
2      val io = IO(new Bundle {
3          val in = Input(Vec(n, UInt(busWidth.W)))
4          val maskIn = Input(UInt(n.W))
5          val lastChunksMask = Input(Vec(n, Bool()))
6
7          val maskOut = Output(UInt(n.W))
8          val haveWinner = Output(Bool())
9          val winnerIndex = Output(UInt(log2Ceil(n).W))
10     })
11     val modifiedInputs = Wire(Vec(n, UInt((busWidth + 1).W)))
12
13     io.in.zipWithIndex.map { case (input, index) =>
14         modifiedInputs(index) := Mux(io.maskIn(index) === 1.U, "b0".U ##
       input, "b1".U ## input)
15     }
16
17     val smallestIndex = Wire(UInt(log2Ceil(n).W))
18     val indexesArray = VecInit(Seq.tabulate(n)(i => i.U(log2Ceil(n).W)))
19
20     smallestIndex := indexesArray.reduce { (indexA, indexB) =>
21         Mux(modifiedInputs(indexA) <= modifiedInputs(indexB), indexA,
       indexB)
22     }
23
24     val equalityMask = Wire(Vec(n, Bool()))
25
```

```
26      for (i <- 0 until n) {
27          equalityMask(i) := modifiedInputs(smallestIndex) ===
    modifiedInputs(i)
28      }
29
30      val countOnes = PopCount(equalityMask.asUInt)
31      val hasOnlyOneOne = countOnes === 1.U
32
33      val andResult = equalityMask.asUInt & io.lastChunksMask.asUInt
34      val andResultEqualsZero = andResult === 0.U
35
36      io.maskOut := Mux(hasOnlyOneOne || andResultEqualsZero,
    equalityMask.asUInt, andResult)
37      io.haveWinner := hasOnlyOneOne || ~andResultEqualsZero
38
39      io.winnerIndex := PriorityEncoder(io.maskOut)
40  }
```

Listing 3.3: A Chisel3 code for the Key Chunks Comparator module.

```
1       io.in.zipWithIndex.map { case (input, index) =>
2           modifiedInputs(index) := Mux(io.maskIn(index) === 1.U, "b0".U ##
    input, "b1".U ## input)
3       }
```

Listing 3.4: A part of the Key Chunks Comparator code that excluded buffers from smallest index search.

3.4.6 Controller

The Controller is responsible for coordination of the modules. It receives commands via AXI Lite interface from the software. In the Figure 3.10, you can find a diagram that describes steps taken by the Controller during the compaction process.

Initially, the Controller module awaits a *start* signal that is set by modifying the register value from the software via AXI Lite. The start signal notifies the module that incoming data needs to be compacted. The KV pairs in our custom format are transferred from the memory to the KV Ring Buffers (we can call them *input buffers*) via AXI Stream. The Controller waits until *active input buffers* are half full. The *active input buffer* means that an input buffer is not empty or did not

see a last KV pair. Then, the Controller sets the Merger by providing an initial mask (discussed in Section 3.4.5) and instructs KV Transfer to transfer key chunks from the active input buffers to the Key Buffer. The Merger starts reading rows of key chunks from the Key Buffer and compares them. When a winner is found, the Merger notifies the Controller. Based on the results, the Controller transfers the winner KV pair from the input buffer to the KV Output Buffer. After that, the Controller advances the read pointers for selected input buffers. Multiple read pointers can be advanced simultaneously if KV pairs from multiple buffers have the exact key. In this case, the most recent KV pair is output, and the older ones are disregarded.

After this, the process for the following KV pairs in input buffers begins again. The Controller does not always wait for buffers to be half-filled. Suppose the Decoder module indicates no more data will be received for a specific buffer. In that case, the Controller no longer needs to wait to fill this particular buffer and is allowed to process KV pairs until the input buffer is empty. If only one buffer is active, the Controller transfers all its KV pairs to the KV Output Buffer.

When no active buffers are left, and the KV Output Buffer is empty, the Controller sends a signal to the Encoder to send the *tlast* signal on the AXI Stream interface, indicating no more data is available for transfer.

The Controller also has a *busy* status output that the software can query via AXI Lite to determine if the compaction is finished.

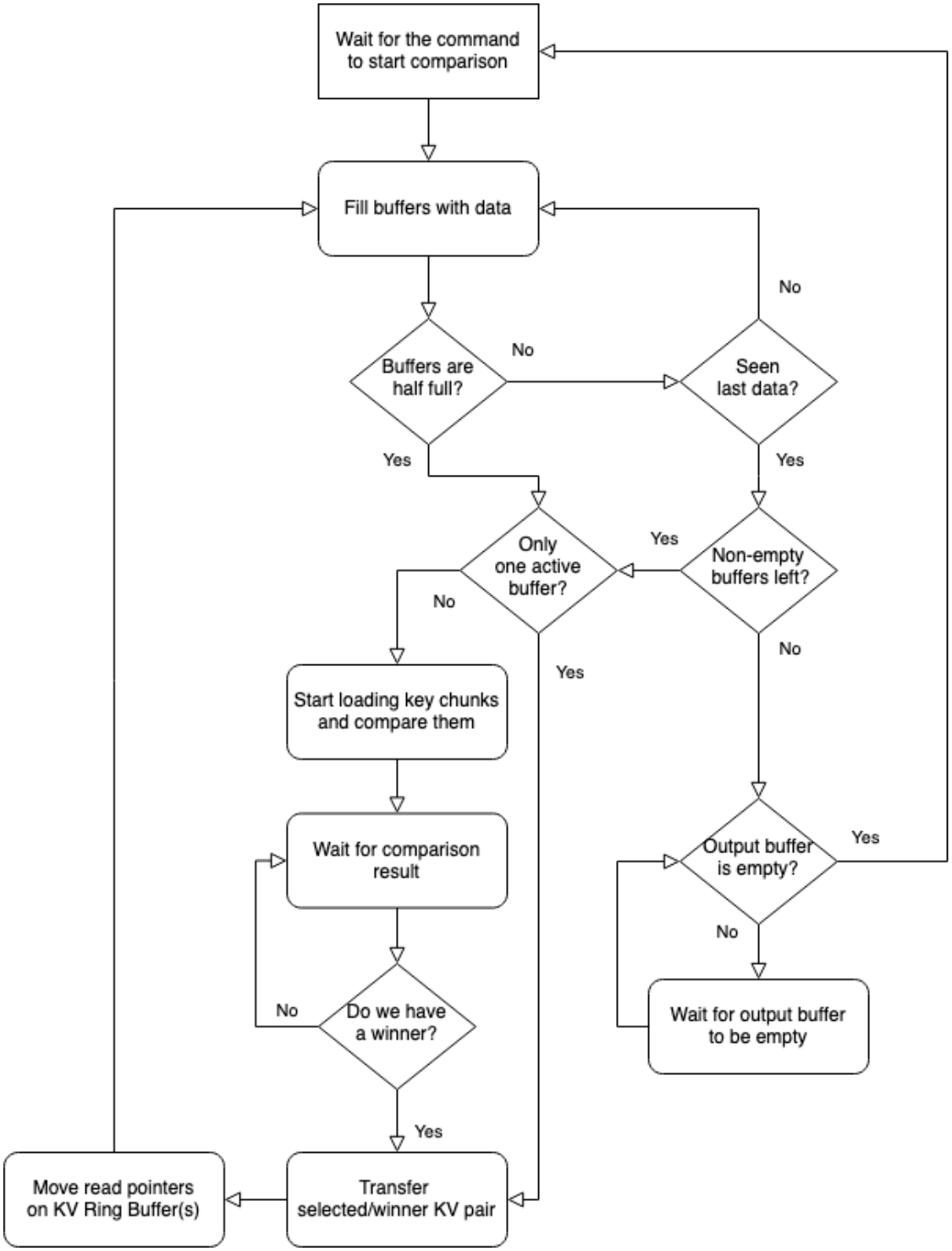

Figure 3.10: A flow chart of the Controller process.

3.5 Software implementation

The purpose of the software is to test the correctness of the hardware. It is split into two groups that are available in the GitHub repository:

- *Python scripts* used to generate random KV pairs and convert them into our custom SSTable format that the CU can directly ingest;
- *C code* used to configure DMA controllers, start data transfer, receive compacted data back, and validate it.

3.5.1 Data preparation

In this part, we will talk about how test data was prepared. We will not discuss how Python scripts work in detail, as it is not essential for this thesis. But, we will share details about how data is formatted.

The ASCII encoding is used for simplicity when generating KV pairs. One Python script generates pseudo-random KV pairs as strings. In the Listing 3.5, you can find a sample of KV pairs. The key part is separated from the value part with ":" (semicolon).

```
AAk:PpKZNNQkEI
AKI:JTCSGrdetD
BeVcerzBq:HvtNieXemG
CsWeLOLBY:KMDsrdIfzw
EEpnMAKos:cvqCTeGiEb
```

Listing 3.5: A sample of KV pairs generated.

The other Python script converts data into a bit-packed format ready to be ingested by the accelerator. In the Listing 3.6, you can see an example of a first KV pair taken from the Listing 3.5. The KV pair is a bit packed. The first value is the metadata field discussed in the Section 3.4.1. Key and value chunks follow afterward. In our chosen bus width of 32-bit, we can pack 4 ASCII characters (8 bits per character).

```
0x00030100  // metadata field, 1 key chunk and 3 value chunks
0x41416B00  // key chunk, 0x41 = A, 0x6B = k
0x50704B5A  // value chunk
0x4E4E516B  // value chunk
0x45490000  // value chunk
```

Listing 3.6: A sample of bit packed KV pair.

3.5.2 Control flow

This short part will discuss the software's steps to compact data. To control the accelerator, we developed C code that is deployed alongside the accelerator. This solution depends on the type of hardware you are targeting. In our case, we have an FPGA board and CPU together on a chip.

Below are the general steps that the C code performs to compact data:

1. Software loads SSTable files into a memory;
2. Software allocates memory for the merge results;
3. Software writes to control registers inside CU that there is data incoming;
4. Software orders DMA controllers to start transferring SSTable files from memory to CU;
5. CU unit accepts data, processes it, and outputs the merged SSTable file to memory via the DMA controller;
6. Software polls status register of CU to determine when a merge is completed;
7. When CU is done, the software can process the merged data further as necessary.

More details can be found in the code source in the GitHub repository.

Chapter 4

Evaluation

In this chapter, we discuss how the Compaction Unit (CU) was tested and verified, and then we will evaluate CU using microbenchmarks. We discuss the results, bottlenecks of our solution, and possible improvements.

4.1 Unit and Integration Tests

Throughout the development of LSM-Compactron3000, unit and integration tests were heavily used. The tests are written in Scala using the ChiselTest library. We have written a unit test for each module in CU to confirm that the module works as expected. If a bug was found, the test case was added to replicate the bug and verify, after the fix, that the bug was gone. Most tests only validate a single module, but some test multiple modules together and are called integration tests. Using unit and integration tests allowed us to confidently refactor the code and add new features in the later stages of the development. It saved us a lot of time that would have been spent on debugging and manually verifying the correctness of the design.

In total, we have 37 tests. The ratio of the test code to the hardware code is 2.5:1. We can see an example of the test results in Figure 4.1. The green color indicates that the test is passed. As visible at the bottom of the figure, all tests take around 13 seconds to run. Tests are grouped into suites (suffixed with **Spec*), and each suite tests a different part of the design. One of the most critical test suites is *CompactionUnitSpec*. It tests the whole CU from ingesting data via AXI Stream into Decoders to outputting the compacted data from Encoder.

```
[info] DummyEncoderSpec:
[info] - Transfer KV Pair from output buffer to AXI Stream interface
[info] ControllerSpec:
[info] - Should correctly control modules from beginning to the end
[info] KvTransferSpec:
[info] - Should stop loading key chunks when requested
[info] - Should load a single key chunk with delayed ready
[info] - Should load a single key chunk with on-time ready
[info] - Should transfer key chunks from selected buffers to deq
[info] - Should output deq.valid == false when clearing Key Buffer
[info] - Should transfer KV pair from selected buffer to deq output
[info] KeyBufferSpec:
[info] - Should load key chunks rows and read them back
[info] - Should write key chunks rows with some chunks shorter and read them back
[info] - Should write key chunks rows and read them back with no delay
[info] - Should reset buffer
[info] - Should load key chunks rows and read them back with delay
[info] KvRingBufferSpec:
[info] - Check default output values
[info] - Should not be ready when full
[info] - Should not overwrite memory if valid is not set and buffer is full
[info] - Should put a single KV value and read it back
[info] - Should read only key
[info] - Should write two KV pairs and read one back
[info] - Should write two KV pairs and read two back
[info] - Should move to read the next KV pair
[info] - Should wait until 'ready' asserted to provide next Key chunk
[info] - Should wait until 'ready' asserted to provide next Key or Value chunk
[info] - Should move read pointer when requested
[info] - Should reset reading to the beginning of KV pair when requested
[info] NextIndexSelectorSpec:
[info] - Different inputs produce correct outputs
[info] TopKvTransferSpec:
[info] - Should load key chunks from all mock buffers
[info] DummyDecoderSpec:
[info] - Transfer last KV Pair from AXI to buffer
[info] DummyKvPairFifoSpec:
[info] - AXI S to KV output buffer to AXI M
[info] MergerSpec:
[info] - One buffer wins and advances
[info] - Some buffers are equal, both advance
[info] - Works correctly after having a winner and starting a new comparison round
[info] KeyChunksComparatorSpec:
[info] - Different input combinations produce correct results
[info] KvRingBuffersAndKvTransferAndKeyBufferSpec:
[info] - Should transfer rows of key chunks from buffers
[info] - Should correctly select winner buffers and read KV pairs from output buffer
[info] CompactionUnitSpec:
[info] - Two buffers, short compact
[info] - Two buffers, long compact
[info] Run completed in 13 seconds, 457 milliseconds.
[info] Total number of tests run: 37
[info] Suites: completed 13, aborted 0
[info] Tests: succeeded 37, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

Figure 4.1: An example of tests for CU in LSM-Compactron3000.

4.2 Setup

For our microbenchmarks, we compare CU from LSM-Compactron3000 (called *accelerator CU* for the rest of the chapter) and the simple counterpart CU written in C with no SIMD support (called *software CU* for the rest of the chapter). They receive the same input, two SSTables in our custom format, and produce the compacted output in the same format. We measure the *relative* time expressed as a number of clock cycles that lapsed from the beginning of the compaction to the end within the CPU. It is essential to mention that our setup compares only the compaction process. In addition, we are using a custom format for SSTables, not the formats used in the production databases. The compaction is a part of the whole database system, and to get the actual performance and conclusive results, we would need to benchmark all parts together. Therefore, we advise treating the results as informational, not as a final conclusion.

The benchmarking is done on the Zybo Zynq-7000 board (a picture can be found in the Figure 4.2). The board has 512 MB of DDR3 RAM and tightly integrates an ARM Cortex-A9 processor with Xilinx 7-series FPGA. The CPU has two cores. Both accelerator and software CUs run on a bare-metal system. For a production system, a Linux DMA driver will likely be required. The software CU is run on a single core and cannot be preempted. The clock frequency of the CPU is set to 650 MHz, and the clock frequency of the accelerator CU in the FPGA is set to 100 MHz.

The test data is generated using the Python scripts described in the Section 3.5.1. In the Table 4.1, we can see parameters of generated test data and clock cycle results for accelerator and software CUs. The first three columns describe the parameters of the test data. *Parameters* is a string that describes parameters used to generate the KV pairs. For example, *K_3_10_V_10* means that the keys are between 3 and 10 ASCII characters long, and the values are 10 ASCII characters long. *N1* and *N2* are the number of KV pairs in the first and second SSTable, respectively. *Similarity* is the ratio of the duplicate keys in both SSTables. The value of 0.6 means both SSTables share 60% of KV pairs. The last two columns show the time lapsed for our design and software counterpart.

The maximum key length supported is 32 ASCII characters, and the maximum value length is 112 ASCII characters. This is due to the limited resources of the FPGA.

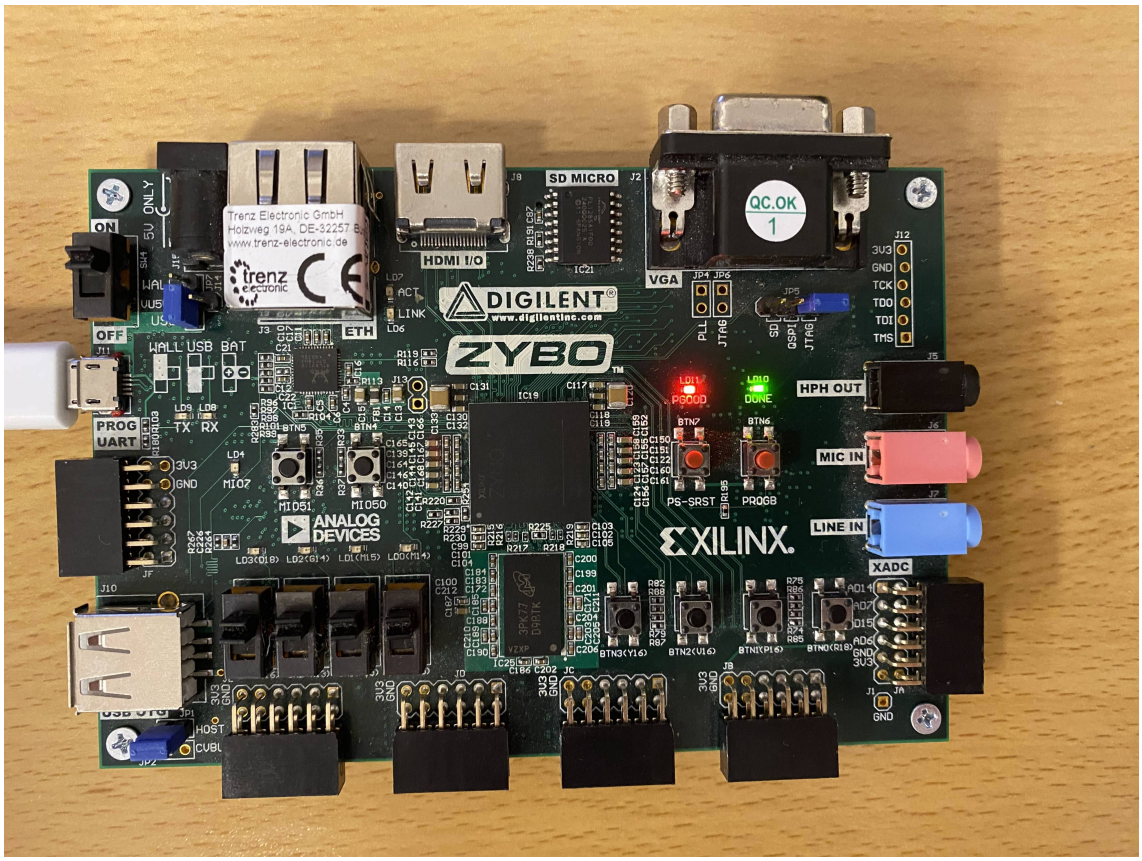

Figure 4.2: A picture of Zybo Zynq-7000 board.

4.3 Results

The results plot based on Table 4.1 is shown in the Figure 4.3. The results show that the LSM-Compactron3000 accelerator is slower than the CPU implementation by 1.5 to 2 times across all test points. It is important to remember that the clock frequency of the CPU is 650 MHz vs. 100 MHz for the FPGA. If we consider that and equalize the clock frequencies, the accelerator should be faster by around a factor of 3, which means the accelerator requires fewer operations to compact the SSTables. This is a good sign and expected as the accelerator is specialized hardware, and the CPU is a general-purpose processor. However, in production deployment, downclocking the CPU is not feasible. The obvious way to speed up the accelerator is to adapt design to run on higher clock frequencies. This will require analysing design timing and fixing timing violations. We will discuss some other ways to improve the performance later in the section.

Besides the accelerator being overall slower than its software counterpart, it leaves the CPU idle and allows the CPU to perform other tasks, such as serving requests to database clients.

Parameters	N1	N2	Similarity	Clk. cycles, accelerator	Clk. cycles, software
K_3_10_V_10	50	30	0.6	15752	7220
K_3_10_V_50	50	30	0.6	20076	13410
K_3_10_V_100	50	30	0.6	25408	17726
K_10_20_V_10	50	30	0.6	16858	8414
K_10_20_V_50	50	30	0.6	21380	18852
K_10_20_V_100	50	30	0.6	26692	26224
K_20_32_V_10	50	30	0.6	18724	10654
K_20_32_V_50	50	30	0.6	23538	14894
K_20_32_V_100	50	30	0.6	28986	20126

Table 4.1: The parameters of the test data and the benchmark results.

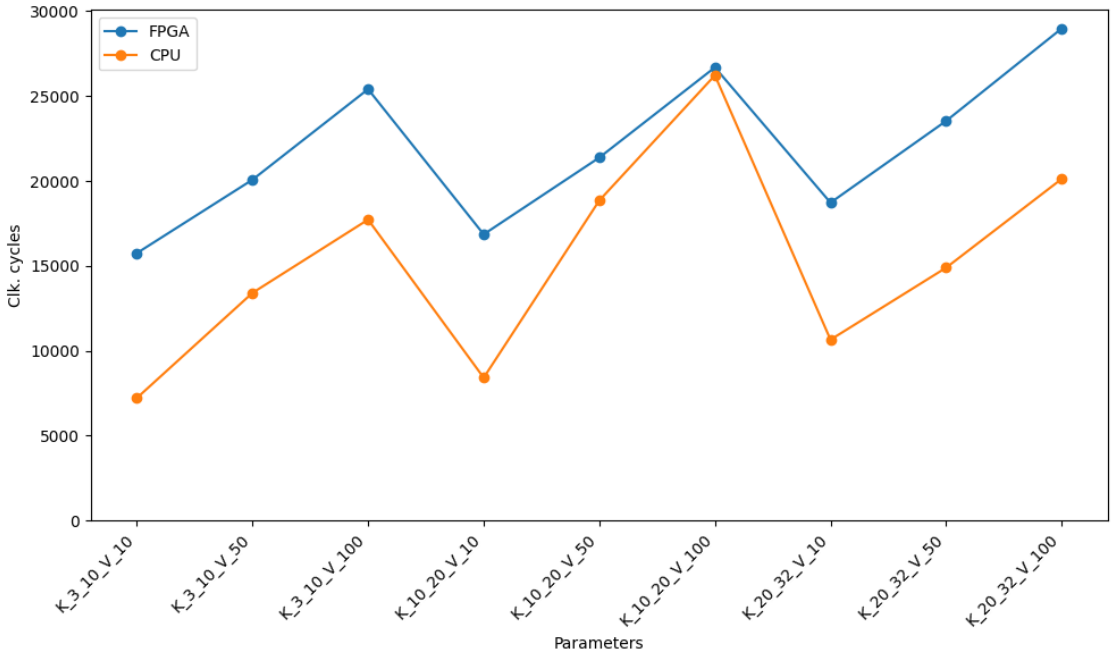

Figure 4.3: A plot of the FPGA vs CPU results is presented in Table 4.1. The lower, the better.

In the Figure 4.4, we can see the waveform output of the Encoder module. The Encoder's input is directly connected to the KV Output Buffer. The waveform shows that the Encoder has a significant delay between two consecutive outputs (around 20 clock cycles) in the later stages of compaction. This is caused by the KV Transfer module being unable to fill the KV Output Buffer fast enough.

When a winner KV pair is selected, the KV Transfer transfers the pair from the KV Ring Buffer to the KV Output Buffer. At this time, no comparison in the Merger can occur. While comparison is stalled, the input buffers are being filled with new KV pairs. It effectively makes the accelerator speed bounded by the KV Transfer

and Merger modules similarly to what Zhang et al. [5] observed. The Figure 4.5 waveform, acquired during the integration test, illustrates this problem in details.

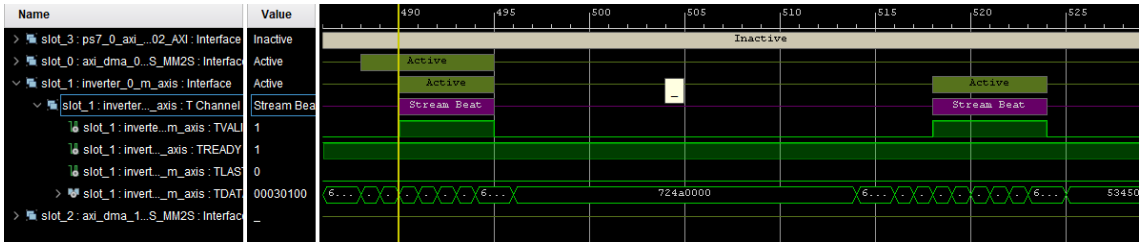

Figure 4.4: A waveform of the stalled Encoder output.

To see where the data is transferred on the waveform, follow the *valid* signals for each module. On top of the waveform, there are multiple markers.

- Between markers A and B, KV Transfer is transferring key chunks for comparison;
- When a winner pair is selected, it is transferred to the KV Output Buffer between markers C and D;
- The markers E and F indicate the next comparison round.

If we count the amount of pairs processed between markers A and D, buffers 1 and 2 were filled with around 3 new KV pairs each while only one KV pair was transferred to the KV Output Buffer.

This bottleneck can be potentially improved in a few ways. We could allow to read data in the KV Ring Buffer from two different memory locations at the same time. It can be done with two memory units that receive duplicated writes. This would allow KV Transfer to transfer the winner KV pair while filling the Key Buffer with key chunks of the next KV pairs for comparison. However, this solution will increase the resource utilization of the FPGA.

Building on a previous idea, we could predict the next KV pair that the Merger will select, similar to how modern CPUs predict branching instructions. With the ability to read from two different memory locations, the predicted KV pair can be transferred while the comparison is ongoing. If the prediction is correct, the KV pair is already in the KV Output Buffer, and the Encoder does not stall. If the prediction is incorrect, the transfer operation can be aborted, and the correct KV pair can overwrite the incorrect one. In theory, adding prediction should not add any performance penalties for a pair transfer.

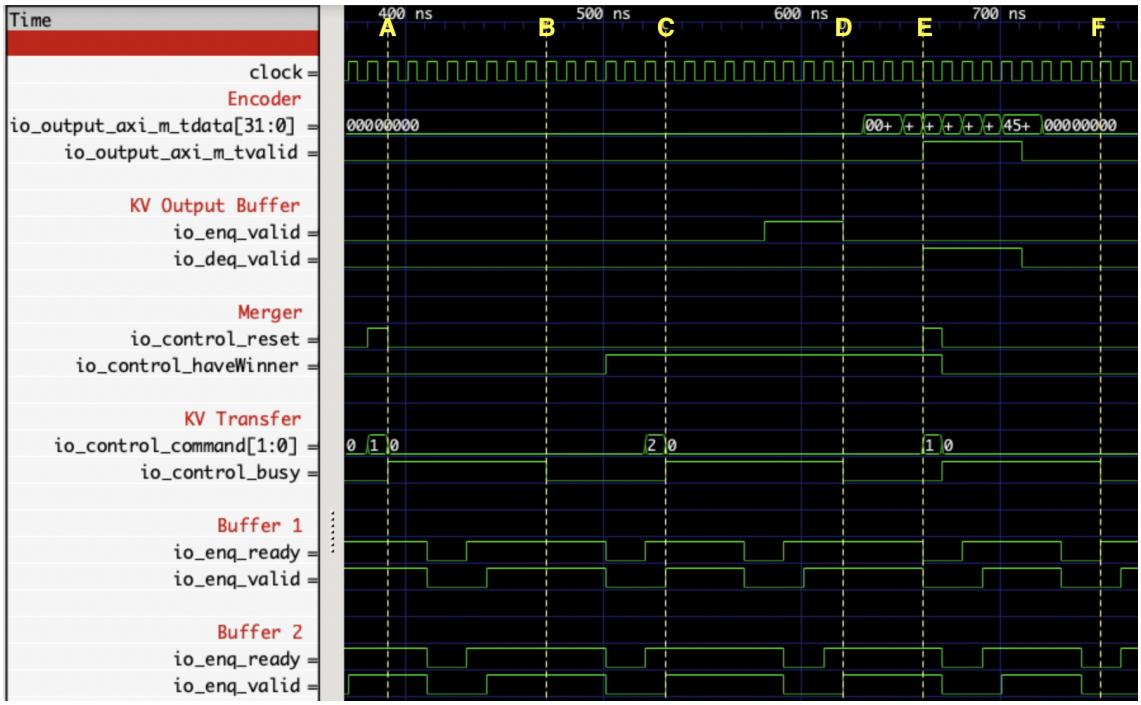

Figure 4.5: A waveform from CU integration test.

Chapter 5

Conclusion

In this thesis, we proposed and implemented an FPGA-based compaction accelerator called *LSM-Compactron3000*. It is the first open-source project of its kind, released under a permissive MIT open-source license on GitHub. The repository can be found at `https://github.com/VMois/LSM-Compactron3000`. We encourage the reader to check the project and contribute to it.

We compared the performance of the LSM-Compactron3000 with the CPU-only baseline. The results indicate that the accelerator requires less number of operations to compact the data. Still, due to the difference in clock frequencies between the CPU and the FPGA, the accelerator is overall slower than the CPU-only baseline by around 1.5-2 times. Despite being slower, the accelerator can help to offload the computations and free up CPU resources for other tasks. In addition, we analyzed current bottlenecks in the Section 4.3 and proposed possible solutions to improve the accelerator's performance. The project's open-source nature makes it easier to build upon the results.

5.1 Future work

It is essential to discuss the future work that can be done to improve the *LSM-Compactron3000* project.

As mentioned, the Encoder and Decoder modules are specific to the database. In our case, we developed a custom format for SSTables, but to make the project more useful, we need to support formats used by production LSM-tree-based KV databases. We believe that the core of the accelerator (all modules except Encoder and Decoder) can be reused. Ideally, adding support to a new database should only require the development of new Encoder and Decoder modules.

To develop a generic solid core for the accelerator, we need to understand the requirements of different production databases. For example, from the Section 2.2.1, we know that databases need to store tombstone to mark deleted records. What other metadata do LSM-tree-based KV databases store? Can we find other similarities? Extracting common features from different formats and adding support to the accelerator's core is a good starting point for future work. The possible approach could be to develop a proof of concept implementation for some of the popular databases, for example, RocksDB [4]. The experience gained during this development, combined with research on other production databases, will help adjust the design of *LSM-Compactron3000* and make it more generic. After the strong core is developed, we can proceed with the performance improvements and add support for more databases.

5.2 Economics

To gain the benefits of FPGA-based accelerators, *LSM-Compactron3000* or any other compaction accelerator needs to be production-ready. This requires a lot of effort and resources. In this section, we will share our opinion on what type of companies can benefit from the hardware accelerators and which companies are unlikely to gain much.

First, it is crucial to understand that the LSM-compaction accelerator is highly specialized hardware. Not every company using LSM-tree databases operates on such a large scale that they need to use hardware accelerators. Therefore, there is no economic sense to produce such accelerators in big quantities. We can exclude the possibility of developing such hardware as ASIC. The flexible FPGAs provide a better alternative and can be reused for other projects if necessary. But, even FPGAs are far from acceptable solutions for most companies. For many companies, their main business does not involve developing hardware. In many cases, even software is only a means to an end.

We should look at companies specializing in developing databases or offering database services. But we need to distinguish even between the companies in that group. Some companies are developing databases but do not operate on their infrastructure, for example, ScyllaDB [32]. Some companies are developing databases and operating on their infrastructure, for example, Alibaba, AWS, and Google Cloud.

As discovered by Zhang et al. [5], their FPGA implementation increased the energy efficiency by 31.7%. For companies that do not operate on their infrastructure, besides the potential performance improvements, the FPGA accelerators are not worth

the effort of spending resources on the hardware and skilled engineers. They are more likely to focus on developing new features and maximizing software optimizations in their databases. On the other hand, energy efficiency combined with performance improvements can motivate companies that operate on their infrastructure, usually cloud providers. We believe that the FPGA-based compaction accelerators might benefit the cloud providers with LSM-tree database service offerings.

Many cloud providers also offer access to FPGAs. Some companies can develop FPGA compaction solutions for databases and sell them as a service for other businesses that operate in the cloud. If the onboarding does not require much effort, companies working with big data might be interested in subscribing.

More calculations are required to determine the economic feasibility of the accelerators, but this can be another direction for future work.

Acknowledgements

I would like to sincerely thank my supervisor, Dr. Ákos Nagy, whose invaluable guidance was instrumental in shaping the overall direction of this thesis and proved indispensable in resolving specific challenges encountered along the way. His expertise and willingness to engage with the intricacies of the project were essential in overcoming hurdles and refining the quality of this work.

Bibliography

[1] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996. ISSN 1432-0525. DOI: 10.1007/s002360050048. URL https://doi.org/10.1007/s002360050048.

[2] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4:1–4:26, June 2008. ISSN 0734-2071. DOI: 10.1145/1365815.1365816. URL https://doi.org/10.1145/1365815.1365816.

[3] LevelDB. URL https://github.com/google/leveldb.

[4] RocksDB, . URL https://rocksdb.org/.

[5] Teng Zhang, Jianying Wang, Xuntao Cheng, Hao Xu, Nanlong Yu, Gui Huang, Tieying Zhang, Dengcheng He, Feifei Li, Wei Cao, Zhongdong Huang, and Jianling Sun. FPGA-Accelerated compactions for LSM-based Key-Value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 225–237, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/zhang-teng.

[6] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 651–665, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450356435. DOI: 10.1145/3299869.3314041. URL https://doi.org/10.1145/3299869.3314041.

[7] Xuan Sun, Jinghuan Yu, Zimeng Zhou, and Chun Jason Xue. Fpga-based compaction engine for accelerating lsm-tree key-value stores. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1261–1272, 2020. DOI: 10.1109/ICDE48307.2020.00113.

[8] Zigang Zhang, Yinliang Yue, Bingsheng He, Jin Xiong, Mingyu Chen, Lixin Zhang, and Ninghui Sun. Pipelined compaction for the lsm-tree. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 777–786, 2014. DOI: 10.1109/IPDPS.2014.85.

[9] Hui Sun, Wei Liu, Jianzhong Huang, and Weisong Shi. Collaborative Compaction Optimization System using Near-Data Processing for LSM-tree-based Key-Value Stores. *Journal of Parallel and Distributed Computing*, 131:29–43, 2019. ISSN 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.04.011. URL https://www.sciencedirect.com/science/article/pii/S0743731518308645.

[10] Chen Ding, Jian Zhou, Jiguang Wan, Yiqin Xiong, Sicen Li, Shuning Chen, Hanyang Liu, Liu Tang, Ling Zhan, Kai Lu, and Peng Xu. Dcomp: Efficient offload of lsm-tree compaction with data processing units. In *Proceedings of the 52nd International Conference on Parallel Processing*, ICPP '23, page 233–243, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400708435. DOI: 10.1145/3605573.3605633. URL https://doi.org/10.1145/3605573.3605633.

[11] Peng Xu, Jiguang Wan, Ping Huang, Xiaogang Yang, Chenlei Tang, Fei Wu, and Changsheng Xie. Luda: Boost lsm key value store compactions with gpus, 2020.

[12] Hui Sun, Wei Liu, Jianzhong Huang, Song Fu, Zhi Qiao, and Weisong Shi. Near-data processing-enabled and time-aware compaction optimization for lsm-tree-based key-value stores. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362955. DOI: 10.1145/3337821.3337855. URL https://doi.org/10.1145/3337821.3337855.

[13] Hui Sun, Qiang Wang, Yin Liang Yue, Yuhong Zhao, and Song Fu. A storage computing architecture with multiple NDP devices for accelerating compaction performance in LSM-tree based KV stores. *Journal of Systems Architecture*, 130:102681, 2022. ISSN 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2022.102681. URL https://www.sciencedirect.com/science/article/pii/S1383762122001874.

[14] Hui Sun, Bendong Lou, Chao Zhao, Deyan Kong, Chaowei Zhang, Jianzhong Huang, Yinliang Yue, and Xiao Qin. An asynchronous compaction acceleration scheme for near-data processing-enabled lsm-tree-based kv stores. *ACM Trans. Embed. Comput. Syst.*, sep 2023. ISSN 1539-9087. DOI: 10.1145/3626097. URL https://doi.org/10.1145/3626097. Just Accepted.

[15] Jonas Dann, Daniel Ritter, and Holger Fröning. Non-relational databases on fpgas: Survey, design decisions, challenges. *ACM Comput. Surv.*, 55(11), feb 2023. ISSN 0360-0300. DOI: 10.1145/3568990. URL https://doi.org/10.1145/3568990.

[16] Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, 2017. ISBN 978-1-4919-0310-0. URL https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/.

[17] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. Compactionary: A dictionary for lsm compactions. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2429–2432, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. DOI: 10.1145/3514221.3520169. URL https://doi.org/10.1145/3514221.3520169.

[18] Chisel3, . URL https://www.chisel-lang.org/.

[19] Scala. URL https://www.scala-lang.org/.

[20] Rocket Chip, RISC-V SoC Generator, . URL https://github.com/chipsalliance/rocket-chip.

[21] Experiences Building Edge TPU with Chisel - Google. URL https://www.youtube.com/watch?v=x85342Cny8c.

[22] Sifive. URL https://www.sifive.com/.

[23] Chiseltest, . URL https://github.com/ucb-bar/chiseltest.

[24] Verilator. URL https://www.veripool.org/verilator/.

[25] Mufan Xiang, Yongjian Li, and Yongxin Zhao. Chiselfv: A formal verification framework for chisel. In *2023 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, April 2023. DOI: 10.23919/DATE56975.2023.10137221.

[26] Paul Lennon and Richard Gahan. A comparative study of chisel for fpga design. In *2018 29th Irish Signals and Systems Conference (ISSC)*, pages 1–6, 2018. DOI: 10.1109/ISSC.2018.8585292.

[27] Matti Käyrä and Timo D. Hämäläinen. A survey on system-on-a-chip design using chisel hw construction language. In *IECON 2021 - 47th Annual Conference of the IEEE Industrial Electronics Society*, pages 1–6, Oct 2021. DOI: 10.1109/IECON48115.2021.9589614.

[28] CIRCT compiler project. URL https://circt.llvm.org/.

[29] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, pages 2–14. IEEE Press, 2021. ISBN 9781728186139. DOI: 10.1109/CGO51591.2021.9370308. URL https://doi.org/10.1109/CGO51591.2021.9370308.

[30] TensorFlow MLIR. URL https://www.tensorflow.org/mlir.

[31] Mojo programming language. URL https://www.modular.com/mojo.

[32] ScyllaDB. URL https://www.scylladb.com/.