AN OPEN SOURCE FRAMEWORK FOR OFFLOADING BIG DATA AND AI TASKS (OFFLOAD) TO HETEROGENEOUS COMPUTE UNITS

A Thesis in

Electrical & Computer Engineering

Presented to the Faculty of the University

of Missouri–Kansas City in partial fulfilment of the requirements for the degree

MASTER OF SCIENCE

by

Satya Sai Siva Rama Krishna Akula

Kansas City, MO, USA

Kansas City, Missouri 2024

ABSTRACT

The ever-increasing demands of artificial intelligence (AI) and big data processing have spurred the rapid development of novel hardware architectures specifically designed for computationally intensive tasks. Alongside these advancements, software solutions are emerging to exploit this specialised hardware by offloading tasks. However, proprietary software often necessitates a substantial learning curve for users, hindering widespread adoption and flexibility.

This paper proposes OFFLOAD, an open-source, hardware-agnostic software-hardware framework. OFFLOAD facilitates the distribution of tasks across diverse hardware units, encompassing both cutting-edge accelerators and existing system-on-chip (SoC) architectures. Our framework seamlessly integrates with popular databases and application development tools. Through the utilisation of multi-level abstractions implemented at the compiler, operating system, and driver levels, OFFLOAD translates high-level code and data into hardware-optimised binary instructions. To the best of our knowledge, OFFLOAD represents a ground-breaking approach within this domain.

The feasibility of OFFLOAD is demonstrably validated by its integration with prevalent tools such as MySQL, Apache Spark, and Apache Arrow within a user-friendly Python environment. Subsequently, tasks are offloaded for execution on hardware leveraging memory-mapped I/O. This is exemplified by integrating OFFLOAD with Raspberry Pi devices, showcasing the entire workflow from software-based data query to hardware execution.

Index Terms: Distributed computing, Hardware accelerators, Custom-designed hardware network, Big data analysis, Machine learning.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Computing and Engineering, have examined a thesis titled "AN OPEN SOURCE FRAMEWORK FOR OFFLOADING BIG DATA AND AI TASKS (OFFLOAD) TO HETEROGENEOUS COMPUTE UNITS" presented by Satya Sai Siva Rama Krishna Akula, a candidate for the Master of Science degree, and hereby certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Rahman Mostafizur, Ph.D., Committee Chair

Department of Division of Energy and Systems, UMKC

Preetham Goli, Ph.D. Committee Member 1

Department of Division of Energy, matter, and systems, UMKC

Mahbube K. Sidiki, Ph.D. Committee Member 2

Department of Computer Science Electrical Engineering, UMKC

CONTENTS

ILLUSTRATIONS

Figure **Page**

TABLES

Table **Page**

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude and appreciation to all those who have contributed to the completion of this thesis. Their support, guidance and encouragement were invaluable

throughout this journey. First and foremost, I am deeply grateful to my thesis advisor, Dr Rahman, for their unwavering guidance, expertise and continuous support. Their insightful feedback, patience and dedication have been instrumental in shaping this research project. I would like to extend my gratitude to the members of this thesis committee, Dr. Preetham Goli, Dr. Mahbube Sidikki for their valuable time, expertise, and constructive feedback. Their valuable insights and suggestions have contributed to the overall quality and rigour of this thesis.

My heartfelt appreciation goes to my family and friends for their unwavering support, encouragement and understanding throughout my academic journey. Their love, motivation, and belief in me have been my driving force, and I am forever grateful for that. Presence in my life. Their support has made this research experience memorable and enjoyable. Lastly, I would like to acknowledge the participants who generously volunteered their time and participated in the data collection process. Their contribution is invaluable and has made this research possible. To everyone who has played a part, big or small, in the competition of this thesis, I extend my deepest gratitude. Your support and encouragement have been invaluable and I'm truly grateful for the opportunity to have worked on this research project.

Thank you all.

CHAPTER 1
INTRODUCTION

The rapid evolution of artificial intelligence (AI) and big data has reshaped the requirements for computing resources, especially as enterprises and researchers tackle increasingly complex and data-intensive tasks. Traditional central processing units (CPUs) have reached their practical limits in

handling the concurrent and high-throughput demands posed by modern applications, resulting in delays, inefficiencies, and increased costs. Addressing these challenges requires a new approach that not only leverages specialized hardware accelerators [1] but also provides an open-source, flexible framework for developers to seamlessly integrate diverse hardware units without the constraints of proprietary solutions. This phenomenon is not only reshaping the landscape of computing but also driving innovation at an unprecedented pace. Let's delve deeper into why this competition is unfolding and the implications it carries.

Increasing Complexity of AI and Big Data Tasks: The complexity of AI and big data tasks is growing exponentially as applications expand in scale, sophistication, and impact across industries. Modern AI models, especially deep neural networks, require substantial computing power to handle massive datasets, intricate calculations, and intensive training processes. For example, training a large language model or a convolutional neural network for image recognition can involve billions of parameters, necessitating prolonged computation that could take days or weeks on conventional CPUs. The need to process and analyze vast volumes of data, often in real time, further complicates these tasks, as AI-driven applications like autonomous vehicles, real-time fraud detection, and personalized recommendations demand fast, precise computations with low latency. Big data applications are similarly affected, with the need to parse structured and unstructured data from diverse sources—social media, IoT sensors, transaction logs, and more—requiring a high degree of parallelism to efficiently manage and extract valuable insights. Traditional CPUs, built for general-purpose tasks, are typically inadequate for such workloads, as they lack the parallel processing capabilities and specialized architecture that high-performance AI and big data tasks require. Consequently, the escalating complexity of these workloads drives the demand for specialized hardware accelerators, like GPUs, TPUs, and FPGAs, which are tailored to handle parallel data processing, optimize matrix and tensor operations, and deliver the necessary speed and efficiency.

Example: Training a deep learning model on a massive dataset can take weeks or even months using conventional CPUs. With specialised accelerators like Google's TPU [3] or Nvidia's GPU, this process can be accelerated significantly, reducing training times to hours or even minutes.

Optimised Architectures for Specific Workloads: Unlike general-purpose CPUs, which are designed to handle a wide range of tasks, specialised accelerators are tailored for specific workloads. This specialisation allows for optimizations that can greatly enhance performance and energy efficiency for targeted tasks such as matrix multiplication or convolutional operations commonly found in AI algorithms. Current trends in hardware accelerators reflect the demand for specialized, high-performance solutions across fields like AI, big data, and edge computing, driving innovation in heterogeneous computing architectures. Tailored accelerators, such as GPUs, TPUs, and AI-specific ASICs, are gaining traction due to their efficiency in tasks like deep learning and data processing. Concurrently, edge computing accelerators address real-time processing needs in environments with constrained resources, such as autonomous vehicles and IoT applications, while early-stage quantum accelerators hold potential for solving complex, intractable problems. Domain-specific accelerators are also emerging, designed to optimize unique workloads in genomics, financial modeling, and cybersecurity. Open-source hardware efforts, like RISC-V and OpenPOWER, aim to reduce vendor lock-in, fostering flexibility and interoperability for diverse applications. Together, these trends underscore the shift towards specialized, scalable, and energy-efficient hardware solutions that support the growing computational requirements of modern applications.

Example: NVIDIA's GPUs are specifically optimized for matrix multiplication, a core operation in many deep learning algorithms, where large matrices are processed in parallel to accelerate tasks like neural network training. By tailoring the architecture with thousands of small, highly efficient cores that handle parallel tasks, GPUs achieve remarkable speed and efficiency in these computations, significantly outperforming traditional CPUs. In contrast, Google’s TPUs (Tensor Processing Units) are optimized for tensor operations, a type of multidimensional matrix computation frequently used in machine learning. TPUs feature a systolic array design that allows them to handle tensor-based operations even faster than GPUs, making them ideal for large-scale machine learning workloads. Meanwhile, FPGAs (Field-Programmable Gate Arrays) are optimized for flexibility, allowing developers to reconfigure the hardware to support specific tasks such as data encryption or image processing. This adaptability makes FPGAs suitable for applications that require low latency and high customization, like autonomous vehicles or high-frequency trading. Each of these architectures—GPUs for matrix-heavy calculations, TPUs for tensor operations, and FPGAs for adaptable workflows—demonstrates how hardware designs are evolving to meet the unique demands of diverse computational tasks, achieving performance gains tailored to specific use cases.

Scalability and Efficiency in Data Processing: Scalability and efficiency are crucial in data processing as datasets grow larger and more complex across diverse applications. Efficient data handling becomes paramount to ensure high throughput, optimal resource utilization, and energy efficiency, especially in distributed systems where data is processed across multiple nodes or devices. Specialized accelerators can manage vast datasets while reducing power consumption and processing times, providing significant advantages over traditional processing methods. In our project, we incorporate Apache Arrow, a framework designed specifically for efficient data processing and interoperability. Apache Arrow's in-memory columnar storage format accelerates analytic workloads by minimizing serialization and deserialization overhead, which is essential in environments where data needs to be accessed and processed across different components seamlessly. Arrow also supports zero-copy reads, which eliminate unnecessary data transfers and streamline operations, making it ideal for tasks that demand high performance and low latency. By using Apache Arrow, our project benefits from efficient data sharing between various systems, ensuring that the hardware accelerators are fed data in an optimized format, thus maximizing processing speed and system efficiency. This approach not only enables horizontal scalability as more data is generated but also maintains system performance as the workload scales, demonstrating the importance of tools like Apache Arrow in achieving scalable and efficient data processing in modern computing frameworks.

Vendor Lock-in and Software Ecosystems: Vendor lock-in is a significant challenge in hardware acceleration, where proprietary software ecosystems often tie users to specific hardware platforms, limiting flexibility and interoperability. Each hardware accelerator—such as NVIDIA GPUs, Google TPUs, or Intel FPGAs—typically includes unique SDKs, libraries, and toolchains tailored to its architecture, which can create a closed ecosystem. For instance, developers working with NVIDIA GPUs rely on CUDA, a proprietary parallel computing platform and API. While CUDA provides powerful tools for performance optimization on NVIDIA hardware, it also makes it difficult for applications to migrate to other platforms, as they would require extensive code rewriting and adaptation for alternative architectures. This limitation restricts organizations, developers, and researchers from freely adopting new or alternative accelerators, potentially increasing costs and reducing adaptability.

In our project, we address this problem through an open-source, hardware-agnostic framework that allows for seamless integration across diverse hardware platforms. By building on top of existing protocols like SPI and flexible tools like Apache Arrow for data interoperability and employing a modular architecture, our framework supports multiple hardware accelerators without locking users into

any specific vendor's ecosystem. For example, data is handled using a standardized columnar format with Arrow, which can be readily interpreted and processed by different accelerators, ensuring that the data pipeline remains flexible and compatible across platforms. This approach enables our project to bypass vendor-specific requirements, reducing dependency on any single hardware provider and allowing users to choose or switch to the most suitable accelerators for their needs. Through this open and adaptable design, our project mitigates the issue of vendor lock-in, providing users with greater control, lower costs, and increased freedom to leverage advancements across the rapidly evolving landscape of hardware accelerators.

In conclusion, the competition among chip manufacturers to develop specialised hardware accelerators is being driven by the increasing demand for efficient computing solutions in AI and Big Data analytics. These accelerators offer unprecedented performance gains, optimised architectures for specific workloads, scalability, and energy efficiency. However, the challenge of vendor lock-in underscores the importance of developing OFFLOAD framework, an open and interoperable software solution to ensure seamless integration and adoption across diverse computing environments.

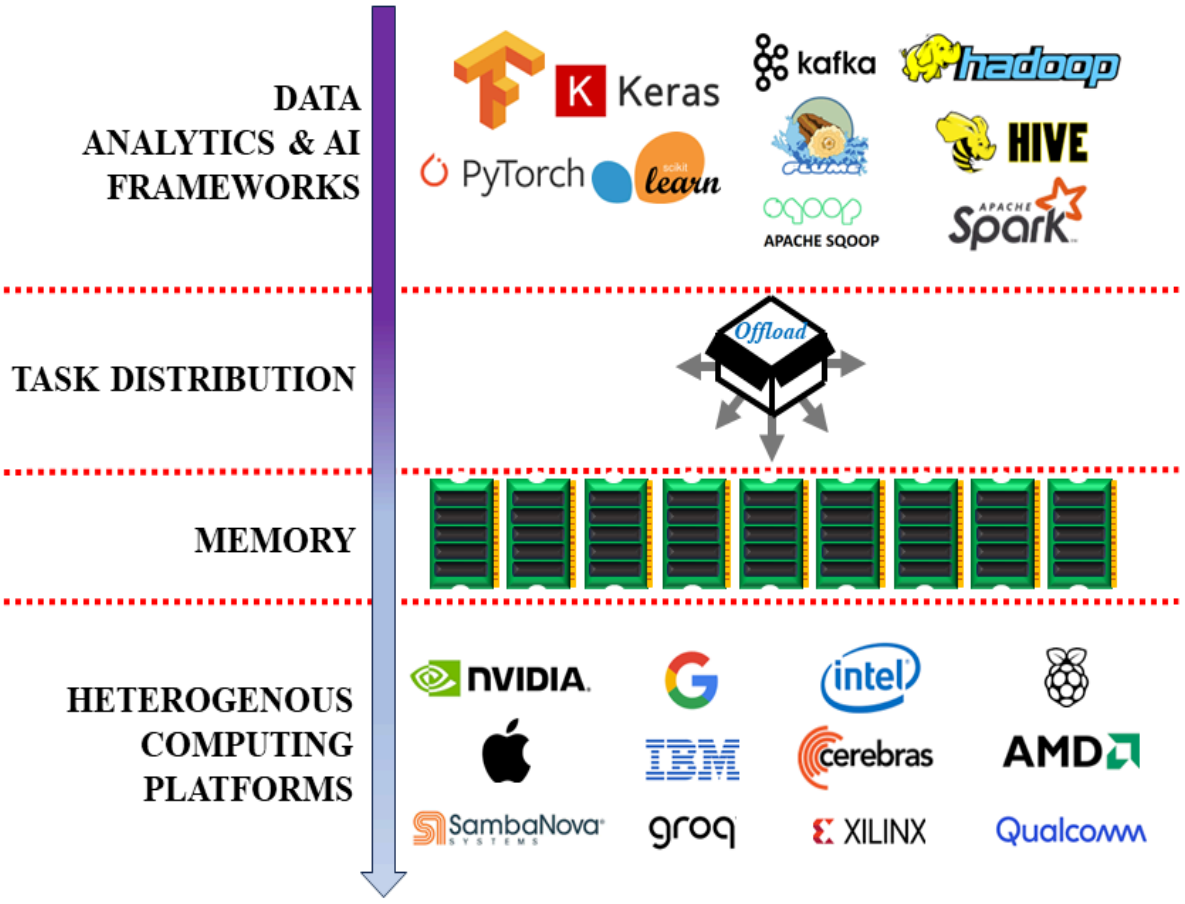

Fig. 1. Layered Architecture of the OFFLOAD Framework for Integration with Hardware Accelerators

Overview of the OFFLOAD Framework: The OFFLOAD framework is an open-source, hardware-agnostic platform designed to distribute compute-intensive tasks across heterogeneous hardware accelerators. This flexibility allows it to adapt to diverse hardware configurations, from GPUs and TPUs to FPGAs and other specialized devices, providing an efficient and scalable solution for AI and big data applications.

As Figure 1 illustrates the layered integration of the OFFLOAD framework with various hardware accelerators, highlighting each component's role in the data processing pipeline. At the base, Layer-1 represents the flexible hardware platforms that can serve as accelerators, ranging from GPUs to FPGAs and other specialized processors. Layer-2 encompasses memory components, such as flash memory, essential for storing and managing large datasets during processing. Layers 3 and 4 depict custom firmware and task distribution mechanisms that interact with higher-level software frameworks, ensuring that workloads are effectively managed and distributed across the hardware network. This layered design underscores OFFLOAD's ability to optimize data flow and computational efficiency, making it suitable for demanding AI and big data applications.

OFFLOAD's architecture is composed of several integrated layers: a software stack, custom firmware, and a dedicated hardware network. The software stack facilitates data movement and computation through an intuitive application interface, primarily built in Python, which interacts seamlessly with tools like Apache Spark and Apache Arrow. Apache Spark serves as the main engine for data processing and initial transformations, while Apache Arrow optimizes data storage and transfer, enabling zero-copy reads that streamline the data flow from storage to hardware accelerators. This columnar data format significantly enhances performance, as it allows for efficient data manipulation across different components.

The custom firmware acts as the intermediary between the software and hardware layers, optimizing data for hardware processing by performing functions like data splitting, formatting, and metadata management. This ensures that each hardware unit receives data tailored to its capabilities, reducing latency and maximizing computational efficiency. The firmware also manages task allocation and scheduling, distributing work evenly across the hardware network to prevent bottlenecks and ensure

optimal utilization of each accelerator.

At the hardware level, OFFLOAD includes a central master accelerator and multiple worker devices that work in parallel. The master device orchestrates task distribution and data flow, leveraging shared memory to facilitate fast communication between itself and the worker devices. Each worker is equipped with specialized firmware to execute assigned tasks independently, allowing the framework to handle large-scale computations in a distributed, parallel manner.

This multi-layered approach enables OFFLOAD to support complex workflows, such as data-intensive AI and big data operations, by efficiently coordinating resources across various hardware platforms. The Flow of Information chapter will delve further into this process, illustrating how data moves seamlessly through the system to ensure high performance, scalability, and adaptability across diverse computing environments.

CHAPTER 2

FLOW OF INFORMATION

The data flow within the system is structured into two main stages: data distribution and query distribution, each playing a crucial role in the overall processing pipeline.

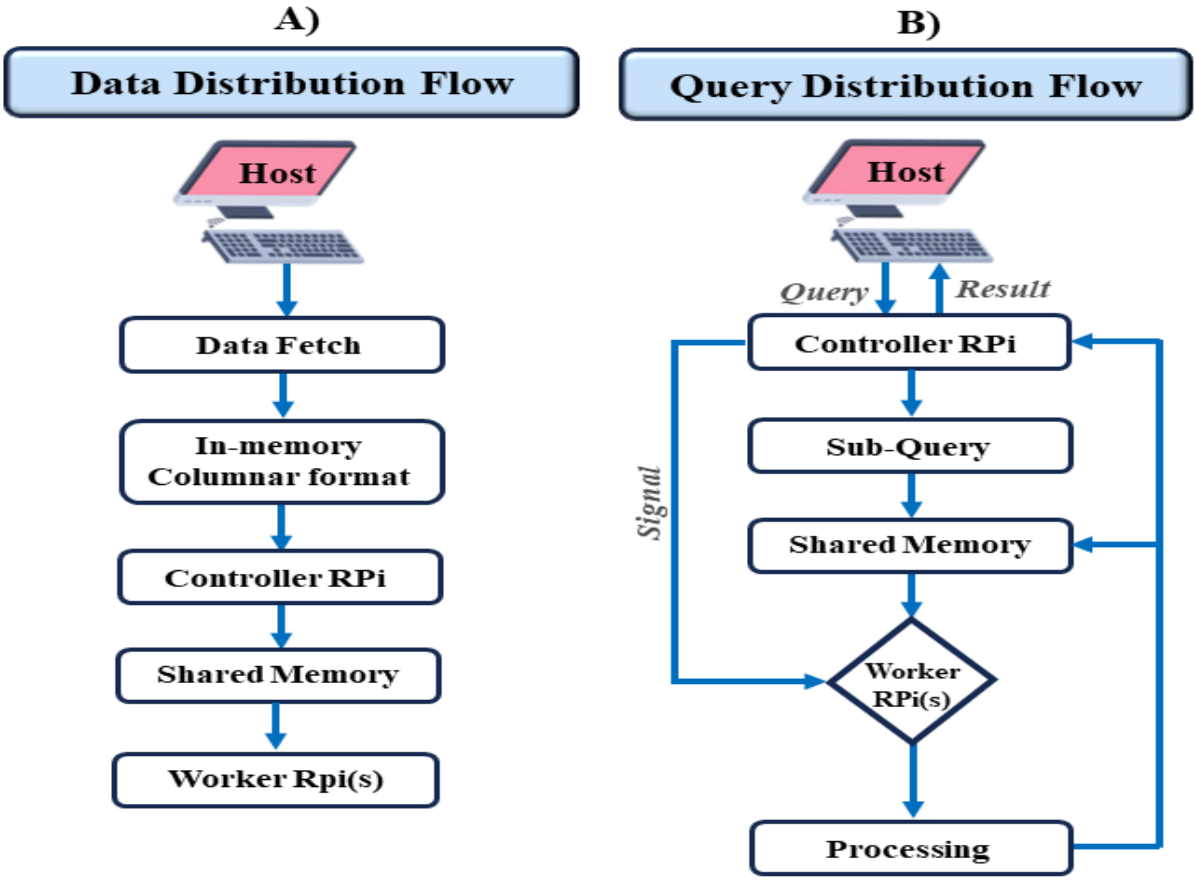

Fig. 2. Flow of (A) Data Distribution and (B) Query Distribution

2.1 Data Distribution

The data distribution stage orchestrates the seamless movement of data from the database to the accelerator hardware network for processing. When a user query specifies an operation on a MySQL database table, Apache Spark, a big data processing framework, initiates the process by utilising JDBC

connectors to connect to the database and retrieve the requested data efficiently. Once the data is retrieved, custom firmware steps in to perform two critical functions. Firstly, it splits the data into smaller subsets tailored to the number of available worker accelerators, such as Raspberry Pi [18][19] devices, within the network. This division facilitates parallel processing, ensuring that the workload is evenly distributed across the hardware accelerators.

Secondly, Apache Arrow is employed to convert these data subsets into the space-optimised Parquet format [20]. Parquet is a columnar storage format known for its efficient data storage and retrieval capabilities, particularly on disk. By leveraging Apache Arrow's in-memory data structure specification, the system further enhances performance, providing a standardised method to represent data in memory for seamless communication across different programming languages.

Subsequently, the master device retrieves the prepared data from the server using the efficient TCP/IP protocol in conjunction with the Paho MQTT messaging library [21]. The master device then splits the data into subsets corresponding to the number of available worker devices. These data subsets are written to designated shared memory locations using a shared memory driver and a Shared Bus interface driver. The final step in the data distribution stage involves the master device signalling the worker Raspberry Pi devices to confirm data reception and readiness for processing, ensuring that each device is prepared to execute its assigned tasks.

2.2 Query Distribution Flow

The query distribution flow manages the translation and execution of user queries within the distributed processing environment, distinguishing this system from traditional database servers that directly parse and execute SQL queries. The distributed nature of the data stored in shared memory across multiple accelerators necessitates a custom firmware approach to handle queries effectively.

The process begins when a user submits a query, ideally adhering to standard SQL syntax. This query is then sent to the master accelerator, such as a Raspberry Pi. The custom firmware on the master accelerator intercepts the query and performs parsing to understand the requested operation. Once the query is parsed, the master accelerator leverages shared memory to distribute sub-queries or tasks to the respective worker accelerators, ensuring that each part of the query is assigned to the appropriate device for processing.

Each worker accelerator receives its assigned sub-query and utilises the optimised Apache Arrow format for efficient processing. Apache Arrow's in-memory data structure facilitates rapid data handling and minimises processing overhead, allowing the worker devices to execute their tasks efficiently. Upon completion of their assigned tasks, the results from each worker device are sent back to the master accelerator using shared memory. This method of communication ensures that data transfer is fast and reliable, maintaining the overall efficiency of the system.

Finally, the custom firmware on the master accelerator aggregates the individual results from the worker devices. This aggregation process may involve performing additional operations as required to compile the final outcome. The master then populates the final result for the user, completing the query execution process. By facilitating parallel processing of the query across multiple accelerators, this approach aims to achieve significant performance improvements for complex data operations, ensuring that the system can handle large-scale queries quickly and accurately.

CHAPTER 3
SOFTWARE STACK

The software stack behind this project orchestrates data movement and processing across several layers, facilitating efficient distributed computing on hardware accelerators.

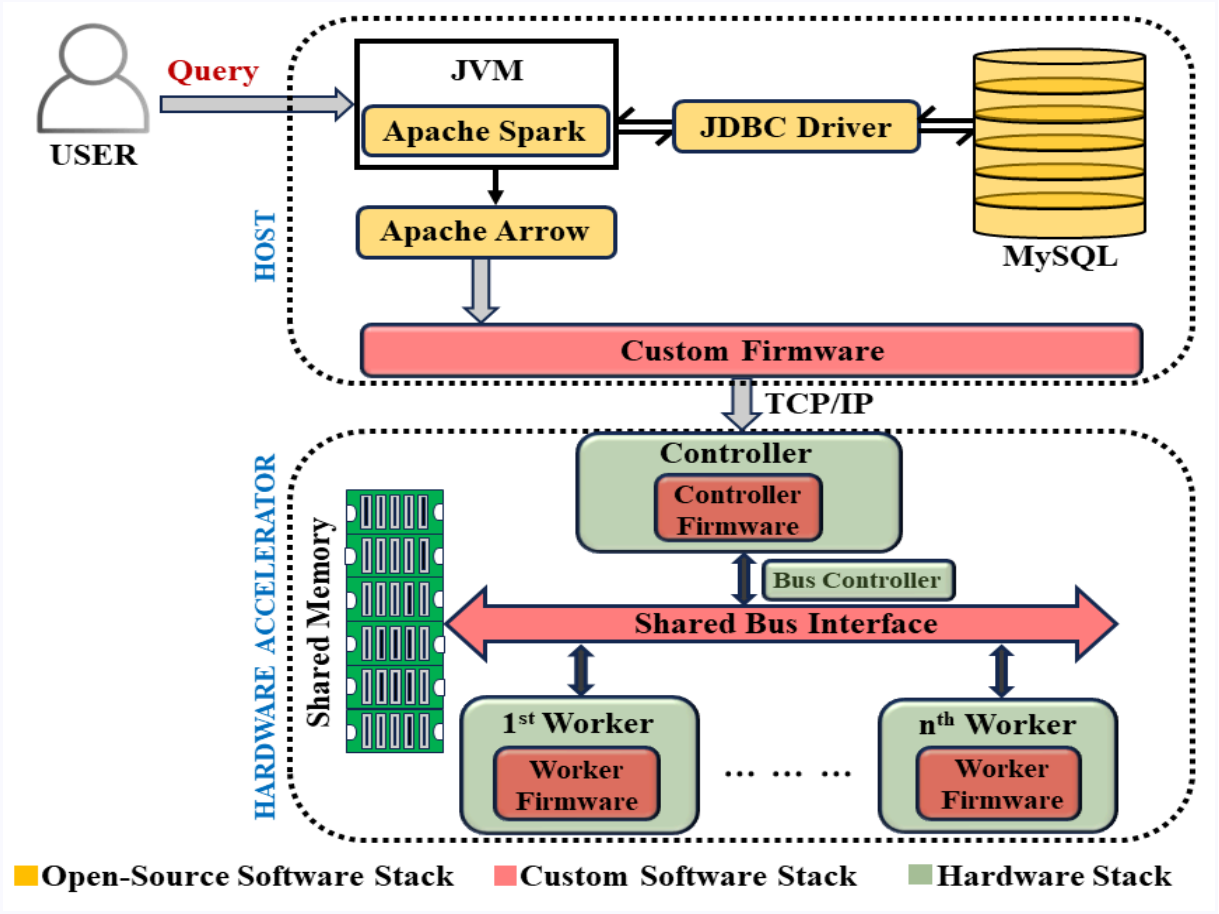

Fig. 3 Software & Hardware Stack of OFFLOAD Framework

3.1 Application Layer

The application layer is a crucial component of the software stack, designed to facilitate seamless interaction between high-level software and underlying data processing frameworks. Here, the integration of Python, Apache Spark, JDBC, and Apache Arrow creates a robust and efficient environment for big data processing and analytics.

3.2 Python Development Environment

The application is primarily developed in Python, a versatile and widely-used programming language known for its simplicity and extensive libraries. Python's rich ecosystem enables rapid

development and easy integration with various tools and frameworks essential for big data processing. In this project, Python serves as the primary language for writing application logic, data manipulation scripts, and orchestrating data flow between different layers of the stack.

3.3 Apache Spark

Apache Spark is at the heart of the data processing framework in the application layer. Spark is a powerful open-source big data processing engine that provides an interface for programming entire clusters with implicit data parallelism and fault tolerance. By running within the Java Virtual Machine (JVM), Spark can efficiently manage and process large datasets across distributed computing environments. Key features of Apache Spark include:

Apache Spark significantly enhances data analytics tasks through its in-memory processing capabilities, which allow data to be processed directly in memory rather than relying on traditional disk-based methods. This approach dramatically speeds up data processing, reducing the latency typically associated with reading and writing to disk and enabling faster, more efficient data analytics. Additionally, Spark's Resilient Distributed Datasets (RDDs) offer a fault-tolerant collection of elements that can be manipulated and transformed in parallel across a distributed computing environment. This parallelism not only increases the efficiency of data operations but also ensures reliability and robustness, as RDDs can recover from failures and continue processing without data loss.

Furthermore, Spark provides powerful tools for interacting with structured data through its SQL and DataFrame API. Spark SQL enables users to execute SQL queries on large datasets, leveraging the familiarity and expressive power of SQL for complex data analysis. This integration allows for seamless querying and data manipulation within the Spark environment. DataFrames, on the other hand, offer a high-level abstraction for working with tabular data, making it easier to perform operations such as filtering, aggregation, and joining data. DataFrames provide a user-friendly interface that simplifies data manipulation tasks while maintaining the performance benefits of Spark's underlying execution engine. Together, these features make Apache Spark a versatile and powerful platform for big data processing and analytics.

3.4 JDBC Integration

Java Database Connectivity (JDBC) drivers are crucial in establishing a seamless connection between the Spark application and the MySQL [11], database server, enabling efficient data exchange and manipulation. JDBC, being a standard API for database interactions, allows the application to execute SQL queries [12] [13], facilitating the retrieval, manipulation, and storage of data within the MySQL database using SQL commands. This capability is essential for performing complex data operations and analyses directly from the Spark [14] environment, leveraging the powerful querying capabilities of SQL. Additionally, JDBC supports transaction management, ensuring data integrity and consistency by providing transactional support. This feature allows multiple operations to be executed as a single unit of work, maintaining the database's reliability and correctness even in the event of failures. Furthermore, JDBC enhances the application's portability and flexibility through its cross-platform compatibility, offering a uniform interface for interacting with various database systems. This standardisation simplifies the integration of the Spark application with different databases, allowing it to function seamlessly across diverse environments and enhancing its adaptability in various use cases. Overall, the use of JDBC drivers in connecting Spark with MySQL significantly boosts the application's functionality, reliability, and versatility in managing and processing large datasets.

3.5 Apache Arrow

Apache Arrow [15] is seamlessly integrated into the application layer to optimise data storage and distribution for hardware accelerators, significantly enhancing overall performance. Arrow, a cross-language development platform for in-memory data, is specifically designed to improve the efficiency of data transfer between different data processing systems. One of the key benefits of using Apache Arrow is its columnar data format, which is particularly well-suited for analytic workloads. This format provides efficient data access patterns, reducing the overhead associated with serialisation and deserialization processes and thereby speeding up data processing tasks. Additionally, Arrow supports zero-copy reads, allowing data to be shared between systems without the need for additional memory copies. This capability minimises data movement overhead and enhances processing speed by enabling direct access to data in its native format. Furthermore, Arrow's interoperability is another significant advantage, as it supports a wide range of programming languages and frameworks. This broad compatibility facilitates smooth data exchange between the Python application, Apache Spark, and hardware accelerators, ensuring that data flows seamlessly across different components of the system. By leveraging these features, Apache Arrow plays a crucial role in optimising data handling and

processing within the application layer, contributing to a more efficient and performance computing environment.

3.6 Data Preparation and Transfer

Once data is retrieved from the MySQL database and processed by Apache Spark [16][17], it undergoes further handling to ensure optimal performance in subsequent stages. This processed data is then divided and transferred to the next layer by sophisticated custom firmware, which acts as a critical intermediary in the data processing pipeline. The primary role of the custom firmware is to prepare and optimise the data for efficient handling by the hardware accelerators. To achieve this, the firmware performs several key functions to ensure the data is in the best possible state for hardware processing.

One of the firmware's essential tasks is data splitting, where it divides the processed data into smaller, manageable chunks. This division is crucial for distributing the workload evenly across multiple hardware accelerators, ensuring balanced and efficient processing. Additionally, the firmware takes care of data formatting, converting the data into a structure and format that the hardware accelerators can process most effectively. This step minimises the need for additional data transformations at the hardware level, streamlining the processing pipeline.

Moreover, the firmware applies various optimization techniques to enhance data processing efficiency. These optimizations might involve data compression, error correction, and other preprocessing steps that prepare the data for rapid and accurate computation by the hardware accelerators. By optimising the data, the firmware ensures that the hardware can execute tasks more quickly and with greater accuracy.

Another crucial function of the custom firmware is metadata management. The firmware manages and attaches necessary metadata to the data chunks, including information such as data type, size, and processing instructions. This metadata is essential for the hardware accelerators to execute their tasks correctly and efficiently, providing the contextual information needed for precise processing.

In summary, the custom firmware plays a pivotal role in the data processing pipeline by preparing and optimising data for hardware accelerators. Through data splitting, formatting, optimization, and metadata management, the firmware ensures that data is in the best possible state for efficient and accurate processing by the hardware accelerators, ultimately enhancing the overall performance of the system.

3.7 Streamlining Subsequent Processing Steps

By performing these intermediary tasks, the custom firmware ensures that the data is meticulously prepared to achieve an optimal state for efficient handling by the hardware accelerators. This preparation plays a critical role in streamlining subsequent processing steps, effectively minimising the workload on the hardware accelerators and maximising their computational efficiency. By optimising the data's format, structure, and metadata, the firmware enables the hardware accelerators to execute tasks faster and with greater accuracy. This streamlined approach not only reduces processing time but also enhances the overall system performance significantly. The well-prepared data facilitates smoother data flows within the system, ensuring that computational resources are utilised more effectively and enabling the hardware accelerators to perform complex computations swiftly and reliably. Thus, the meticulous preparation by the custom firmware contributes directly to improving the system's speed, accuracy, and overall efficiency in data processing tasks.

3.8 Integration with Hardware Accelerators

The optimised and formatted data is seamlessly transferred to the custom-designed hardware network, comprising a central accelerator master and multiple accelerator worker devices. Together, these components efficiently process the data to meet computational demands. The central accelerator master orchestrates task distribution and data transfer using a shared memory interface, ensuring efficient communication with individual worker devices. Each worker device, equipped with specialised firmware, executes assigned tasks using the optimised data prepared by the custom firmware intermediary.

In summary, the custom firmware serves a crucial role in the data processing pipeline by meticulously preparing and optimising data for hardware accelerators. Its intermediary functions include data splitting, formatting, and optimization, which collectively enhance efficiency and streamline subsequent processing steps. By preparing data to meet hardware requirements, the firmware maximises system performance, ensuring tasks are executed swiftly and accurately across the hardware network. This integrated approach optimises resource utilisation and supports seamless data processing within the system.

CHAPTER 4

HARDWARE SETUP

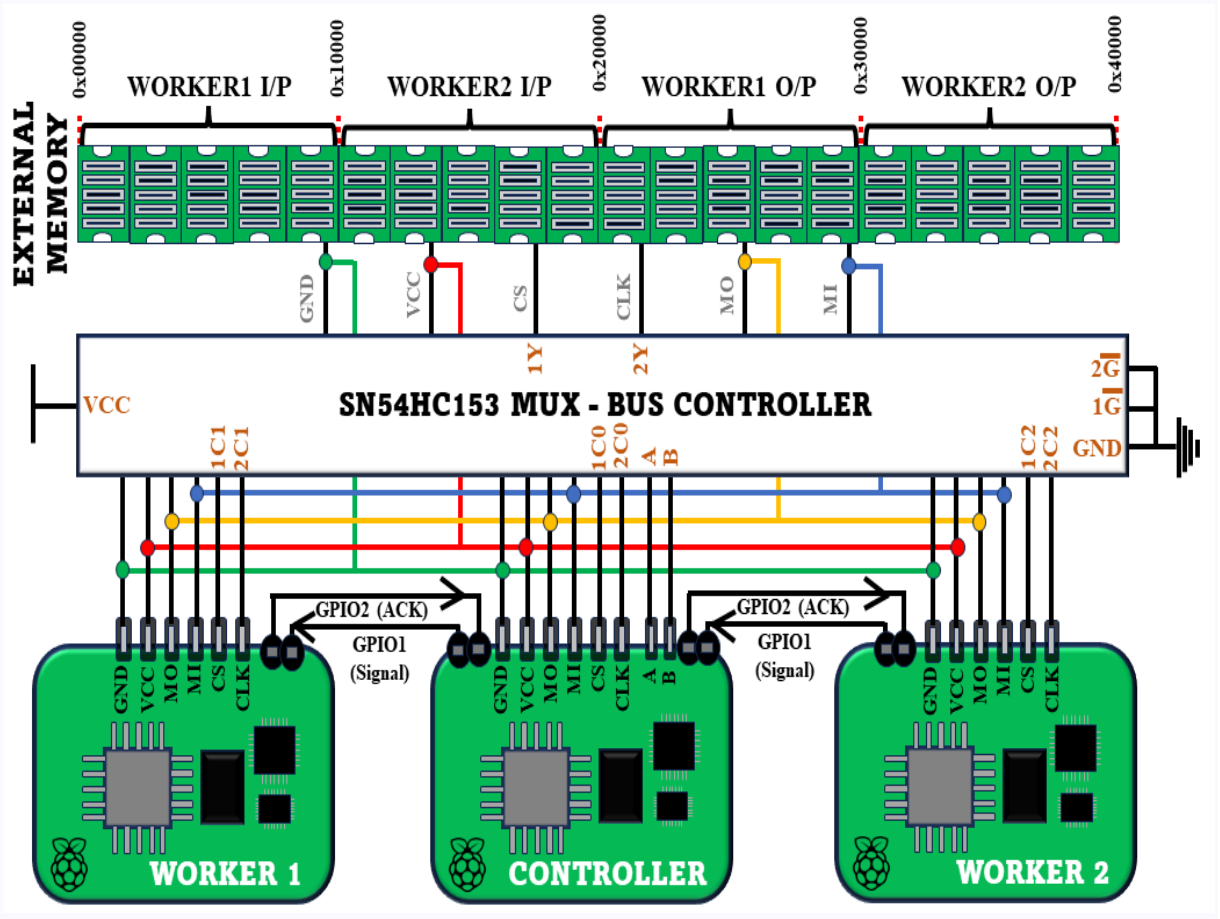

Fig. 4: Hardware Setup Block Diagram

The hardware setup can be visualised through a block diagram where the Master Raspberry Pi (RPi) acts as the central hub for user interaction and communication. The Master RPi establishes direct connections to both worker devices using General Purpose Input/Output (GPIO) pins. This allows for low-level control and data exchange between the Master RPi and the workers. Additionally, the Master RPi interfaces with external memory through a dedicated Bus Controller. This controller, managed by the Master RPi, plays a crucial role in managing access to the external memory for the worker devices. By granting and revoking access privileges, the Bus Controller ensures efficient memory utilisation and avoids potential conflicts during data operations initiated by the workers. This configuration facilitates a centralised control structure where the Master RPi coordinates communication and memory access for the worker devices within the accelerator framework.

The hardware setup of the framework can be visualised through a block diagram outlining the interconnection between its key components. These components include: a Master Raspberry Pi (RPi), one or more worker Raspberry Pi(s), an SN54HC153 Multiplexer (MUX) acting as the Bus Controller, and a shared memory module, the AT45DB321E [22].. For stable power distribution throughout the system, all components share common ground and VCC connections. The Master RPi communicates with worker-1 using two General Purpose Input/Output (GPIO) pins [23]. GPIO-1 transmits signals from the Master to worker-1, while GPIO-2 serves as the acknowledgment channel from worker-1 back to the Master. Following the master-worker paradigm, GPIO-1 is configured as an output pin on the Master and an input pin on worker-1. Conversely, GPIO-2 acts as an input pin for the Master and an output pin for worker-1. This communication pattern is replicated for worker-2, utilising GPIO-3 and GPIO-4 on the Master RPi.

The hardware framework facilitates data transfer between the master device, shared memory, and worker devices using the SPI protocol. However, to ensure only one device communicates with the shared memory at a time, a specific connection scheme is employed. Firstly, all MOSI (Master Output/worker Input) pins from the master, workers, and shared memory are tied together. This configuration reflects the single master design, where only one master communicates with the shared memory. Similarly, all MISO (Master Input/worker Output) pins are shorted together, allowing data transfer between any device and the master. For controlled access to the shared memory, a critical role is played by the SN54HC153 multiplexer (MUX). The master’s CS (Chip Select) pin connects to input 1C0 of the MUX, while its SCLK (System Clock) connects to 2C0. This establishes the master as the default device communicating with the shared memory.

TABLE I

A	B	C0	C1	C2	Y	Device connected to shared memory
L	L	L	X	X	L	MASTER
		H	X	X	H	
H	L	X	L	X	L	SLAVE-1
		X	H	X	H	
L	H	X	X	L	L	SLAVE-2
		X	X	H	H	

● L→Low (0) ● H→High (1) ● X→Unused

Table. 1: SN54HC153 Multiplexer Configuration to select Master and worker Devices

However, the framework allows selective access for worker devices. worker-1's CS pin connects to input 1C1 of the MUX, while worker-2's CS pin connects to 1C2. Similarly, worker-2's SCLK connects to input 2C2 of the MUX. This configuration empowers the master to control which device (itself or a worker) interacts with the shared memory. The core concept lies in granting exclusive access to the shared memory for read/write operations. To achieve this, the shared memory connects to all devices (master and workers). However, through the MUX, only one device can access it at a time. The master firmware dictates this access by controlling the A and B pins of the MUX using two GPIO pins. By default, both A and B pins are LOW (A=0, B=0). This configuration selects the output connected to C0, effectively connecting the master to the shared memory. When the master sets A to HIGH (A=1) while keeping B LOW (B=0), the output connected to C1 is selected, granting access to worker-1. Similarly, setting A to LOW (A=0) and B to HIGH (B=1) routes the shared memory connection to worker-2. This design ensures controlled and efficient communication between the master, shared memory, and worker devices within the SPI framework.

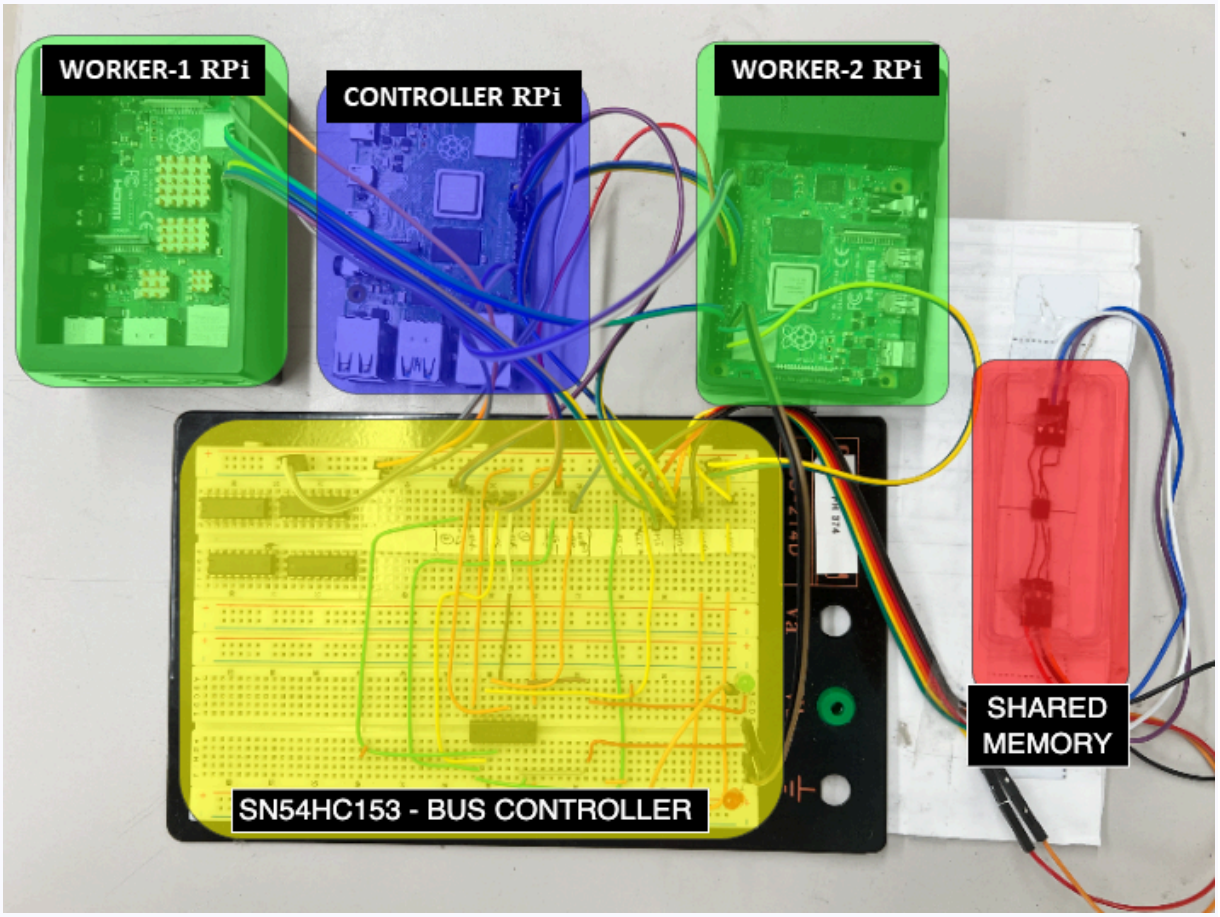

Fig. 5: Actual Hardware Setup

CHAPTER 5

HARDWARE NETWORK AND TASK DISTRIBUTION

5.1 Custom-Designed Hardware Network

After the data has been optimally prepared by the custom firmware, it undergoes transfer down the software stack to a sophisticated, custom-designed hardware network meticulously engineered for high-performance computing tasks. This specialised hardware network is crafted to operate efficiently and effectively under rigorous computational demands. It integrates advanced components such as a central accelerator master and multiple accelerator worker devices, each equipped with specialised firmware tailored for rapid and precise data processing. The network's design emphasises high throughput and low latency, crucial for handling intensive computing tasks with optimal efficiency. This architecture ensures that the prepared data flows seamlessly through the hardware network, leveraging its capabilities to maximise overall system performance and support complex computational operations reliably.

5.2 Shared Memory Device Driver Interface

Central to the functionality of the hardware network is its shared memory device driver interface, a critical component enabling seamless communication between the central accelerator master and multiple accelerator worker devices. This interface facilitates high-speed data transfer, optimising the movement of data between the master and worker devices to minimise delays and eliminate bottlenecks in the system. Moreover, the shared memory device driver interface supports concurrent access, enabling multiple devices to access shared memory simultaneously. This capability enhances the hardware network's efficiency by facilitating parallel processing and coordinated execution of tasks across the worker devices, ensuring that computational resources are fully utilised and contributing to overall system performance.

5.3 Central Accelerator Master

At the core of the hardware network lies the central accelerator master, a pivotal device that directs the entire data processing workflow using the shared memory interface to coordinate and distribute tasks effectively. One of its primary functions is task distribution, where the master device decomposes intricate processing tasks into smaller, more manageable units. These units are subsequently allocated to individual accelerator worker devices for execution, optimising workload distribution and enhancing

overall efficiency. Additionally, the central accelerator master manages data allocation by assigning prepared data chunks to the respective worker devices. This allocation ensures that each device receives the necessary data required for its specific tasks, facilitating synchronised processing and minimising latency in data handling. By overseeing task distribution and data allocation through the shared memory interface, the central accelerator master plays a critical role in optimising resource utilisation and maintaining robust performance across the hardware network.

5.4 Accelerator worker Devices

The accelerator worker devices form the backbone of the hardware network, each equipped with specialised firmware meticulously crafted to execute specific computational tasks with efficiency and precision. These devices receive tasks and data from the central accelerator master through the shared memory interface and operate concurrently to maximise system throughput. Key features of the accelerator worker devices include optimised computation capabilities achieved through tailored firmware designed to fully leverage the hardware's computational resources. This optimization ensures rapid and accurate processing of data, crucial for handling complex computational tasks effectively. Moreover, the worker devices support parallel processing, allowing multiple devices to operate simultaneously on different parts of the workload. This parallelism significantly enhances the overall processing speed and throughput of the hardware network, enabling it to tackle large-scale computations swiftly and efficiently. By combining optimised computation with robust parallel processing capabilities, the accelerator worker devices contribute to the high-performance and scalability of the hardware network, meeting demanding computational requirements with efficiency and reliability.

5.5 Coordinated Processing

The interaction between the central accelerator master and the accelerator worker devices is a meticulously orchestrated process critical to the smooth operation of the hardware network. The master device maintains constant vigilance over the progress of each worker device, actively monitoring their performance and workload. This oversight enables the master to dynamically reallocate tasks and data as needed, ensuring that processing remains balanced and efficient across the network. By continually optimising task distribution and data allocation, this coordinated approach maximises the hardware network's capability to handle extensive data processing tasks swiftly and seamlessly. This systematic management not only enhances operational efficiency but also guarantees that the hardware network operates at its peak potential, delivering reliable and high-performance computing solutions for demanding computational requirements.

5.6 Integration and Optimization

The seamless integration of Apache Spark, Apache Arrow, and custom firmware with a sophisticated hardware network culminates in a highly efficient and scalable system for distributed computing on hardware accelerators. This multi-faceted approach is meticulously designed to ensure rapid and precise data processing, leveraging both advanced software capabilities and powerful hardware optimizations to deliver exceptional performance.

5.7 Apache Spark Integration

Apache Spark occupies a pivotal role within the system, serving as a robust framework designed for large-scale data processing. Its capability to execute computations in-memory brings significant acceleration to data processing tasks, bypassing traditional disk-based limitations. Spark integrates seamlessly into the system by first retrieving large datasets from the MySQL database with efficiency and then conducting initial data preprocessing tasks. This initial stage establishes a foundation for subsequent processing steps by leveraging Spark's capabilities in managing extensive data transformations and conducting complex analyses. Furthermore, Spark enhances processing speed and efficiency through its support for parallel processing, enabled by resilient distributed datasets (RDDs) and DataFrame APIs. These features empower the system to distribute tasks across multiple computing nodes concurrently, optimising resource utilisation and accelerating overall data processing. By harnessing Spark's robust functionalities, the system achieves heightened performance and scalability, effectively meeting the demands of rigorous data processing tasks in diverse computational environments.

5.8 Apache Arrow Optimization

Apache Arrow significantly enhances the performance of the system by optimising both data storage and transfer mechanisms. One key feature is Arrow's adoption of a columnar data format, which ensures that data is stored in a highly efficient manner. This format facilitates faster access and manipulation of data, particularly beneficial for analytic workloads and processing by hardware accelerators. The columnar storage organises data by columns rather than rows, reducing the overhead associated with accessing and processing large datasets. Additionally, Arrow supports zero-copy reads, a capability that minimises data movement overhead within the system. This feature enables components to share data directly without duplicating it unnecessarily, enhancing overall system efficiency and reducing latency. By integrating Apache Arrow into the system architecture, these optimizations

contribute to accelerated data processing speeds and improved performance, making it well-suited for handling complex computational tasks and large-scale data analytics with heightened efficiency.

5.9 Custom Firmware's Role

The custom firmware serves as a critical intermediary within the system, playing a pivotal role in preparing and optimising data specifically tailored for the hardware accelerators. Its primary function includes comprehensive data preparation, where the firmware divides, formats, and optimises data to ensure it is in an optimal state for efficient hardware processing. This preparation phase is crucial as it maximises the effectiveness and performance capabilities of the hardware accelerators, enabling them to process tasks swiftly and accurately. Additionally, the firmware excels in task coordination by facilitating seamless data transfer to the hardware network. It manages the distribution of tasks across the network, ensuring each hardware component receives the necessary data in an optimised format to perform its designated functions effectively. By orchestrating these critical tasks, the custom firmware plays a key role in enhancing overall system efficiency, streamlining data processing workflows, and maximising the utilisation of hardware resources within the system architecture.

5.10 Hardware Network Efficiency

The custom-designed hardware network, featuring a central accelerator master and multiple accelerator worker devices, is meticulously engineered to deliver peak performance within the system architecture. At its core, the network utilises a shared memory interface that facilitates high-speed communication and efficient data transfer between the central accelerator master and the worker devices. This interface plays a crucial role in enabling seamless task distribution and execution across the network, optimising resource utilisation and minimising latency.

Parallel processing is another cornerstone of the hardware network's design, where multiple worker devices operate concurrently to execute tasks in parallel. This parallelism significantly enhances the overall processing speed of the system, allowing it to handle complex computations and large-scale data processing tasks with heightened efficiency.

The integration of these advanced components results in a system renowned for its superior performance capabilities. Scalability is a key feature, as the system is designed to effortlessly accommodate growing data volumes and computational demands without compromising on

performance. This scalability ensures that the system can adapt to evolving needs and expand its capabilities as required.

Moreover, the system excels in efficiency by leveraging the synergies between software and hardware components. By harnessing the strengths of both domains, the system achieves exceptional performance levels, seamlessly navigating through intricate data processing tasks while maintaining high levels of accuracy and reliability. This integrated approach underscores the system's ability to deliver robust performance across diverse computing environments, making it a reliable choice for demanding applications in data-intensive industries.

CHAPTER 6

SOFTWARE & FIRMWARE IMPLEMENTATION

The project codebase has been released as open-source on GitHub [24]. It is organised within a folder named "bist," signifying Built-In Self-Test [25]. This folder contains the source code modules that comprise the system. The initial development of the bist module targeted the creation of a Proof-of-Concept (POC) test suite. This test suite facilitates comprehensive integrated testing between various system components including the Master Raspberry Pi (RPi), worker Raspberry Pis (RPis), External Shared Memory module, and Shared Bus controller interface.

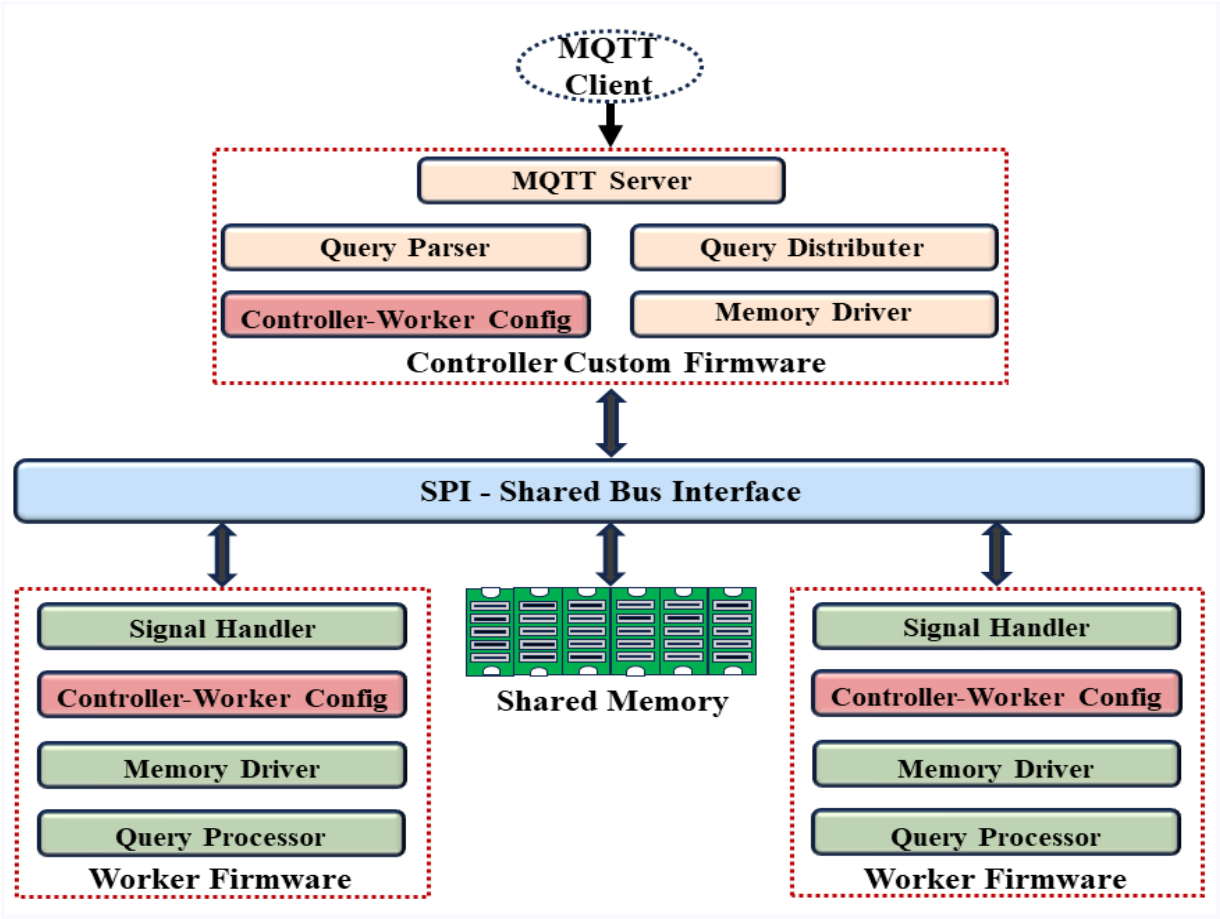

Fig 6: Firmware Layers

The **AT45DB321E.py** module serves as the driver component, encapsulating the commands responsible for read, write, and erase operations on the external shared memory module. It also includes all associated routines pertinent to these operations. Notably, the AT45DB321E.py module resides on both the Master and worker RPis. This ensures consistency in how memory operations are handled

across the different devices in the network.

The **bist_master.py** module functions as the entry point for the accelerator network. It furnishes the interface for receiving user queries from the host system. The execution of this module is initiated using the command `python bist_master.py`. It plays a crucial role in managing the overall workflow and communication between the user interface and the accelerator network.

The **bist_worker.py** module resides on each worker RPi within the system. Its execution is invoked using the command `python bist_worker.py --worker worker_NUMBER`. The specific configuration for each worker RPi is established within this module by parsing the ext_mem_config.json file. This allows each worker device to be uniquely configured and managed within the network, ensuring that they can process their allocated tasks effectively.

The **gpio.py** module is accountable for managing the GPIO (General Purpose Input/Output) configurations for both the Master and worker RPis. Proper configuration of GPIO pins is essential for facilitating communication and control signals between the Master and worker devices.

The **ext_mem_config.json** configuration file is present on both Master and worker RPis. It stores the JSON formatted configuration details for all worker RPis, including the designated input and output regions within the shared memory allocated to each worker RPi. For example:

```
{
"worker_1_INPUT_ADDR" : 0,
"worker_2_INPUT_ADDR" : 65536,
"worker_1_OUTPUT_ADDR" : 131072,
"worker_2_OUTPUT_ADDR" : 196608
}
```

This configuration ensures that each worker has a defined memory space for input and output operations, facilitating orderly and conflict-free data processing.

The **query_handler.py** module executes the routines tasked with parsing SQL queries and subsequently distributing these parsed queries to the worker RPis within the system. This module is crucial for translating user queries into actionable tasks that the accelerator network can process.

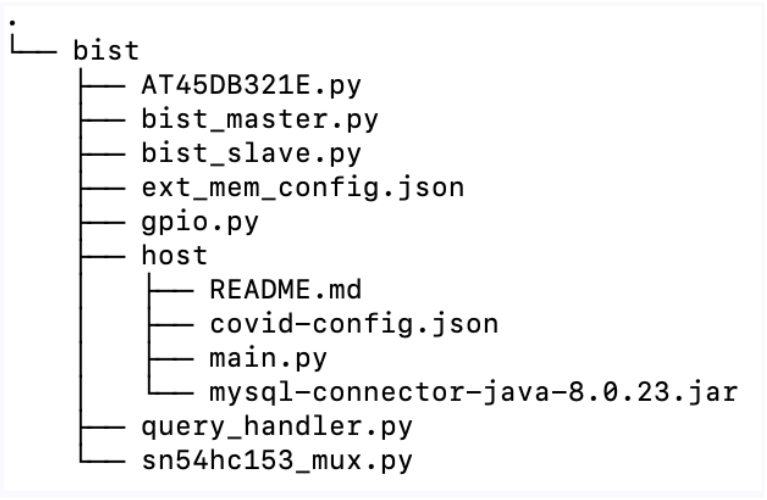

Fig. 7: Code-base folder structure

The **sn54hc153_mux.py** module serves as the bus controller component utilised by the Master RPi to govern access to the shared memory by all worker RPis. This module ensures that memory access is coordinated and controlled, preventing data conflicts and ensuring smooth operation of the system.

The **host** subfolder within the bist folder constitutes the direct user interface. The host system can reside on a cloud platform, a personal computer, or a big data server. This subfolder encompasses the modules that interact with the user, the database, and retrieves queries to be processed by the accelerator network.

The **covid-config.json** configuration file specifies the details for connecting to the MySQL database server, including the connector driver information, username, and password. This configuration file ensures that the system can securely and effectively connect to the database to retrieve and process data.

The **main.py** module represents the user-level application that leverages Apache Spark. It is responsible for receiving user queries and delivering them to the Master node within the accelerator network. This module is essential for integrating big data processing capabilities into the system, ensuring that large datasets can be efficiently managed and processed.

Finally, the **mysql-connector-java-8.0.23.jar** file comprises the JDBC driver, a software component that permits Apache Spark to establish a connection to the MySQL database server [26]. This driver is necessary for facilitating communication between Spark and the database, enabling seamless data retrieval and processing.

CHAPTER 7

RESULTS

The OFFLOAD framework's design prioritises simplicity and generality. This strategic approach enables effortless porting across various architectures, eliminating limitations imposed by fixed instruction set architectures. As a result, the framework presents itself as a valuable tool applicable to big data analytics tasks within heterogeneous computing environments.

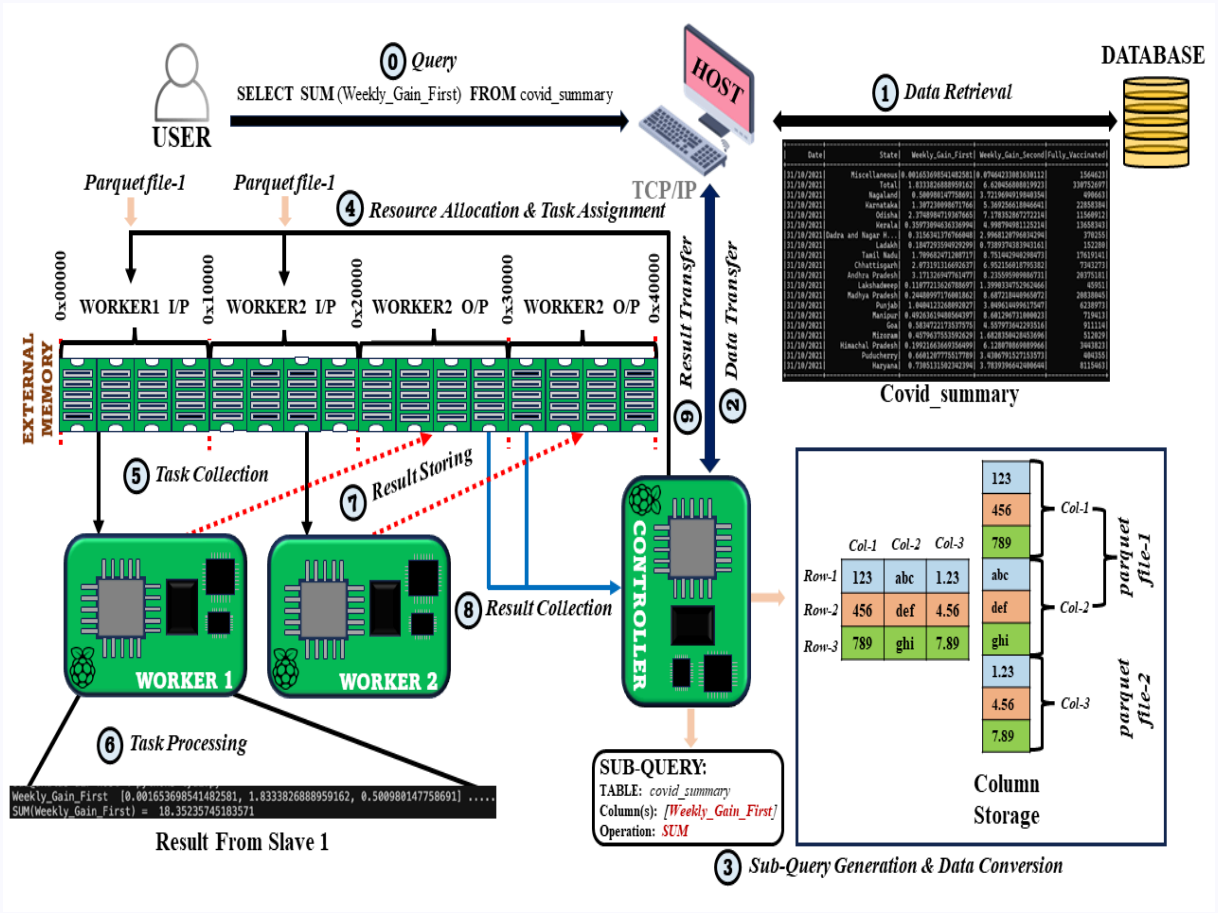

Fig. 8:Demonstration of query handling from host to distributed accelerator network

To achieve accelerated and parallel processing within the distributed accelerator network, query handling occurs primarily at three distinct levels.

7.1 Host Side

The Host entirely handles queries or part of queries involving data retrieval from the database. For instance, consider the query **SELECT SUM (Weekly_Gain_First) FROM** covid_summary, as shown in Figure 8. In this query, **SELECT** specifies data retrieval, SUM indicates the summation operation of

the (**Weekly_Gain_First**) column, and **FROM** covid_summary identifies the specific table in the database from which the data to be fetched. The Host processes this query, accesses the database, and retrieves the requested data (in this case covid_summary table) before passing it to subsequent stages for further processing or analysis within the accelerator network.

7.2 Controller Side

Once the requested table is retrieved from the database, it is transmitted to the Controller Raspberry Pi (RPi) via a standard communication protocol like TCP/IP network. The Controller analyses the query to determine the specific columns (in this case Weekly_Gain_First) and operations (SUM) required, then identifies the appropriate Worker RPi for the task based on the configuration file. The Controller divides the table into smaller, manageable chunks, taking into account the number of Worker RPis connected to the network. To optimise processing, it employs a column-based splitting strategy using Apache Arrow, which efficiently stores the data in Parquet format, as shown in Figure 8. This method organises the data by columns, enhancing accessibility for parallel processing. After converting Manuscript submitted to ACM the data into Parquet files, the Controller copies these files into designated shared memory locations, ensuring each Worker has access to the necessary data. Finally, the Controller signals the Workers that the data is ready for processing, initiating the distributed computing tasks.

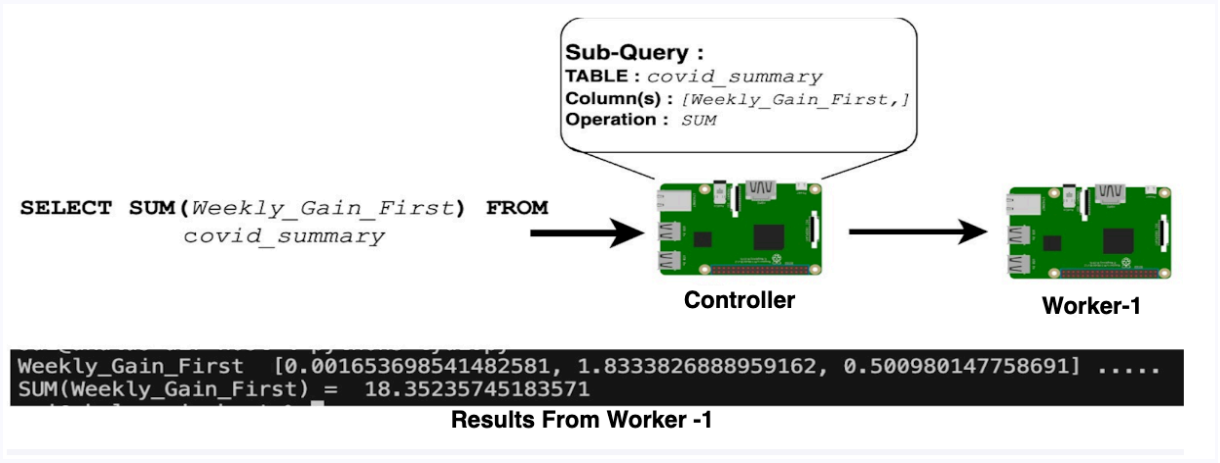

Fig. 9: Master dividing query into subqueries for offloading to workers

7.3Worker Side

The Worker RPis handle queries or portions of queries involving data processing. They start the process by reading input data from designated locations in the shared memory. Each Worker RPi then

performs the specified operation, such as summing the data from the designated column (Weekly_Gain_First), as shown in Figure 8. After processing, the Worker writes the results back to the assigned output memory location and signals the Controller upon completion. Once all Workers have acknowledged completion, the Controller retrieves the results, aggregates them, and transmits the final outcome back to the Host via the network, completing the query execution.

7.4 Timing Analysis

To demonstrate the efficacy of our framework, we have segmented the entire workflow into distinct sections and measured timing values for each, as detailed in Table 1. Key timings include:

- GPIO Initialization Time: GPIO initialization time represents the duration to initialize the GPIO pins of Raspberry Pi, a one-time activity crucial for setting up the system. Our measurements show that GPIO pin initialization on the Raspberry Pi takes 36,489,587 nanoseconds using the OFFLOAD framework.
- Memory Initialization Time: Memory initialization indicates the time necessary to set up the shared memory, another one-time process essential for the system's proper functioning. This setup ensures that all components can communicate efficiently and access shared memory as required. For the OFFLOAD framework, memory initialization takes just 92,711 nanoseconds.
- Writing to Memory: This measures the time needed to write sample data into shared memory. This operation, although performed once, benefits from an efficient block writing mechanism, ensuring that the time does not increase linearly with data size. For our prototype, writing to memory consumes only 136,723 nanoseconds.
- Bus Controller Time: Bus Controller Time refers to the duration required by the RPi controller node to hand over the control of shared memory to RPi worker nodes for processing. This hand-off, essential for each query, ensures that worker nodes can access the memory needed for their tasks. In the OFFLOAD framework, this process takes 19,111 nanoseconds per query.
- GPIO De-initialization Time: This refers to the time needed to safely reset and clear the GPIO hardware, releasing resources and preparing the system for shutdown or reconfiguration. As a one-time finalization step, it completes the necessary hardware disengagement. In the OFFLOAD framework, GPIO de-initialization takes 1,061,017 nanoseconds.
- Memory De-initialization Time: Memory de-initialization represents the time required to

fully reset and disengage the shared memory hardware setup, ensuring that all resources are released and the system is properly concluded. This process is essential to finalize operations and prepare the system for shutdown or future adjustments. For the OFFLOAD framework, this de-initialization takes 67,259 nanoseconds, marking the end of memory-related activities.

- Acknowledge Workers: This reflects the time required by the controller node to assign and communicate specific tasks to the worker nodes, a crucial step that must be performed for every query to ensure that each worker knows its role in the processing operation. This coordination process ensures that the distributed computing tasks are efficiently managed across the network. In the context of the OFFLOAD framework, this task assignment process takes 29,111 nanoseconds for each query.

TABLE 2

SECTION	Initialization Time (ns)	Processing Time Per Query (ns)
GPIO Initialization Time	36,489,587	-
Memory Initialization Time	92,711	-
Writing to Memory	136,723	-
Bus Controller Time	-	19,111
GPIO De-initialization Time	1,061,017	-
Memory De-initialization Time	67,259	-
Acknowledge Workers	-	29,111
Total	37,847,297 (Fixed)	48,222

Table 2. Timing Values Collected from the framework

Based on the analysis of our framework, we have formulated the total time equation as a sum of initialization time and processing time. The equations are defined as follows:

$$T_{Memory} = T_{Memory\ Initialization} + T_{Memory\ Deinitialization} \quad (1)$$

$$T_{GPIO} = T_{GPIO\ Initialization} + T_{GPIO\ Deinitialization} \quad (2)$$

$$T_{Initialization} = T_{GPIO} + T_{Memory} + T_{Memory\ Write} \quad (3)$$

$$T_{Processing\ per\ Query} = T_{Bus\ Controller} + T_{Acknowledge\ Slaves} \quad (4)$$

$$T_{Total} = T_{Initialization} + \frac{N_Q \times T_{Processing\ per\ Query}}{N_S} \quad (5)$$

Here, TInitialization represents the total initialization time, which is a one-time overhead and includes the time taken for GPIO initialization, memory initialization, GPIO de-initialization, memory de-initialization, and memory write operations. TProcessing per Query denotes the time taken for each query, involving the bus controller time and the time to acknowledge worker nodes. Finally, TTotal is the total time, which includes both the initialization time and the processing time per query, distributed across NS worker nodes, for NQ queries. To visualize the efficiency and scalability of our framework, we conducted a predictive analysis as depicted in Figure 9. In this analysis, the initialization time remains constant at 37.85 ms, while the processing times vary according to the number of queries and the number of worker nodes. For example, with 4 worker nodes, the total processing time for 10,000 queries is 158.40 ms. This time significantly reduces to 39.73 ms when the number of worker nodes is increased to 256 for the same number of queries. As the number of worker nodes increases, the processing time decreases significantly, highlighting the framework's capability to effectively manage and distribute workloads.

7.5 Comparisons

OFFLOAD stands out as a device-agnostic framework, distinguishing itself from Fletcher and TaPaSCo. One of its key strengths lies in its ability to support a variety of devices seamlessly, compared to the FPGA-centric nature of Fletcher and TaPaSCo. While Fletcher and TaPaSCo rely on FPGA toolchains and are limited to FPGA devices, OFFLOAD operates on memory-mapped I/O, ensuring compatibility with a broader spectrum of hardware. Moreover, OFFLOAD offers flexibility in task partitioning strategies through its custom driver, enabling integration with custom architectures—an advantage unavailable in Fletcher and TaPaSCo. This adaptability empowers developers to tailor their systems precisely to their needs, maximizing performance and efficiency

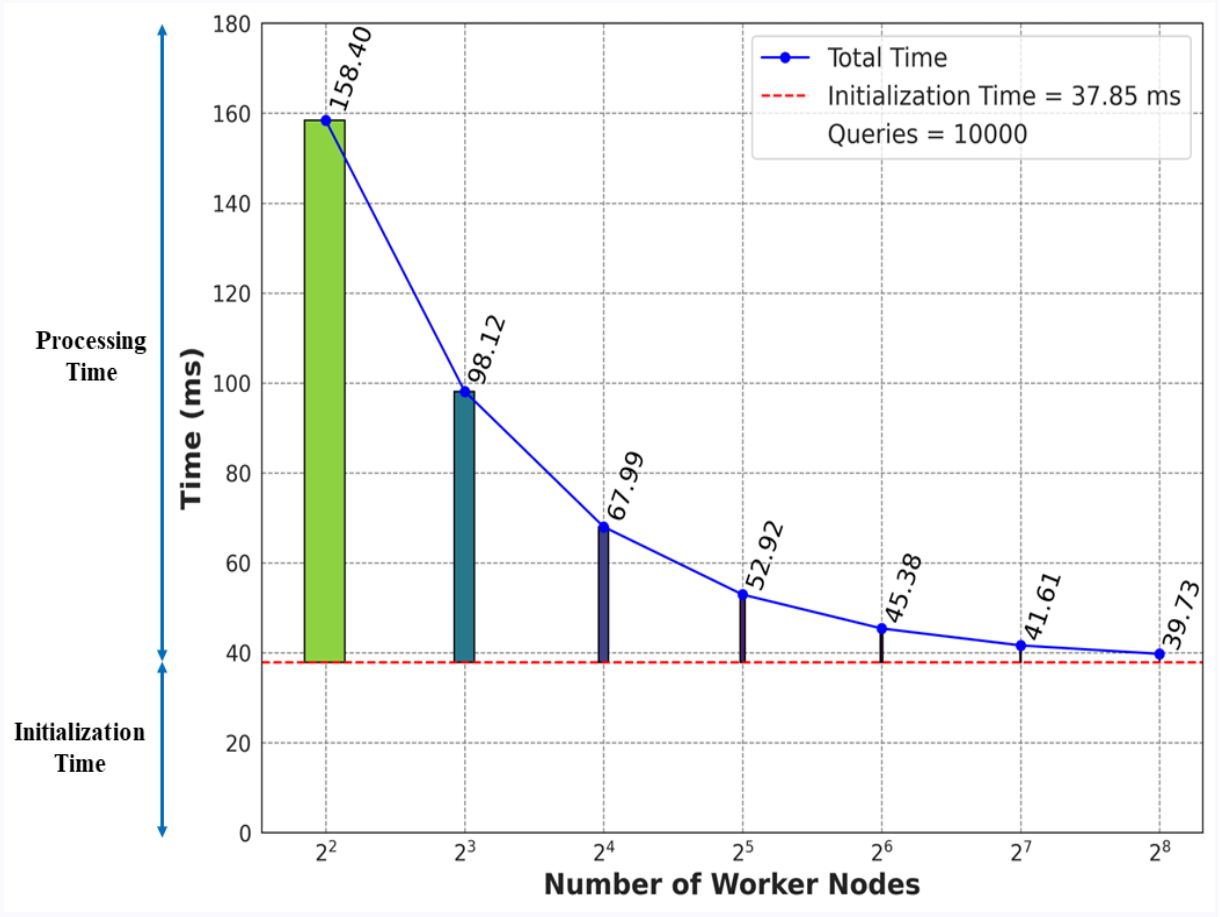

Fig. 10. Predictive Analysis of Processing Time vs. Number of Worker Nodes

As shown in Table 3, OFFLOAD excels with its simple and straightforward syntax, making it more accessible compared to the complex structures of Fletcher and TaPaSCo. This simplicity lowers the learning curve, allowing developers to optimize performance without relying on complex compiler configurations. OFFLOAD also offers high modularity, enabling parallel task distribution, unlike the more monolithic or limited approaches of Fletcher and TaPaSCo. Its high portability further enhances its adaptability across various platforms, a significant advantage over Fletcher's lower and TaPaSCo's moderate portability. OFFLOAD's open-source nature fosters collaboration and eliminates the financial barriers associated with proprietary FPGA tool-chains, as seen with Fletcher and TaPaSCo. This flexibility, combined with its seamless integration with widely used tools like Apache Spark, Apache Arrow, MySQL, and Python, positions OFFLOAD as a versatile solution for big data and AI tasks. This broad applicability makes OFFLOAD particularly well-suited for heterogeneous computing environments, providing a comprehensive and user-friendly alternative to device-constrained frameworks like Fletcher and TaPaSCo.

In summary, the results demonstrate that the OFFLOAD framework, with its flexible design and efficient use of hardware and software resources, is well-suited for big data analytics tasks in heterogeneous computing environments. The ability to effortlessly port the framework across different architectures further enhances its applicability, making it a valuable tool for a wide range of data processing applications.

TABLE 3

Feature	OFFLOAD	Fletcher [7]	TaPaSCo [8]
Syntax	Simple	Complex	Complex
Modularity	High	Low	Moderate
Task Distribution	Parallel	Monolithic	Parallel
Platform	Generic	Only FPGAs	Only FPGAs
Standard Libraries	Limited	Extensive	Limited
Memory Manageme	Firmware	User	Kernel
Exception Handling	Firmware	Operating System	Kernel
Compiler Support	Not Needed	Yes	Yes
Portability	High	Low	Moderate
Need for Standard IS	No	Yes	Yes
Apache Spark	Yes	Yes	No
Apache Arrow	Yes	Yes	No
MySQL	Yes	No	No
Python	Yes	Yes	No

Table 3. Comparison of OFFLOAD with other Frameworks

Furthermore, OFFLOAD's emphasis on simplicity and ease of use is evident in its streamlined syntax and compiler-independent nature. Unlike Fletcher and TaPaSCo, which require specific compiler configurations, OFFLOAD simplifies the development process by eliminating such dependencies. This simplification reduces the learning curve for developers and speeds up the implementation of new features and functionalities.

Additionally, OFFLOAD's high portability makes it exceptionally adaptable across diverse environments, enhancing its versatility and usability. Its seamless integration with popular tools such as Apache Spark, Apache Arrow, MySQL, and Python further underscores its suitability for a wide range

of applications, particularly in big data and AI tasks. By leveraging these widely-used technologies, OFFLOAD can be easily incorporated into existing workflows, providing a powerful and flexible solution for complex data processing challenges.

CHAPTER 8

CONCLUSION

The OFFLOAD framework represents a pioneering approach in the realm of offloading compute-intensive tasks to a variety of hardware. In this paper, we have meticulously detailed the framework and its core aspects, encompassing everything from data source and application coupling to binary instruction and data generation. We have validated the framework's efficacy by demonstrating hardware task offloading for MySQL database queries. Our demonstration utilised Apache Spark and Apache Arrow in Python, two of the most popular tools for database management and dataset de-segmentation.

On the hardware side, Raspberry Pis were employed to exemplify heterogeneous hardware configurations, showcasing the framework's compatibility with diverse devices. The task offloading process was centralised through memory-mapped I/O, ensuring efficient data transfer and processing. This methodology underscores the framework's versatility and flexibility, making it adaptable to a wide range of hardware environments.

The scalability and open-source nature of the OFFLOAD framework are pivotal in paving the way for easier adoption of emerging machine learning and data analytics hardware. By providing a robust, flexible, and scalable solution, OFFLOAD offers significant potential to enhance performance in big data analytics tasks within heterogeneous computing environments. This framework not only simplifies the integration of various hardware accelerators but also fosters innovation and collaboration within the community, setting the stage for future advancements in compute task offloading and hardware integration.

REFERENCES

[1] K. Neshatpour, A. Sasan and H. Homayoun, ”Big data analytics on heterogeneous accelerator architectures,” 2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Pittsburgh, PA, USA, 2016, pp. 1-3. keywords: Acceleration;Field programmable gate arrays;Hardware;Big data;Energy efficiency;Accelerator architectures.

[2] Peccerillo, Biagio, Mirco Mannino, Andrea Mondelli, and Sandro Bartolini. ”A survey on hardware accelerators: Taxonomy, trends, challenges, and perspectives.” Journal of Systems Architecture 129 (2022): 102561.

[3] Kimm, H., Paik, I. and Kimm, H., 2021, December. Performance comparision of tpu, gpu, cpu on google colaboratory over distributed deep learning. In 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC) (pp. 312-319). IEEE.

[4] Abts, D., Kim, J., Kimmell, G., Boyd, M., Kang, K., Parmar, S., Ling, A., Bitar, A., Ahmed, I. and Ross, J., 2022, August. The groq software-defined scale-out tensor streaming multiprocessor: From chipsto-systems architectural overview. In 2022 IEEE Hot Chips 34 Symposium (HCS) (pp. 1-69). IEEE Computer Society.

[5] Emani, M., Vishwanath, V., Adams, C., Papka, M.E., Stevens, R., Florescu, L., Jairath, S., Liu, W., Nama, T. and Sujeeth, A., 2021. Accelerating scientific applications with sambanova reconfigurable dataflow architecture. Computing in Science Engineering, 23(2), pp.114-119.

[6] Zong, Z., Ge, R. and Gu, Q., 2017. Marcher: A heterogeneous system supporting energy-aware high performance computing and big data analytics. Big data research, 8, pp.27-38.

[7] Peltenburg, Johan, Jeroen van Straten, Matthijs Brobbel, Zaid Al-Ars, and H. Peter Hofstee. ”Generating high-performance fpga accelerator designs for big data analytics with fletcher and apache arrow.” Journal of Signal Processing Systems 93 (2021): 565-586.

[8] Heinz, Carsten, Jaco Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. ”The TaPaSCo Open-Source Toolflow: for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems.” Journal of Signal Processing Systems 93 (2021): 545-563.

[9] Mittal, Sparsh, and Jeffrey S. Vetter. ”A survey of CPU-GPU heterogeneous computing techniques.” ACM Computing Surveys (CSUR) 47, no. 4 (2015): 1-35.

[10] S. Wang, A. Prakash and T. Mitra, ”Software Support for Heterogeneous Computing,” 2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Hong Kong, China, 2018, pp. 756-762, doi: 10.1109/ISVLSI.2018.00142. keywords: Graphics processing units;Field programmable gate arrays;Kernel;Parallel processing;Computer architecture;Heterogeneous computing, scheduler, compiler, power/thermal management.

[11] Pereira, A.L., Raoufi, M. and Frost, J.C., 2012. Using MySQL and JDBC in new teaching methods for undergraduate database systems courses. In Data Engineering and Management: Second International Conference, ICDEM 2010, Tiruchirappalli, India, July 29-31, 2010. Revised Selected Papers (pp. 245-248). Springer Berlin Heidelberg.

[12] Gupta, Anand, Hardeo Kumar Thakur, Ritvik Shrivastava, Pulkit Kumar, and Sreyashi Nag. ”A big data analysis framework using apache spark and deep learning.” In 2017 IEEE international conference on data mining workshops (ICDMW), pp. 9-16. IEEE, 2017.

[13] Borthakur, D., 2008. HDFS architecture guide. Hadoop apache project, 53(1-13), p.2.

[14] Chebotko, A., Kashlev, A. and Lu, S., 2015, June. A big data modeling methodology for Apache Cassandra. In 2015 IEEE International Congress on Big Data (pp. 238-245). IEEE.

[15] van Leeuwen, Lars TJ, Zaid Al-Ars, and H. Peter Hofstee. ”Fletcher: A Framework to Efficiently Integrate FPGA Accelerators with Apache Arrow.”

[16] Gould, C., Su, Z. and Devanbu, P., 2004, May. JDBC checker: A static analysis tool for SQL/JDBC applications. In Proceedings. 26th International Conference on Software Engineering (pp. 697-698). IEEE.

[17] Hildebrandt, Juliana, Dirk Habich, and Wolfgang Lehner. ”Integrating Lightweight Compression Capabilities into Apache Arrow.” In DATA, pp. 55-66. 2020.

[18] Upton, E. and Halfacree, G., 2016. Raspberry Pi user guide. John Wiley Sons.

[19] Leens, F., 2009. An introduction to I 2 C and SPI protocols. IEEE Instrumentation Measurement Magazine, 12(1), pp.8-13.

[20] Peltenburg, J., Van Leeuwen, L.T., Hoozemans, J., Fang, J., Al-Ars, Z. and Hofstee, H.P., 2020, December. Battling the CPU bottleneck in apache parquet to arrow conversion using FPGA. In 2020 international conference on Field-Programmable technology (ICFPT) (pp. 281-286). IEEE.

[21] Bender, M., Kirdan, E., Pahl, M.O. and Carle, G., 2021, January. Open-source mqtt evaluation. In 2021 IEEE 18th Annual Consumer Communications Networking Conference (CCNC) (pp. 1-4). IEEE.

[22] Bez, R., Camerlenghi, E., Modelli, A. and Visconti, A., 2003. Introduction to flash memory. Proceedings of the IEEE, 91(4), pp.489-502.

[23] Cicolani, J. and Cicolani, J., 2018. Raspberry pi gpio. Beginning Robotics with Raspberry Pi and Arduino: Using Python and OpenCV, pp.103-128.

[24] Source code of the project in https://github.com/saiakula997/lab518.git

[25] Agrawal, V.D., Kime, C.R. and Saluja, K.K., 1993. A tutorial on built-in self-test. I. Principles. IEEE Design Test of Computers, 10(1), pp.73-82.

[26] Silva, Yasin N., Isadora Almeida, and Michell Queiroz. (2016). ”SQL: From traditional databases to big data.” In Proceedings of the 47th ACM Technical Symposium on Computing Science Education, pp. 413-418.

VITA

Satya Sai Siva Rama Krishna Akula, a native of India, holds a Bachelor of Technology degree in Electrical and Electronics Engineering, which he obtained in 2018 from Gudlavalleru Engineering College, affiliated with Jawaharlal Nehru Technological University, Kakinada, India. Following his undergraduate studies, he gained significant professional experience as a firmware engineer at Qualcomm and Sasken Technologies, where he focused on embedded systems firmware development, contributing to various high-impact projects.

Currently, Mr. Akula is pursuing a Master’s degree at the School of Computing and Engineering, University of Missouri-Kansas City. His academic work is centered around his master's thesis, *OFFLOAD: An Open-source Framework for Distributing Big Data and AI Tasks to Heterogeneous Compute Units*, which aims to address the challenges of distributing computational workloads across

diverse processing units in big data and artificial intelligence applications. Through this research, he seeks to advance the field of distributed computing by developing open-source solutions that leverage heterogeneous computing environments.