An Opensource Framework for Offloading Big Data and AI Tasks (OFFLOAD) to Heterogenous Compute Units

*Note: Sub-titles are not captured in Xplore and should not be used

1st Satya Sai Siva Rama Krishna Akula
School of Science and Engineering, University of Missouri Kansas City MO, USA, sa7gb@umsystem.edu

2nd Chowdhury, Rownak
School of Science and Engineering, University of Missouri Kansas City, MO, USA, rhctmc@umkc.edu

3rd Sapireddy, Srinivas Rahul
School of Science and Engineering, University of Missouri Kansas City, MO, USA, ssdx5@umkc.edu

4th Rahman, Mostafizur
School of Science and Engineering, University of Missouri Kansas City, MO, USA, rahmanmo@umkc.edu

***Abstract*—The ever-growing demand for efficient AI and Big Data processing has fueled a rapid development of new hardware architectures specifically designed for compute-intensive tasks. Alongside these advancements, software solutions are emerging to leverage this specialized hardware by offloading tasks. However, proprietary software often presents a significant learning curve for users, hindering adoption and flexibility. This paper proposes OFFLOAD, an open-source, hardware-agnostic software-hardware framework. OFFLOAD distributes tasks to various hardware units, including both novel accelerators and existing system-on-chip (SoC) architectures. Our framework seamlessly interfaces with popular databases and application development tools. Through multi-level abstractions at the compiler, operating system, and driver levels, OFFLOAD translates high-level code and data into hardware-optimized binary instructions. To our knowledge, OFFLOAD represents a novel approach in this domain. We demonstrate the feasibility of OFFLOAD by integrating it with popular tools like MySQL, Apache Spark, and Apache Arrow in Python at the user level. Tasks are then offloaded for execution on hardware using memory-mapped I/O. Raspberry Pis serve as examples in this demonstration, 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.**

I. Introduction

The evolving field of Artificial Intelligence and Big Data analytics [1] have sparked a chips race for developing hardware accelerators [2] capable of handling a variety of compute intensive tasks. Examples include Google's TPU [3], Nvidia's Groq [4], SambaNova's DSA [5], Intel's Hailo [6] and others. As these chips with new computing architectures are emerging, so is their counterpart in software for offloading tasks [7] [8]. A challenge with such software-hardware pairing is the locking in vendor specific ecosystem and significant development cycle to adapt to new software framework [9] [10]. To overcome this, we propose OFFLOAD: an open-source, hardware-agnostic framework that bridges the gap between software and hardware. OFFLOAD enables task distribution across diverse hardware architectures, including cutting-edge accelerators and established system-on-chip (SoC) solutions. This flexibility is achieved through multi-level abstractions, where OFFLOAD translates high-level code and data used in popular application tools into hardware-optimized binary instructions for the target hardware. This approach not only simplifies software development but also unlocks the full potential of the underlying hardware, regardless of its origin.

A variety of higher level application development framework can be supported in OFFLOAD. This includes database management systems such as MySQL [11], Oracle, SQLite, MongoDB, etc., and data processing frameworks like Spark [12], Hadoop [13], Cassandra [14], Flume, etc., which are combined in development frameworks like python, java, c++. Our framework also supports data storage formats like Arrow, Parquet, etc. In this paper, we demonstrate the overall integrated framework with MySQL, Apache Spark, Apache Arrow and python in the software side, and an x86 machine, 3 raspberry PIs and Flash memory on the hardware side as a proof of concept.

MySQL is a free data management system that allows connectivity through various APIs. Apache Spark, a widely used big data processing engine, bridges the gap between MySQL and OFFLOAD. Spark's high-level API simplifies job control, making it easier to manage complex workflows. Apache Arrow complements this ecosystem with its efficient data storage and processing techniques. Crucially, Arrow enables parallel data

tokenization, making it ideal for distributed execution within the framework.

To effectively distribute computations across diverse hardware units (including both novel accelerators and existing SoCs), OFFLOAD utilizes a custom-designed master-slave architecture. Master receives data and queries from Server then the master hardware processes data and instructions, then offloads the optimized binary information to a shared memory space. Slave devices then access their relevant portions of this memory, which acts as the central communication hub for input and output.

We demonstrate the framework's functionality using two Raspberry Pi devices as slaves. This showcase encompasses the entire workflow, from a software-based data query (using a real-estate sales database as an example) to its execution on the hardware. Additionally, the paper presents a qualitative comparison with relevant academic work applied to FPGAs and SoCs.

This paper presents a comprehensive exploration of the OFFLOAD framework. Section II provides an overview of its entire software and hardware stack, along with a detailed explanation of the information flow between these components. Section III delves into the hardware setup employed for the proof-of-concept implementation. Section IV details the firmware implementation. Section V serves the dual purpose of demonstrating the problem-solving efficacy of the proof-of-concept and presenting an analysis of the OFFLOAD framework. Finally, Section VI concludes the paper by summarizing the key takeaways.

II. Approach for the Framework

A. Software Stack

The software stack behind this project orchestrates data movement and processing across several layers, facilitating efficient distributed computing on hardware accelerators. At the highest level, the application, developed in Python, leverages the rich functionalities of Apache Spark [15], a big data processing framework. Spark, running within the Java Virtual Machine (JVM), utilizes JDBC [16] drivers to seamlessly connect with a MySQL database server. This enables the application to execute SQL queries and retrieve data, benefitting from established and standardized interfaces typically used in big data environments. Apache Arrow [17] further enhances this process by optimizing data storage and distribution for accelerators. Its columnar format and efficient processing techniques contribute to improved performance.

Data retrieved from the database and prepped by Spark is then split and transferred to the next layer by custom firmware, which will be elaborated on later. This custom firmware acts as an intermediary, preparing the data for efficient processing by the hardware accelerators. The prepared data is then transferred down the stack to a custom-designed hardware network. This network utilizes a shared memory device driver interface, facilitating communication between a central accelerator master and multiple accelerator slave devices. The master leverages this shared memory interface to distribute tasks and the prepared data to the individual slaves. Each slave device executes its assigned tasks using custom accelerator slave firmware, specifically designed to optimize computations on the hardware accelerators.

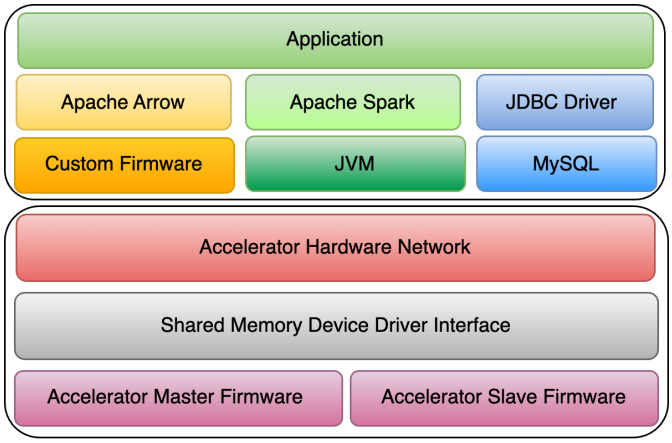


Fig. 1: Software Stack

B. Hardware Stack

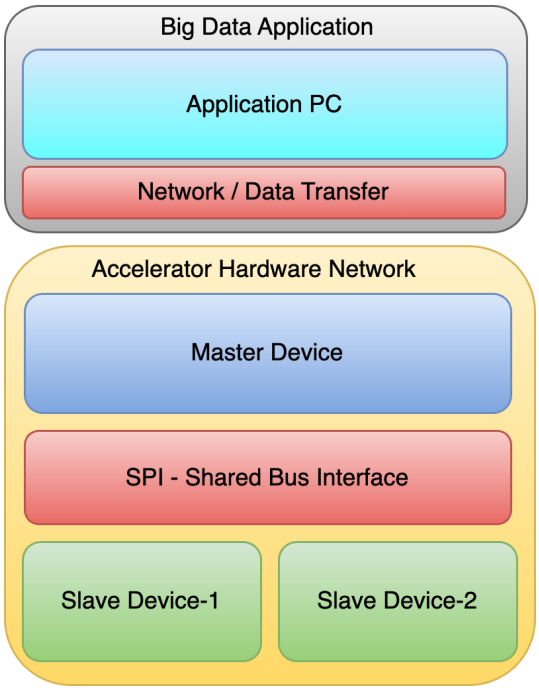


Fig. 2: Hardware Stack

The hardware stack comprises two distinct sections: the user application environment and the accelerator hardware network. The user application resides on a standard Windows PC connected to a network utilizing the ubiquitous TCP/IP protocol. This allows for seamless integration with existing computing infrastructure. The accelerator hardware network, on the other hand, provides a dedicated environment for

distributed computations. It features a Raspberry Pi (RPi) [18] acting as the master node, serving as the entry point for the network. The master RPi connects to the external world through the familiar TCP/IP protocol, ensuring smooth communication with the user application. Internally, the master communicates with a shared bus controller, implemented using a SN54HC153 multiplexer (MUX). This MUX facilitates efficient data exchange between the master and multiple slave accelerator RPis. All communication within the accelerator hardware network leverages the widely adopted Serial Peripheral Interface (SPI) bus protocol [19], ensuring reliable and standardized data transfer between processing units.

C. Flow of Information

The data flow within the system can be separated into two main stages: data distribution and query distribution.

1) Data Distribution: The data distribution flow orchestrates the movement of data from the database to the accelerator hardware network for processing. First, upon receiving a user query specifying an operation on a MySQL database table, Apache Spark, a big data processing framework, utilizes JDBC connectors to seamlessly connect and retrieve the requested data. Custom firmware then intercepts this retrieved data and performs two key actions. It first splits the data into smaller subsets based on the number of available slave accelerators (Raspberry Pi devices) within the network, facilitating parallel processing. Secondly, Apache Arrow is employed to convert these data subsets into a space-optimized Parquet format [20]. Parquet, a columnar storage format, offers efficient data storage and retrieval on disk. Additionally, Apache Arrow, with its in-memory data structure specification, further enhances performance by providing a standardized way to represent data in memory for communication across different programming languages.

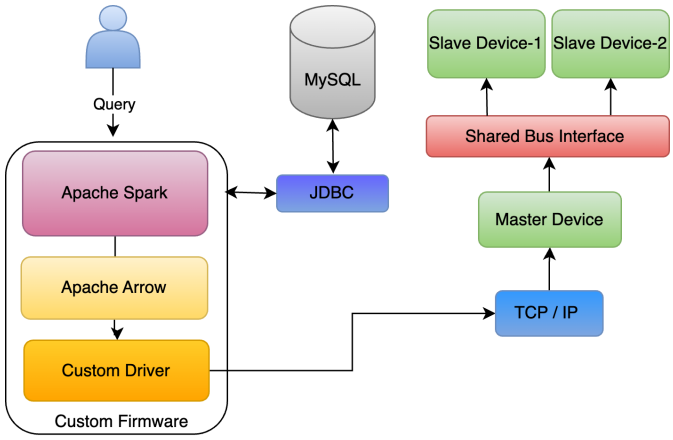


Fig. 3: Data Distribution Flow

Finally the master device retrieves data from the server using the efficient TCP/IP protocol with the Paho MQTT messaging library [21]. It then splits the data into subsets based on the number of available slave devices. These subsets are written to designated shared memory locations using a shared memory driver and a Shared Bus interface driver. Finally, the master signals the slave Raspberry Pi devices to confirm data reception and readiness for processing.

2) Query Distribution Flow: The query distribution flow manages the translation and execution of user queries within the distributed processing environment. Unlike traditional systems where the database server directly parses and executes SQL queries, this project utilizes a custom firmware approach due to the distributed nature of the data stored in shared memory across multiple accelerators.

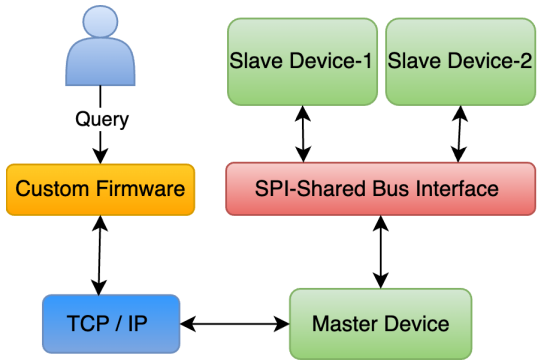


Fig. 4: Query Distribution Flow

The process begins with the user submitting a query, ideally adhering to standard SQL syntax. This query is then sent to the master accelerator (Raspberry Pi). The master's custom firmware intercepts the query, performs parsing to understand the operation requested, and leverages shared memory to distribute sub-queries or tasks to the respective slave accelerators. The specifics of this task distribution using shared memory will be further elaborated upon in a later section.

Each slave accelerator receives its assigned sub-query and utilizes the optimized Apache Arrow format for efficient processing. Upon completion, the results from each slave are sent back to the master accelerator using shared memory. Finally, the master's custom firmware aggregates these individual results, potentially performing additional operations as required, and ultimately populates the final outcome for the user. This approach facilitates parallel processing of the query across multiple accelerators, aiming to achieve significant performance improvements for complex data operations.

III. Hardware Setup

The hardware setup can be visualized 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 slave devices using General Purpose Input/Output (GPIO) pins. This allows for low-level control and data exchange between the Master RPi and the slaves. 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

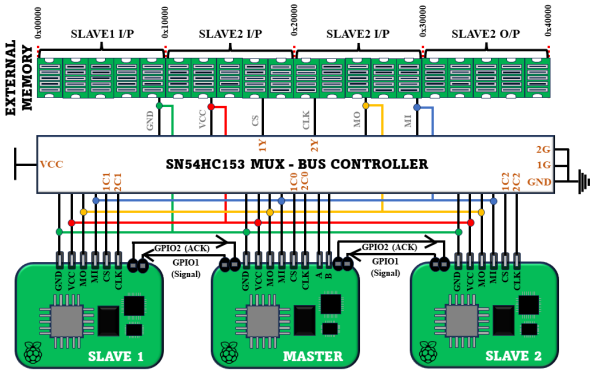


Fig. 5: Hardware Setup Block Diagram

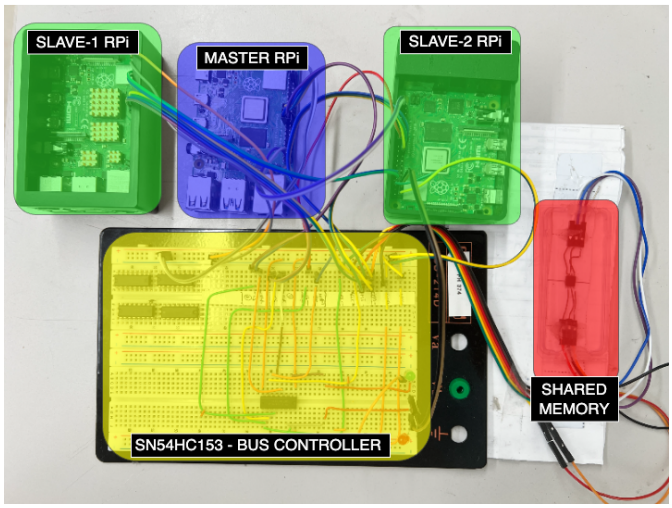


Fig. 6: Hardware Setup Real Image

slave devices. By granting and revoking access privileges, the Bus Controller ensures efficient memory utilization and avoids potential conflicts during data operations initiated by the slaves. This configuration facilitates a centralized control structure where the Master RPI coordinates communication and memory access for the slave devices within the accelerator framework.

The hardware setup of the framework can be visualized through a block diagram outlining the interconnection between its key components. These components include: a Master Raspberry Pi (RPi), one or more Slave 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 Slave-1 using two General Purpose Input/Output (GPIO) [23] pins. GPIO-1 transmits signals from the Master to Slave-1, while GPIO-2 serves as the acknowledgement channel from Slave-1 back to the Master. Following the master-slave paradigm, GPIO-1 is configured as an output pin on the Master and an input pin on Slave-1. Conversely, GPIO-2 acts as an input pin for the Master and an output pin for Slave-1. This communication pattern is replicated for Slave-2, utilizing GPIO-3 and GPIO-4 on the Master RPi.

The hardware framework facilitates data transfer between the master device, shared memory, and slave 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/Slave Input) pins from the master, slaves, 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/Slave 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.

However, the framework allows selective access for slave devices. Slave-1's CS pin connects to input 1C1 of the MUX, while Slave-2's CS pin connects to 1C2. Similarly, Slave-2's SCLK connects to input 2C2 of the MUX. This configuration empowers the master to control which device (itself or a slave) 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 slaves). 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.

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

● L→Low (0) ● H→High (1) ● X→Unused

Fig. 7: SN54HC153 Multiplexer Configuration to select Master and Slave Devices

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 Slave-1. Similarly, setting A to LOW (A=0) and B to HIGH (B=1) routes the shared memory connection to Slave-2.

This design ensures controlled and efficient communication between the master, shared memory, and slave devices within the SPI framework.

IV. Software & Firmware Implementation

The project codebase has been released as open-source on GitHub `link` [24] is organized 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), Slave Raspberry Pis (RPis), External Shared Memory module, and Shared Bus controller interface.

```
.
└── bist
    ├── AT45DB321E.py
    ├── bist_master.py
    ├── bist_slave.py
    ├── ext_mem_config.json
    ├── gpio.py
    ├── host
    │   ├── README.md
    │   ├── covid-config.json
    │   ├── main.py
    │   └── mysql-connector-java-8.0.23.jar
    ├── query_handler.py
    └── sn54hc153_mux.py
```

Fig. 8: Code-base folder structure

`AT45DB321E.py`: This 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 Slave RPis.

`bist_master.py`: This 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`.

`bist_slave.py`: This module resides on each Slave RPi within the system. Its execution is invoked using the command `python bist_slave.py --slave SLAVE_NUMBER`. The specific configuration for each Slave RPi is established within this module by parsing the `ext_mem_config.json` file.

`gpio.py`: This module is accountable for managing the GPIO (General Purpose Input/Output) configurations for both the Master and Slave RPis.

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

```
{
    "SLAVE_1_INPUT_ADDR" : 0,
    "SLAVE_2_INPUT_ADDR" : 65536,
    "SLAVE_1_OUTPUT_ADDR" : 131072,
    "SLAVE_2_OUTPUT_ADDR" : 196608
}
```

`query_handler.py`: This module executes the routines tasked with parsing SQL queries and subsequently distributing these parsed queries to the Slave RPis within the system.

`sn54hc153_mux.py`: This module serves as the bus controller component utilized by the Master RPi to govern access to the shared memory by all Slave RPis.

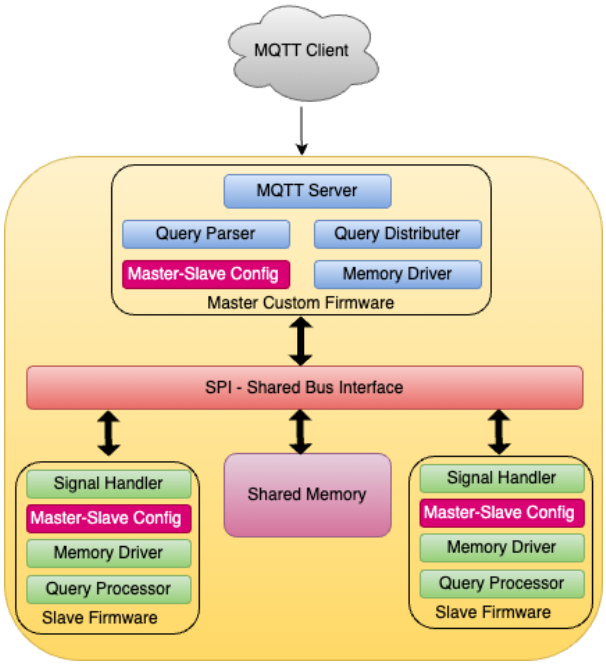


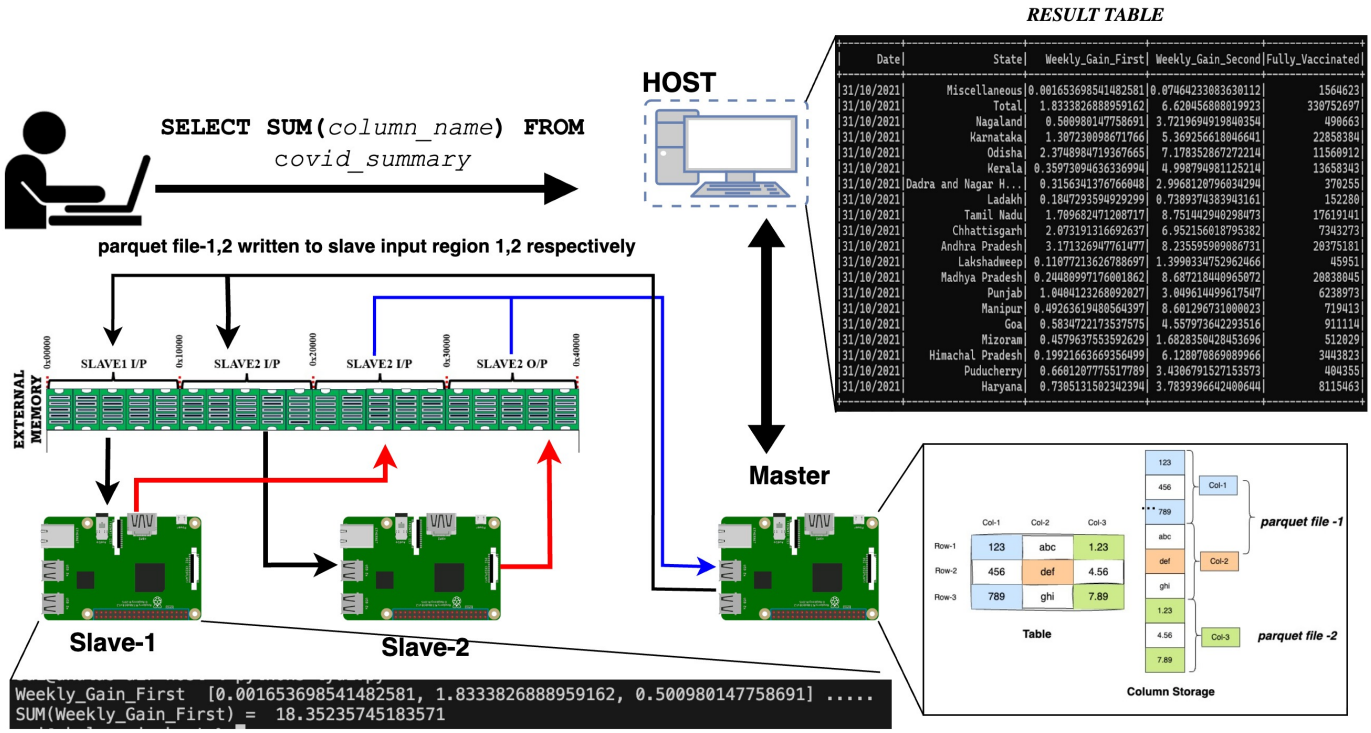
Fig. 9: Query Distribution - Firmware Layers

`host`: This 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.

`covid-config.json`: This configuration file specifies the details for connecting to the MySQL [26] database server, including the connector driver information, username, and password.

`main.py`: This 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.

`mysql-connector-java-8.0.23.jar`: This file comprises the JDBC driver, a software component that permits Apache Spark to establish a connection to the MySQL database server.



Date	State	Weekly_Gain_First	Weekly_Gain_Second	Fully_Vaccinated
31/10/2021	Miscellaneous	0.0016536985414825 81	0.07464233083630112	1564623
31/10/2021	Total	1.8333826888959162	6.620456808019923	330752697
31/10/2021	Nagaland	0.5009801477586 91	3.721969491984035 4	490663
31/10/2021	Karnataka	1.3072380986717 66	5.3692566184646 41	22858384
31/10/2021	Odisha	2.3748984719367665	7.1783528672722 14	11568912
31/10/2021	Kerala	0.4597309463633699 4	4.9987949811252 14	13658343
31/10/2021	Dadra and Nagar H...	0.3156341376766048	2.9968120796034294	370255
31/10/2021	Ladakh	0.1847293594929299	0.7389374383943161	152280
31/10/2021	Tamil Nadu	1.7096824712087 17	8.7514429482984 73	17619141
31/10/2021	Chhattisgarh	2.0731913166926 37	6.9521560187955 382	7343273
31/10/2021	Andhra Pradesh	3.1713269477614 77	8.2355959098867 31	20375181
31/10/2021	Lakshadweep	0.1107721362678 8697	1.3990334752962 466	45951
31/10/2021	Madhya Pradesh	0.2448099771600 1862	8.6872184480965 072	20838045
31/10/2021	Punjab	1.0404123268892027	3.0496144996175 47	6238973
31/10/2021	Manipur	0.4926361948056 4397	8.6012967310000 23	719413
31/10/2021	Goa	0.5834722173537575	4.5579736422935 16	911114
31/10/2021	Mizoram	0.4579637553592629	1.6828350428453 696	512029
31/10/2021	Himachal Pradesh	0.1992166366935 6499	6.1280708690899 66	3443823
31/10/2021	Puducherry	0.6601207775517 7789	3.4306791527153 573	404355
31/10/2021	Haryana	0.7305131502342 394	3.7839396642400 644	8115463

Fig. 10: Complete Task OFFLOADING Cycle

V. Results

The OFFLOAD framework's design prioritizes 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.

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

1) **Host Side:** Queries pertaining solely to data retrieval from the database, such as "select * from table_name," are processed entirely at the host level.
2) **Master Side:** Once the requested table is fetched from the database, it is transmitted to the Master Raspberry Pi (RPi) over a TCP/IP network. The Master RPi then partitions the table into multiple chunks based on the number of Slave RPis connected to the network (identified through the `ext_mem_config.json` configuration file). Two primary data splitting methods exist: splitting by the number of rows or splitting by the number of columns. While both methods offer advantages, column-based splitting leverages the accelerated performance of Apache Arrow. Apache Arrow stores data in Parquet file format, which, along with hardware accelerators, offers additional performance benefits due to contiguous memory locations. This eliminates the CPU cycles required for traversing through non-contiguous memory locations. For current implementations, exploiting ARM SIMD/NEON ISA (instruction set architecture) can be beneficial. However, the design holds the potential for exponential performance gains with the integration of dedicated accelerator hardware in the future. Consequently, the data is split by columns into Parquet files and subsequently copied into designated shared memory locations based on the pre-defined configuration.
3) **Slave Side:** Queries requiring processing, such as "SELECT SUM(column_name) FROM table_name," are handled differently. The host transmits the query to the Master RPi. The Master then analyzes the query to extract information regarding the specific column and operation to be performed. Based on this analysis and the configuration file, the Master identifies the appropriate Slave RPi for the task. The Master subsequently writes a subquery into the designated Slave memory location and invokes the corresponding Slave RPi. The Slave RPi then initiates the process by reading the input data from its designated columns within the shared memory using the Apache Arrow Parquet format. The Slave RPi executes the query on the designated column, writes the results to its assigned output memory location, and signals the Master upon completion. Upon receiving acknowledgments from all Slaves, the Master retrieves the results from their respective output memory locations and transmits the combined outcome back to the host via the network, finalizing the query execution.

In scenarios where multiple Slaves are involved, the Master

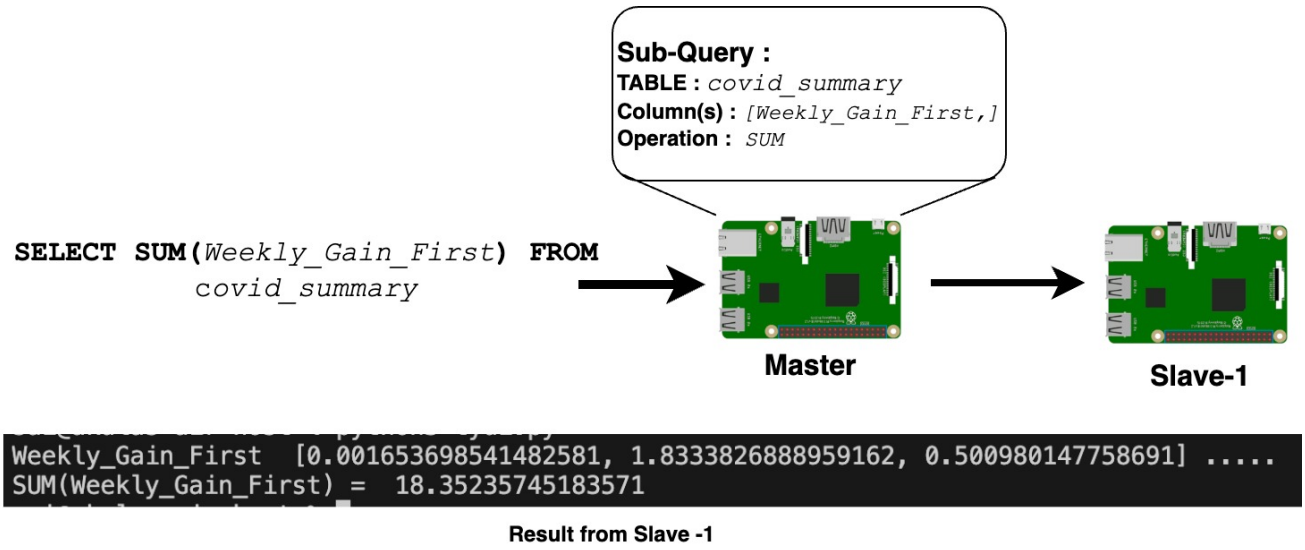


Fig. 11: Master dividing query into sub-queries for offloading to slaves

waits for acknowledgments from all participating Slaves before retrieving the results from their respective output memory locations. The Master then combines these results and transmits the final outcome back to the host via the network, finalizing the query execution.

TABLE I
Comparison of OFFLOAD with other Frameworks

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

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 unparalleled 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.

Another significant aspect is that OFFLOAD is the only open-source toolchain among the three frameworks. In contrast, Fletcher and TaPaSCo are built on FPGA toolchains, placing them under proprietary and costly constraints. This distinction not only fosters a more collaborative development environment but also eliminates the financial barriers associated with proprietary solutions.

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.

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.

VI. Conclusion

The OFFLOAD framework is the first of its kind and provides a pathway for offloading compute intensive tasks to a variety of hardware. In this paper, we have detailed the framework and its core aspects from data source, application coupling to binary instruction and data generation. We have shown validation of the framework by demonstrating hardware task offloading for MySQL Database queries. Our demonstration used Apache Spark and Apache Arrow in Python which are one of the most popular tools for database management and de-segmentation of datasets. On the hardware side Raspberry PIs were used to represent heterogeneous hardware. The task offloading was centralized with memory mapped I/O. The versatility, flexibility and scalability of this open-source framework can be vital in paving way for easier adaption of emerging machine learning and data analytics hardware. .

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 chips-to-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.