Scaling up data analytics in Python using multiple FPGAs

Shashank Aggarwal

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on 05.08.2020 at 12:00.

Student number:	4790626	
Thesis number:	Q&CE-CE-MS-2020-08	
Project duration:	February 1, 2020 – August 5, 2020	
Thesis committee:	Prof. Dr. Zaid Al-Ars,	TU Delft, supervisor
	Dr. Jan S. Rellermeyer,	TU Delft
	Prof. Peter Hofstee,	TU Delft
	Dr. Joost Hoozemans,	TU Delft

TUDelft

Abstract

Big data applications are becoming more commonplace due to an abundance of digital data and increasingly powerful hardware. One of these classes of hardware devices are FPGAs, which are being used today in various ways such as data centers and embedded systems. High performance, power efficiency, and reprogrammability are the primary reasons behind their wide use.

Another trend over the previous years has been to use distributed data processing frameworks such as Apache Spark to improve the performance of big data applications. Traditionally, such frameworks are deployed on commodity hardware to save costs. This approach is fairly popular, with organizations often having on-premise compute clusters or using a cloud provider to access a managed cluster.

This project attempts to combine the above-mentioned worlds - FPGAs and distributed data processing. We have designed an architecture that allows us to use FPGAs as end-devices in a compute cluster to perform the actual computation instead of CPUs. This architecture is designed by composing together several open source technologies and allows us to interact with an FPGA cluster using Python. Using a high-level programming language such as Python makes this system easy to use for software developers and data scientists, and also abstracts away the internal communication within the cluster. We have built prototypes based on this architecture for 3 hardware platforms (FPGA families) and 3 specific applications to demonstrate general applicability. We have observed noticeable performance gains in these applications by scaling up the FPGA cluster.

Preface

I am writing my Master thesis with the Accelerated Big Data Systems (ABS) group at TU Delft. The thesis is in fulfillment of the requirements of the EIT (European Institute of Innovation and Technology) Digital Master School program.

My technical major in this program is *Distributed Data processing*, and hence this project, which deals with parallelization of data analytic tasks, is very closely related to my interests. My previous experience with data analysis and distributed systems motivated me to choose this project, and fortunately, I was able to apply my knowledge in the project. Interestingly, being completely from a software background, I was pushed into the realm of hardware in this project, which took some effort to grasp initially. While admittedly there were parts when I was unsure of the underlying technology, I can now safely say that I have a much better understanding of FPGAs and their applications. I have also learned about several open-source technologies such as Apache Arrow, Dask, and Fletcher, which I hope to use again in real-world applications.

This was one of the most interesting and novel projects I have worked on, and for this, I would like to thank my supervisor, Prof. Dr. Zaid Al-Ars, for giving me this opportunity, and for his constant feedback and suggestions. Also, I am grateful to Joost Hoozemans, who patiently helped resolve all my queries and gave very insightful feedback. I would also like to thank other members of the Accelerated Big Data Systems group, especially Johan Peltenburg and Matthijs Brobbel, for their advice and tips throughout the project.

As goes without saying, I thank my family and loved ones for their support and encouragement.

Delft, August 2020

Contents

1 Introduction

1.1. Context

Data analysts and researchers are increasingly faced with the problem of extremely large data sets and computationally expensive operations. Distributed data processing frameworks (such as Apache Hadoop and Spark) running on top of large clusters are now the de facto method used for solving these problems at scale.

While horizontal scaling and distributed filesystems have solved the problems of large datasets, an issue that remains is the poor performance of the hardware itself. An example is deep neural networks, whose training and inference stages can be very computationally expensive, and not feasible to run on a CPU (even in a large cluster of CPUs). GPUs offer a solution to this problem due to their fast arithmetic capabilities.

Another type of specialised hardware that is used for data analysis is Field Programmable Gate Arrays (FPGA). These offer the advantages of performance, reprogrammability, and high power efficiency. In view of this, large cloud providers such Amazon and Microsoft now provide FPGAs as generic cloud resources, along with more traditional resources such as CPUs and GPUs.

As was the case with CPUs, horizontal scaling lets us extract more utility from FPGAs. Creating a cluster of FPGAs lets us perform data analytics jobs that are too large for a single FPGA instance. This is where this project fits in. During the course of the thesis, we have developed several prototypes as proofs of concepts to demonstrate the possibility of using multiple FPGAs to improve the performance of data analytic jobs. We have also implemented distributed versions of existing FPGA examples built previously by students at TU Delft and demonstrated the performance gain obtained due to parallelization.

A primary focus has been to make this process friendly for software developers by using a high-level programming language and popular data processing frameworks.

1.2. Problem statement

While FPGAs have been shown to be useful for several common data analysis tasks, there are certain problems we observe when trying to use them in real-world applications:

1. Scalability issues
2. Absence of support in high-level programming languages

1.2.1. Scalability issues

FPGAs have inherent limitations when trying to work on large data analysis jobs:

1. *Area limitations of FPGAs*: FPGAs have limited memory buffers and programmable logic elements. In this case, we cannot scale up our data task beyond what a single FPGA can handle.
2. *Optimizing kernel design is hard*: Accelerators can be optimized to reduce synthesis time and increase runtime performance with several strategies. However, for a hardware engineer, it is often time-consuming to look for bottlenecks and optimize the kernel.

With the increasing size of data we process in modern data analysis pipelines, it is often not possible to have a single FPGA perform the entire computation for us. Of course, it is possible to use larger FPGAs with a high number of logic cells, memory bandwidth, I/O segments, etc., but these are often very expensive, and difficult to design for in an optimized manner.

1.2.2. Absence of support in high-level programming languages

Designing FPGA accelerators is usually much more difficult and time-consuming than their software counterparts, mainly due to complex tools and slower debug cycles. Hence, FPGA design is usually done by a few skilled and experienced hardware engineers. Hence, for software engineers to be able to use FPGAs for data analysis, it is important to be able to deploy and reuse existing FPGA designs.

Software developers and data engineers find it difficult to use FPGAs for day to day tasks, even if there is an existing FPGA accelerator available with the required functionality. This is because FPGAs are not directly accessible via high-level programming languages such as Python. The functionality of an FPGA is often exposed by binary executables or other low-level APIs (such as OpenCL), which are unwieldy to use via programming languages such as Python or have a very steep learning curve. It is for this reason that libraries such as Pynq [7] are being developed, which let software developers use Python to interact with an FPGA without having to think about the underlying hardware.

1.2.3. Research questions

This project investigates the above mentioned problems, and aims to provide solutions to mitigate them. More precisely, the following research questions are answered:

1. Is it possible to improve the run time performance of existing FPGA data analytics applications by using multiple FPGAs in parallel instead of one?
2. Is is possible to interact with the FPGA cluster using a high level programming language (such as Python) transparently?
3. Can we further optimise the performance of the above applications using a columnar data format to represent our application data?

1.3. Solution approach and contributions

To answer the above mentioned research questions, we propose a simple architecture with 2 main components:

1. *A cluster of FPGAs*: Instead of using a single FPGA, we connect several FPGAs together. Since FPGAs are usually attached to a complete CPU host, this means that we connect several hosts together into a local network.
2. *A Python interface to the cluster*: We should be able to communicate with the cluster directly via Python. This means that we should be able to issue commands and transmit data between our Python process and all the FPGAs. Additionally, we should be able to split our task easily into smaller chunks based on the number of connected FPGA instances.

With this approach, given access to an FPGA cluster, software engineers and data scientists will be able to scale up their tasks transparently to the entire cluster without having to think about the internal workings of the cluster. The technical architecture of the system will be explained in subsequent chapters.

The main contributions of this project are as follows:

- We design an architecture to enable parallel data analysis on multiple FPGAs geared towards software engineers.
- We develop a few prototypes to validate our architecture. These prototypes use a combination of different software and hardware platforms to demonstrate the general applicability of our solution.
- We provide performance benchmarks to demonstrate speedup due to the use of multiple FPGAs instead of one.

1.4. Thesis structure

The structure of this report is as follows: In Chapter 2, we first discuss some existing research explaining use cases where FPGAs have been shown to be useful. Then we look at existing examples of using multiple FPGAs in a cluster/data center. This is closely related to our work and provided us the initial motivation for the project. Then we give a quick background of the tools and concepts we have used to build our solution.

In Chapter 3, we discuss the architecture of the proposed solution and specify the steps we have taken to build a few prototypes based on the architecture. We also provide a few performance benchmarks to demonstrate the speedup obtained by using multiple FPGAs.

In Chapter 4, we demonstrate additional prototypes we build using a modified architecture. In this case, we use the Apache Arrow data format to represent our input data to the FPGAs. We discuss the advantages of this architecture, followed by some performance benchmarks. The final chapter discusses our conclusions and future work.

2 Background and related work

In recent years, FPGAs have been deployed in a variety of applications. Owing to their ability to be reprogrammed and their low power consumption, they have been used in machine learning, graph processing, database applications, etc. In the following sections, we first present existing research demonstrating the use of FPGAs in various types of systems - streaming applications, machine learning, and bulk data transformation. Then, we discuss a few systems that utilize multiple FPGAs to scale up computational tasks. Many of these are based on open source technologies such as Spark and Pynq and aim to solve the same problem as ours.

Finally, we explain the important tools and concepts that we have used throughout the project to provide readers some context.

2.1. FPGA applications

FPGAs have been shown to be able to accelerate many software-based systems, especially for compute-intensive big data applications, such as genomics algorithms [29][19], data decompression [16], and image processing [18]. By offloading the compute-heavy or I/O heavy operations to FPGAs, these systems have been able to improve their performance. In this section, we describe some categories of these systems and how they use FPGAs.

2.1.1. Streaming applications

Networking devices such as routers and switches have to be able to process traffic at high throughput, and also comply with changing standards. In this case, FPGAs can offer performance at par with dedicated network processors. In [14] for example, the authors developed a switch fabric entirely using FPGAs and found the port-to-port latency to be similar to that of existing ASIC-based switches. FPGAs are also used for implementing network security through packet filtering. Network intrusion detection systems (NIDS) identify packets containing viruses, trojans, etc. using regular expression matching. In [20], and FPGA-based string matcher was shown to outperform the GNU regex program by 600 times for large regular expression patterns.

Digital signal and image processing is also a suitable application for FPGAs. Due to a fixed transformation logic, which is generally simple, highly pipelined accelerators can be built to achieve high throughput. For example, authors in [21] developed a Fast Fourier Transformation architecture for FPGAs with low power consumption and low latency. Background detection, which is used in applications such as video surveillance and traffic monitoring, requires hardware processing for large output frame rates. While GPU implementations have been developed, they are unsuitable for embedded systems due to their power constraints. In [17], a real-time background identification circuit was implemented on an FPGA with the ability to process video with a high frame size and frame rate.

In [15], the role of FPGAs in several Internet of Things (IoT) applications is discussed, and compared to traditional IoT edge devices such as microcontrollers and ASICs. Some of the applications discussed include cryptographic algorithms and image processing.

2.1.2. Machine learning

Several common machine learning algorithms have been implemented on FPGAs, and have been shown to outperform their corresponding software implementation, often by an order of magnitude or more.

K-means clustering is a very popular algorithm which is used in several data analysis applications such as pattern recognition, image processing, business analytics, etc. This algorithm is inherently parallel, and also has a simple control flow. Hence, it is especially suitable to be implemented on an FPGA. In [35], the popular modification of the K-mean algorithm called the *filtering* algorithm was implemented on an FPGA. In [13], a distributed version of the K-means algorithm is considered. The implementation was deployed on an FPGA cluster with 3 nodes connected via Ethernet, and a speedup of over 15x was observed as compared to the software implementation.

Another class of ML algorithms that have been implemented in FPGAs are Neural Networks (NN). Network models such as convolutional NN and recurrent NN have improved performance over traditional ML algorithms, especially in areas of image, video, and speech processing. Several FPGA accelerators have been built for such neural networks, and these offer advantages over CPUs and GPUs due to better performance and power efficiency, respectively. In [22], [25] and [26] for example, FPGA accelerators have been build for binarized neural networks, where the weights and/or activations are restricted to be 1- bit boolean values. Binarized neural networks lead to more compact models, and the authors above show their models to be comparable in performance to GPUs with better energy efficiency.

2.1.3. Data transformation

Here, we discuss some commonly performed data transformations such as data compression and encryption, and their FPGA implementations.

Generally, CPU based compression and decompression implementations provide limited throughput due to memory and CPU constraints. In [28], the authors developed an FPGA hardware architecture for GZIP compression and decompression by offloading CPU intensive operations to a cluster of 4 Intel Cyclone-III (formerly Altera) FPGA devices and observed better disk utilization and (de)compression throughput. In [32], a decompressor for Snappy-compressed files was implemented on an FPGA and the input throughput of the decompressor was demonstrated to exceed that of a single i7 core. Furthermore, placing more such engines on the same FPGA led to a corresponding improvement in performance.

FPGAs can also be coupled with Database Management Systems to offload some expensive data analytics operations. Apart from the traditional transactional queries which are performed on a database, it is often necessary to perform ad-hoc analysis on the data to gather business insights. In [33], such CPU-intensive analytics queries were offloaded to FPGAs. These operations include row decompression and predicate evaluation. An end-to-end performance improvement of up to 6.2x was observed, including the data transfer time and time spent on the FPGA.

2.1.4. Fully HW vs HW/SW co-design

An important aspect to consider in the above applications is whether the system is deployed entirely on specialized hardware, or only partially. These two approaches are generally referred to in the literature as fully hardware design and HW/SW co-design.

In the *fully hardware-implemented* approach, the entire task (machine learning algorithms in most cases) is implemented on the FPGA. In [12] and [24] for example, the entire map and reduce stages for applications such as K-Means clustering and Finite impulse response (FIR) filter are implemented on the FPGA, with the host CPU only responsible for distributing and collecting data. While this provides high performance, it also requires more effort during the design phase and is also more HW resource-heavy.

Co-designed algorithms, on the other hand, offload only the time-consuming parts of the algorithm to the FPGA. While this does not achieve as much speedup as in the fully-hardware accelerated case, it requires lesser FPGA resources and is also simpler to design. In [27], various machine learning algorithms (such as K-means, SVM, etc.) are analyzed for performance bottlenecks (called ’hotspots’). These bottlenecks are then implemented on the FPGA using High Level Synthesis (HLS) tools. HLS is the process of transforming high-level code (in C/C++, for example) into lower-level code suitable to be deployed on FPGAs. Of course, even when using high-level programming languages, the designer has to optimize it to fully utilize the parallelizable nature of FPGAs using techniques such as loop unrolling (running multiple loop iterations in parallel) and pipelining (parallel execution of different stages of a pipeline).

2.2. Parallelization using multiple FPGAs

Here we discuss frameworks built to enable the use of a cluster of FPGAs to improve the performance of data analysis tasks. These frameworks are not built for specific applications, but instead, propose general architectures allowing for the deployment of any specific application.

SPynq [23] is a framework that allows us to use Apache Spark, a popular distributed data processing framework, to perform data analysis on a cluster of FPGAs, which in their case was a cluster of 4 Xilinx Pynq-Z1 FPGA boards. While the example of logistic regression has been implemented on the cluster, it is possible to replace it with any other operation/algorithm, provided one can develop the required FPGA hardware designs. The resulting architecture has been compared to both high-performance cloud computing Xeon clusters and low-power embedded processors and been shown to perform faster with better energy efficiency. An important focus has been to expose an easy to use API for software developers to enable them to interact with the FPGA, i.e., allocate/deallocate buffers, obtain results from the programmable logic. This work is in close relation to what our project aims to do, with the following differences:

- SPynq is based on the Spark environment (JVM), while our system is based on the Python ecosystem.
- SPynq works with Xilinx's Pynq framework, and hence supports only compatible FPGA boards. Using Fletcher on the other hand, as we have, lets us integrate with other platforms as well (OpenCAPI, for example).

There is a large body of research on making FPGAs available as generic cloud resources. The intention with such systems was to allow organizations to procure FPGA instances instead of ordinary VMs from the cloud provider. Since FPGAs are generally more performant than VMs and also power efficient, this stands to benefit both the cloud provider and its customer.

In [10], FPGA resources were made available as OpenStack cloud resources, allowing users to quickly scale up/down the size of their clusters. FPGAs were presented as virtual cloud accelerators and were shown to be able to boot up faster than virtual machines and be more performant.

A similar framework was developed by IBM in [11], which abstracted FPGA as OpenStack resources. Sample applications, such as load balancing, encryption, etc. were implemented and the overhead due to virtualization was shown to be minimal.

2.3. Background about used frameworks/technologies

This project integrates several hardware and software frameworks together in order to build a complete data analytics workflow. Each of these components was chosen based on their ease of use, as well as their availability (since some of them are proprietary) on TU Delft's infrastructure. In the following paragraphs, we explain the important ones in a ”chronological” sequence, in the sense that we start with tools we used to build/test FPGA accelerators, then the platforms to deploy these accelerators, and finally the frameworks to build and use multi-node clusters of the deployed accelerators.

2.3.1. Apache Arrow

Arrow is a memory format based on a columnar representation of data structures. This format is designed to be used for data analysis tasks since it allows CPUs to take advantage of optimizations such as SIMD (Single instruction, multiple data) and cache locality. Arrow also provides software libraries in more than 10 commonly used programming languages to create and transfer Arrow-formatted data structures (called Record batches and Tables). The memory layout is designed such that no (de)serialization is needed when transferring data over the wire or between processes. In this project, we have used Arrow for representing our dataset in memory and distributing it to the FPGAs. This has helped to reduce the communication overhead between the cluster nodes and also lets us make use of Fletcher (discussed subsequently) to interface with the FPGAs.

2.3.2. Fletcher

Fletcher [31] is a collection of tools and libraries that an FPGA developer can use to build and use accelerators which consume and/or produce data in the Apache Arrow format. Fletcher is based on the idea that columnar in-memory data formats such as Arrow are highly suitable for FPGA accelerators due to a contiguous memory layout which allows for faster data traversal. Fletchgen, which is one of these tools, creates boilerplate Hardware Description Language (HDL) templates based on the schema of the Arrow data structure expected by the accelerator during runtime. The developer provides the so-called Arrow *schema*, and Fletchgen outputs VHDL code for the accelerator kernels. This code sets up memory input and output streams, which the developer can hook on to, and then implement the logic of the kernel. This decreases the setup time for developing a new FPGA kernel. An important consideration here is that Fletcher supports any arbitrarily complex Arrow schema, which is necessary to support a broad range of applications. Fletcher does this by implementing components such as *Column Readers/Writers*, *Buffers* to support corresponding Arrow data structures such as arrays and buffers [30].

Fletcher also provides platform-agnostic host libraries that allow us to send and receive Arrow data structures from the FPGA during runtime. This allows the same data-analytics code to be reused on any underlying hardware platform supported by Fletcher (for example - AWS F1, CAPI SNAP, etc.)

2.3.3. Hardware Description Language (HDL)

HDLs are computer languages that are used to describe digital circuits. VHDL and Verilog are popular examples. These languages allow a hardware developer to specify the layout of the digital circuit using programming language constructs. While these look similar to traditional programming languages such as C/C++, they work on the notion of *data flow*, i.e., the logic of transforming input data into output data as opposed to *control flow*, which is an imperative list of instructions for the CPU to follow. HDL code is used for one or both of the following:

- *Synthesis*: This is the process of generating the actual digital circuit description in terms of logic gates, which is then deployed on real hardware such as an Application-specific integrated circuit (ASIC), FPGA, etc.

- *Simulation*: Simulation tools allow the developer to test the HDL program without using real hardware. This makes it easier to debug and optimize the circuit.

2.3.4. Pynq

Pynq (abbreviated from Python + Zynq) is an open-source framework from Xilinx designed originally for their Zynq SoCs. It allows accelerator designers to provide an easy-to-use interface to their accelerators using Python. This way, software developers can utilize accelerator wrappers called *overlays* without having to worry about the interface of the internal accelerator kernel. Using pre-installed Jupyter notebooks on the SoC, the developer can download and instantiate pre-built overlays and use them directly from Python.

The underlying FPGA accelerator can be built using any Xilinx Synthesis tool (such as Vivado and Vitis, etc.). The *Pynq* python package can then be used to set register and buffer values on the Programmable Logic (PL) of the FPGA.Xilinx also provides pre-built overlays for a few common use-cases. One of these which we have used in this project is the Quantized Neural Network (QNN) overlay. These neural network overlays let us perform image classification on popular image datasets (MNIST, Cifar10).

The reason we use Pynq for this project is that its Python support makes it work well with *Dask*, which is a Python library for distributed data processing.

2.3.5. Dask

Dask is a Python framework which lets us parallelize data analytics task natively in Python. It allows us to split up large data processing tasks into smaller chunks, and distributes them in one of two ways:

1. Large multi-node clusters connected by a network, or supercomputers.
2. A single machine, in which case the data task is split across the available CPU cores, also letting us use local disk in case of memory constraints.

Dask also provides parallelized versions for several functions from common Python libraries such as Numpy and Scikit-learn, allowing the programmer to parallelize them without having to learn a new API. In this project, however, we use the lower level functionality offered by Dask, which lets us specify arbitrary data sets and custom business logic. For example, as we will discuss later, we use Dask's *scatter* and *submit* functions to distribute binary data to several worker nodes.

3

Parallelization on multiple FPGAs

The first half of this project dealt with developing proofs of concept to demonstrate that any data analytics task can be split up to run on multiple FPGAs. That is, using concepts of data parallelism, we can use popular data processing frameworks to execute the same data transformation function on all the subsets of a dataset simultaneously on different FPGAs.

3.1. Solution architecture

We now describe the general architecture of the system we have built. The general idea was to combine different (mostly open-source) tools and layer them in such a way that a Python data analytics script would internally be able to communicate with remotely connected FPGAs and distribute the task transparently among them. A schematic diagram of the architecture is shown in Figure 3.1.

The main components of the system are as follows:

1. *Dask client*: This is a Python class that acts as the entry point to access a Dask cluster. Initialising a client requires a Dask-scheduler to be running (possibly on another machine) beforehand. The client can then connect to this scheduler, and provides the users an API to submit tasks to the scheduler. Optionally, as in our case, the client can distribute the data to the workers before submitting the computations to the scheduler. The other possibility is that the workers can directly access the data from a distributed file system/storage.

2. *Dask-scheduler*: This is a process that coordinates all the connected worker nodes. Given a data analytics task (in the form of a task graph), it is responsible for executing it in parallel on the workers. Depending on the type of scheduler instantiated by the user, it may distribute the task between different threads, processes, or machines. In this project, the Dask client and the scheduler were running on the same machine. The client would read the input data (either from a

file or from an in-memory python data structure such as a Numpy array), scatter it to the workers, and then submit the task to the scheduler, which would, in turn, trigger the required computations on the workers.

3. *Dask worker(s)*: These are processes that perform the actual computations using a thread pool. The size of this thread pool is by default equal to the number of CPU cores. In our case, since we have only one FPGA instance connected per worker, we set the number of threads in the pool to be 1. The workers also hold the computed results unless explicitly asked for by the client or other workers. Workers also spill this data to disk if the memory usage exceeds user-specified thresholds.

4. *FPGA interface library*: These are Python libraries that let us interface with the accelerator. These expose APIs to send data to the FPGA, set specific registers, and get the results back. In this part of the project, we used the Pynq library for this purpose. Pynq provides APIs to download the bitstream in the Programmable Logic, and then interact with it by setting register values.

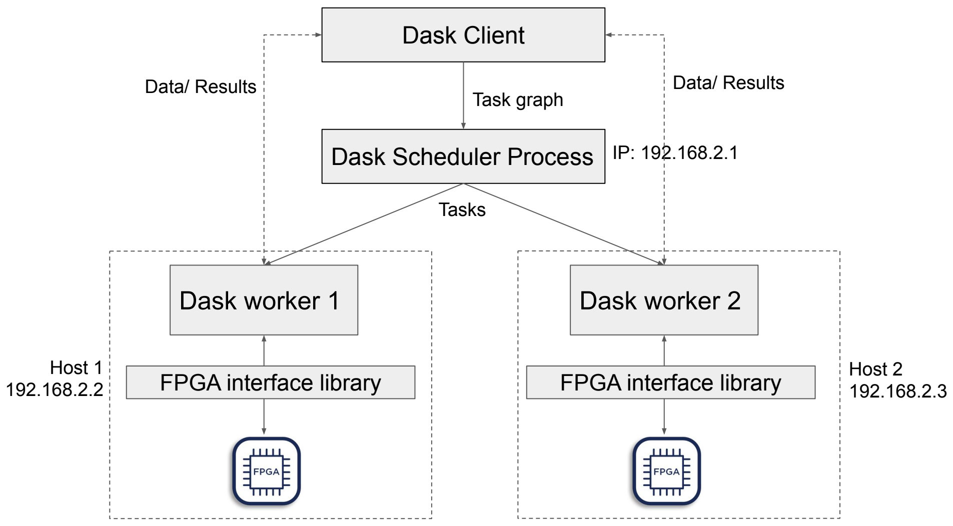

Figure 3.1: Architecture of the system

In the following section, we describe the steps of setting up this system in more detail:

1. Build/reuse suitable accelerator(s) to install on the FPGAs

2. Setup a small *Dask* cluster of 2 FPGAs over a local network. These nodes should communicate with a Dask-scheduler running on a third host.

3. Using Python, one should be able to specify an input dataset, and a custom data processing method one wants to run on the dataset.

3.2. Multi-FPGA system setup

3.2.1. Build/Reuse appropriate accelerator bitstreams

In order to use the Programmable Logic (PL) of the FPGA, we needed to deploy an accelerator overlay on the Pynq boards. We can design the PL ourselves using Xilinx's Vivado software. This will generate the bitstreams (binary files) that we can install on the board. Also, a Python driver can be optionally implemented which wraps the functionality of the bitstream into easy to use methods.

In our case, however, we reuse existing overlays provided by Xilinx. In particular, we experimented with an image resizer accelerator and an image-classification neural network overlay.

Image resizer overlay

This [8] is a pre-built bitstream provided by Xilinx. It accepts an image in the form of a buffer array and then resizes it based on a user-specified scaling factor. A purely-software implementation based on OpenCV is also provided for performance benchmarking.

There is no Python 'driver' provided for the overlay, and hence the process of calling the accelerator kernel is fairly low level. Some of these APIs (provided by the Pynq package) we have used in our python code are:

- `Xlnk.cma_array(shape, dtype)`

  Create a contiguous buffer of *shape*, data type *dtype*

- `DMA.sendchannel.transfer(buffer)`

  Transfer *buffer* from memory to the DMA write channel, used to send input data to the FPGA.

- `DMA.recvchannel.transfer(buffer)`

  Transfer data from the DMA read channel into *buffer*, used to read output data from the FPGA.

- `Pynq.overlay.DefaultIP.write(offset, value)`

  Write *value* to the address *offset* of the MMIO device.

However, the hardware developer can choose to provide a driver to make it easier to access the accelerator's functionality by extending Pynq's *DefaultIP* class, which is the default driver in case a more specific driver is not provided.

Image classification overlays

These overlays are a set of 5 quantized neural network overlays provided by Xilinx to perform image classification. The neural networks are based on the topologies described in the FINN paper [34]. The overlays support different precision for the weight and activation of the neurons in the network, namely either 1-bit or 2-bit precision. The networks are pre-trained on various datasets such as MNIST [6] and CIFAR10 [4]. Quantized neural networks help overcome the limitations of limited resource and power budgets in floating-point based neural networks. A purely-software implementation is also provided for performance benchmarking.

For this overlay, a python driver in the form of the *bnn* package is provided which exposes convenient wrappers around the overlay functionality:

- `bnn.CnvClassifier(bnn.NETWORK_CNVW2A2,'cifar10',bnn.RUNTIME_HW)`

  Instantiate a classifier instance, passing in as arguments the type of neural network, the dataset name, and whether to run the software or the hardware implementation.

- `classifier.classify_image(img)`

  Return the class to which the image *img* belongs

Vector addition

The latest version of the Pynq package, released recently, also supports Xilinx's Alveo FPGA cards and AWS-F1 instances. TU Delft's data cluster has a server installed with two Alveo FPGA cards, and hence we chose this specific overlay as well to demonstrate our results on Alveo. This overlay is provided in the official Alveo Pynq repository [1], and performs 2D vector addition. This example allows for straightforward parallelism by allowing us to split the dataset into smaller parts and performing addition separately on each part.

3.2.2. Setup a cluster of FPGAs

FPGAs used

The first step was to build a small cluster of 2 FPGAs. We used two different FPGA devices:

1. *Pynq-Z1*: These boards belong to Xilinx's Zynq-7000 family of FPGAs. FPGAs from this family consist of a programmable FPGA fabric along with an ARM-based processor. The Pynq-Z1 board, specifically, consists of a dual-core ARM A9 processor and runs on a Linux OS. More importantly, it is pre-installed with a Jupyter Notebook Web server and the Pynq python package:

   - *Jupyter Notebook*: Jupyter notebook is a web application that allows users to run programming languages such as Python on the browser. The interface allows for quick development and debugging cycles, and is the de facto standard used today for data exploration/analysis tasks in Python.
   - *Pynq* : Pynq is a Python package that allows users to include pre-built FPGA accelerators called *overlays* into their Python code, similar to including software packages. Pynq also lets the programmer interact with the FPGA, for example, allocate memory on the Dynamic RAM (DRAM), stream audio/video from the on-board mic and HDMI ports, etc.

   Pynq-Z1 FPGAs are particularly suited for quick prototypes due to their small dimensions and host of IO ports. Two Z1 boards for this project were provided by TU Delft's Accelerated Big Data Systems group.

2. *Alveo*: These are larger, more expensive FPGA cards aimed at large data processing workloads. We had access to 2 Alveo cards installed on a single server - the Alveo U200 card, and the larger Alveo U280 card. The Pynq library supports multi-device operations as well, i.e., it is possible to specify which card we would like to run the bitstream on using simple arguments to the Pynq API. In our case, we ran two dask workers, each interacting with a separate card to enable parallel execution.

Setup a Dask cluster
A Dask cluster is a set of worker nodes and a single scheduler node. The worker nodes perform the actual data analysis, while the scheduler is responsible for managing the worker nodes. This involves distributing and collecting data to/from the worker nodes, building task graphs and scheduling them, etc. A dask cluster can be set up in various ways:

- Manual setup using helper command-line utilities provided by Dask.
- Kubernetes: Dask provides Helm charts to quickly set up a cluster where schedulers and workers are deployed as pods in a Kubernetes cluster.
- Hadoop YARN cluster: Dask can also deploy itself on YARN cluster, allowing one to use YARN's resource management capabilities

In our case, we chose to use the first method, i.e., manually setting up the cluster using *dask-scheduler* and *dask-worker* utilities because of its flexibility and ease-of-use. The following steps were performed to obtain a fully functioning Dask cluster. These steps serve as guiding principles for anybody looking to get an FPGA cluster up quickly.

1. In the case of Pynq Z1, connect the 2 boards to a local Laptop via a Gigabit router and Ethernet. This setup is shown in Figure 3.2. This step is not needed for the Alveo cards since they are attached to the same host.
2. Install `dask-distributed` on the 3 nodes (2 FPGA node, one local scheduler laptop). Python's `venv` module can be used here for ensuring isolation and reproducibility of the setup environment.
3. Run `dask-scheduler` on the laptop. This runs a scheduler process on the node and spits out an IP address for workers to connect to.
4. Run `dask-worker <ip-of-scheduler>` on each of the two Z1 boards (after connecting via SSH). This spawns worker processes on the FPGA which are internally connected to the scheduler on the laptop. In the case of Alveo, we ran this command twice in two separate shells and used an environment variable to specify which device/card to use.

At this point, we had a fully functional Dask cluster. This was verified by running a simple matrix transpose Dask example on the scheduler node. Using Dask's dashboard, we could see the data computation being split up between the two workers. Note that at this point, we were not using the nodes' capabilities as an FPGA, but only as CPU nodes.

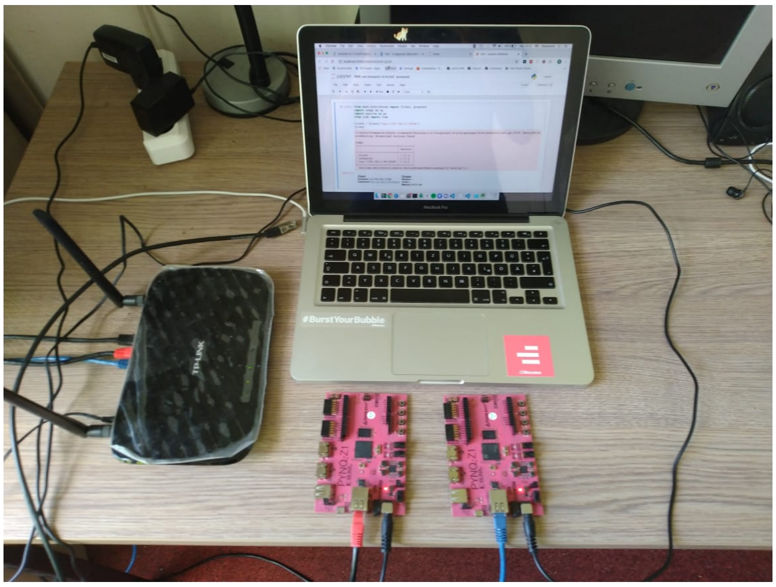

Figure 3.2: Cluster setup

3.2.3. Use Dask to distribute the data analysis

When running in distributed mode, Dask provides broadly 2 ways of distributing work:

1. *High-level collections*: Dask implements scalable versions of common Python data structures often used for data analytics. For example, Dask provides an Array API, which is functionally similar to the Numpy API, but internally it executes many of its operations in parallel on the cluster. Similarly, it provides the Dataframe API as a parallel version of Panda's Dataframe API.

   This way of using Dask is especially useful if we want to parallelize an already existing codebase with minimal code changes, and is also easier to understand for people familiar with the Python data science ecosystem.

2. *Low-level interfaces*: Dask also provides a low-level API to allow us to execute custom algorithms on arbitrary data structures. This API lets us submit functions along with their arguments to the Dask-scheduler, and the Dask-scheduler executes them on the cluster.

For our project, we have used the second approach since our computation was highly custom and application-specific. For example, we wrote custom functions that contain the logic to communicate with the FPGAs. Following is a Python snippet that demonstrates the typical flow in our examples (The entire example is in Appendix A).

```python
# Import the python driver for the overlays
from dask.distributed import Client

# Instantiate Dask client to communicate with the cluster
client = Client(IP_OF_DASK_SCHEDULER)

# Read input data (from a file or an in-memory Python object)
```

```
input_data = read_input_data()

# Split data into based on number of workers present
split_data = split_data(input_data, number_of_workers)

# Scatter data to the workers
distributed_data = client.scatter(split_data)

# Instruct the scheduler to run the function 'run_on_worker' on all the workers
futures = client.map(run_on_worker, distributed_data)

# Wait until all the operations complete, and then get the results
results = client.gather(futures)
```

Here, `run_on_worker` is the function that is executed on the workers. This method is responsible for sending the data to the FPGA, fetching the results back, and returning it to the Dask client. Depending on the chosen overlay (refer to Section 3.2.1), we use the appropriate API provided by the overlay to communicate with the FPGA.

3.3. Experimental results

To verify the performance gain obtained due to parallelism, we carried out experiments to measure the time to completion of the above examples for two cases:

1. Single Dask worker: Only one dask-worker was instantiated. This served as the basis to measure our speedup against since it represents a non-parallelized version of the application.

2. Two Dask workers: Two dask-workers were instantiated, and each worked on only half of the input dataset. This represents the parallelized version.

This performance analysis was carried out for two of the three examples in section 3.2.1 - the image classification example and the vector sum example.

3.3.1. Image classification using QNN on Pynq-Z1

This example involved the classification of 10000 32*32 color images into 10 classes - airplane, bird, cat, etc. using a quantized convolutional neural network with 2-bit weights and activations.

Experimental setup: Two Pynq-Z1 FPGA devices were connected via a Gigabit router to a laptop. These devices were loaded with a bootable Linux image available from the Pynq website [7]. This image includes the *Pynq* python package (version v2.5.1). A Dask cluster was created using the Dask Python package with Python 3.6. The test batch of the CIFAR-10 [4] dataset was used for testing the neural network. Pre-trained neural network models were obtained using Xilinx's [3] Python package. This package provides bitstreams for different neural networks, including the 2-bits weight and activation NN we have used for this experiment. It also includes a software-only implementation of the network built using Xilinx's C++ deep learning framework Tiny-CNN [9] .

Attempting to run the software version of the network on the Pynq-Z1 led to a timeout. This is likely due to the small ARM Cortex-A9 processor present on the board.

The FPGA implementation, however, completed successfully. For classifying the entire dataset using 1 dask worker, it took a total of 38s. This includes the time taken to read the input file, send it to the remote worker, execute the inference on the FPGA, and get the result back. On scaling the setup to 2 Dask workers, the total time taken was reduced to 22 seconds, implying a speedup of 1.7x (Figure 3.3).

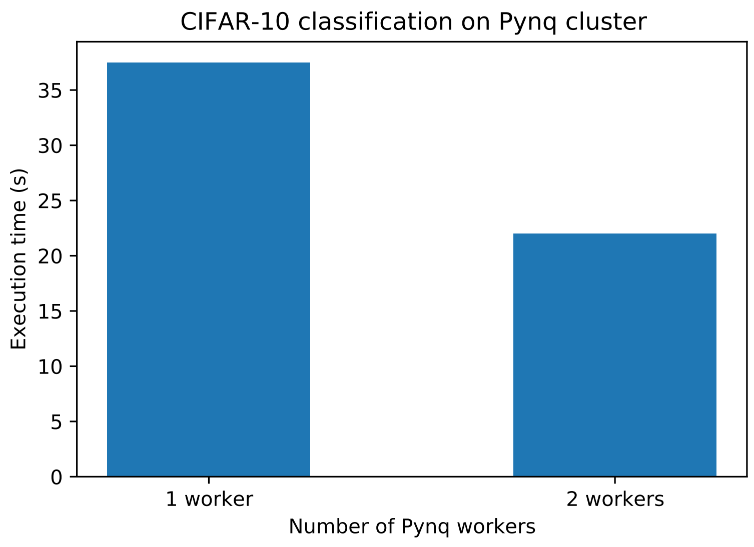

Figure 3.3: Performance comparison between 1 and 2 FPGAs

Ideally, a speedup of 2x should be observed. However, all stages of the process are not parallelized, for example reading the file from disk. This can be mitigated by having the workers pull the data themselves in parallel from the client or a distributed storage (Amazon S3, for example).

3.3.2. Vector sum on Alveo

This example involved the addition of 2 vectors of size 4096*4096 using the overlay provided in the official Pynq Alveo repository [1].

Experimental setup: This experiment was run on two Alveo FPGA cards (U200 and U280) attached to a single machine. Dask with Python 3.6 was used to create a virtual cluster consisting of two workers and a scheduler (also running on the same machine). The vectors to be added were populated by randomly-generated unsigned integers.

As in the previous example, we compared the performance of using 1 Alveo accelerator card vs using both of them. In the case of 2 workers, the data was split equally, i.e., into two 2D arrays of shape 2048*4096. We also performed a more detailed split-up of the total elapsed time, by splitting the entire run-time into 3 components:

1. *Time to scatter*: Time taken to send the chunks of data to their respective workers. This is done using Dask's *scatter* method, which allows us to distribute our data

across the cluster directly from the client to the workers, without having to go through the scheduler.

2. *Time to execute*: This is the time spent inside the workers, and consists of the time taken to allocate the required buffers, call the FPGA accelerator and get the results.

3. *Time to gather*: This consists of the time taken to transmit the result back from the workers to the clients.

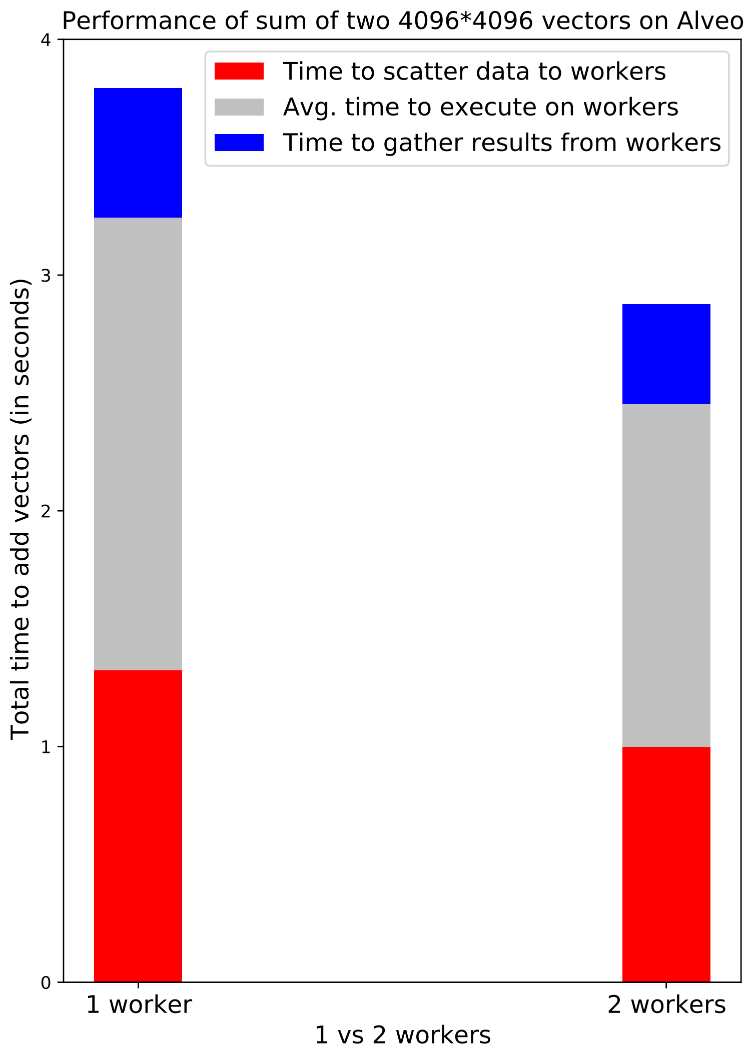

Figure 3.4: Performance comparison between 1 and 2 FPGAs

The results of the experiment are shown in Figure 3.4. As expected, a major portion of the time was spent in data transmission (scattering the input data and collecting the results). Also visible is a speedup of ~1.3x obtained in scaling our setup from 1 to 2 FPGAs. This speedup is lesser than the previous example because of the large non-parallelizable components of data scattering and gathering.

4 Parallelization optimization using Apache Arrow

In the previous chapter, we showed examples where the data was in an arbitrary memory format. For example, in the image resizing and classification prototypes, the input data sent from the client to the workers was of type `PIL.image`, which is a Python-specific data format for representing images. In the 2D vector sum example, we used Numpy 2D arrays as the input data format. During the second half of this project, we developed a prototype that used Apache Arrow as the data format for communication between the hosts and the FPGA. Fletcher, which easily lets us build and communicate with such accelerators, was used in the prototype. In the following sections, we justify the use of Apache Arrow and also explain our prototype in more detail.

4.1. Advantages of using Arrow

One of the bottlenecks observed when designing an FPGA accelerated system is the cost of serialization and deserialization which occurs at the interface of different programming languages, as well as when transmitting data over the wire. In many cases, this serialization is slower than the bandwidth of the FPGA accelerator interface, and hence reducing this overhead leads to a direct and noticeable improvement in runtime.

Apache Arrow was designed keeping such problems in mind. Instead of using Python-specific data representation formats such as Numpy arrays or Pandas Dataframes, using Arrow-based data structures for communicating between the Dask client and workers led to significant improvement in the time taken to scatter data.

The Arrow memory format is columnar, and this provides many benefits for big data analytic systems:

- *Better data locality*: Since all values for a particular column are stored together, it is possible to run aggregation and selection queries without having to scan the entire memory occupied by the table.

- *Support for SIMD*: Columnar data allows compilers to use Single Instruction, multiple data (SIMD) on modern CPUs to allow for data parallelism.

Since a major portion of the total runtime for experiments was the time taken to scatter the input data to the workers, decreasing the serialization cost (which is a significant portion of the scatter time) was an important priority. Fortunately, Dask provides custom support for the Arrow data format. Dask uses different serialization methods based on the type of data being moved. While for some data types, the simple `pickle` serializer is used, Dask maintains several custom serialization families for some special cases, with Arrow being one of them. For Arrow `RecordBatch` and `Table`, Dask uses `PyArrow`'s APIs (the official Python bindings for Apache Arrow) to serialize them.

To quantify the performance gain obtained when using Arrow, we compared the time for scattering data (read from CSV files) from a client to a worker for both Arrow Tables and Pandas Dataframes, which is a popular format used in data analysis. This time includes the time for serialization at the sender, network transmission, and deserialization at the receiver. The results are shown in Figure 4.1. Arrow was found to be roughly 2x faster in these experiments.

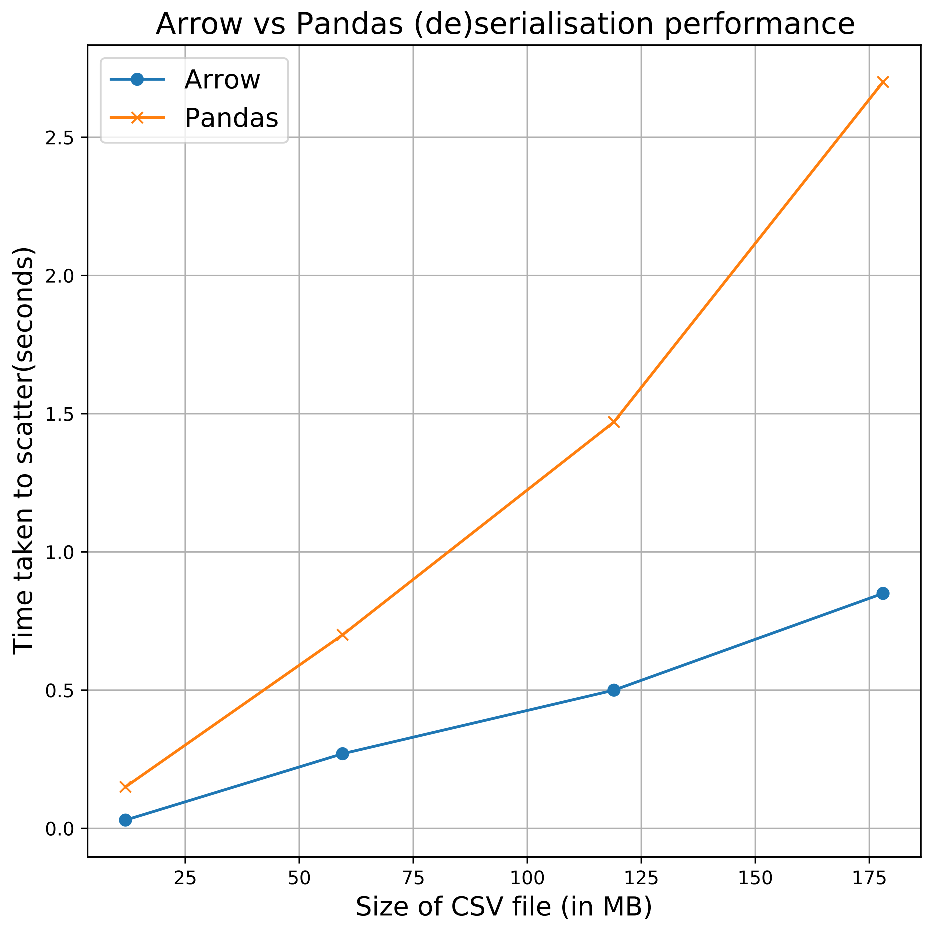

Figure 4.1: Scatter performance - Arrow Table vs Pandas Dataframe

Using Arrow also enabled us to use the Fletcher runtime libraries for communicating with our FPGA kernel. In the previous chapter, the `Pynq` package was used for this purpose. Fletcher also provides the functionality, with the additional advantage of supporting more than one platform. In our experiments, we used Fletcher on the OpenCAPI and AWS-F1 environments.

4.2. Solution architecture

While the architecture, in this case, is similar to the previous experiments from the last chapter, there are some differences:

- The bitstream loaded on the FPGA should be built using Fletcher's interface generation tool (Fletchgen) to allow us to send Record Batches to it. Fletchgen generates streams of the same data type as the Arrow schema specified during the kernel design stage.
- Fletcher's runtime library in Python (Pyfletcher) is used to connect to the FPGA from the Python code. Pyfletcher auto-detects the underlying hardware platform (`aws`, `capi` or `echo`), and hence the resulting Python code is platform agnostic.
- Arrow is used as the data format used for communication between the client and the workers, and further between the workers and the FPGA.

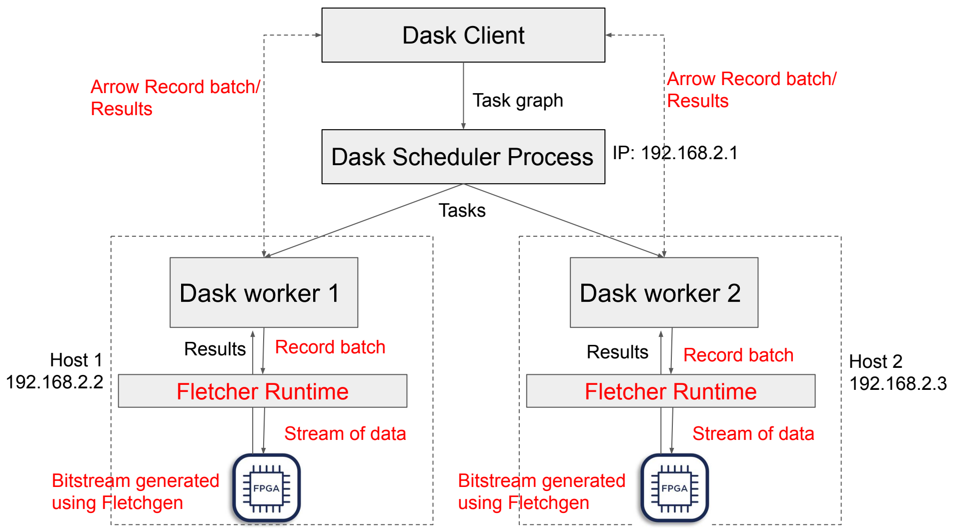

Figure 4.2: Architecture of the system (Differences from previous architecture highlighted in red)

4.3. Example prototype

One of the platforms supported by Fletcher is AWS-F1. Amazon's EC2 F1 instances are high performance compute systems (upwards of 122 GiB memory and 8 vCPUs, where each vCPU is a core on the 2.3 GHz Intel Xeon processor) with Xilinx Virtex

FPGAs. For our experiments, we chose the `f1.2xlarge` instance which contains 1 such FPGA. For the purpose of measuring scalability improvements, we created up to 3 such instances. They were all co-located in the same availability zone to maximize network throughput between them. The Dask-scheduler was running on one of these instances, and the 3 workers were distributed across the 3 instances.

The example we used was a regular expression matching example available as one of the Fletcher examples for AWS integration. The accelerator performs a search for multiple regex patterns in a text input file. The input data format is an Arrow `RecordBatch` whose schema consists of a single column of strings, where each string is a line from a text file. The Verilog source files for the accelerator are present on the Github repository [5], and the run-time python code was adapted from the host code available in the same repository.

To synthesis and deploy the bitstream on the 3 instances, the following steps were taken:

1. Generate a design checkpoint (DCP) for the regexp example using Vivado and the hardware development kit (HDK) of the AWS EC2 FPGA Development kit [2].
2. Upload the DCP to AWS S3 using the AWS CLI.
3. Use the `create-fpga-image` AWS command to create an Amazon FPGA Image (AFI) from the DCP stored in S3. An AFI contains the required bitstream and has a unique AFI ID, which can be used to deploy the bitstream on any F1 instance using the `fpga-load-local-image` command of the FPGA kit.

Once the bitstream is loaded on the 3 instances, we can run interact with it via Python using Pyfletcher. This is done in the following manner (see Appendix for the complete code):

```python

# Auto detect the hardware platform.
platform = pf.Platform()

# Initialize the Platform.
platform.init()

# Create a Context for our data on the Platform.
context = pf.Context(platform)

# Queue the RecordBatch 'batch' to the Context.
context.queue_record_batch(batch)

# Enable the Context, (potentially transferring the data to FPGA).
context.enable()

# Set up an interface to the Kernel, supplying the Context.
kernel = pf.Kernel(context)

# Start the kernel.
kernel.start()

```

```
# Wait for the kernel to finish, and get the result back.
kernel.wait_for_finish()

result = kernel.get_return(np.dtype(np.uint32))
```

Running the above code as a Dask task on the workers lets us split up the text file, and perform the pattern search on all of the workers simultaneously. To check the usefulness of this experiment, we first compared the FPGA accelerated version to an optimized CPU implementation, which used OpenMP to build a multithreaded parallelized version of the regular expression matching example.

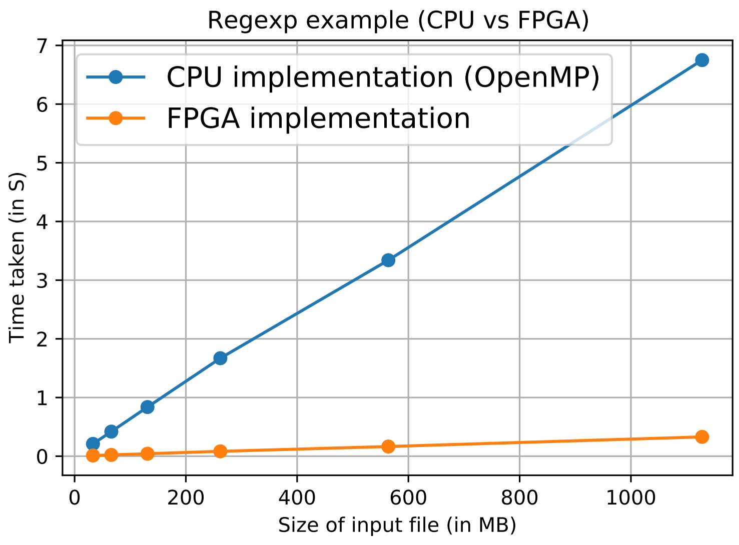

Figure 4.3: Performance comparision between CPU and FPGA-accelerated versions

From Figure 4.3 , it is evident that using an FPGA provides a significant benefit over a CPU implementation. In the next section, we look at how to further optimize the FPGA runtime using Dask parallel processing.

4.4. Experimental results

In addition to comparing the execution times of the CPU and the FPGA versions, we also perform a more detailed split-up of the entire runtime along with the Dask setup, including the time needed to read, prepare and distribute the data for both the versions. The results are shown in Figure 4.4.

Apart from the obvious performance difference between the FPGA and the CPU versions, two other things can also be inferred from Figure 4.4:

1. Using two Dask workers provides an improvement in runtime for both the versions, especially the CPU version where the execution time dominates the total runtime.

2. For the FPGA version, the actual execution time is a very small percentage of the total runtime.

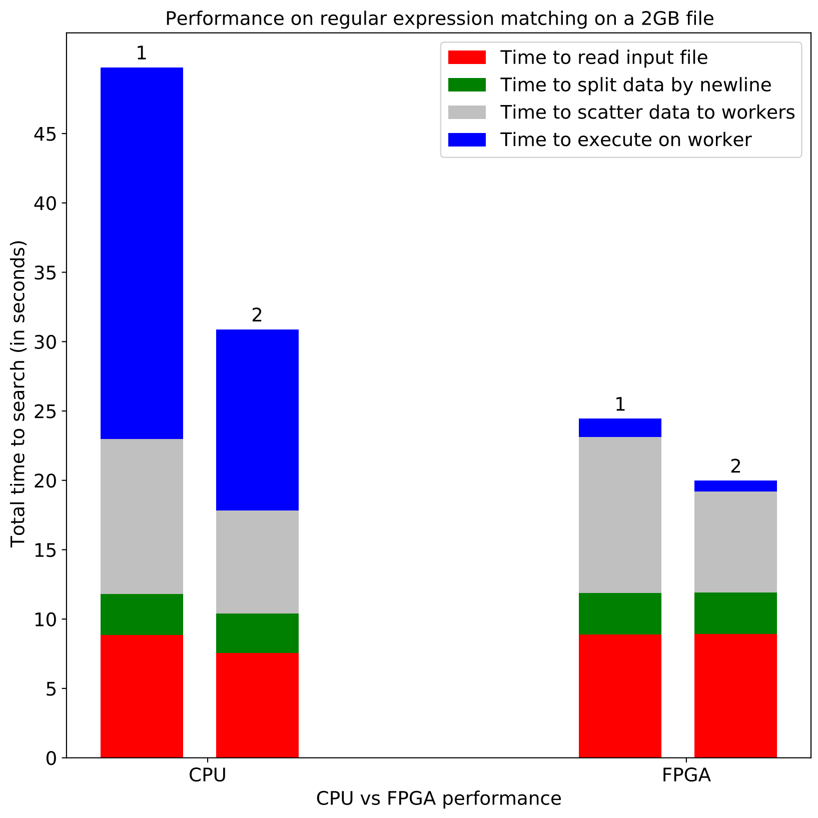

Figure 4.4: Performance comparision between CPU and FPGA-accelerated versions (the numbers on top of the bar indicate the number of Dask workers)

Point 2 above hints to the need to optimize the scatter time since it constitutes a major portion of the total time. Using more than 1 worker helps us do this. Hence, we performed a more detailed analysis by using up to 3 Dask workers and various input sizes. The results are summarized in Figure 4.5.

In the cases in Figure 4.5, it can be seen that using more than one worker reduces the runtime significantly. Up to 1.45x speedup is obtained when using 3 workers instead of 1. Parallelising the process has reduced the scatter time and execution time, reducing the overall time noticeably.

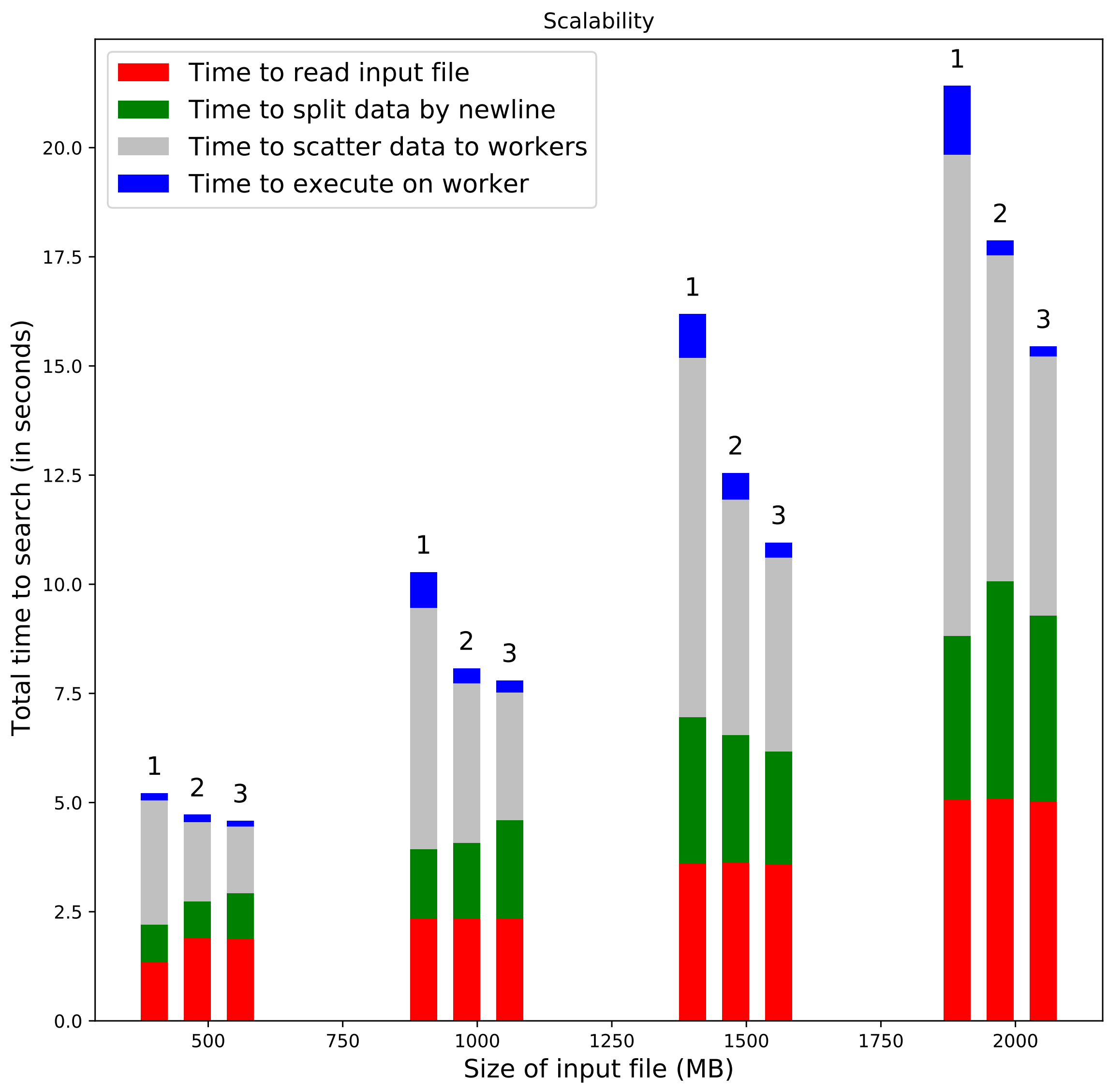

Figure 4.5: Scaling of Dask workers (the numbers on top of the bar indicate the number of Dask workers)

5 Conclusions and future work

5.1. Conclusions

In this project, we have proposed an architecture to be able to tie together several FPGA devices, running the same pre-built accelerator, into a cluster and to execute our computation tasks on the cluster using a high-level programming language. We have also used different combinations of software and hardware platforms to demonstrate the general applicability of our solution.

More precisely, we have been able to address the research questions formulated in the beginning of the project.

The first research question involved determining whether we can improve an application's performance using multiple FPGAs. This was answered by implementing distributed versions of existing FPGA applications on two or three FPGA devices, and comparing them to the single-FPGA implementation. Three such applications were examined:

1. Image classification using a quantised neural network: Total time for classifying 10000 images was reduced from 38 to 22 seconds on scaling from 1 to 2 Pynq-Z1 FPGA devices.
2. Addition of 2 integer vectors : A performance gain of 1.3x was obtained by using 2 Alveo FPGAs devices. Performance gain was observed both in the execution phase and the data distribution phase.
3. Regular expression matching for multiple patterns on a text file: With large text files and 4 regular expressions, the application was scaled to up to 3 AWS FPGA instances. For a text file of size 1.5GB, the total runtime reduced from 16 to 11 seconds (1.45x speedup).

The second research question dealt with using a high level programming language to interact with the cluster of FPGAs. This project used simple Python scripts and packages to do the following:

- Given a set of hosts connected over a network, the Python framework Dask was used to connect them together into a simple scheduler-worker architecture, with an FPGA card attached to each of the workers.
- Python libraries (Pynq and Pyfletcher) were used by the Dask workers to interact with the FPGA. This helps us avoid writing complex host binaries (often written in C/C++) to use the accelerator.
- All the data handling, i.e. reading, distributing and collecting results, was completely performed using Python. This lets us use the rich Python ecosystem of data analytics libraries such as Pandas and Numpy.

This makes it possible for data scientists and software engineers to reuse highly specialized and optimized FPGA implementations without having to deal with low-level communication drivers. The user also does *not* have to think about the internal workings of the cluster (how nodes communicate with each other, data distribution, network error handling, etc.), and can treat the system as a single entity which internally provides parallelism.

The third research question dealt with using a columnar memory format to further optimise the application. We have demonstrated the performance gains due to the use of Apache Arrow, a popular framework for in-memory data, for representing our program data instead of traditional data structures. The regular expression matching example mentioned above uses Apache Arrow to reduce the serialisation and deserialisation costs of the program data. In a standalone experiment involving the scattering of a large file over the network, Arrow performed over twice as fast when compared to Pandas, a popular Python framework for representing in-memory data in big data applications.

From the experiments in this project, it can be seen that there exist several cases where a distributed FPGA cluster can provide better performance when compared to a single FPGA or a software implementation. However, there still exists a scope for improving the general adoption of FPGAs in the industry. Similar to software libraries, being able to use off-the-shelf accelerators built by highly skilled hardware designers is one way of improving the adoption of FPGAs. In this project, we have developed prototypes that demonstrate ways to achieve this goal.

5.2. Future Work

There are a few improvements to this system that can make it further useful. Most importantly, we would like to use remotely available data instead of reading data serially from a filesystem. This includes data from HDFS, S3, etc. This should help parallelize the data-reading process by allowing the workers to pull chunks of data themselves in parallel instead of the Dask client reading all the data serially and then distributing to workers. Then the client only has to provide pointers/offsets to the data to the workers (HDFS filenames, S3 object keys, etc.), and network communication will be minimized/-parallelized.

We would also like to provide easy-to-use cluster setup utilities to allow users to easily deploy our architecture stack on various operating systems and FPGA devices.

This task is fairly tedious due to a host of software requirements (Dask, Fletcher, Arrow) as well as hardware-specific requirements (AWS FPGA SDK, etc.)

A

Programming snippets

- Python code to run the image resizer overlay. Shows some of the capabilities of the Pynq framework.

```python
#Import necessary methods from Pynq Python package
from pynq import Xlnk, Overlay

#Download the bitstream on the FPGA
resize_design = Overlay("resizer.bit")

#Create references to the DMA and the resizer IPs
dma = resize_design.axi_dma_0
resizer = resize_design.resize_accel_0

#Create input and output buffers in the on-chip DRAM
in_buffer = xlnk.cma_array(shape=(old_height, old_width, 3),
                           dtype=np.uint8, cacheable=1)
out_buffer = xlnk.cma_array(shape=(new_height, new_width, 3),
                            dtype=np.uint8, cacheable=1)

# Copy image data from Python local memory to input buffer
in_buffer[:] = np.array(original_image)

#Set values in the predefined MMIO registers on the FPGA
resizer.write(0x10, old_height)  # number of rows for original picture
resizer.write(0x18, old_width)   # number of columns for original picture
resizer.write(0x20, new_height)  # number of rows for resized picture
resizer.write(0x28, new_width)   # number of columns for resized picture


#Run the accelerator kernel
dma.sendchannel.transfer(in_buffer)
dma.recvchannel.transfer(out_buffer)
resizer.write(0x00,0x81) # start
dma.sendchannel.wait()
dma.recvchannel.wait()
```

```

#Output image
resized_image = Image.fromarray(out_buffer)
```

- Python code to run the image classification overlay. Demonstrates code simplicity when using a Python driver instead of making low level Pynq API calls.

```
# Import the python driver for the overlays
import bnn

#Import one of the hardware overlays
# W1A1 - 1 bit weight and 1 bit activation
hw_classifier = bnn.CnvClassifier(bnn.NETWORK_CNVW1A1,'cifar10',bnn.
    RUNTIME_HW)

#OR W1A2 - 1 bit weight and 2 bit activation
hw_classifier = bnn.CnvClassifier(bnn.NETWORK_CNVW1A2,'cifar10',bnn.
    RUNTIME_HW)

#OR W2A2 - 2 bit weight and 2 bit activation
hw_classifier = bnn.CnvClassifier(bnn.NETWORK_CNVW2A2,'cifar10',bnn.
    RUNTIME_HW)

# Call accelerator kernel
inferred_class = hw_classifier.classify_image(img)
```

- Python code for a simple Dask example to resize an image using Pynq-Z1 FPGA:

```
# Function which is executed on every worker
def run_on_worker(data):
    from multiprocessing import Process,Queue
    from PIL import Image

    def use_overlay(queue, file_path):
        import bnn
        from pynq import Xlnk

        hw_classifier = bnn.CnvClassifier(bnn.NETWORK_CNVW2A2,'cifar10',
    bnn.RUNTIME_HW)
        result_W2A2 = hw_classifier.classify_cifars(file_path)
        xlnk = Xlnk()
        xlnk.xlnk_reset()
        queue.put(result_W2A2)

    # Writing to a file is necessary since this overlay expects a file
    path present on the Pynq board
    file_path = "input_data.bin"
    with open(file_path, "wb") as outfile:
        outfile.write(data)

    '''
    We need to run the Pynq overlay in a new forked process since
    the Pynq library cannot be run in a non-main thread, which happens to
    be
    the default Dask behaviour for all workers
    '''
    queue = Queue()
    p = Process(target=use_overlay, args=(queue,file_path))
    p.start()
    result = queue.get()
    p.join()
    return result



#The following code is executed on the machine acting as the Dask Client.
from dask.distributed import Client
client = Client("tcp://192.168.2.1:8786") # IP of scheduler


# Split up the CIFAR-10 dataset into equal sized chunks based on number of
     available dask workers
num_of_workers = len(client.scheduler_info()["workers"])
data_split = []
with open("cifar-10-batches-bin/data_batch_1.bin", "rb") as ifile:
    total = ifile.read()
    start = 0
    chunk_size = int(len(total)/num_of_workers)
    for i in range(num_of_workers):
        data_split.append(total[start: start+chunk_size])
        start += chunk_size


# Scatter the data to the workers before calling run_on_worker on the
    workers
distributed_data = client.scatter(data_split)
futures = client.map(run_on_worker, distributed_data)

# Get the output returned by the workers
result = client.gather(futures)
```

- Python code to demonstrate regular expression matching using Apache Arrow. Tested on up to 3 AWS F1 instances, but also works on any Fletcher-supported hardware platform.

```

class RegExCore(pf.UserCore):
    #... Class definition available at https://github.com/abs-tudelft/
    fletcher-example-regexp/blob/master/software/python/regexp.py#L49
    #... Not provided here for brevity

# function which executes on the workers
def run_on_worker(batch, regexes):

    platform = pf.Platform()  #Auto detect the platform
    context = pf.Context(platform)
    rc = RegExCore(context)

    # Initialize the platform
    platform.init()

    # Reset the UserCore
    rc.reset()

    # Prepare the column buffers
    context.queue_record_batch(batch)
    context.enable()

    # Set required parameters of the accelerator
    rc.set_reg_exp_arguments(0, batch.num_rows)

    # Start the matchers and poll until completion
    rc.start()
    rc.wait_for_finish(10)


    # Get the number of matches from the UserCore
    matches = rc.get_matches(len(regexes))

    return (matches, t2-t1)



#The following code is executed on the machine acting as the Dask Client.

# Initialise Dask client
client = Client(IP_OF_SCHEDULER)

#Read input data
filename = 'input_data.txt'
f = open(filename, 'r')
strings_native = f.readlines()


# Declare the Arrow schema for our input data
column_field = pa.field("text_corpus", pa.string(), False)
schema = pa.schema([column_field])

# Split input data into chunks, where each chunk is a record batch of
    schema `schema`
num_rows = len(strings_native)
num_of_workers = len(client.scheduler_info()["workers"])
data_split = []
chunk_size = int(num_rows/num_of_workers)
start = 0
for w in range(num_of_workers):
    data_split.append(pa.RecordBatch.from_arrays([pa.array(strings_native[
    start: start+chunk_size])], schema))
    start += chunk_size

# The regex patterns to search for
regexes = [".*(?i)bird.*", ".*(?i)bunny.*", ".*(?i)cat.*", ".*(?i)dog.*",
    ".*(?i)ferret.*", ".*(?i)fish.*",
                ".*(?i)gerbil.*", ".*(?i)hamster.*"]

```

```python

# Scatter the data to the workers before calling run_on_worker on the
    workers
scattered_data = client.scatter(data_split)
futures = client.map(run_on_worker, scattered_data, [regexes]*
    num_of_workers)

# Get the output returned by the workers. Note that these results need to
    be merged to get the total matches
results = client.gather(futures)

#Combine matches from all the workers
total_matches = [0]*len(regexes) # total matches = [0,0,0,....], where
    each index corresponds to a single regex pattern
for result in results:
    total_matches  = [sum(x) for x in zip(total_matches, result)]
```

Bibliography

[1] Official examples for using pynq with alveo. URL `https://github.com/Xilinx/Alveo-PYNQ`.

[2] Aws ec2 fpga development kit. URL `https://github.com/aws/aws-fpga`.

[3] Quantized neural network (qnn) on pynq. URL `https://github.com/Xilinx/BNN-PYNQ`.

[4] The cifar-10 dataset. URL `https://www.cs.toronto.edu/~kriz/cifar.html`.

[5] Fletcher regular expression matching example. URL `https://github.com/abs-tudelft/fletcher-example-regexp`.

[6] The mnist database. URL `http://yann.lecun.com/exdb/mnist/`.

[7] Pynq: Open source project from xilinx, . URL `http://www.pynq.io/`.

[8] Pynq image resize example from xilinx, . URL `https://github.com/Xilinx/PYNQ-HelloWorld`.

[9] tiny-cnn: A header only, dependency-free deep learning framework in c++11. URL `https://github.com/Xilinx/xilinx-tiny-cnn`.

[10] Stuart Byma, J Gregory Steffan, Hadi Bannazadeh, Alberto Leon Garcia, and Paul Chow. Fpgas in the cloud: Booting virtualized hardware accelerators with openstack. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 109–116. IEEE, 2014.

[11] Fei Chen, Yi Shan, Yu Zhang, Yu Wang, Hubertus Franke, Xiaotao Chang, and Kun Wang. Enabling fpgas in the cloud. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, pages 1–10, 2014.

[12] Y. Choi and H. K. So. Map-reduce processing of k-means algorithm with fpga-accelerated computer cluster. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 9–16, 2014.

[13] Yuk-Ming Choi and Hayden Kwok-Hay So. Map-reduce processing of k-means algorithm with fpga-accelerated computer cluster. In *2014 IEEE 25th International Conference on Application-Specific Systems, Architectures and Processors*, pages 9–16. IEEE, 2014.

[14] Zefu Dai and Jianwen Zhu. Saturating the transceiver bandwidth: Switch fabric design on fpgas. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 67–76, 2012.

[15] Mohammed Elnawawy, Abid Farhan, Ahmad Al Nabulsi, AR Al-Ali, and Assim Sagahyroon. Role of fpga in internet of things applications. In *2019 IEEE International Symposium on Signal Processing and Information Technology (ISSPIT)*, pages 1–6. IEEE, 2019.

[16] Jian Fang, Jianyu Chen, Jinho Lee, Zaid Al-Ars, and H Peter Hofstee. Refine and recycle: A method to increase decompression parallelism. In *2019 IEEE 30Th international conference on application-specific systems, architectures and processors (ASAP)*, volume 2160, pages 272–280. IEEE, 2019.

[17] Mariangela Genovese and Ettore Napoli. Asic and fpga implementation of the gaussian mixture model algorithm for real-time segmentation of high definition video. *IEEE transactions on very large scale integration (VLSI) systems*, 22(3): 537–547, 2013.

[18] Joost Hoozemans, Rolf Heij, Jeroen van Straten, and Zaid Al-Ars. Vliw-based fpga computation fabric with streaming memory hierarchy for medical imaging applications. In *International Symposium on Applied Reconfigurable Computing*, pages 36–43. Springer, 2017.

[19] Ernst Joachim Houtgast, Vlad-Mihai Sima, Koen Bertels, and Zaid Al-Ars. Hardware acceleration of bwa-mem genomic short read mapping for longer read lengths. *Computational biology and chemistry*, 75:54–64, 2018.

[20] Brad L Hutchings, Rob Franklin, and Daniel Carver. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 111–120. IEEE, 2002.

[21] Preston A Jackson, Cy P Chan, Jonathan E Scalera, Charles M Rader, and M Michael Vai. A systolic fft architecture for real time fpga systems. Technical report, MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB, 2005.

[22] Li Jiao, Cheng Luo, Wei Cao, Xuegong Zhou, and Lingli Wang. Accelerating low bit-width convolutional neural networks with embedded fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[23] Christoforos Kachris, Elias Koromilas, Ioannis Stamelos, and Dimitrios Soudris. Spynq: Acceleration of machine learning applications over spark on pynq. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 70–77. IEEE, 2017.

[24] Zhongduo Lin and Paul Chow. Zcluster: A zynq-based hadoop cluster. pages 450–453, 12 2013. ISBN 978-1-4799-2198-0. doi: 10.1109/FPT.2013.6718411.

[25] Duncan JM Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong. High performance binary neural networks on the xeon+ fpga™ platform. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[26] Hiroki Nakahara, Tomoya Fujii, and Shimpei Sato. A fully connected layer elimination for a binarizec convolutional neural network on an fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[27] K. Neshatpour, M. Malik, M. A. Ghodrat, A. Sasan, and H. Homayoun. Energy-efficient acceleration of big data analytics applications using fpgas. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 115–123, 2015.

[28] Jian Ouyang, Hong Luo, Zilong Wang, Jiazi Tian, Chenghui Liu, and Kehua Sheng. Fpga implementation of gzip compression and decompression for idc services. In *2010 International Conference on Field-Programmable Technology*, pages 265–268. IEEE, 2010.

[29] Johan Peltenburg, Shanshan Ren, and Zaid Al-Ars. Maximizing systolic array efficiency to accelerate the pairhmm forward algorithm. In *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 758–762. IEEE, 2016.

[30] Johan Peltenburg, Jeroen van Straten, Matthijs Brobbel, H Peter Hofstee, and Zaid Al-Ars. Supporting columnar in-memory formats on fpga: The hardware design of fletcher for apache arrow. In *International Symposium on Applied Reconfigurable Computing*, pages 32–47. Springer, 2019.

[31] Johan Peltenburg, Jeroen van Straten, Lars Wijtemans, Lars van Leeuwen, Zaid Al-Ars, and Peter Hofstee. Fletcher: A framework to efficiently integrate fpga accelerators with apache arrow. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 270–277. IEEE, 2019.

[32] Yang Qiao. An fpga-based snappy decompressor-filter. *Master's thesis*, 2018.

[33] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database analytics acceleration using fpgas. In *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 411–420. IEEE, 2012.

[34] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74, 2017.

[35] Felix Winterstein, Samuel Bayliss, and George A Constantinides. Fpga-based k-means clustering using tree-based data structures. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–6. IEEE, 2013.