

CSC4006

Research and Development Project

Karl McCoubrey

40129688

Software Development Report

Contents

Introduction	2
System Design and Specification	3
PYNQ K-Means Test Implementation	3
Analysis of K-Means Limitations in Python	4
K-Means vs K-Means++ for Image Quantization	6
Video Capture Method	7
K-Means Image Segmentation from Video Feed	7
Hardware Acceleration of Data Analytics	8
IP Block Design	8
Hardware Design Environment Issue	9
K-Means IP Block	10
Conclusion	10
References	11

Introduction

Field Programmable Gate Arrays offer a powerful and efficient deployment for data analysis when combined with an effective hardware configuration and low-level programming. Embedded systems requiring low-power yet demanding computation are rapidly increasing through fields such as autonomous driving and widespread artificial intelligence solutions. FPGAs are limited by their compatibility of programming techniques. The addition of support for a high-level programming on an FPGA creates both new opportunities and potential problems, both of which will be explored. The PYNQ project aims to apply this high-level programming aspect by running a linux distribution on the board with Python programming capability. Throughout we will test this new technology's potential for powerful computation, and its' current limitations. The PYNQ project aims to enable software developers to design powerful systems by accelerating programming functions by designing dedicated hardware blocks to handle proprietary functions. One such explored opportunity is the implementation of a K-Means clustering algorithm as a pre-processing stage, for image analysis onboard an FPGA device. Existing research provides exemplary material proving the power of FPGA hardware enabling real-time image processing over slower high-level language implementations. Real-time processing can also be made available by altering the standard Lloyds algorithm for k-means to a histogram-based algorithm using the low-level programming capabilities [1].

This research aims to explore the acceleration of data analytics for image processing on an FPGA, specifically the speedup of K-Means algorithms hardware acceleration. Python is not supported by FPGA boards traditionally due to its' high-level nature, meaning this code stands many layers from machine code itself. Python must first be run through an interpreter program converting the python code into a low-level language such as 'C'. A second program, called a compiler, takes the interpreted code and converts into machine code therefore enabling the processor to run. Python traditionally does not have direct access to hardware resources and configurations necessary for speedup in FPGA algorithms.

Xilinx, the inventor and manufacturer of the FPGA, released in 2016 the PYNQ project. The goal of PYNQ is to enable easier exploitation of the unique benefits of Xilinx devices for designers of embedded systems [2]. Using Python libraries and language, designers can now exploit the benefits of programmable logic and microprocessors in Zynq to build more capable embedded systems. Programmable logic circuits are presented with hardware libraries named *overlays*. Along with overlays, Xilinx produces a High-Level-Synthesis tool (Vivado HLS) enabling hardware acceleration.

The K-Means algorithm is a commonly used unsupervised machine learning technique to a partition a data set into k groups. It proceeds by selecting k initial cluster centers and then iteratively refining them. Explained in [3] by:

1. Each instance d_i is assigned to its' closest cluster center
2. Each cluster center C_j is updated to be the mean of its constituent instances.

"It is well known that K-Means clustering is highly sensitive to proper initialization. The classic remedy is to use a seeding procedure [...] known as k-means++" [4]. K-Means initializes its beginning cluster centroid points randomly, and algorithmically searches for better centroids through follow-up iterations. K-Means++ by Arthur and Vassilvitskii [5] begins by initializing one centroid and allocates the remaining cluster centroids relative to the first point. Both techniques have advantages and disadvantages regarding precision, speed, and overheads and these will be examined further. Real-time processing can also be made available by altering the standard Lloyds algorithm for k-means to a histogram-based algorithm using the low-level programming capabilities [1].

To solve this problem and test various methods, a PYNQ instance is setup enabling Python production on an FPGA. In this scenario, the PYNQ-Z2 board will be used for testing and analysis. Configurations are defined throughout.

System Design and Specification

PYNQ is an open source project from Xilinx enabling easier design of embedded system on the Xilinx ZYNQ SoC System on Chips. PYNQ effectively deploys a bootable Linux image providing high-level Python programmability via a Jupyter Notebook environment. The PYNQ runs Python programs on its' 650MHz ARM A9 dual-core processor, with 512MB DDR3 with 16-bit bus at 1050Mbps. The storage medium for the operating system and static memory relies on a MicroSD card. In the following tests, a Lexar 633x 128GBUHS-I class 10 card is used. The operating system version began with the PYNQ v2.3 but after recurring boot and connectivity issues, the image was updated to v2.4 [6]. The device was controlled via ethernet-to-ethernet port access, connecting via a local IP within the host computer's web browser.

The aim is the acceleration of data analytics using the python programmed FPGA hardware. The final goal is to utilize the FPGA properties on the PYNQ device, to accelerate data analytics over video streams, and run k-means clustering at real-time speeds. This is enabled by implementing a pre-processing K-Means IP block for streamed image data which is then accessible to the Python frontend. As a new technology, **the PYNQ-Z2 board is yet to receive substantial testing** on its' limitations regarding python implementations, due to its' rapidly evolving operating system versions and limited hardware resources. **Extensive iterative testing** is crucial as existing designs and solutions may not be entirely stable between versions. **No strict prior system design is produced** as the solution to achieve the image processing efficiency is likely to change throughout the component testing phases. The latest OS version available and used throughout is the PYNQ v2.4 [6]. Initial test and implementations focus on the python development environment with K-Means clustering over image data. The tests will include a function for K-Means clustering provided by the industrial grade 2D image processing OpenCV library. This library is mature and stable, improving the reliability of results. Primary testing will approach from a purely software developer approach to explore any limitations of the python environment. This will read image data from its' microSD non-volatile external memory storage, for alternating K-Means algorithm types (regular vs K-Means++), on various colour formats. Upon completion, secondary testing will be conducted to analyse performance of a video capture options. The Linux backend which the PYNQ system is built on, recognizes HDMI as input unlike other system options. This enables testing initiated via OpenCV options, compared to HDMI input options available through the 'base' overlay. The primary acceleration methodology for the PYNQ is through the hardware design synthesis available. Designs will be iteratively testing for standard data transmission and further into image processing, aiming toward real-time analysis.

PYNQ K-Means Test Implementation

Prior to K-Means implementation, a previously functional version of HOG human detection had been run on the PYNQ to test potential limitations. The program worked efficiently on regular PC hardware, but unknown errors became apparent when produced on the PYNQ. This may be due to the limited resources available to the board in terms of RAM and overall memory busses for data transfer. Therefore, emphasis on iterative testing of the current PYNQ setup must be made throughout to determine any potential issues or limitations.

Analysis of K-Means Limitations in Python

To begin development, the PYNQ-Z2 board is selected as the FPGA platform for our K-Means implementation. The 'PYNQ-Z2 v2.3 PYNQ Image' is downloaded and the OS flashed to a MicroSD card as the board's boot device. The image includes Jupyter Notebooks and board specific example overlays. With the boot device ready, it is inserted into the board. The FPGA is powered via Micro-USB and connection into the board is made via the ethernet port. The device is controlled and accessed via a web browser on a local computer. This is made possible by connecting the ethernet port of the board, to the network router or network switch. In this case, the board is connected to an ethernet switch and accessed over the local network by directing to 'pynq:9090' within the web browser. Further into development this method changed due to connectivity issues between the OS and the network setup. OS v2.4 was then used with direct access to the board without the intermediary router. The device is booted and the connection directly into Jupyter Notebooks established by browsing to address 'http://192.168.2.99:9090/'.

Jupyter is an open-source web application that enables the creation of documents containing live code, visualization, and narrative text. This can be used for data cleaning, transformation, statistical modelling machine learning, and much more. Jupyter will be the primary development environment. To begin testing, example source code supplied including very basic Python implementation were ran to ensure Python compilation worked. With no errors, the next stage is to test the import of libraries and modules which may be used. Scikit-learn was not readily available from the OS image, and therefore had to be installed. This library is widely used for machine learning in python, and with extensive support available online, it was chosen as the initial test framework. Jupyter allows terminal access, and the install process was identical to existing Linux distributions through the command line. Installation speeds proved extremely slow, in some cases taking over one hour for a small dependency package to be applied.

Prior to K-Means implementation, a previously working version of HOG human detection had been run on the PYNQ to test potential limitations. The program worked efficiently on regular PC hardware, but unknown errors became apparent when produced on the PYNQ. This may be due to the limited resources available to the board in terms of RAM and overall memory busses. Therefore, emphasis on iterative testing of the current PYNQ setup must be made throughout to try and pre-determine any unexpected issues later.

With the setup complete the first k-means algorithm can be implemented. The goal is for image processing using K-Means clustering. An existing project is reviewed for example Python functions, therefore obtaining a reference result and output to ensure accuracy and dependability of Python on the PYNQ [7]. The input data from the example remains the same, a 100 x 100 image 'colors.jpg'. The image is downloaded on the local computer and transferred to the PYNQ board via the network. The structure of the code remains similar, to reduce potential user errors as the main goal here is to ensure python stability. The first test was to ensure importing the image worked correctly. Initially this was done by displaying the image directly using standard image displaying methods within Python.

```
In [12]: # Displaying image using Python
from IPython.display import Image
Image("img/colors.jpg")

Out[12]:
```




Figure 1 Python Image Import

The image imported correctly, providing reassurance of python calls from external file directories, stored in MicroSD memory to Jupyter. To obtain visible results from our k-means, we need to test visualization methods. In this case we imported cv2, part of the OpenCV library, to plot data points on a 3D graph relative to its RGB values ($x = r$, $y = g$, $z = b$). The results displayed the following.

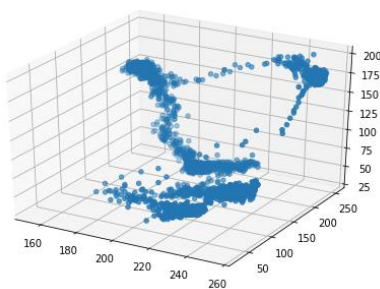


Figure 2 3D Plot of RGB Pixels

From the 3D graph we can visualize the image colours with multiple regions. These regions represent the colours in the image. With the addition of K-Means, we should be able to cluster these colours. The next step is to apply the K-Means implementation using Scikit-learn. With K-Means implemented similarly to the example and applied to the same input data, we receive the outputs of 5 K-Clusters:

```
[[240 99 69], [253 236 175], [163 210 154], [199 191 50], [233 33 67]]
```

This data represents each centroid point for the k-means cluster where $k=5$. The data matches the sample program assuring that the Scikit-learn implementation of K-Means on the PYQN board runs accurately when applied to a 100 x 100 pixel image. The cluster centroid points can further be plotted onto the 3D graph to visualize which pixel regions belong to their cluster colour.

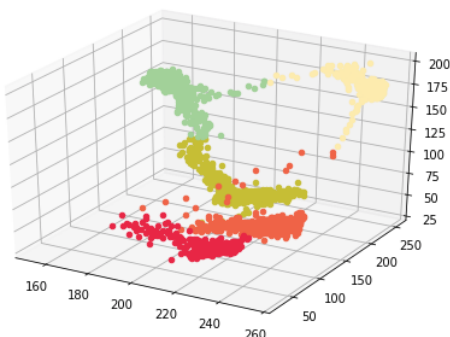


Figure 3 Clustered RGB Pixels

K-Means vs K-Means++ for Image Quantization

The next stage is to begin the implementation of k-means over image data. Image colour quantization applies clustering to reduce the number of distinct colours in an image. The main reason this may be performed is to enable the rendering of an image in devices supporting only a limited number of colours, or to highlight precise image regions by cluster colour allocation. This pre-processing stage using K-Means is common throughout many image processing designs. D. Arthur [5] reports observations of k-means++ consistently out-performing regular k-means, in many cases more than 50% faster in execution time. The nature of the k-means++ algorithm having a pre-processing step will initially create extra overheads, but over the large datasets used by Arthur [5] such as the *intrusion* dataset [8] consisting of 494019 point with 35 dimensions, this step greatly reduces the overall computation. Arthur also used large cluster numbers from 10, 25, 50 again enabling greater optimization with the k-means++ pre-processing stage. K-Means++ is explored on the PYNQ board to analyse if the implementation method of the algorithm will improve performance in aiding to achieve real-time image processing.

The test designed implements an OpenCV K-Means function [9], which enables parameters to define random initial cluster centroids (k-means) and pre-processed centroids (k-means++). The function also takes parameters for number of 'k' clusters, and a number for max iterations of the algorithm. Iterations remained as default and clusters set to 4 to suit the test image for visual analysis of the post-processed image rebuild. Image data is taken as a matrix of 3-dimensional data in the form an a 'numpy array', enabling the OpenCV application to apply the k-means function. The matrix is flattened into 1 dimension and

The results from testing k-means vs k-means++ indicates that processing high definition images such as 1920x1080px image results in very slow performance. With computation timing of 49 and 46 seconds respectively, it is vastly far from achieving real-time processing. K-means++ provides a consistent improvement for the higher resolution image, due to a larger set of datapoints to be processed and the pre-processing stage of cluster initialization becoming more effective with size. Image resolution was resized within the algorithm to then test an approach for speedup. With the reduction of image pixels, the execution time vastly

RGB Resolution	Clusters	Method	Time(s)
1920x1080	4	K	49.845
1920x1080	4	K++	46.680
355x200	4	K	1.740
355x200	4	K++	1.820
177x100	4	K	0.569
177x100	4	K++	0.570
Greyscale			
1920x1080	4	K	31.448
1920x1080	4	K++	26.967

Table 1 Results of Differing K-Means Algorithms

drops, but during testing the k-means++ approach could prove slower. This will be due to the extra computation overhead outweighing its benefit over larger sets, because of the downsizing.

Image colour format is also tested with OpenCV colour conversion taking place prior to k-means analysis from BGR to Greyscale. This effectively reduces the dataset size again for the image as each individual pixel has less data, from three colour values to one. K-Means clustering again improves in execution time over the smaller set.



Figure 4 Original Image, K=4 RGB Clustering, K=4 Greyscale Clustering

Video Capture Method

The end goal of real-time image processing means several components must be tested to diagnose any potential performance bottlenecks, one of these potential issues is in relation to the method used to capture the live feed of image data. The linux distribution acting as an operating system for the PYNQ 2.4 image recognizes HDMI ports as input devices. This enables the current OpenCV formatting already tested to recognize and capture an image data stream from the HDMI-IN port on the board, without the need for USB interface. The OpenCV model uses a VideoCapture class with 'cv2.VideoCapture()' and 'cv2.VideoCapture.read()' to initialise the input interface and read the next frame from the input. It returns an array of image data. The 'base' overlay provided with the PYNQ image also enables video class for analysis using the default block design and IP provided for HDMI connectivity between both IN and OUT ports, and the ZYNQ7020 chip. This enables full access to the VideoDMA of the block design, enabling image packets to be pulled from the channel and returned to the user as a Numpy array. The data is obtained after initial processing stages of colour conversions and pixel packing. The video stream was sent from a laptop at 1280x720 as a screen duplicate. Each implementation methodology iterates through 600 frame captures, with each capture waiting for a new frame.

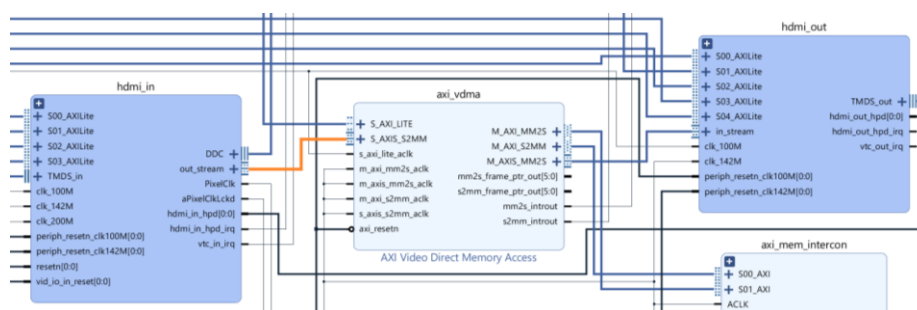


Figure 5 Overlay HDMI Video Block Design

Both methods achieved real-time capture of frames around 60fps, similarly to the framerate sent from the signal device. Therefore, either capture method should not provide bottlenecks to further development into this methodology.

K-Means Image Segmentation from Video Feed

With video capture tested, the next test designed is for the PYNQ boards capabilities to apply the k-means function tested over the video stream. This was aimed to test a differing video stream from a live feed via a camera through previous OpenCV implementation method over USB webcam interface. The USB interface provides real-time speeds through Python without the application of

the k-means function. Therefore, any variability will be due to the frame processing for image segmentation. The design setup is similar to that of the previous section 'Video Capture Method', but loops through a defined number of frames for testing. The setup used K=3 clusters, and reduced the sample iterations within the k-means method to 1 to best achieve speeds. This reduces the reliability for cluster accuracy, but the main code at this point is to test the potential calculation speed.

```
for i in range (num_frames):
    # read next image
    ret, frame_vga = videoIn.read()
    if (ret):
        outframe = hdmi_out.newframe()
        #cv2.cvtColor(frame_vga, cv2.COLOR_BGR2RGB)
        Z = frame_vga.reshape((-1,3))    ## reshape the image(numpy Array) to be a list of pixels
        Z = np.float32(Z)

        criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,1, 1.0)
        K=3
        ret, label1, center1 = cv2.kmeans(Z, K, None,criteria, 1, cv2.KMEANS_RANDOM_CENTERS)
        center1 = np.uint8(center1)
        res1 = center1[label1.flatten()]
        frame_vga = res1.reshape((frame_vga.shape))

        #outframe[0:480,0:640,:] = frame_vga[0:480,0:640,:]
        outframe[:] = frame_vga
        hdmi_out.writeframe(outframe)
```

Figure 6 K-Means Code for Camera Input Stream

Test	Avg. Frames Per Sec
1	4.215446784118083
2	4.242484753878919
3	4.196583752157522
4	4.216135613984127
5	4.219863438556211

Table 2 Results of K-Means Performance in Python

Hardware Acceleration of Data Analytics

IP Block Design

The final stage is to prepare an IP and block design capable of processing the K-Means clustering algorithm. The initial approach to the problem is to create an IP intercepting the image data between existing blocks of the base overlay. This way, the user still has full control over the extensive video functionality included with the board and not only a clustering block. A software engineer will be less capable of creating full block designs from scratch with the k-means IP, and the passing of data between board components. A compatible IP block capable of dropping in to one data stream is ideal, enabling simple instructions to be followed for implementation onto the base overlay design. Iterative testing for production began small with the creation of a simple adder block design for passing integer data via DMA direct memory access. This adder was nested into a loop to add some minor complexity for timing and resource testing.

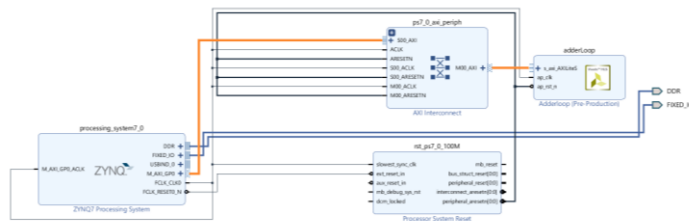


Figure 7 Block Design of adderLoop

The adderLoop block design ran effectively without issues.

To begin analysing video frames, a simple pixel streaming block was designed and implemented into the base overlay design. The next design uses pre-processing to the VDMA video direct memory access within the 'base' overlay, which allows the Python code to handle full video frames in and out of the VDMA block. This solution with k-means implemented would be ideal as a drop in processing pipeline, easily allowing an in-experienced hardware designed to apply one block within a data path.

```
#include "stdint.h"
#include "hls_stream.h"
#include "ap_axi_sdata.h"

typedef ap_axiu<32,1,1,1> pixel_data;
typedef hls::stream<pixel_data> pixel_stream;

void pixStream(pixel_stream &src, pixel_stream &dst)
{
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE axis port=&src
    #pragma HLS INTERFACE axis port=&dst
    #pragma HLS PIPELINE II=1

    pixel_data p;
    src >> p;
    dst << p;
}
```

Figure 8 C++ Code for Stream IP Block

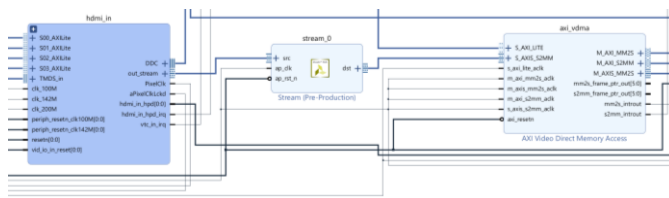


Figure 9 Stream IP Implemented into Base Overlay

Hardware Design Environment Issue

The previous stream IP was built, and the IP block generated correctly. The block must be placed within the full 'base' block design between the HDMI-IN and VDMA components. The stream simply intercepts the AXI Stream data and pipelines it out, as shown in the diagram above. The next stage is to validate the design and build the bitstream, which will be used as the overlay for Python interaction. The design validated correctly but an error occurred consistently when generating the bitstream. Extensive time was spent trying to debug the issue of the streaming block, but eventually the fault was found in the official overlay design supplied by the manufacturers. The original overlay design was re-downloaded to remove any potential issues from editing, but the fault reoccurred consistently on the default build with repeated "place_design error" faults. The fault takes approx. 1 hour 30 minutes to occur during the generation of the bitstream, whilst validation of the design remained successful. Heavy amounts of time were dedicated to trying to resolve the issue or bypass the problem, but from

software engineering background the depth of the issue was too much for the limited time for development. The Vivado tool involved were found to be difficult in debugging design issue.

K-Means IP Block

The C++ K-Means code “kmeansImage.cpp” developed was not able to be fully implemented within the design because of the outstanding issue with the supplied ‘base’ overlay. Another approach had been previously considered, by retrieving the image data from the installed ‘base’ overlay, storing, and sending the image data via DMA or VMDA in a new overlay design. Having to dynamically reload overlays between processing is unsuitable for effective performance due to loading and driver setup. Data transmission is limited by size with the use of DMA to memory mapped I/O and remains unsuitable data handling of large image resolutions on the PYNQ board. A processing stage to build the pixel stream into an array of data, to then pass the full image data into the k-means function is a possibility.

```
1: Function Lloyd (inputImage, K)
2:   for all  $c \in 0..K-1$  do  $\text{mean}[c] = \text{Initial value}$ 
3:   do
4:     totalDistance = 0
5:     for all  $x, y \in \text{domain}(\text{inputImage})$  do
6:        $p = \text{inputImage}[x, y]$ 
7:       find  $c \in 0..K-1$  such that  $\text{abs}(p - \text{mean}[c])$  is minimum
8:       totalDistance +=  $(p - \text{mean}[c])^2$ 
9:       sum[c] += p
10:      size[c] += 1
11:    for all  $c \in 0..K-1$  do
12:      if (size[c]  $\neq$  0)  $\text{mean}[c] = \text{sum}[c] / \text{size}[c]$ 
13:    while (totalDistance has not converged)
14:    return result, means
15: end
```

Figure 10 Pseudocode for Lloyd's K-Means Clustering

A further stage was designed for further acceleration of image clustering by improving the k-means algorithm from Lloyds' based, to histogram-based algorithm as proposed by T. Deng [1]. The algorithm was taken and setup similarly to that of the previous Lloyd's based k-means, but no full overlay implementation was available due to the outstanding build version problems provided by Xilinx.

Conclusion

Research proves the potential for high performance applications launched on FPGA hardware. For a software developer with strictly a high-level experience background, implementation of unsupervised machine learning image processing designs can be implemented successfully on the PYNQ python front-end, whilst remaining stable for k-means clustering. Alternative may prove volatile under other conditions. For complex acceleration designs to be produced, a developer should consider a standalone block design without the inference of an existing setup. A newer technology such as PYNQ is rapidly evolving, with modifications to the operating system frequently changing, and support for such a device is limited. One should approach the problem from a hardware design perspective with intentions of designing a standalone hardware solution without the inference of any existing solutions. Stability and reproducibility for the PYNQ project remains a great issue for largescale adaption by high-level software developers, with very little support available for the specific interaction for the PYNQ front-end to hardware. A hybrid device supporting both high- and low-level compatibility certainly has potential, but availability of reproducible designs and support must be improved for widespread adaption to new markets.

References

- [1] T. Deng, D. Crookes, F. M. Siddiqui and R. Woods, "A New Real-Time FPGA-Based Implementation of K-Means Clustering for Images," in *1st International Conference on Intelligent Manufacturing and Internet of Things, IMIOT 2018 and International Conference on Intelligent Computing for Sustainable Energy and Environment, ICSEE 2018*, Chongqing, 2018.
- [2] Xilinx, "PYNQ Introduction," Xilinx, 2018. [Online]. Available: <https://pynq.readthedocs.io/en/v2.3/>. [Accessed 21 11 2018].
- [3] K. Wagstaff, C. Cardie, S. Rogers and S. Schrod, "Constrained K-Means Clustering With Background Knowledge," in *International Conference on Machine Learning*, Williamstown, 2001.
- [4] O. Bachem, M. Lucic, S. H. Hassani and A. Krause, "Approximate K-Means++ in Sublinear Time," in *AAAI Conference on Artificial Intelligence*, Phoenix, 2016.
- [5] D. Arthur and S. Vassilvitskii, "k-means++: The Advantages of Careful Seeding," in *SODA '07 Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithm*, New Orleans, 2007.
- [6] Xilinx, "Xilinx Github Repository," [Online]. Available: <https://github.com/Xilinx/PYNQ/releases>. [Accessed 12 April 2019].
- [7] S. K. Thakkar, "Dominant colors in an image using k-means clustering," [Online]. Available: <https://buzzrobot.com/dominant-colors-in-an-image-using-k-means-clustering-3c7af4622036>. [Accessed 3 11 2018].
- [8] *KDD Cup 1999 dataset*, <http://kdd.ics.uci.edu/databases/kddcup99/kddcup.html>.
- [9] OpenCV, "Clustering - Kmeans," [Online]. Available: <https://docs.opencv.org/2.4/modules/core/doc/clustering.html?highlight=kmeans>. [Accessed 20 04 2019].