Accelerated Data Analytics Using Python-Programmed FPGA Hardware

K. R. McCoubrey
Queen's University Belfast, UK
kmccoubrey02@qub.ac.uk

***Abstract*—The PYNQ project enables the application of high-level programming languages such as Python to be implemented onboard a Field Programmable Gate Array. This high-level adaption enables non-expert developers with reduced hardware experience to program an FPGA board. FPGAs are powerful devices with configurable programmable logic, enabling versatile optimization of applications. The addition of high-level access and Python implementation may incur additional problems which are latter studied. Introduction of high-level inclusivity results in optimization at low-levels becomes more difficult. We research viable options for acceleration within the Python code using overlays and low-level options through high-level synthesis. In this research article we explore the methods available for high-level developers to accelerate data analytics functions on an FPGA. We tailor this acceleration towards the K-Means clustering algorithm for image processing.**

***Index Terms*—FPGA Acceleration, K-Means Clustering, PYNQ, Python**

I. Introduction

Field Programmable Gate Arrays offer an efficient deployment for data analysis when combined with an effective hardware configuration and low-level programming. FPGAs are limited by their compatibility of programming techniques. The addition of support for a high-level programming language creates both new opportunities and potential problems, both of which will be explored. 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 combined with an FPGA enabling real-time image processing. Real-time processing is 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 on an FPGA, specifically the speedup of K-Means algorithms using Python. Python is not supported by FPGA boards traditionally due to it being a high-level language, meaning this code is 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 method to automatically 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 all randomly, and algorithmically searches for better centroids throughout. K-Means++ 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.

To solve this problem and test various methods, a PYNQ instance is setup enabling Python code production on an FPGA. In this scenario, the PYNQ-Z2 board will be used for testing and analysis. Configurations defined are discussed in the latter section "Progress of Research".

K. R. McCoubrey is with the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, UK (e-mail: kmccoubrey02@qub.ac.uk).

II. Literature Review

Initial optimization/acceleration methods have been researched in source [5] regarding the performance of FPGA acceleration using various methods. The author outlines the options for using *high-level synthesis* to address design productivity. HLS enables raising the programming level from the low RTL used in languages such as Verilog, to higher level languages of C or C++. Although helpful, HLS still requires the expertise of low-level design engineering, restricting its validity of the article topic regarding general purpose acceleration. Restrictions to productivity are outlined that with HLS tools there remains prohibitive compilation times impeding many mainstream adoptions. A common problem with FPGA acceleration are the methods involved needing tailored toward specific FPGA implementations. A solution is produced with the engineering of a programmable course-grained hardware abstraction layer, named an *overlay*. Overlays are deployed to the developer as a programming method, enabling various improvements. Better portability across devices, design reuse, and rapid configuration with orders of magnitude faster than other reconfigurable approaches. The article proceeds to analyze overlay options between time-multiplex and spatially-configured. Overheads are discussed regarding the respective overlay types, but the author does not proceed to conclude which type may or may not suit application implementations for a high-level engineer. Benchmarks are run testing various overlay and hardware configurations, with discussion on hardware performance penalties, and improvements of route time and hardware kernel switching time. The author neglects to conclude the real-time effects this may cause on general purpose applications in which the report is based around. Focusing on portability and mainstream adoption of FPGA based accelerators, there is a clear gap in the analysis. Future testing for our K-Means algorithms will need to further explore overlay types and experiment with individual performance effects, specifically for our chosen algorithms. This experimentation will bridge the gap of information identified in the article.

The next publication [6] starts by addressing the target audience of developers who are non-experts in the field of HW description languages or DPR (Dynamic Partial Reconfiguration). The author explains how overlay architectures abstract the complexity and FPGA design challenges whilst maintaining high performance. Evidence is sourced from multiple publications regarding the sustainability of performance using overlays. This makes for easy evaluation of overlay effects on performance for the reader. The article then explores the option of implementing the readily available base overlays, along with partial reconfigurable overlays. Partial configurable overlays aim to make overlays more versatile and to suit better for multiple applications. This may provide a viable option for our acceleration due to its high-level access via Python and ability for hardware configuration. In the related works section, the author explains well the PYNQ project basics, and its ability for redistribution via Python package automated installation on other PYNQ systems with 'pip' package manager. This is useful for future reference for further redeployment of PYNQ machines. The author proceeds to explain the ability to produce dynamic partial reconfiguration overlays. This approach benefits from HW acceleration, while having a shorter reconfiguration time compared to full bitstream overlays due to the inclusion of replacement strategies for the reconfigurable partitions (programmable logic). The author did this by including strategies known from cache memories such as 'least recently used' and 'first in first out'. These methods reduced the resource usage by 40% in preliminary tests. In evaluation of the article, the author has produced a great description of the acceleration possibilities for a high-level developer using Python. Effective strategies have been conveyed and tests ran as evidence. The code used has been released on GitHub which may provide guidance to our custom overlays which may be produced in future. One gap in the research is the comparison of partial reconfigurable overlays against high-level synthesis, which may provide further understanding of the overheads associated with overlay implementation.

Journal article [7] focusses on the Scikit-learn Python module which integrates a wide range of state-of-the-art machine learning algorithms including the k-means at interest. This module emphasizes primarily on performance and ease of use whilst containing minimal dependencies. Immediately this shows itself as a potential opportunity for our k-means implementation. Since the module is extensively tried and tested, efficient performance and ensuring minimal dependencies is critical when applied to the PYNQ board. Dependencies may lead to potential issues due to the OS configuration with an FPGA, but due to Scikit-learn being only dependent on 'NumPy' and 'SciPy' which are both distributed in the PYNQ official package, this ensures correct functionality and facilitates easy distribution. Further depth of information is discussed regarding individual functions within scki-kit learn and its' corresponding dependencies, providing sufficient confidence on how the library will run on a PYNQ board. The author evaluates sci-kit-learn against various machine learning libraries exposed in Python by running benchmarks on a Medalon dataset with multiple algorithms each including k-means. Scikit-learn proves to be superior in elapsed time of task completion. This information is useful for our choice of Python library during implementation. One gap in the test is the lack of a control, or raw coded algorithm as a reference point without overheads of calling a library. Some libraries such as OpenCv which includes a K-Means function are not included in this test, lacking a comparison between two of the largest machine learning Python libraries. In conclusion the author has presented sound evidence that Scikit-learn is a capable library and suitable for implementation on the PYNQ board due to its small dependencies but lacks a comparison with OpenCv which will need to be reviewed in future.

In conclusion to the literature review, sufficient information has been extracted in relation to accelerating a k-means function. Scikit-learn provides a safe implementation of k-means whilst ensuring efficient performance. Acceleration

may be added through overlays, enabling a Python implementation of hardware configuration. Further hardware acceleration may be explored through partial reconfigurable overlays or high-level synthesis.

III. Progress of Research

Use 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. With the device booted, the connection directs into Jupyter Notebooks. 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. 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.

With the setup complete the first k-means algorithm can be implemented. The goal is for image processing using K-Means clustering. A pre-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 [8]. 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 over the network. The structure of the code remains similar to reduce potential errors on this end, with the exception of some alterations for image directories and Jupyter configuration. 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]:

Fig1. Basic Image Import

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

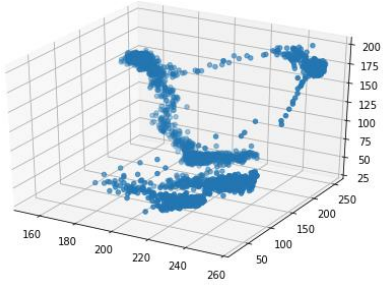

Fig2 3D plot of Image Pixels

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 output:

```
[[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. The result is displayed below.

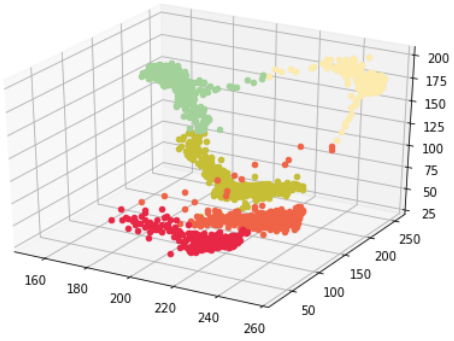

Fig3 3D plot of K-Means Clusters

Currently, development is active on using a video format as input and separating the frames into an image sequence, with further k-means analysis on each frame. The goal is to achieve as close to real-time image analysis as possible from a video feed.

Source code for the main K-Means implementation is available in the appendix, and the full Jupyter Notebook test available from GitLab upon request.

IV. Appendix

1. K-Means test on PYNQ-Z2

```
import cv2
from sklearn.cluster import KMeans

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

class DominantColors:

    CLUSTERS = None
    IMAGE = None
    COLORS = None
    LABELS = None

    def __init__(self, image, clusters=3):

        self.CLUSTERS = clusters
        self.IMAGE = image

    def dominantColors(self):

        #read image
        img = cv2.imread(self.IMAGE)
        #img = cv2.imread('img/colors.jpg')

        #convert to rgb from bgr
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        #reshaping to a list of pixels
        img = img.reshape((img.shape[0] * img.shape[1], 3))

        #save image after operations
        self.IMAGE = img

        #using k-means to cluster pixels
        kmeans = KMeans(n_clusters = self.CLUSTERS)
        kmeans.fit(img)

        #the cluster centers are our dominant colors.
        self.COLORS = kmeans.cluster_centers_

        #save labels
        self.LABELS = kmeans.labels_

        #returning after converting to integer from float
        return self.COLORS.astype(int)


    def rgb_to_hex(self, rgb):
        return '#%02x%02x%02x' % (int(rgb[0]), int(rgb[1]), int(rgb[2]))

    def plotClusters(self):
        #plotting
        fig = plt.figure()
        ax = Axes3D(fig)
        for label, pix in zip(self.LABELS, self.IMAGE):
            ax.scatter(pix[0], pix[1], pix[2], color = self.rgb_to_hex(self.COLORS[label]))
        plt.show()


img = 'img/colors.jpg'
clusters = 5
dc = DominantColors(img, clusters)
colors = dc.dominantColors()
dc.plotClusters()
```

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*, Chogqing, 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. Schrodl, "Constrained K-Means Clustering With Background Knowledge," in *Internation 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] A. K. Jain, D. L. Maskell and S. A. Fahmy, "Are Course-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs?," in *2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*, Auckland, 2016.

[6] B. Janben, P. Zimprich and M. Hubner, "A Dynamic Partial Reconfigurable Overlay Concept for PYNQ," in *27th International Conference on Field Programmable Logic and Applications (FPL)*, Ghent, 2017.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel and B. Thirion, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825-2830, 2011.

[8] 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].