Principal Component Analysis for Data Compression on FPGA

1st Anshul Maurya
Florida Institute of Technology, Melbourne, USA
amaurya2022@my.fit.edu

***Abstract*—Principal Component Analysis (PCA) is a widely used dimensionality reduction technique in machine learning and signal processing applications. It is commonly used for tasks such as higher dimensional dataset visualization, image and video compression, feature extraction, and pattern recognition. However, PCA can be computationally intensive, especially for large datasets, which results in the sluggish execution of PCA-based algorithms popular in Machine Learning Tasks.**

In this paper, we propose an implementation of PCA on a heterogeneous architecture provided by the PYNQ-Z2 board (Zynq-7000 SoC). PYNQ, an open-source project from Xilinx®, offers a convenient and user-friendly platform for leveraging the power of Xilinx platforms. By utilizing the Python language and its rich libraries, designers can effectively harness the potential of programmable logic and microprocessors to develop sophisticated and innovative electronic systems.

Experimental results show that our FPGA-based PCA implementation achieves significant speedup compared to the software-based implementation, with reduced processing time and improved throughput. Furthermore, our implementation demonstrates efficient utilization of FPGA resources and low power consumption, making it suitable for embedded and real-time applications. Our work demonstrates the feasibility and advantages of implementing PCA on an FPGA platform, offering a scalable and high-performance solution for accelerating PCA computations in various applications. The proposed FPGA-based PCA implementation has the potential to enable real-time processing of large datasets, making it applicable to a wide range of fields, including image and video processing, data analytics, and machine learning.

***Index Terms*—PCA, FPGA, PYNQ, Overlay, Vitis, HLS**

I. Introduction

FPGA (Field-Programmable Gate Array) is a reconfigurable hardware device that allows digital circuits to be implemented and customized at the chip level. Unlike application-specific integrated circuits (ASICs) that are fixed in their functionality, FPGAs can be programmed and reprogrammed to perform various tasks by configuring the logic gates and interconnects on the chip. This flexibility makes FPGAs ideal for a wide range of applications, including digital signal processing, machine learning, communication systems, embedded systems, and more. FPGAs offer high performance, low latency, and parallel processing capabilities, making them well-suited for real-time and high-throughput applications. Using FPGA in support of the CPU can provide additional speedup in CPU tasks. Such Performance gain can be very beneficial in the execution of computationally expensive tasks such as machine learning (ML) applications where we deal with a great amount of complex data. In this paper, we will show the performance gain in PCA (Principal Component Analysis) algorithm which is the data preparation stage in many ML algorithms. PCA is a widely used dimensionality reduction technique in machine learning and signal processing. It is a statistical method that transforms a dataset with multiple correlated variables into a smaller set of uncorrelated variables called principal components which contains most of the information of the dataset. In the past, there are many works on PCA implementation on FPGAs like [4] [5], these implemeations are purly on the FPGA and uses only HLS , makes them very complex and modularity wise less suitable, in our implementation, we will be using PYNQ's framework which provides dynamic usage of the PL (Programmable Logic) part and provides more control over the algorithm modeling with FPGA. The PYNQ (Python + Zynq) framework is an open-source project [6] that combines Python programming language and the Zynq System-on-Chip (SoC) from Xilinx, a leading FPGA manufacturer. PYNQ is built on top of the Xilinx Zynq SoC, which integrates a dual-core ARM Cortex-A9 processor with an FPGA fabric. This allows developers to leverage the power of software processing using Python while also taking advantage of the reconfigurability and parallelism of hardware acceleration using FPGA. PYNQ provides a rich set of pre-built overlays, which are hardware libraries that can be loaded onto the FPGA to provide specific functionalities. We implemented PCA on the Zynq-7000 SoC board and parallelized the major matrix multiplication task to the PL, our experimental results show around $3\times$ time gain in algorithm execution with respect to the only CPU execution model and with more floating accuracy in the result.

II. Background

In this paper, the reader will need background knowledge on two major topics: PCA and PYNQ-based development details. We will provide more information on these topics in the implementation section.

We would like to express our sincere gratitude to Dr. Naveed Mahmud for his valuable guidance and insightful suggestions that played a crucial role in the successful implementation and completion of this project.

A. PCA (Principal Component Analysis)

PCA a data reduction technique works by identifying the principal components of a dataset, which are linear combinations of the original features that capture the largest amount of variance in the data. The first principal component (PC1) is the direction along which the data varies the most, and subsequent principal components (PC2, PC3, and so on) capture the remaining variance in descending order.
The PCA algorithm involves the following steps:

1. Data Preparation and Standardization: The input data is typically organized as a matrix where each row represents an observation or data point, and each column represents a feature or variable. PCA is sensitive to the scale of the features, so it is common practice to standardize the data by subtracting the mean and dividing by the standard deviation of each feature. This ensures that all features have the same scale and prevents one feature from dominating the analysis due to its larger magnitude.

2. Covariance Matrix Computation: The covariance matrix is calculated from the standardized data, and it provides information about the relationships between pairs of features in the data.

4. Eigenvalue and Eigenvector Computation: The eigenvalues and eigenvectors of the covariance matrix are calculated. The eigenvalues represent the amount of variance explained by each principal component, and the corresponding eigenvectors represent the directions or axes of the principal components.

5. Principal Component Selection: The eigenvectors are ranked based on their corresponding eigenvalues, and the top k eigenvectors (principal components) are selected, where k is the desired dimensionality of the reduced data.

6. Data Transformation: The original data is projected onto the selected principal components, resulting in a lower-dimensional representation of the data. This is done by multiplying the standardized data matrix with the selected eigenvectors.

B. PYNQ Framework and Development

The PYNQ framework is an open-source project that provides a Python-based platform for programming and designing embedded systems using Xilinx Zynq System-on-Chip (SoC) devices. PYNQ stands for "Python Productivity for Zynq" and aims to make it easy for software developers to leverage the power of FPGA-based hardware acceleration using familiar Python programming paradigms.

PYNQ provides a software platform that includes a Linux-based operating system running on the ARM processor of the Zynq SoC, along with Python libraries and tools for programming the FPGA fabric [6]. The framework allows developers to use Python to program both the software running on the ARM processor and the hardware logic implemented in the FPGA fabric. This enables the development of custom accelerators, interfaces, and other digital circuits using Python, which is a popular language for data science, machine learning, and other application domains.

The PYNQ framework also provides a Jupyter notebook environment, which allows developers to create interactive, web-based notebooks for developing and testing FPGA-based designs. These notebooks provide a graphical interface for designing digital circuits using high-level hardware abstraction libraries, and they can be used for rapid prototyping, testing, and debugging of FPGA-based designs.

PYNQ supports a wide range of FPGA development boards from Xilinx, including the Zynq-7000 and Zynq UltraScale+ SoC platforms. These boards provide a combination of ARM processors and FPGA fabric, allowing for tight integration of software and hardware accelerators.

The development process with PYNQ typically involves designing the desired digital circuit using high-level hardware abstraction libraries in Python, compiling the design into a bitstream that can be loaded onto the FPGA, and then running Python applications on the ARM processor that interact with the custom hardware accelerators. PYNQ provides a rich ecosystem of software libraries, examples, and documentation to help developers get started with FPGA development using Python and the Zynq SoC platforms.

Overall, the PYNQ framework and development process provides a user-friendly and productive platform for designing embedded systems with FPGA-based hardware acceleration using Python, making it accessible to software developers with little or no prior FPGA experience.

III. Implementation

In this section, we will present our implementation of Principal Component Analysis (PCA), which involves utilizing the CPU for the initial steps of the algorithm, while parallelizing the data projection onto the principal axes (III-A) using programmable logic to achieve significant speedup. Specifically, in our implementation, the CPU is responsible for centering the data and performing eigenvector computation.
While centering the data is relatively computationally inexpensive, eigenvector computation involves costly multiplication and division operations. Thus, parallelizing these steps using programmable logic has the potential to significantly enhance the speedup of our implementation. This aspect could also be considered as part of our future work.

The data projection step involves a dot product between eigenvectors and the data matrix. This computationally intensive dot product we will deploy on PL logic. PYNQ allows for the dynamic integration of hardware libraries, enabling the PL component of the system-on-chip (SoC) to connect with the system at runtime. To perform the dot product, we utilized a custom overlay that performs floating point matrix multiplication, in the form of $C = A * B$, {where A, B, and C are 128×128 floating-point matrix}. The custom overlay used in this project [1] was developed in C++ using High-Level Synthesis (HLS) techniques, based on Xilinx's original application note [2]. This overlay is implemented as an Intellectual Property (IP) and utilizes the AXI protocol. By connecting this IP to the AXI protocol, it gains access to the Dynamic Random Access Memory (DRAM) module of the Processing System (PS), enabling it to transfer data between the PL and the PS efficiently.

Prior to accessing the DRAM of the Processing System (PS), it is necessary to allocate buffers for the IP block. These

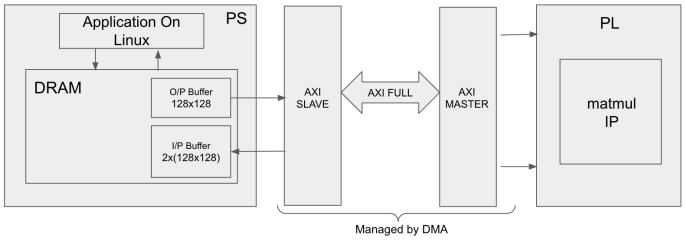

Fig. 1. Implemented Architecture on PYNQ-Z2

buffers are reserved for the IP's use and the size and address of the memory are passed to the IP in the PL, then DMA manages the data transfer based on the processor's request. To create these buffers, PYNQ offers the allocate method (pynq.allocate) [3], which creates a contiguous memory block that serves as a buffer for the IP block to read data from the DRAM on the PS side. For our implementation, we created two buffers to hold the input and output data for the IP. The input buffer is used to store the data matrix and the eigenvector matrix, both of which have a size of 128×128 floating point of 32 bytes. Therefore, the size of the input buffer is $2 * (128 \times 128)$. Similarly, an output buffer can be created with the same size as the data. To create these buffers, we used the following code snippet:

```
DIM1 = 128
DIM2 = 128

in_buffer = allocate(shape=(2, DIM1, DIM2), dtype=np.float32, cacheable=True)
out_buffer = allocate(shape=(DIM1, DIM2), dtype=np.float32, cacheable=True)
```

Fig. 2. Example code snippet for Buffer Allocation

With the above settings complete, our PL block is now ready to be used, and we can proceed to the initialization step. IPs created in Vivado that use the AXI protocol by default produce a control register at the 0x00 address. The below table explains the bit-wise details of the control register.

TABLE I
IP'S CONTROL REGISTER

Bit *Number*	***Bit Description***
0	ap_start (Read/Write/COH)
1	ap_done (Read/COR)
2	ap_idle (Read)
3	ap_ready (Read)
7	auto_restart (Read/Write)
others	reserved

Our design needs the start and auto restart bit to be set to 1, which mean bit 1 and 7 should be set to 1. This will enable our IP block to be active all the time.

A. PCA Implementation with NumPy

In our implementation, we used NumPy as our primary computational tool for performing the necessary calculations for the PCA. NumPy is a popular numerical computing library in Python that provides an extensive set of functions for scientific and mathematical operations, including operations on arrays and matrices [7].

Our dataset was stored in the form of a NumPy array, which allowed us to efficiently process the data using NumPy's capabilities. NumPy employs various techniques to enable faster computation of matrix operations, such as vectorization and other linear algebra rules, which make it well-suited for performing complex calculations on large datasets. Our implementation of the PCA algorithm was divided into three functions, each of which performed a specific part of the algorithm. The first function computed the covariance of the data after the centering operation. To perform this computation, we utilized NumPy's built-in numpy.cov [8] function, which provides an efficient and reliable way to compute the covariance matrix of a dataset.

The second function took the covariance matrix computed by the previous function and computed its eigenvectors and eigenvalues. We then sorted the eigenvectors in ascending order based on their corresponding eigenvalues. To achieve this, we utilized the numpy.argsort [9] function, which allows us to obtain the indices that would sort the array. We then used these indices to sort the eigenvector matrix accordingly. This sorted eigenvector matrix was returned from the function.

The final function was where the actual dimension reduction step was performed. Here, we simply took the dot product of the data matrix with the sorted eigenvector matrix obtained from the previous function. This resulted in the data being projected onto the most important eigenvectors in ascending order.

By dividing the PCA algorithm into these three functions, we were able to modularize the implementation and leverage the PL acceleration effectively. In the final function, where we performed the dot product, we provided two variants. The first variant utilized NumPy to perform the dot product on the CPU. The second variant utilized the PL to perform the dot product and was more suitable for large datasets.

In the second variant, we utilized the numpy.stack function to stack both matrices and write them onto created input buffer. The IP then executed the floating-point matrix multiplication on the input data and returned the output data. By using the PL variant of this function, we were able to accelerate the computation significantly and reduce the execution time for large datasets.

IV. EXPERIMENTAL RESULTS

After implementing our PCA algorithm on the hardware using PL acceleration, we wanted to verify its correctness and performance. To do so, we compared the results of our implementation with the industry-standard PCA implementation provided by scikit-learn.

Furthermore, we conducted a T-test on randomly generated 128x128 data to compare the mean values obtained from the PL acceleration and the CPU-based implementation. In this T-test, we did not need to create a hypothesis since we had already verified the results with the scikit-learn library. We used two datasets for the T-test, which were the output values obtained from the PL and CPU-based implementations for the randomly generated data.

Based on the table presented, we can observe that the mean value of the output obtained from our implementation is quite

TABLE II
T-TEST COMPARISON TABLE

Statistical *Values*	***On PL Result***	***On CPU only Implementation***
Mean	7.15e-07	-5.77e-15
Std. Deviation	285.89774	285.89775
Variance	81737.52	81737.52

close to the expected mean value of 0, due to the centering of the input data. However, there is a slight difference between

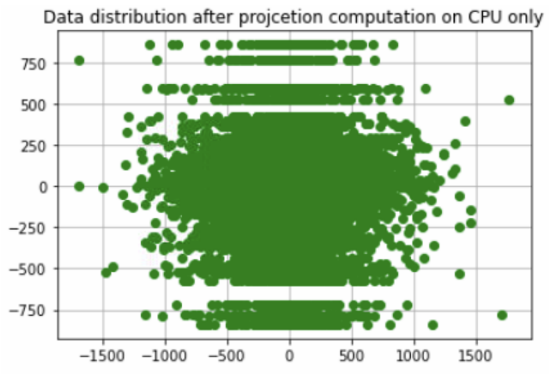

Fig. 3. Projected Data Distribution On CPU Only

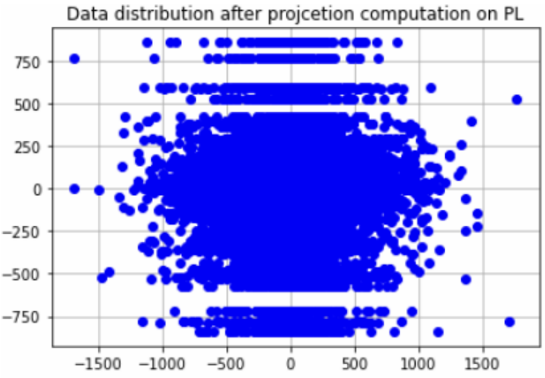

Fig. 4. Projected Data Distribution On PL

the mean values obtained from the PL and CPU, which can be attributed to the variation in the result produced by the two methods. Nevertheless, the variance and standard deviation of the results are consistent, and the data distribution graph is also identical, thus indicating the successful passing of our T-Test.

To analyze the PCA execution time and the speedup achieved using the PL and the PS, we conducted a timing analysis on four different random datasets of varying sizes: 128x128, 256x256, 384x384, and 512x512. We recorded the execution time and calculated the speedup. Fig. 5 depicts the PCA execution timings for each of the mentioned data sizes and shows the time distribution in terms of the percentage of the retained dimension. Based on Fig. 5, we observe that the PL-based implementation takes only one-third of the PS execution time. The speedup achieved for each of

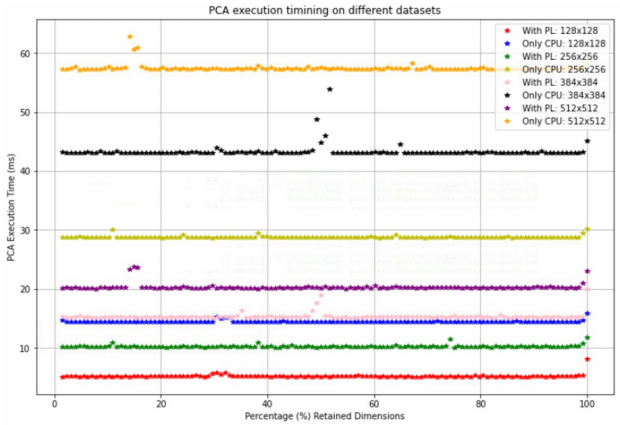

Fig. 5. PCA Execution Timing Comparison

the mentioned data sizes is depicted in Figure 6. It can be observed that larger datasets achieve higher speedups when no compression is applied, and all features are retained in the PCA output. The implementation attains a maximum speedup of 3 and an average speedup of approximately 2.8, with some fluctuations in the graphs. We believe that the fluctuations may be due to scheduling jitters in the operating systems because these fluctuations are not consistent.

Fig. 6. PCA Execution Speedup

Our analysis revealed some inconsistencies between the results produced by PL and PS. As shown in Fig. 7, there is a difference between the corresponding data points for the 128x128 dataset, with most differences being in the magnitude of around 10^{-5}. Additionally, Fig. 8 displays the mean log average difference for each of the 128 features, which varies in magnitude by approximately 4×10^{-5}.

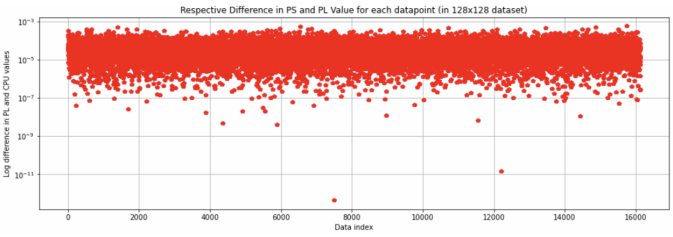

Fig. 7. Difference in computed value from PS and PL for each datapoint

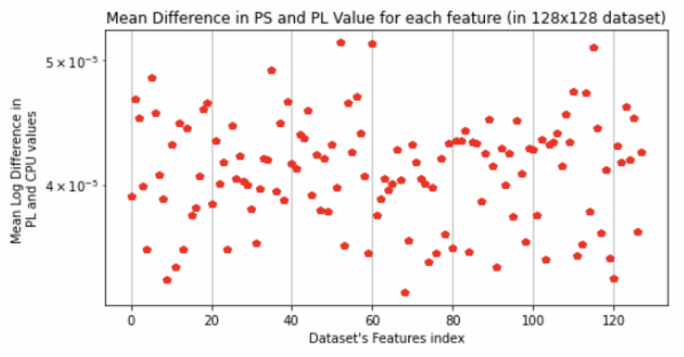

Fig. 8. The row-wise difference in computed value from PS and PL

Conclusion and Future Work

In this project, we utilized the PYNQ's capability of dynamically attaching hardware implemented in the PL to accelerate the application's performance. We have demonstrated the method of using PYNQ's overlay with the application using AXI protocol, and our experimental results showed around 3 times speedup in the PCA algorithm's performance.

This implementation can be further improved for more speedup by adding the PL part in places where we are using NumPy's functions that perform costly computations, as part of future work. This can be achieved by exploring the possibility of implementing those functions in the PL using hardware accelerators, which will reduce the execution time further. Overall, the use of PYNQ's dynamic reconfiguration feature has shown promising results in accelerating the performance of the PCA algorithm and can be extended to other applications as well.

References

[1] https://github.com/twaclaw/matmult
[2] A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS (XAPP1170) [https://docs.xilinx.com/v/u/en-US/xapp1170-zynq-hls]
[3] https://pynq.readthedocs.io/en/v2.5/pynq_libraries/allocate.html#example
[4] M. A. Mansoori and M. R. Casu, "Efficient FPGA Implementation of PCA Algorithm for Large Data using High Level Synthesis," 2019 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), Lausanne, Switzerland, 2019, pp. 65-68, doi: 10.1109/PRIME.2019.8787782.
[5] R. Endluri, M. Kathait and K. C. Ray, "Face recognition using PCA on FPGA based embedded platform," 2013 International Conference on Control, Automation, Robotics and Embedded Systems (CARE), Jabalpur, India, 2013, pp. 1-4, doi: 10.1109/CARE.2013.6733778.
[6] http://www.pynq.io/
[7] https://numpy.org/about/
[8] https://numpy.org/doc/stable/reference/generated/numpy.cov.html
[9] https://numpy.org/doc/stable/reference/generated/numpy.argsort.html