HLS-Based Flexible Hardware Accelerator for PCA Algorithm on a Low-Cost ZYNQ SoC

HLS-Based Flexible Hardware Accelerator for PCA Algorithm on a Low-Cost ZYNQ SoC

Mohammad Amir Mansoori, Mario R. Casu
Department of Electronics and Telecommunications, Politecnico di Torino, Turin, Italy.
e-mail: {*mohammadamir.mansoori, mario.casu*}*@polito.it*

***Abstract*—Principal Component Analysis (PCA) is a widely used approach for dimensionality reduction in image processing. In microwave imaging, for example, it is used as an intermediate step toward image reconstruction. An FPGA hardware implementation of PCA is highly beneficial, especially as an accelerator for a low-cost embedded environment. In this paper we propose a flexible PCA hardware accelerator that can be used for different input data dimensions and input precisions. In addition, it supports both floating-point and fixed-point arithmetic representations. The target hardware is a ZYNQ SoC. We used High Level Synthesis (HLS) to quickly explore the design space and so to find the best implementation for a given setting of the application parameters and given the characteristics of the target hardware. We show the impact on performance of different hardware optimization techniques enabled by HLS. The proposed method outperforms a similar state-of-the-art HLS design in terms of latency and resource usage.**

***Index Terms*—PCA, FPGA, HLS, ZYNQ SoC, Hardware Optimization**

I. Introduction

Principal Component Analysis (PCA) is a statistical method useful for data dimensionality reduction. PCA starts by finding the directions of data (e.g. a set of images or signals) corresponding to maximum variances, which are called Principal Components (PCs). Then, by ignoring the minimum PCs, the data are projected into lower dimensions without significant loss of information. PCA is useful in various applications. In machine learning, the feature extraction can use PCA as the first step to obtain the most uncorrelated features from a dataset, which is also important for the accuracy of the classification step [1], [2]. In Microwave Imaging (MI), which is becoming popular in medical image processing, PCA is one of the main approaches for dimensionality reduction [3]–[5].

PCA is computationally expensive as it typically deals with large number of arithmetic operations on large datasets. To accelerate its execution, a dedicated hardware implementation is extremely helpful, especially in low-cost embedded systems with limited computing and power capabilities. In the last two decades there has been a growing interest in using FPGAs as hardware accelerators in a variety of applications including machine learning and image processing. FPGAs can process large data in parallel with lower power consumption compared to GPUs or multicore CPUs. For this reason, a significant research effort has been invested into designing FPGA-based hardware architectures to accelerate the PCA execution. One of the critical steps in PCA is the Singular Value Decomposition (SVD) of the input data, for which there has been a considerable effort toward designing efficient hardware implementations. For example, in [6] a novel architecture was proposed based on two CORDIC modules that proved useful to reduce the total resource usage. In [7] a modified CORDIC algorithm was introduced for SVD calculation based on the Jacobi method, which works also for complex matrices. In [8]–[10] other RTL implementations of SVD were presented.

Besides SVD, the three other important steps in PCA are "Mean computation","Covariance calculation", and "Projection computation," which are discussed thoroughly in the next sections. Although they are less complex than SVD, these steps process a large amount of input data, which affects the computation time and makes it difficult to design an efficient low-cost hardware that includes all parts of PCA. For instance, in [11] PCA is used in a fall detection system as a feature extraction step, but the PC coefficients are not computed in hardware due to the high resource usage, and only the Projection is implemented in hardware. In the application domain of hyperspectral imaging, Fernandez et al. [12] use a powerful Virtex-7 FPGA to implement PCA for large data dimensions. Due to the large dimension of data, Mean and Covariance calculation were performed in software. Lately, in [14] a new design for PCA acceleration was introduced for large data without including covariance and mean computations.

Although manual RTL design using VHDL or Verilog is still predominant in FPGA design, the design approach using High-Level Synthesis (HLS), which leverages programming languages like C or C++, is gaining momentum. Through the processes of scheduling, allocation, and binding, HLS converts a software code into its corresponding RTL description. The main advantage of HLS is the ability to explore the design space in search of Pareto-optimal solutions more efficiently and quickly than with a hand-coded RTL design.

Recently, HLS designs have been shown to be effective for PCA hardware acceleration. I. Bravo et al. [13] redesigned their previous RTL implementation of the Jacobi algorithm using HLS and obtained better performance. In [15] the authors present a HLS-based design targeting a low-cost Xilinx ZYNQ for computing PCA of spectral images with dimensions of 12 spectral bands. The SVD part was implemented in software in the Processing System (PS) of the ZYNQ while the other parts were designed in the Programmable Logic (PL) of the ZYNQ and took most of the FPGA resources.

The previous PCA designs either use manual RTL or cannot implement a complete PCA algorithm efficiently in hardware.

The design in [14], for example, uses HLS for hardware design, but does not include all parts of PCA. In contrast with previous works, we propose in this paper a flexible hardware architecture based on HLS that fully implements all the steps of PCA in a low-cost ZYNQ (xc7z020clg484-1) and reduces the overall computing latency with respect to state of the art [15]. The flexibility is enabled primarily by the use of HLS, which allows us to use different data dimensions and precision and to support both floating- and fixed-point computations.

In the following, we first describe briefly the PCA algorithm. Then we explain the proposed design and different HLS-based hardware optimization techniques in Sec. II. In Sec. III we discuss the results for different dimensions and data types. Finally, we draw the conclusions in Sec. IV.

A. PCA description

Consider the input data as a matrix of size $R \times C$, in which R is the number of data and C are the original dimensions. For example, each of the R data vectors can be a signal (i.e. its samples) or an image (i.e. its pixels represented as a single vector), whereas C can be the number of spectral bands (of signals or images). PCA computes the principal components corresponding to maximum variations of data by using the following procedure:

1- **Mean computation:** The first step in PCA is data normalization which requires the mean values of input matrix:

$$[Mean]_{1\times C} = \frac{1}{R}\sum_{i=1}^{R}[X_i]_{1\times C} \tag{1}$$

where $[X]_i$ is the ith row of the input matrix X. To use a matrix representation, $[M]_{R\times C}$ is derived from vector $Mean$:

$$[M_i]_{1\times C} = [Mean]_{1\times C}, i = 1, 2, ..., R \tag{2}$$

where $[M_i]$ is the ith row of matrix M.

2- **Covariance calculation**: To find the correlation between input data samples, the covariance matrix is computed:

$$[COV]_{C\times C} = \frac{1}{R-1}(X-M)^T \times (X-M) \tag{3}$$

3- **SVD of covariance**: The next step is to find the singular values and vectors of the covariance matrix:

$$COV = U\Sigma U^T \tag{4}$$

where Σ is a diagonal matrix containing all the singular values of covariance matrix, and U contains the singular vectors.

4- **Sort and selection**: The singular values and vectors are sorted in descending order and after selecting a proper threshold, the first few components ($L < C$) are preserved:

$$\Sigma^s, U^s = Sort(\Sigma, U) \tag{5}$$

$$[PC]_{C\times L} = Select(\Sigma^s, U^s) \tag{6}$$

The selection of PCs in (6) is based on the cumulative energy of singular values. At first, the total energy of singular values (E) is obtained:

$$E = \sum_{i=1}^{C}\sigma_i, \tag{7}$$

where σ_is are the singular values from Σ^s. After that, the first L components are selected in such a way that their cumulative energy is no less than a predetermined fraction of total energy, the threshold T (%):

$$100 \times \frac{\sum_{i=1}^{L}\sigma_i}{E} \geq T \tag{8}$$

5- **Projection**: Finally, the normalized input data is projected into the new base by the following equation:

$$[Y]_{R\times L} = (X-M) \times PC \tag{9}$$

B. PCA application

Although the application of interest for PCA is MI for this paper, to the best of our knowledge there are no previous FPGA implementations specifically for MI that we could use as a reference for comparison. Therefore, we selected the application of spectral images for which a similar PCA hardware design is found [15]. The images are obtained in several bands in the electromagnetic spectrum and are organized as a matrix with the rows R equal to the number of pixels and the columns C equal to the number of bands, thus in the following we use terms "columns" and "bands" interchangeably.

II. Proposed method

A. Hardware Design

The proposed hardware for PCA implementation is described in Fig. 1. It consists of one single HLS design (the large green box) with two core functions, Dispatcher and PCA cores. Dispatcher reads input data X from an external memory using a memory controller (the yellow box, a pre-designed Xilinx IP), and dispatches them to three sets of FIFOs. These FIFOs are connected to three main PCA components that directly work on input data: Mean, Covariance (COV), and Projection Unit (PU). At first, the Mean unit reads data, computes the mean values according to (1) and stores them in an internal memory. After that, the COV unit reads data from its corresponding FIFO and mean values from the memory (to normalize data) and computes the covariance matrix, which is stored in another internal memory. The SVD unit reads the covariance matrix from this memory and computes singular values and vectors. Finally, after sorting and selecting the principal components, the Projection Unit reads the selected PCs and multiplies them by the normalized input data, which are obtained from the FIFOs and the Mean memory. The final output from PU is written back to the external memory.

The procedure for reading data from FIFOs is as follows. In the Mean unit, the mean of every column of input matrix is computed as in (1). To compute all the C mean values in parallel, C FIFOs are used so that the Mean unit reads C inputs at once from these FIFOs and accumulates them. When all of the input rows are read from FIFOs, the accumulated columns are divided by the number of rows (R) to store the final means in the internal memory. It is possible to read C values in each clock cycle as long as the number of FIFOs is less than F_{max}, otherwise the external memory bandwidth

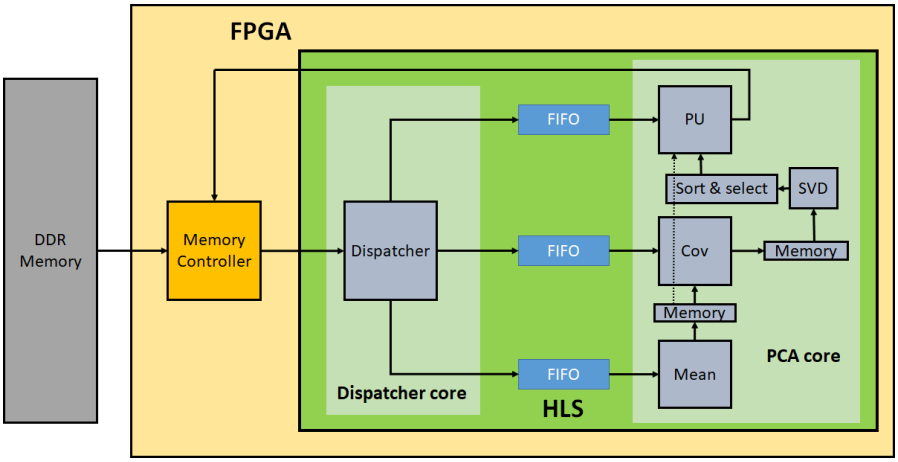

Fig. 1. Block diagram of the proposed hardware implementation for PCA.

saturates. When $C > F_{max}$ it is not possible to process all the columns at the same time and the minimum achievable Initiation Interval (II), the time to receive a new set of data, is equal to C/F_{max} clock cycles.

In the Projection Unit the same approach is adopted and every input row (containing all the columns) is received from C FIFOs and multiplied, after normalization, by the principal components according to (9).

For the Covariance unit, due to the high II of reading data from COV FIFOs, there is no benefit in using C FIFOs. Even by using one FIFO there is enough time for the dispatcher to write all the required data to the single FIFO while COV is processing the previous set of data. This will be illustrated in detail in subsection II-C.

SVD is less critical, performance-wise, than the rest, because it does not work directly on the large input data rows. In this design, we used a built-in HLS SVD function. Although the latency of SVD is negligible, its resource usage is comparable to other PCA parts.

It is important to note that all of the previous parts are designed using a single HLS code, which is advantageous in providing various flexibility options.

B. Flexibility

HLS allows us to quickly reconfigure the design without changing significantly the original code. This allows us to make our PCA design more general and easy to deploy in different application scenarios. For example, different sensors used in MI or spectral imaging may have different resolutions resulting in different data precision.

In this design, we assume the input data are integers stored in the external memory with data widths of 8, 16, or 32 bits, and the outputs are stored as 32-bit floating-point numbers. The number of columns (or bands in spectral images) is variable up to a maximum determined by the available hardware resources.

The computations in PCA core can be either in floating- or fixed-point arithmetic. However, the built-in SVD function works on floating-point values only, therefore the fixed-point version of our design requires data conversion before/after the SVD function. A custom fixed-point version of SVD is left for future work.

C. HLS Optimizations

HLS provides a hardware designer with the possibility to create the desired hardware and improve its efficiency by applying different optimization techniques to the same design. The hardware interfaces, the level of parallelism in loops, the loop pipelining option, and specific resource allocation options can be all determined by proper HLS directives.

Among these directives, the Dataflow one allows two functions to operate concurrently by creating channels between them. In our design, the two main HLS sub-functions are connected by using a Dataflow optimization directive. We specified the channels to be FIFOs to allow data streaming. In addition, the input partitioning, and so the number of FIFOs, follows the number of columns (except for $C > F_{max}$) to maximize the parallelism.

The total latency of the initial unoptimized design was excessive. To reduce it, we optimized one by one the main components of the PCA core that were the bottlenecks of the design: Mean, COV, and PU. The impact of these subsequent optimizations on the total latency and resource usage is shown in Fig. 2. Here we report, as an example, the case with $R = 640 \times 480$, $C = 12$, and 8 bits for the input data width. Since the dispatcher input is limited to 256 bits in the target FPGA, this combination results in $F_{max} = 256/8 = 32$.

The default unoptimized design has the least amount of resource usage and the highest latency: 9.06 s, which includes 5.4 s for COV, 3.04 s for PU, and 0.52 s for Mean units.

We first optimized the Mean unit because it affects other PCA parts as they use its output for normalization. The Mean unit implements two loops on the rows and columns to compute the sum of input data for each column. The code snippet is shown in Algorithm 1. It stores data from input FIFOs (Din_Mean) in a_mean memory and accumulates the

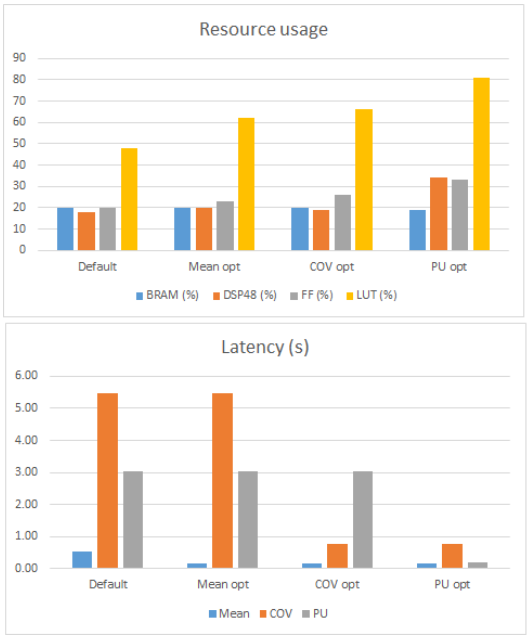

Fig. 2. Impact of HLS optimizations on latency and resource usage.

results in tmp_mean. By using the "array partitioning" HLS directive for a_mean and tmp_mean, the loops are unrolled and the input array split so that all of the C mean values are computed in parallel reading inputs from multiple FIFOs. In addition, a "pipeline" directive is used for the first loop to improve the throughput and reduce the latency.

Algorithm 1 (Mean computation)

```
mean_row_loop:
for (int r=0;r<R;r++){
#pragma HLS PIPELINE
        mean_col_loop:
        for(int c=0;c<C;c++){
                a_mean[c]=Din_Mean[r][c];
                tmp_mean[c]+=a_mean[c];}}
Divide_loop:
for(int c=0;c<C;c++)
        a_mean[c]=tmp_mean[c]/R;
```

Notice that since the output array a_mean is partitioned, this optimization affects also the other PCA parts that use this partitioned array as an input. This is the reason why we tackle the Mean optimization first.

For floating-point arithmetics, the minimum II for loop pipelining in Mean unit is 5 clock cycles. This is because the floating-point adder has a 5-cycle latency and since the accumulation of tmp_mean over C values introduces a loop-carried dependency, it is not possible to exploit finer-grain pipelining. For fixed-point instead, II can be reduced to 1 clock cycle because the fixed-point adder has a 1-cycle latency. These optimizations reduce the Mean latency in the floating-point case from 0.52 s to 0.15 s.

The next optimization is for COV unit. Due to the symmetry of the covariance matrix, it is sufficient to compute $N = (C \times (C+1))/2$ values rather than all of its C^2 elements. The

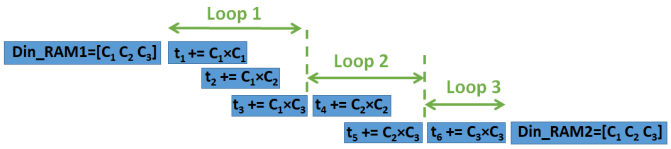

Fig. 3. Covariance loops pipelining with C=3.

internal hardware for COV stores the received data from its FIFO in a memory and updates the N covariance components based on (3). The code snippet is shown in Algorithm 2.

In the RAM_LOOP, normalized input data are stored in Din_RAM memory and in the next loops the multiplications between the required elements are computed and stored in tmp_cov of size N. By applying loop pipelining and unrolling (as shown in Algorithm 2 by the respective "pragma HLS" directives), different iterations of COV_COL2 loop can be pipelined to reduce the latency.

Algorithm 2 (COV computation)

```
COV_Loop_Rows:
for(int r=0;r<R;r++){
 RAM_LOOP:
 for(int c=0;c<C;c++){
#pragma HLS PIPELINE
  Din_RAM[c]=Din_Cov[r][c]-a_mean[c];}
  COV_COL1:
  for(int c1=0;c1<C;c1++){
#pragma HLS UNROLL
    COV_COL2:
    for(int c2=c1;c2<C;c2++){
#pragma HLS PIPELINE
      ... //Indexing
tmp_cov[Index]+=Din_RAM[c1]*Din_RAM[c2];}}}
```

Fig. 3 shows the schedule for the COV unit when $C = 3$. C_1 to C_3 are 3 column values of one row in input data which are received from a single FIFO. The three loops in COV_LOOP1 are executed sequentially, but each loop is pipelined with II=1. The impact on resources is minimal, but we cannot further optimize the latency by unrolling the loops: if we unrolled them, the hardware usage would exceed the available FPGA resources. Therefore, this optimization choice is a trade-off between resource usage and latency. Nonetheless, applying these optimizations (and keeping the previous directives for Mean unit) results in reducing COV latency significantly, from 5.46 s down to 0.76 s.

As described before, we used only one FIFO for the COV interface. As shown in Fig. 3, the computation time of the $N = 6$ elements of covariance is the sum of iteration latency of the loops (the number of clock cycles to complete one loop iteration), and COV has to wait until these computations are finished before it receives the next set of data. This is the reason of high II of COV for reading from the FIFO. By using one FIFO, the Dispatcher writes data in the FIFO while COV is computing and using more FIFOs would be useless.

The final optimization is for the PU for which the code snippet is presented in Algorithm 3. Three loops multiply normalized data and principal components.

Algorithm 3 (PU computation)

```
Projection_Loop:
for(int r=0;r<R;r++){
    COLB:for(int c1=0;c1<L;c1++){
#pragma HLS PIPELINE
        tmp=0;
        ...
        COLA:for(int c2=0;c2<C;n++){
            tmp+=(Din_Nrml[c2]*PC[c2][c1];}
        Data_Transformed[r][c1]=tmp;}}
```

The best optimization in this case is to pipeline the middle loop, which unrolls the inner loop and makes the unrolled loops operate in parallel, resulting in a reduced latency with an II of 2. This also requires partitioning the arrays involved in the multiplication. After applying all of these hardware directives, the PU latency decreases from 3.04 s to 0.18 s.

III. Results

The proposed PCA hardware flexible accelerator is evaluated on the low-cost ZYNQ SoC xc7z020clg484-1 used in the Zedboard ZC702. The frequency of operation is 100 MHz. The flexibility is implemented through a set of variables to be set at design time, which define the value of design parameters like the AXI width for the Dispatcher input, the number of FIFO channels, the input data width, and the data dimensions. Some of these parameters are automatically determined by HLS based on the maximum bandwidth toward the external memory (like the maximum number of FIFO channels), whereas the others depend on the application. Vivado HLS 2018.2 is used for the evaluation and performance measurements.

In the following experimental results, we set the data size of rows to 640×480 pixels and measure the performance in terms of latency and resource usage when we vary the number of bands (i.e. C, the number of columns), the input data width (DW), and the data representation for internal calculations (32-bit fixed or floating point). In addition, we compare our work with a recent implementation of PCA in the same target FPGA [15]. In this reference, the number of bands is C=12, the number of rows (or pixels) is R=640×480, the input data width is DW=8 bits, and the computations are in floating-point. Notice that our design includes all PCA parts, whereas in [15] the SVD is implemented in software. In addition, the authors used separate HLS blocks and Vivado IPs because the Processing System (PS) controlled the external memory addresses, which were transferred separately to the corresponding blocks (details are found in [15]). In contrast, we use an unified HLS code for all the PCA functions.

The total latency in our design depends only on the number of bands (as long as $C \leq F_{max}$) and is shown in Fig. 4. The figure highlights also the maximum number of bands that the target FPGA can support before saturating the resources. It is evident that the fixed-point arithmetic accepts more bands and results in lower latency compared to floating-point.

The total resource consumption of Dispatcher, FIFO channels, and PCA core is in Fig. 5 for DW=8 and different number of bands. For floating-point representation, LUTs are the bottleneck of the design, while for fixed-point, DSP48s

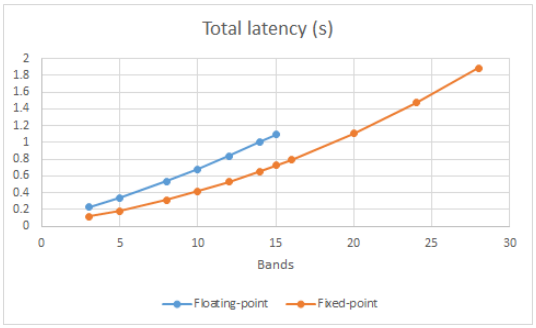

Fig. 4. Total latency for different bands for the proposed PCA accelerator.

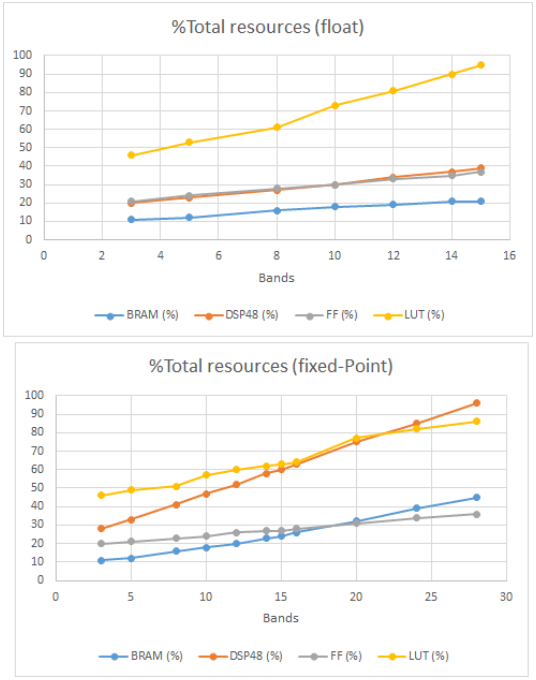

Fig. 5. Total resource usage for different bands (data width = 8 bits)

are the limitation. The maximum number of bands is 15 for floating-point and 28 for fixed-point computations.

When DW increases, the use of memory bandwidth increases. As a result, the maximum number of FIFOs (F_{max}) decreases. When $C \leq F_{max}$, the latency is independent of DW because i) there is enough bandwidth to write all the input data to the FIFOs and ii) the PCA computations are always performed on 32 bits (floating- or fixed-point) regardless of DW. DW only affects the resource utilization of the Dispatcher and its FIFO interfaces.

When $C > F_{max}$, the II for Mean and PU increases by a factor of C/F_{max} due to the bandwidth limitation. Note that in this case we can use fewer FIFOs ($F < F_{max}$) for the same II. For example, when $F_{max} = 16$ and $C = 20$, we obtain II=2 either using 16 or 10 FIFOs. We obtain the optimum number of FIFOs by finding the lowest integer n for which $F = (C+n)/(n+1) \leq F_{max}$. In this way we reduce the resource consumption without affecting the latency.

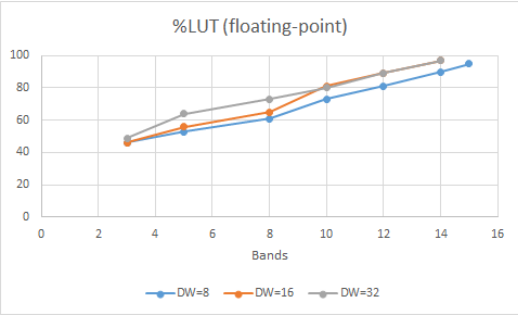

Fig. 6. Total resource usage for different data widths (floating-point).

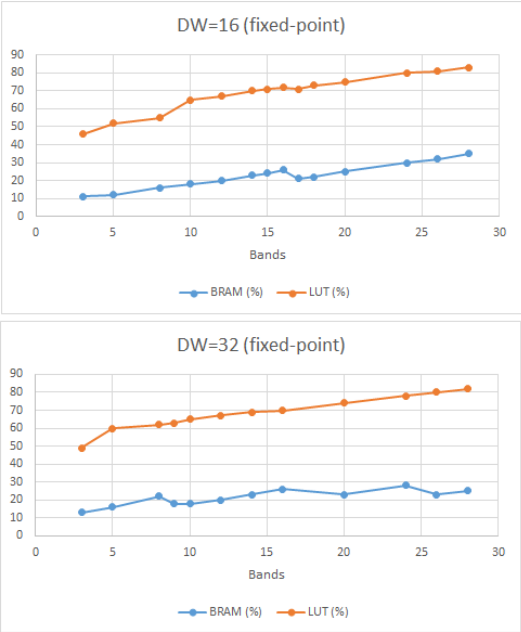

Fig. 7. Total resource usage for fixed-point and data widths 16 and 32 bits.

Figs. 6-7 show the impact of DW on the resource usage when the number of bands C increases, for floating and fixed-point, respectively. The Dispatcher consumes mainly LUT resources, which increase for higher DW. In Fig. 6 the LUT usage is the same for DW=16 and DW=32 when $C \geq$10. This is because for these cases more resources are mapped to BRAMs rather than LUTs. The fact that the BRAM usage does not always increase is because of the reduction of FIFOs when $C > F_{max}$, which is especially evident in Fig. 7.

Changing the input data width also affects the total latency when $C > F_{max}$ due to the increase in the II of Mean and PU. However, since the latency of COV is the dominant contribution, the overall increase is not significant. In the worst case, when DW increases from 8 to 32 and the number of bands is C=28 for fixed-point, the latency increases from 1.88 s to 1.91 s (note that for both values of DW, the COV computations are done in 32-bit fixed-point).

Compared with reference [15], as shown in Table I, the final latency and resource usage improve, except for a slight increase in DSP usage. One possible reason of the superiority of our design is that we use a single HLS code, rather than various blocks implemented individually as done in [15]. This allows the HLS synthesizer to find better solutions that maximize resource sharing between the subfunctions without impacting on the overall latency.

TABLE I
PERFORMANCE COMPARISON BETWEEN THE REFERENCE [15] (SEPARATE DESIGN) AND THIS WORK (SINGLE HLS), C=12, DW=8.

	latency (s)	%BRAM	%DSP48	%FF	%LUT
Ref(float)	1.1	12	19	38	73
Proposed (float)	0.83	4	21	17	46
Proposed (fixed)	0.53	5	39	10	24
Proposed SVD	—	5	13	13	21

The resource usage in the table keeps the SVD unit of the proposed design separated for a fair comparison with the reference. The latency of the proposed design, however, includes the SVD latency (similar to the reference). Even though the reference does not include the Mean computation, we could not separate it from our overall resource usage because, differently from the SVD unit, which is a separate function, the Mean is simply a loop in the main code and Vivado HLS does not report resource usage for individual loops.

IV. Conclusions and Future Work

In this paper, a flexible hardware PCA accelerator is proposed and evaluated on a low-cost ZYNQ SoC. It supports different input data widths and dimensions, as well as both floating and fixed point arithmetic depending on the application needs. The design uses a single HLS code and several HLS optimization strategies to improve the performance. Compared with a reference HLS design implemented as separate HLS blocks, the proposed design achieves lower latency and resource usage. In addition, we show that a fixed-point implementation has lower latency and supports higher data dimensions. In the future, we will increase the flexibility by adding support for different FPGA boards. Since we used a built-in HLS function for SVD based on floating-point, we had to include type conversion. Therefore, we will also work on a fixed-point implementation of SVD to further reduce the resource usage and so to support larger number of dimensions.

Acknowledgment

This work was supported by the EMERALD project funded by the European Unions Horizon 2020 research and innovation programme under the Marie Skodowska-Curie grant agreement No. 764479.

References

[1] S. Khalid, T. Khalil and S. Nasreen, “A survey of feature selection and feature extraction techniques in machine learning,” Science and Information Conference, London, pp. 372–378, 2014.

[2] X. Kang, X. Xiang, S. Li and J. A. Benediktsson, "PCA-based edge-preserving features for hyperspectral image classification," IEEE Transactions on Geoscience and Remote Sensing, vol. 55, no. 12, pp. 7140–7151, 2017.

[3] E. Ricci, S. di Domenico, E. Cianca et al., "PCA-based artifact removal algorithm for stroke detection using UWB radar imaging," Medical & Biological Engineering & Computing, vol. 55, no. 6, pp 909–921, 2017.

[4] M. N. Tabassum, I. Elshafiey and M. Alam, "Enhanced noninvasive imaging system for dispersive highly coherent space," International Conference on Acoustics, Speech and Signal Processing (ICASSP), Brisbane, pp. 912–916, 2015.

[5] Y.Wu, B. Liu, and M. Zhu, "A single-pair antenna microwave medical detection system based on unsupervised feature learning," International Conference on Computational Social Networks (CSoNet): Computational Data and Social Networks, vol. 11280, pp 404–414, 2018.

[6] I. Bravo et al., "Novel HW architecture based on FPGAs oriented to solve the eigen problems," IEEE Trans. VLSI Systems, vol. 16, no. 12, pp. 1722–1725, 2008.

[7] M. Tian, M. Sima, and M. Mcguire, "Behavioral Implementation of SVD on FPGA," International Symposium on Signal Processing and Information Technology (ISSPIT), USA, pp. 495–500, 2018.

[8] M. Franceschi, A. Nannarelli and M. Valle, "Tunable floating-Point for embedded machine learning algorithms implementation," 15th International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), Prague, pp. 89–92, 2018.

[9] Luis M. Ledesma-Carrillo et al., "Reconfigurable FPGA-based unit for singular value decomposition of large $m \times n$ matrices," International Conference on Reconfigurable Computers & FPGAs, pp. 345–350, 2011.

[10] U. Martinez-Corral, K. Basterretxea and R. Finker, "Scalable parallel architecture for singular value decomposition of large matrices," 24th International Conference on Field Programmable Logic and Applications (FPL), Munich, pp. 1–4, 2014.

[11] A. Ali, M. Siupik, A. Amira, F. Bensaali, and P. Higuera, "HLS based hardware acceleration on the Zynq SoC: a case study for fall detection system,", 11th International Conference on Computer Systems and Applications (AICCSA), pp. 685–690, 2014.

[12] D. Fernandez, C. Gonzalez, D. Mozos, and S. Lopez, "FPGA implementation of the principal component analysis algorithm for dimensionality reduction of hyperspectral images," Journal of Real-Time Image Processing, pp. 1–12, 2016.

[13] I. Bravo, C. Vazquez, A. Gardel, J. L. Lazaro, and E. Palomar, "High level synthesis FPGA implementation of the jacobi algorithm to solve the eigen problem," Mathematical Problems in Engineering, vol. 2015, Article ID 870569, 11 pages, 2015.

[14] M. A. Mansoori and M. R. Casu, "Efficient FPGA Implementation of PCA Algorithm for Large Data using High Level Synthesis," 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), pp. 65-68, Switzerland, 2019.

[15] M. Schellhorn and G. Notni, "Optimization of a principal component analysis implementation on Field-Programmable Gate Arrays (FPGA) for analysis of spectral images," Digital Image Computing: Techniques and Applications (DICTA), Canberra, Australia, pp. 1–6, 2018.