POLITECNICO DI TORINO
Repository ISTITUZIONALE

High level design of a flexible PCA hardware accelerator using a new block-streaming method

Original
High level design of a flexible PCA hardware accelerator using a new block-streaming method / Mansoori, M.A., Casu, M.R.. - In: ELECTRONICS. - ISSN 2079-9292. - ELETTRONICO. - 9:3(2020), pp. 449-471. [10.3390/electronics9030449]

Availability:
This version is available at: 11583/2804332 since: 2020-03-18T17:37:32Z

Publisher:
MDPI AG

Published
DOI:10.3390/electronics9030449

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

13 September 2026

electronics

MDPI

Article

High Level Design of a Flexible PCA Hardware Accelerator Using a New Block-Streaming Method

Mohammad Amir Mansoori * and Mario R. Casu

Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy; mario.casu@polito.it

* Correspondence: mohammadamir.mansoori@polito.it

Received: 3 February 2020; Accepted: 4 March 2020; Published: 7 March 2020

Abstract: Principal Component Analysis (PCA) is a technique for dimensionality reduction that is useful in removing redundant information in data for various applications such as Microwave Imaging (MI) and Hyperspectral Imaging (HI). The computational complexity of PCA has made the hardware acceleration of PCA an active research topic in recent years. Although the hardware design flow can be optimized using High Level Synthesis (HLS) tools, efficient high-performance solutions for complex embedded systems still require careful design. In this paper we propose a flexible PCA hardware accelerator in Field-Programmable Gate Arrays (FPGA) that we designed entirely in HLS. In order to make the internal PCA computations more efficient, a new block-streaming method is also introduced. Several HLS optimization strategies are adopted to create an efficient hardware. The flexibility of our design allows us to use it for different FPGA targets, with flexible input data dimensions, and it also lets us easily switch from a more accurate floating-point implementation to a higher speed fixed-point solution. The results show the efficiency of our design compared to state-of-the-art implementations on GPUs, many-core CPUs, and other FPGA approaches in terms of resource usage, execution time and power consumption.

Keywords: FPGA; Principal Component Analysis (PCA); High Level Synthesis (HLS); hardware acceleration; embedded systems; fixed-point implementation

1. Introduction

Principal Component Analysis (PCA) is a widely-used method for reducing dimensionality. It extracts from a set of observations the Principal Components (PCs) that correspond to the maximum variations in the data. By projecting the data on an orthogonal basis of vectors corresponding to the first few PCs obtained with the analysis and removing the other PCs, the data dimensions can be reduced without a significant loss of information. Therefore, PCA can be used in various applications when there is redundant information in the data. For example, in Microwave Imaging (MI), PCA is useful before image reconstruction to reduce data dimensions in microwave measurements [1–3]. In a recent work [4], PCA is used as a feature extraction step prior to tumor classification in MI-based breast cancer detection.

PCA involves various computing steps consisting of complex arithmetic operations, which result in a high computational cost and so a high execution time when implementing PCA in software. To tackle this problem, hardware acceleration is often used as an effective solution that helps reduce the total execution time and enhance the overall performance of PCA.

The main computing steps of PCA, which will be described thoroughly in the next section, include the computation of an often large covariance matrix, which stresses the I/O of hardware systems, and the computation of the singular values of the matrix. Since the covariance matrix is symmetric, the singular values are also the eigenvalues and can be computed either with Eigenvalue

Decomposition (EVD) or with Singular Value Decomposition (SVD): the more appropriate algorithm to implement in hardware is chosen depending on the application and the performance requirements. Other important steps in PCA are the data normalization, which requires to compute the data mean, and the projection of the data on the selected PCs.

In recent years, numerous hardware accelerators have been proposed that implement either PCA in its entirety or some of its building blocks. For example, in [5] different hardware implementations of EVD were compared and analyzed on CPU, GPU, and Field-Programmable Gate Arrays (FPGA), and it was shown that FPGA implementations offer the best computational performance, while the GPU ones require less design effort. A new FPGA architecture for EVD computation of polynomial matrices was presented in [6], in which the authors show how the Xilinx System Generator tool can be used to increase the design efficiency compared to traditional RTL manual coding. Leveraging a higher abstraction level to improve the design efficiency is also our goal, which we pursue using the High-Level Synthesis (HLS) design approach, as we discuss later, as opposed to VHDL- or Verilog-based RTL coding. Wang and Zambreno [7] introduce a floating-point FPGA design of SVD based on the Hestenes-Jacobi algorithm. Other hardware accelerators for EVD and SVD were proposed in [8–11].

In [12] an embedded hardware was designed in FPGA using VHDL for the computation of Mean and Covariance matrices as two components of PCA. Fernandez et al. [13] presented a manual RTL design of PCA for Hyperspectral Imaging (HI) in a Virtex7 FPGA. The Covariance and Mean computations could not be implemented in hardware due to the high resource utilization. Das et al. [14] designed an FPGA implementation of PCA in a network intrusion detection system, in which the training phase (i.e., computing the PCs) was done offline and only the mapping phase (i.e., the projection of the data on the PC base) in the online section was accelerated in hardware. Our goal is instead to provide a complete PCA implementation, which can be easily adapted to the available FPGA resources thanks to the design flexibility enabled by the HLS approach.

Recently, some FPGA accelerators have been introduced that managed to implement a complete PCA algorithm. In [15] such an accelerator was designed in a Virtex7 FPGA using VHDL, but it is applicable only to relatively small matrix dimensions. Two block memories were used for the internal matrix multiplication to store the rows and columns of the matrices involved in the multiplication, which resulted in a high resource usage. Thanks to our design approach, instead, we are able to implement a complete PCA accelerator for large matrices even with few FPGA resources.

FPGAs are not the only possible target for PCA acceleration. In [16], all the PCA components were implemented on two different hardware platforms, a GPU and a Massively Parallel Processing Array (MPPA). Hyperspectral images with different dimensions were used as test inputs to evaluate the hardware performance. It is well known, however, that these kinds of hardware accelerators are not as energy-efficient as FPGAs. Therefore we do not consider them, especially because our target computing devices are embedded systems in which FPGAs can provide an efficient way for hardware acceleration.

To design a hardware accelerator in FPGA, the traditional approach uses Hardware Description Languages (HDL) like VHDL or Verilog. Although this approach is still the predominant design methodology, it increases the development time and the design effort. As embedded systems are becoming more complex, designing an efficient hardware in RTL requires significant effort, which makes it difficult to find the best hardware architecture. An alternative solution that is becoming more popular in recent years is the High Level Synthesis (HLS) approach. HLS raises the design abstraction level by using software programming languages like C or C++. Through the processes of scheduling, binding, and allocation, HLS converts a software code into its corresponding RTL description. The main advantage of HLS over HDL is that it enables designers to explore the design space more quickly, thus reducing the total development time with a quality of results comparable and often better than RTL design. Another important advantage is the flexibility, which we use in

this work and which allows designers to use a single HLS code for different hardware targets with different resource and performance constraints.

Recently, HLS-based accelerators have been proposed for PCA. In [17], a design based on HLS was introduced for a gas identification system and implemented on a Zynq SoC. Schellhorn et al., presented in [18] another PCA implementation on FPGA using HLS for the application of spectral image analysis, in which the EVD part could not be implemented in hardware due to the limited resources. In [19], we presented an FPGA accelerator by using HLS to design the SVD and projection building blocks of PCA. Although it could be used for relatively large matrix dimensions, the other resource-demanding building blocks (especially covariance computation) were not included in that design. In a preliminary version of this work [20], we proposed another HLS-based PCA accelerator to be used with flexible data dimensions and precision, but limited in terms of hardware target to a low-cost Zynq SoC and without the support for block-streaming needed to handle large data matrices and covariance matrices, which instead we show in this paper.

The PCA hardware accelerators proposed in the literature have some drawbacks. The manual RTL approach used to design the majority of them is one of the disadvantages, which leads to an increase in the total development time. Most of the previous works, including some of the HLS-based designs, could not implement an entire PCA algorithm including all the computational units. Other implementations could not offer a flexible and efficient design with a high computational performance that could be used for different data sizes.

In this paper we close the gap in the state of the art and propose an efficient FPGA hardware accelerator that has the following characteristics:

- The PCA algorithm is implemented in FPGA in its entirety.
- It uses a new block-streaming method for the internal covariance computation.
- It is flexible because it is entirely designed in HLS and can be used for different input sizes and FPGA targets.
- It can easily switch between floating-point and fixed-point implementation, again thanks to the HLS approach.
- It can be easily deployed on various FPGA-based boards, which we prove by targeting both a Zynq7000 and a Virtex7 in their respective development boards.

The rest of the paper is organized as follows. At first, in Section 2 the PCA algorithm is described with the details of its processing units. We briefly describe in Section 3 the application of PCA to Hyperspectral Imaging (HI), which we use as a test case to report our experimental results and to compare them with previously reported results. The proposed PCA hardware accelerator and the block-streaming method is presented in Section 4 together with the HLS optimization techniques. The implementation results and the comparisons with other works are reported in Section 5. Finally, the conclusions are drawn in Section 6.

2. PCA Algorithm Description

Let X be an array of size $R \times C$ in which R (*Rows*) is the number of data samples and C (*Columns*) is the main dimension in which there is redundant information. PCA receives X and produces a lower-dimensionality array Y of size $R \times L$ with $L < C$ through the steps shown in Algorithm 1.

In the first step, the mean values of each column of the input data are computed and stored in matrix M for data normalization. The second step is the computation of the covariance of the normalized data, which is one of the most computationally expensive steps of PCA due to the large number of multiplications and additions. After computing the eigenvalues or singular values (and the corresponding eigen/singular vectors) of the covariance matrix by using EVD or SVD in the third step, they are sorted in descending order and the first L eigen/singular vectors are selected as

Principal Components in the fourth step. The selection of PCs is based on the cumulative energy of eigen/singular values. After computing the total energy (E) as in the following equation,

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

where σ_i is the energy of the ith eigen/singular value, 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 (%), as follows:

$$100 \times \frac{\sum_{i=1}^{L} \sigma_i}{E} \geq T. \tag{2}$$

Finally, in the last step, the normalized input is projected into the principal components space to reduce the redundant information.

Algorithm 1: PCA algorithm

Step 1- Mean computation: /* $M_{R\times C}$ is the matrix representation of vector $Mean_{1\times C}$ */
$[Mean]_{1\times C} = \frac{1}{R}\sum_{i=1}^{R}[X_i]_{1\times C}$ /* $[X]_i$ is the ith row of the input matrix $X_{R\times C}$ */
$M_{R\times C} : [M_i]_{1\times C} = [Mean]_{1\times C}, i = 1, 2, ..., R$ /* $[M_i]$ is the ith row of matrix M*/
Step 2- Covariance calculation:
$[COV]_{C\times C} = \frac{1}{R-1}(X - M)^T \times (X - M)$
Step 3- EVD/SVD of covariance:
$COV = U\Sigma U^T$
Step 4- Sort and selection:
$\Sigma^s, U^s = Sort(\Sigma, U)$
$[PC]_{C\times L} = Select(\Sigma^s, U^s)$
Step 5- Projection:
$[Y]_{R\times L} = (X - M) \times PC$

3. Hyperspectral Imaging

Although we are interested in the application of PCA to Microwave Imaging (MI), to the best of our knowledge there is no hardware accelerator for PCA in the literature that is specifically aimed at such application. In order to compare our proposed hardware with state-of-the-art PCA accelerators, we had to select another application for which an RTL- or HLS-based hardware design was available. Therefore, we selected the Hyperspectral Imaging (HI) application.

Hyperspectral Imaging (HI) sensors acquire digital images in several narrow spectral bands. This enables the construction of a continuous radiance spectrum for every pixel in the scene. Thus, HI data exploitation helps to remotely identify the ground materials-of-interest based on their spectral properties [21].

HI data are organized in a matrix of size $R \times C$ in which R is the number of pixels in the image and C is the number of spectral bands. Usually, there is redundant information in different bands that can be removed with PCA. In the following notations, we use interchangeably the terms $R(Rows)$ and pixels, as well as $C(Columns)$ and bands.

For MI, instead, the $R \times C$ matrix could represent data gathered in C different frequencies in the microwave spectrum by R antennas, or a reconstructed image of the scattered electromagnetic field of R pixels also at C frequencies.

4. PCA Hardware Accelerator Design

Figure 1 shows the architecture of the PCA accelerator and its main components. Since the accelerator is developed in HLS using C++, the overall architecture correspond to a C++ function and its components correspond to subfunctions. At the highest level, there are two subfunctions named

Dispatcher and *PCA core*. The Dispatcher reads the input data stored in an external DDR memory and sends them to the PCA core through the connecting FIFOs. The connection with FIFOs is also described in HLS with proper code pragmas.

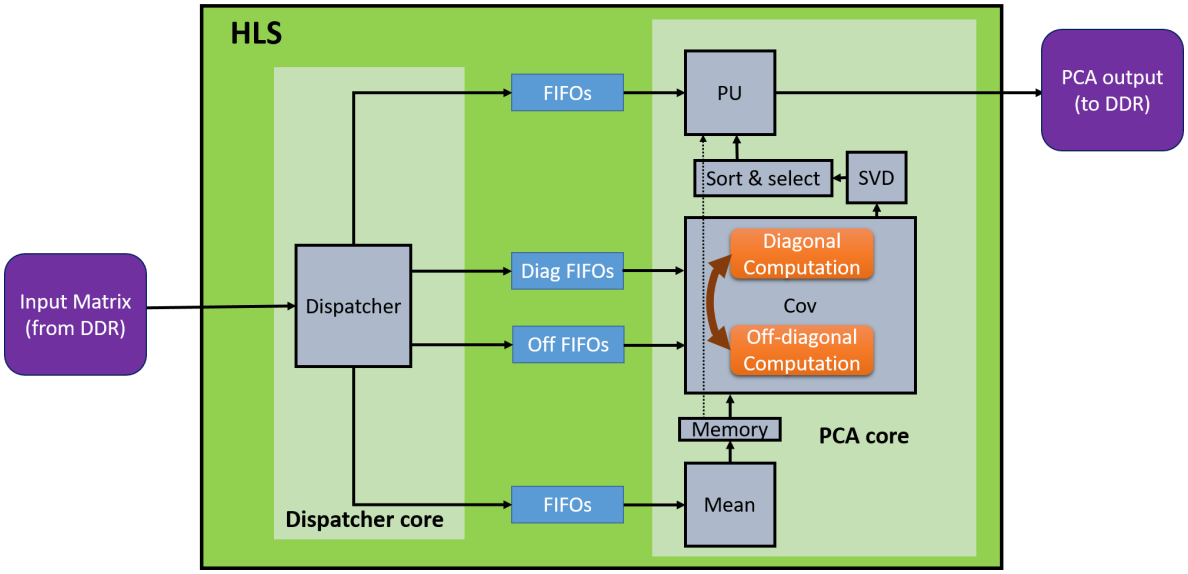

Figure 1. Architecture of the proposed hardware accelerator for Principal Component Analysis (PCA) in Field-Programmable Gate Arrays (FPGA).

The PCA core contains different processing units. The first is the *Mean* unit, which computes the mean vector corresponding to the mean value of each column of the input matrix. This vector is stored in an internal memory that is used by the next processing units, *Cov* and *PU*, for data centering. The Cov unit uses the new block-streaming method for computing the covariance matrix, which will be explained thoroughly in the following subsection. It reads the required data from two sets of FIFOs corresponding to diagonal and off-diagonal computation. Then the *SVD* unit computes the singular values of the covariance matrix, and the *Sort and Select* unit sorts them and retains the first components. Finally, the *Projection* unit reads the input data again and computes the multiplication between the centered data and the sorted and selected PCs to produce the final PCA output data, which are written back to the external DDR memory.

The computational cost of PCA depends on the input dimensions. When the main dimension from which the redundant information must be removed (columns or bands in HI) is lower than the other dimension (rows or pixels in HI), the PCA performance is mainly limited by the computation of the covariance matrix, due to the large number of multiplications and additions that are proportional to the number of pixels. Indeed, in the covariance computation all the pixels in one band must be multiplied and accumulated with the corresponding pixels in all of the bands. This is illustrated in Figure 2 where the bands are specified by the letters a to n and pixels are indicated by the indices 1 to N. The result of the covariance computation, which is the output of the Cov unit, is a matrix of size ($bands \times bands$) that becomes the input of the SVD unit. In HI applications, in which it is true that $pixels \gg bands$, the covariance computation is the major limitation of the whole PCA design, hence its acceleration can drastically enhance the overall performance.

Multiple parallel streaming FIFOs are needed to match the parallelism of the accelerated Cov unit. The number of FIFOs is determined based on the input data dimensions and the maximum bandwidth of the external memory. Streaming more data at the same time through the FIFOs enables a degree of parallelism that is matched to the number of columns of the input matrix. It is important to note that all of the hardware components are described in a single HLS top-level function, which simplifies the addition of different flexibility options to the design such as variable data dimensions, block sizes, and number of FIFOs.

The complexity of the Cov unit compared to other parts raises the importance of the block-streaming method in covariance computation as this method allows using the same design for higher data dimensions or improving the efficiency in low-cost embedded systems with fewer hardware resources.

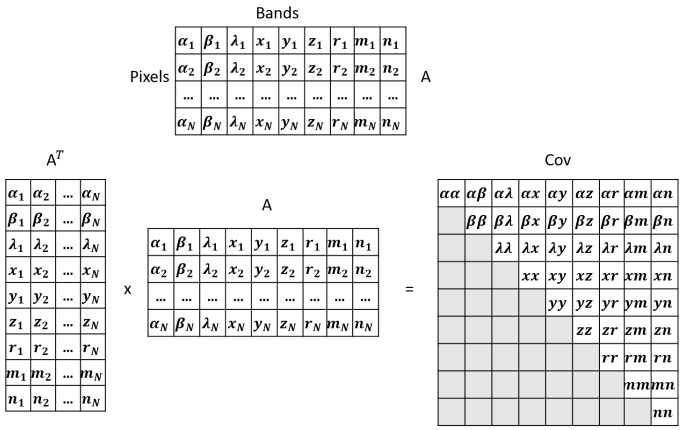

Figure 2. Example of covariance computation with 9 bands and N pixels, $PQ = \Sigma_{i=1}^{N} P_i \times Q_i$, where P, Q are the symbols of bands (α to n).

4.1. Block-Streaming for Covariance Computation

The block-streaming method is helpful whenever there is a limitation in the maximum size of input data that can be stored in an internal memory in the Cov unit. Therefore, instead of streaming the whole data one time, we stream "blocks" of data several times through the connecting FIFOs. There are two internal memories inside the Cov unit each of which can store a maximum number of bands (B_{max}) for each pixel. These memories are used in the diagonal and off-diagonal computations, so we call them "Diag" and "Off-diag" RAMs, respectively. The input data is partitioned into several blocks along the main dimension (bands) with a maximum dimension of B_{max} (block size). Each block of data is streamed through the two sets of FIFOs located between the Dispatcher and Cov unit (Diag and Off-diag FIFOs) in a specific order, and after the partial calculation of all the elements of the covariance matrix for one pixel, the data blocks for the next pixels will be streamed and the partial results accumulated together to obtain the final covariance matrix.

To better understand the block-streaming method, we provide two examples in Figures 3–6.

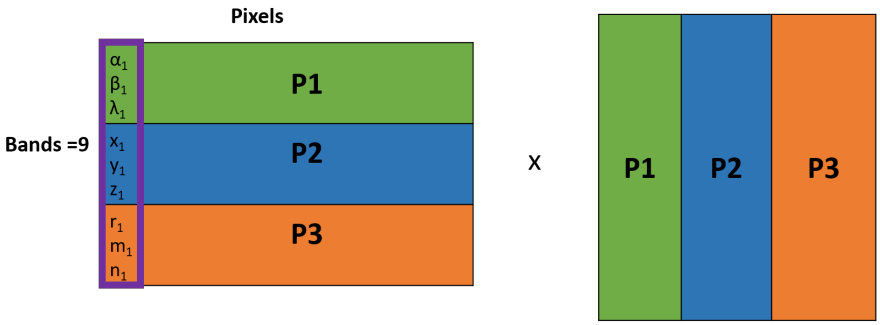

Figure 3. Example of partitioning of input data into blocks. The total number of bands is $B = 9$ and the block size is $B_{max} = 3$.

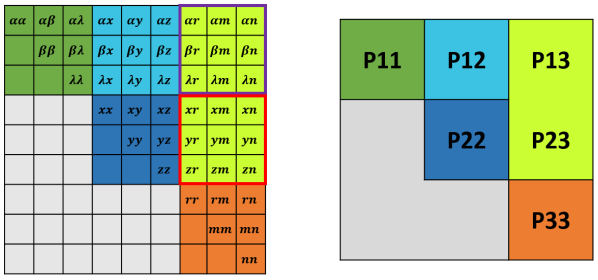

Figure 4. Illustration of an example of covariance computation using the block-streaming method with 3 blocks ($B = 9$, $B_{max} = 3$).

The first example is illustrated in Figure 3 in which the total number of bands is $B = 9$ and the block size is $B_{max} = 3$. Therefore, we have 3 blocks of data that are specified in figure as P1 to P3. The Block-streaming method consists of the following steps that can be realized from Figure 4:

1. **Diagonal computation:**

   The 3 blocks of data (P1 to P3) for the first pixel are streamed in Diag FIFOs one by one and after storage in the Diag RAM, the diagonal elements P11, P22, and P33 are computed.
2. **Off-diagonal computation of the last block:**

   (a) Keep the last block (P3) in the Diag RAM.
   (b) Stream the first block (P1) into Off-Diag FIFOs, store it in Off-Diag RAM, and compute $P13 = P1 \times P3$.
   (c) Stream the second block (P2) into Off-Diag FIFOs, store it in Off-Diag RAM, and compute $P23 = P2 \times P3$.
3. **Off-diagonal computation of the preceding blocks:**

   (a) Update the Diag RAM by the last values of Off-Diag RAM (P2).
   (b) Stream the first block (P1) into Off-Diag FIFOs, store it in Off-Diag RAM, and compute $P12 = P1 \times P2$.
4. **Stream Pixels:** Steps 1 to 3 are repeated for the next pixels and the results are accumulated to obtain the final covariance matrix.

The second example is illustrated in Figure 5 in which the number of blocks is 4 and after the diagonal computation (in green color) there are 3 steps for off-diagonal computations that are indicated in 3 different colors. Figure 6 shows the order of data storage in the Diag and Off-diag RAMs. After the 7th and 9th steps, the Diag RAM is updated by the last value of Off-Diag RAM (P3 and P2).

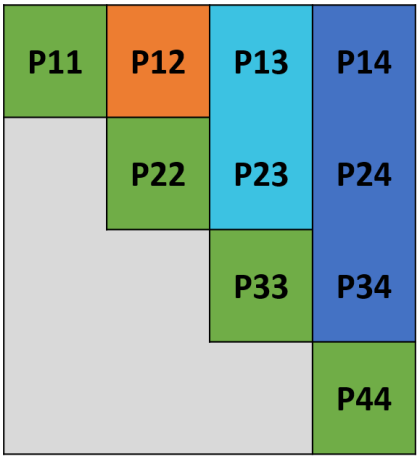

Figure 5. Block-streaming method with 4 blocks ($B/B_{max} = 4$).

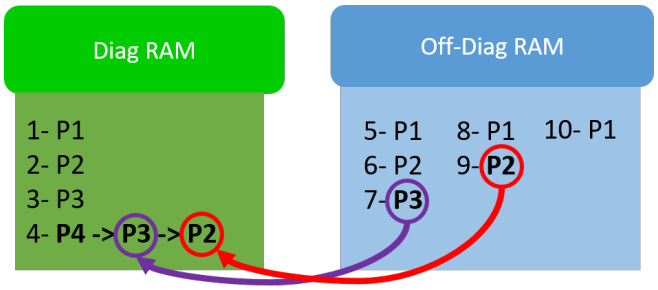

Figure 6. Order of data storage in the Diagonal and Off-diagonal RAMs inside the Cov unit.

4.2. High Level Synthesis Optimizations

4.2.1. Tool Independent Optimization Directives and Their Hardware Implementation

To introduce the principles behind the hardware optimization techniques used in the proposed design, a more abstract description of the high-level directives is presented in this part without references to a specific hardware design tool. In Figure 7 the most widely used optimization directives are illustrated with their corresponding hardware implementation. These directives are *Loop Pipelining*, *Loop Unrolling*, *Array Partitioning*, and *Dataflow*.

Loop Pipelining allows multiple operations of a loop to be executed concurrently by using the same hardware iteratively. Loop Unrolling creates multiple instances of the hardware for the loop body, which allows some or all of the loop iterations to occur in parallel. By using Array Partitioning we can split an array, which is implemented in RTL by default as a single block RAM resource, into multiple smaller arrays that are mapped to multiple block RAMs. This increases the number of memory ports providing more bandwidth to access data. Finally, the Dataflow directive allows multiple functions or loops to operate concurrently. This is achieved by creating channels (FIFOs or Ping-Pong buffers) in the design, which enables the operations in a function or loop to start before the previous function or loop completes all of its operations. Dataflow directive is mainly used to improve the overall latency and throughput of the design. Latency is the time required to produce the output after receiving the input. Throughput is the rate at which the outputs are produced (or the inputs are consumed) and is measured as the reciprocal of the time difference between two consecutive outputs (or inputs).

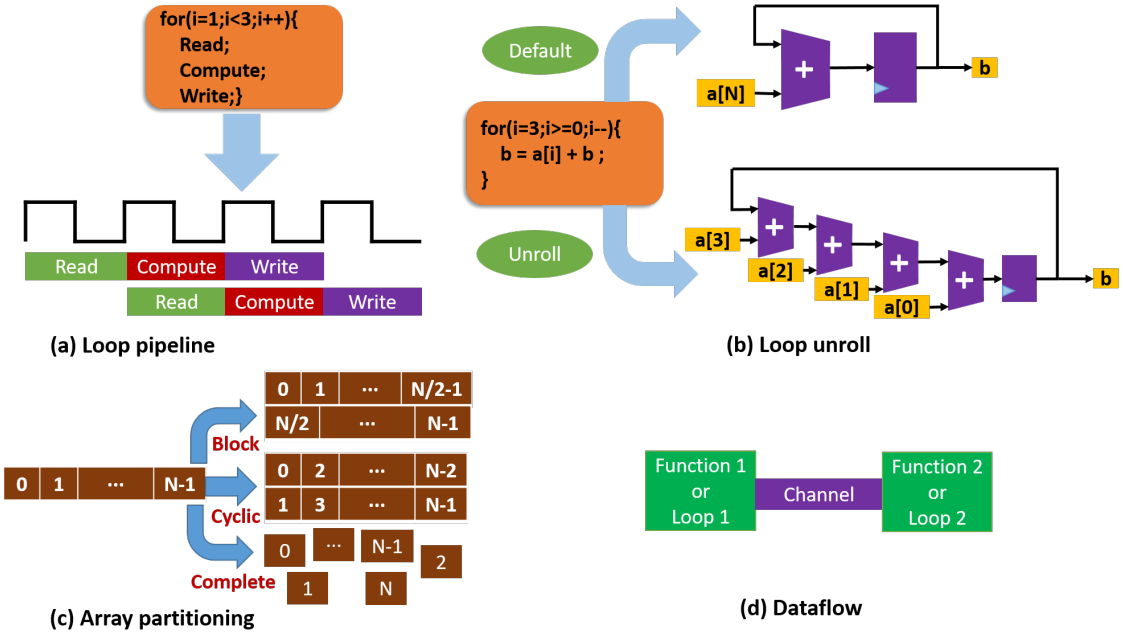

Figure 7. Hardware optimization directives.

The PCA hardware accelerator is designed in C++ using Vivado HLS, the HLS development tool for Xilinx FPGAs. The above-mentioned optimization directives can be applied easily in HLS by using their corresponding code *pragmas*. HLS enables us to specify the hardware interfaces as well as the level of parallelism and pipelined execution and specific hardware resource allocation thanks to the addition of code pragmas. By exploring different combinations of the optimization directives, it is possible to determine relatively easily the best configuration in terms of latency and resource usage. Therefore, several interface and hardware optimization directives have been applied in the HLS synthesis tool, as explained below.

4.2.2. Input Interfaces

The main input interface associated to the Dispatcher input and the output interface associated to the PU output consist of AXI master ports, whose number and parallelism are adapted to the FPGA target. For the Zynq of the Zedboard, four AXI ports (with a fixed width of 64 bits) are connected to the Dispatcher input in such a way to fully utilize the available memory bandwidth. In the Virtex7 of the VC709 board we can use instead only one AXI port with a much larger bit-level parallelism. The output interface for both boards is one AXI master port that is responsible for writing the output to the memory. Other interfaces are specified as internal FIFOs between the Dispatcher and the PCA core. As shown in Figure 1, four sets of FIFOs send the streaming data (containing a data block of bands) from the Dispatcher to the corresponding processing units in the PCA core. Mean and Projection units receive two sets of FIFOs and Cov unit receives another two. Each set of FIFOs is partitioned by B_{max}, the size of a data block, so that there are B_{max} FIFOs in each set.

These FIFOs are automatically generated by applying the Vivado HLS *Dataflow* directive for the connection between the Dispatcher and the PCA core. This directive lets the two functions execute concurrently and their synchronization is made possible by the FIFO channels automatically inserted between them.

4.2.3. Code Description and Hardware Optimizations

In this part the code description for each PCA component is presented. To optimize the hardware of each PCA unit, we analyzed the impact of different HLS optimizations on the overall performance. Specifically, we considered the impact of loop pipelining, loop unrolling, and array partitioning on latency and resource consumption. The best HLS optimizations are selected in such a way that the overall latency is minimized by utilizing as many resources as required.

The Mean unit computes the mean values of all the pixels in each band. The code snippet for the Mean unit is presented in Algorithm 2 and consists of two loops on the rows and columns to accumulate the pixel values and another loop for the final division by the number of rows.

Algorithm 2: Mean computation

```
mean_row_loop:
 for r=0 to R do
   #pragma HLS PIPELINE
    mean_col_loop:
    for c=0 to C do
      a_mean[c] = Din_Mean[r][c];
      tmp_mean[c]+ = a_mean[c];
    end
end
Divide_loop:
 for c=0;c<C;c++ do
   a_mean[c] = tmp_mean[c]/R;
end
```

The best HLS optimization for the Mean unit is to *pipeline* the outer loop (line #pragma HLS PIPELINE in Algorithm 2), which reduces the Initiation Interval (II), i.e., the index of performance that corresponds to the minimum time interval (in clock cycles) between two consecutive executions of the loop (ideally, II = 1). In addition, the memory arrays *a_mean* and *tmp_mean* are partitioned by B_{max} (not shown in the code snippet) to have access to multiple memory locations at the same time, which is required for the loop pipelining to be effective, otherwise the II will increase due to the latency needed to access a single, non-partitioned memory.

The Cov unit uses the block-streaming method to compute the covariance matrix. Its pseudo code is presented in Algorithm 3. The HLS optimizations include loop pipelining, unrolling, and the arrays full partitioning. In Algorithm 3 full indexing is not displayed to make the code more readable and only the relation between the variables and the indexes is shown by the parentheses. For example, DiagFIFO(r,b) indicates that the indexes of variable DiagFIFO are proportional to (r,b). The standard Cov computation is adopted from [20] and is used for diagonal covariance computations. The write function in the last line writes the diagonal and off-diagonal elements of covariance matrix from variables CovDiag and CovOff to the corresponding locations in CovOut. As shown in Algorithm 3, there are two pipeline directives that are applied to the loops on the diagonal and off-diagonal blocks, respectively. The memory arrays need to be fully partitioned, which is required to unroll the inner loops. As described before, a thorough analysis of different possible optimizations was performed to find out a trade-off between resource usage and latency.

Algorithm 3: Cov computation, block-streaming

```
for r=0 to R do  /* Stream Pixels */
    for b=0 to NB do  /* Diagonal Computations, NB = B/Bmax */
        #pragma HLS PIPELINE
        DiagRAM = DiagFIFO(r,b) − a_mean(b);
        /* Start standard Cov computation [20] */
        for c1=0 to B_max do
            for c2=c1 to B_max do
                .../* indexing */
                CovDiag(b, Index) = DiagRAM[c1] ∗ DiagRAM[c2];
            end
        end
        /* Finish standard Cov computation */
    end
    for ct=1 to NB do  /* Off-Diagonal computations */
        for b=0 to NB-ct do
            #pragma HLS PIPELINE
            if Step3(a) then /*refer to Section 4.1, the four steps of block-streaming method*/
                DiagRAM = OffRAM;
            end
            OffRAM = OffFIFO(r,b) − a_mean(b);
            CovOff(b,ct)+ = OffRAM ∗ DiagRAM;
        end
    end
end
/* Write to the final Cov matrix */
CovOut = write(CovDiag, CovOff);
```

The next processing unit is the EVD of the covariance matrix. For real and symmetric matrices (like the covariance matrix) EVD is equal to SVD and both methods can be used. For SVD computation

of floating-point matrices, there is a built-in function in Vivado HLS that is efficient especially for large dimensions. On the other hand, a fixed-point implementation of EVD is highly beneficial for embedded systems due to the lower resource usage (or better latency) compared to the floating-point version of the same design.

For these reasons, in this paper we propose two versions of the PCA accelerator. The first one is the floating-point version, which uses the built-in HLS function for SVD computation. The second one is the fixed-point version, which uses the general two-sided Jacobi method for EVD computation [22,23]. The details of this method for computing EVD is shown in Algorithm 4. As we will show in the results section, there is no need to add any HLS optimization directives to the fixed-point EVD (for our application) because the overall latency of the PCA core, which includes EVD, is lower than the data transfer time, so there is no benefit in consuming any further resources to optimize the EVD hardware. We do not report the code of the floating-point version as it uses the built-in Vivado HLS function for SVD (Vivado Design Suite User Guide: High-Level Synthesis, UG902 (v2019.1), Xilinx, San Jose, CA, 2019 https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf).

Algorithm 4: EVD computation, Two-sided Jacobi method

/*Initialize the eigenvector matrix V and the maximum iterations $max = bands$*/
$V = I$; /* I is the identity matrix */
for *l=1 to max* **do**
  for *all pairs i<j* **do**
    /* Compute the Jacobi rotation which diagonalizes $\begin{bmatrix} H_{ii} & H_{ij} \\ H_{ji} & H_{jj} \end{bmatrix} = \begin{bmatrix} a & c \\ c & b \end{bmatrix}$ */
    $\tau = (b-a)/(2*c)$; $t = sign(\tau)/(|\tau| + \sqrt{1+\tau^2})$;
    $cs = 1/\sqrt{1+t^2}$;$sn = cs*t$;
    /* update the 2×2 submatrix */
    $H_{ii} = a - c*t$;
    $H_{jj} = b + c*t$;
    $H_{ij} = H_{ji} = 0$;
    /* update the rest of rows and columns i and j */
    for *k=1 to bands except i and j* **do**
      $tmp = H_{ik}$;
      $H_{ik} = cs*tmp - sn*H_{jk}$;
      $H_{jk} = sn*tmp + cs*H_{jk}$;
      $H_{ki} = H_{ik}$; $H_{kj} = H_{jk}$;
    end
    /* update the eigenvector matrix V /*
    for *k=1 to bands* **do**
      $tmp = V_{ki}$;
      $V_{ki} = cs*tmp - sn*V_{kj}$;
      $V_{kj} = sn*tmp + cs*V_{kj}$;
    end
  end
end

The last processing unit is the Projection Unit (PU), which computes the multiplication between the centered data and the principal components. Algorithm 5 presents the code snippet for the PU. Similar to the Mean unit, we applied some optimizations to this code. The second loop is pipelined and, as a consequence, all the inner loops are unrolled. In addition, the memory arrays involved in the

multiplication must be partitioned. For more information on the hardware optimizations for Mean and PU and their impact on the latency and resource usage please refer to [20].

Algorithm 5: Projection computation

```
for r=0 to R do
    for c1=0 to L do
        #pragma HLS PIPELINE
         tmp = 0;
        for n=0 to C do
            .../* Index control */
            Din_Nrml[n] = Din_PU[r][n] − a_mean[n];
        end
        for c2=0 to C do
            tmp+ = (Din_Nrml[c2] ∗ PC[c2][c1];
        end
        Data_Transformed[r][c1] = tmp;
    end
end
```

4.3. Fixed-Point Design of the Accelerator

There are many considerations when selecting the best numerical representation (floating- or fixed- point) in digital hardware design. Floating-point arithmetic is more suited for applications requiring high accuracy and high dynamic range. Fixed-point arithmetic is more suited for low power embedded systems with higher computational speed and fewer hardware resources. In some applications, we need not only speed, but also high accuracy. To fulfill these requirements, we can use a fixed-point design with a larger bit-width. This increases the resource usage to obtain a higher accuracy, but results in a higher speed thanks to the low-latency fixed-point operations. Therefore, depending on the requirements, it is possible to select either a high-accuracy low-speed floating-point, a low-accuracy high-speed fixed-point, or a middle-accuracy high-speed fixed-point design. Available resources in the target hardware determine which design or data representation is implementable on the device. In high-accuracy high-speed applications, we can use the fixed-point design with a high resource usage (even more than the floating-point) to minimize the execution time.

To design the fixed-point version of the PCA accelerator, the computations in all of the processing units must be in fixed-point. The total Word Length (WL) and Integer Word Length (IWL) must be determined for every variable. The range of input data and the data dimensions affect the word lengths in fixed-point variables, so the fixed-point design may change depending on the data set.

For our HI data set with 12 bands, we used the MATLAB fixed-point converter to optimize the word lengths. In HLS we selected the closest word lengths to the MATLAB ones because some HLS functions do not accept all the fixed-point representations (for example in the fixed-point square root function, IWL must be lower than WL).

The performance of EVD/SVD depends only on the number of bands. As we will see in the next section, the latency of our EVD design in HLS is significantly higher than the HLS built-in SVD function for floating-point inputs. One possible countermeasure is to use the SVD as the only floating-point component in a fixed-point design to obtain better latency, by adding proper interfaces for data-type conversion. However, when the data transfer time (i.e., the Dispatcher latency) is higher than the PCA core latency, the fixed-point EVD is more efficient because of the lower resource usage, which is the case for a small number of bands.

4.4. Hardware Prototype for PCA Accelerator Assessment with the HI Data Set

The proposed hardware design is adaptable to different FPGA targets and its performance will be evaluated in the results section in particular for two test hardware devices. In this subsection, as an example of system-level implementation using our flexible accelerator we introduce its implementation on a low-cost Zynq SoC mounted on the Zedboard development board. We used this implementation to evaluate our PCA accelerator on the HI data set. The details are illustrated in Figure 8. The input data set is stored in an SD card. The Zynq Processing System (PS) reads the input data from the SD card and writes them to the DDR3 memory. The Zynq FPGA reads the data from the DDR3 memory by using two High Performance (HP) ports (Zynq SoC devices internally provide four HP interface ports that connect the Programmable Logic (PL) to the Processing System (PS)). As the HI data consist of images in 12 bands and each pixel has an 8-bit data width, to match the processing parallelism we need an I/O parallelism of $12 \times 8 = 96$ bits to read all the bands at once. Therefore, we use two 64-bit HP ports for the input interface. After the PCA computation in the accelerator, the output is written back to the DDR3 memory through the first HP port by using an *AXI Smart Connect* IP block (AXI smart connect IP block connects one or more AXI memory-mapped master devices to one or more memory-mapped slave devices). Finally, the Zynq PS reads the PCA outputs from the DDR3 and writes them to the SD card.

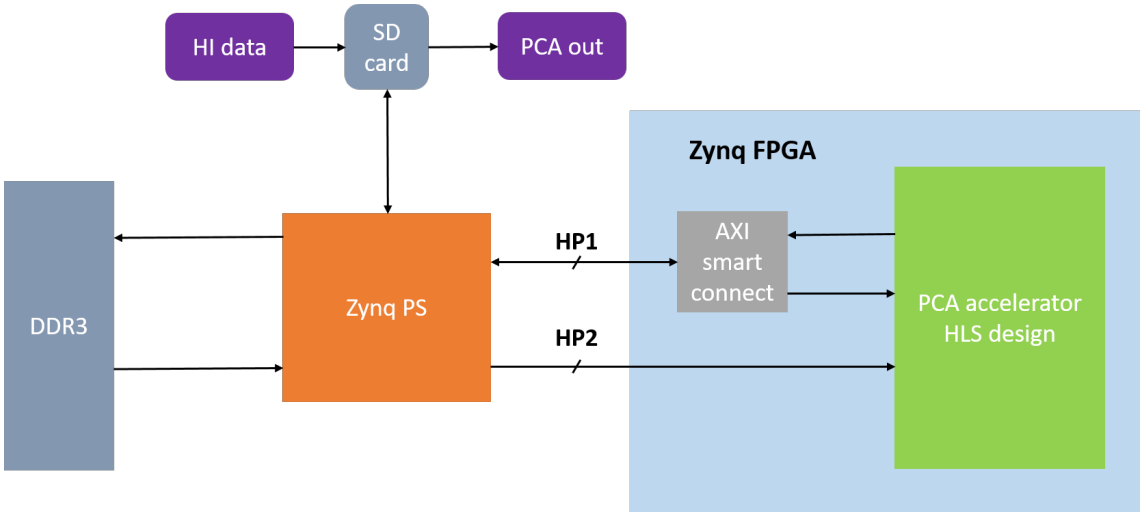

Figure 8. PCA accelerator design in Zedboard.

For other applications or different FPGA targets, the connection of the PCA accelerator to the I/O system or to a processor is easily adapted thanks to the flexibility enabled by the HLS approach. It is important to note that in many designs the hardware might be so fast that its performance becomes limited by the memory speed. To avoid this problem, it is necessary that the consumption rate of the hardware that fetches the data from the DDR memory is matched to the speed of that memory. In particular, in our design, by considering the maximum bandwidth B_{DDR} (Gb/s) of the external DDR memory and the clock frequency F (GHz) of the Dispatcher unit, we can obtain the maximum bit-width as $bw_{max} = B_{DDR}/F$ for the PCA accelerator input connected to the Dispatcher input. Depending on the input data set and the number of bands, we can obtain the maximum required I/O parallelism. For example, if the number of bands is B and the data width of each pixel is DW bits, we need a maximum of $bw = B \times DW$ to read one pixel for all the bands at once in one clock cycle. The final input bit-width of the Dispatcher (bw_{Disp}) is selected in such a way that we do not exceed the memory bandwidth. Therefore, if $bw \leq bw_{max}$, then $bw_{Disp} = bw$. Otherwise, we have to fix the Dispatcher input bit-width to $bw_{Disp} = bw_{max}$ (note that the Dispatcher input is an AXI port and its data width must be a power of 2 and a multiple of 8. In addition, some FPGA targets, like the Zedboard, can read data from memory using separate AXI ports (HP ports)). It should be noted that

all the above-mentioned conditions can be easily described in HLS using a set of variables and C++ macros that are set at the design time. In order to map the design into a new FPGA target, the only required change is to adjust the pre-defined variables based on the hardware device.

5. Results

The proposed PCA accelerator is implemented using Vivado HLS 2019.1. To demonstrate the flexibility of the proposed method, we did the experiments on two Xilinx FPGA devices and their development boards, the relatively small Zynq7000 (XC7z020clg484-1) on a Zedboard and the large Virtex7 (XC7vx690tffg1761-2) on a VC709 board. The results are evaluated for different numbers of bands, blocks and pixels. In addition, for the smaller FPGA with limited resources, we report a comparison between the fixed-point and the floating-point versions of the accelerator. Finally, the HI data set is used to evaluate the performance of the PCA accelerator in the Zynq device. Accuracy, execution time, and power consumption are also measured for both floating- and fixed-point design. Note that we define the execution time or latency as the period of time between reading the first input data from the external memory by the Dispatcher and writing the last PCA output to the memory by the Projection unit.

In the following subsections the impact of input dimensions (bands and pixels), size of the blocks (B_{max} in the block-streaming method), and data format (floating- or fixed-point) on the resource usage and latency is evaluated for the two hardware devices.

5.1. Number of Blocks, Bands, and Pixels

To show the efficiency of the trade-off between latency and resources enabled by the block-streaming method, different numbers of bands and blocks are considered. In the first experiment with the floating-point version on the Virtex7, we consider the total number of bands set at 48 and the size of the block (B_{max}) as a parameter that changes from 4 to 16. Figure 9 shows that, as expected, by using a larger block the total latency decreases in exchange for an increase in the resource usage. The latency for different parts of the accelerator is shown with different colors. The most time-consuming parts are Cov and Dispatcher (Dispatcher latency is the time for data transfer through the FIFOs). By increasing the block size (when $B_{max} = 16$) we can reduce the latency of Cov computation, so that the only limitation becomes the Dispatcher latency. It should be noted that the PCA core and the Dispatcher work concurrently, which reduces the overall latency of the entire design.

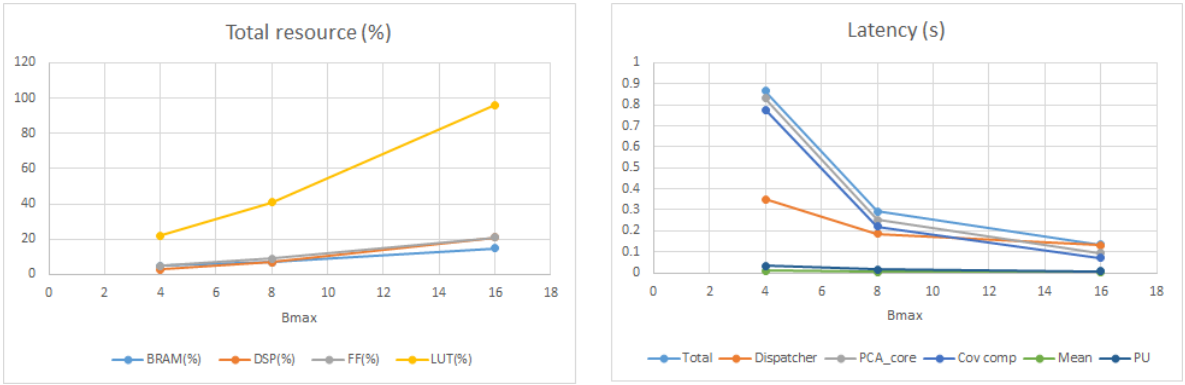

Figure 9. Impact of block size (B_{max}) on the resource usage and latency for the Virtex7, bands = 48, pixels = 300 × 300, floating-point design.

Increasing the number of pixels changes the latency of the design as the time to stream the whole data increases. Figure 10 shows the latency of different parts of the design when changing the total number of pixels. The resource consumption remains constant and does not change with the number

of pixels. As expected, the latency of SVD is also constant because it depends on the number of bands, not on the number of pixels. For the other parts, the latency increases almost proportionally.

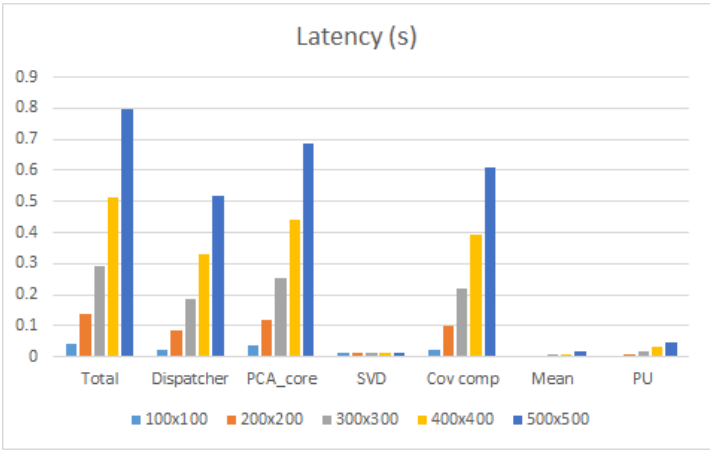

Figure 10. Impact of the number of pixels on the latency for Virtex7, bands = 48, $B_{max} = 8$, floating-point design.

In the next experiment the block size is fixed to $B_{max} = 8$ and the total number of bands is variable. The resource usage in the Virtex7 for the floating-point version of the PCA core without the SVD part (PCA-SVD), and the latency for different bands with a fixed B_{max} are shown in Figure 11. The number of pixels in this case is 300×300. The number of bands has a direct impact on the generated SVD hardware, so the resource usage of SVD unit is excluded from the total resources to obtain a better estimate of the performance of the block-streaming method.

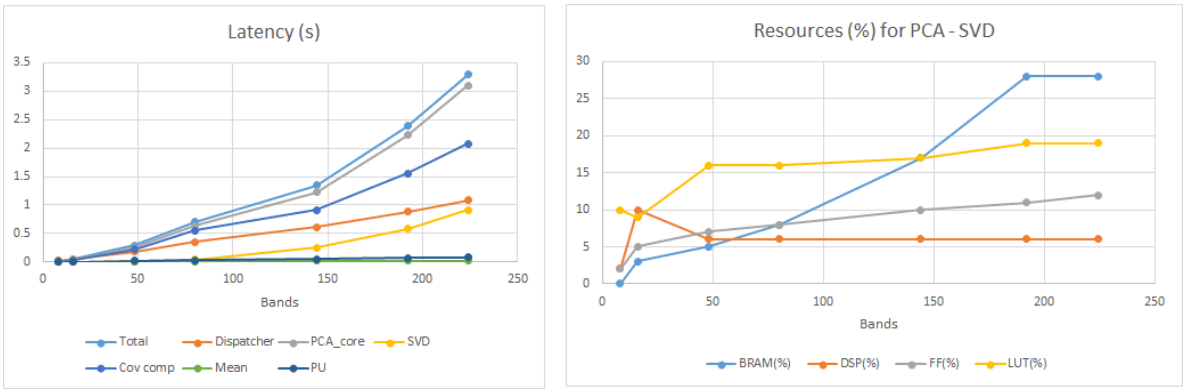

Figure 11. Latency and resource usage for Virtex7 with a fixed block size ($B_{max} = 8$), floating-point design.

As shown in Figure 11, the latency increases with the number of bands because the computational time depends on the main dimension. From the resource usage, it is evident that the FFs, DSPs, and LUTs are almost constant (except for a slight increase due to other components of PCA core like Mean and PU). The number of BRAMs, however, increases because in the HLS design there are other two memory arrays in addition to Diagonal and Off-diagonal RAMs to store the temporary values of the computations (CovDiag and CovOff in Algorithm 3) and the dimensions of these arrays depend on the ratio between the total bands and B_{max}. Still, up to a very large number of bands, the total resource usage of the most critical component is well below 30%.

5.2. Fixed-Point and Floating-Point Comparison

The fixed-point design of the PCA accelerator is evaluated on the Zynq7000 for different numbers of bands and blocks and is compared with the floating-point design. We first obtained the word

lengths using the MATLAB fixed-point converter and then used the nearest possible word lengths in the HLS design.

The total resource usage for the fixed- and floating-point design is shown in Figure 12 for a fixed number of bands ($B = 12$). In the floating-point design (histogram on the left side of Figure 12), the maximum block size is $B_{max} = 3$ because the LUTs required for larger block values exceed the LUTs available. For the fixed-point design (histogram on the right side of Figure 12), however, the block size can be up to 4. The comparison of the floating- and fixed-point designs for the same block size ($B_{max} = 3$) shows that there is a reduction in the resource usage for the fixed-point design except for the DSP usage. This is because to obtain a similar accuracy the fixed-point representation requires a larger bit-width. As a consequence, the HLS design requires more DSPs to implement the same operations in fixed-point.

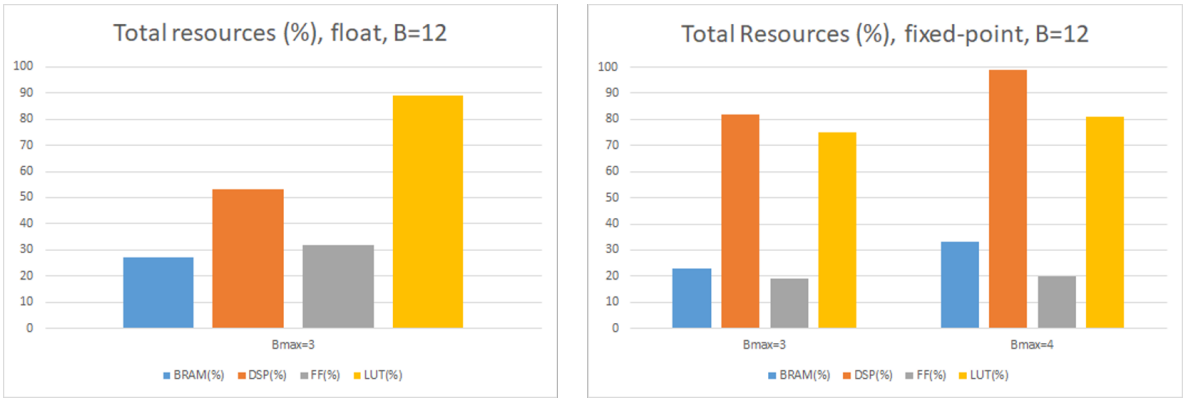

Figure 12. Resource usage for Zedboard for fixed- and floating-point design, B = 12, pixels = 300 × 300.

The larger amount of resources needed by fixed-point design is counterbalanced by the lower latency, as shown in Figure 13 for $B = 12$ and $B_{max} = 3, 4$. The fixed-point design has a lower latency at the same block size and even less latency when using a larger block ($B_{max} = 4$). This is because the latency of fixed-point operations is lower than the floating-point ones. For example, the fixed-point adder has a latency of 1 clock cycle, while the floating-point adder has a latency of 5 clock cycles.

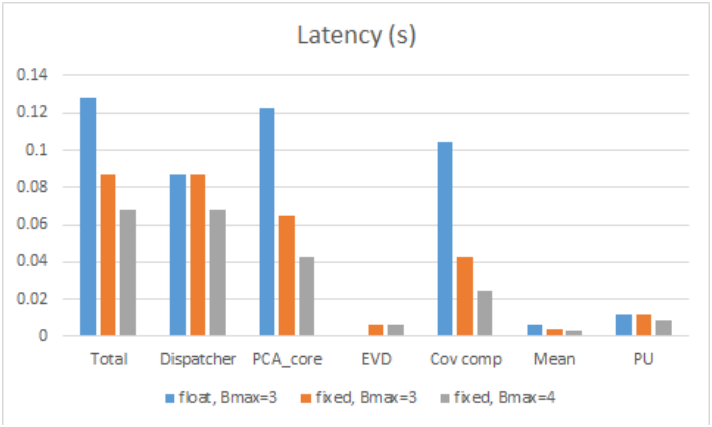

Figure 13. Comparison of the latency of the fixed- and floating-point design for Zedboard, B = 12, pixels = 300 × 300.

Figures 14 and 15 illustrate the total latency of the PCA accelerator and its resource usage for different numbers of bands for a fixed block size ($B_{max} = 3$). As shown in Figure 14, the latency of the floating-point design is limited by the PCA core function, whereas in the fixed-point design the Dispatcher latency is the main limitation. This is because the PCA core and the Dispatcher operate concurrently, as noted before, and therefore the total latency is basically the maximum between the

two latencies, which may change depending on the implementation details (in this case floating- versus fixed-point data representation). The comparison of the resource usage in Figure 15 shows that except for an increase in the DSP usage, other resources are reduced in the fixed-point design. As explained before, the increase in DSP usage is due to the larger bit-width needed for the fixed-point data representation.

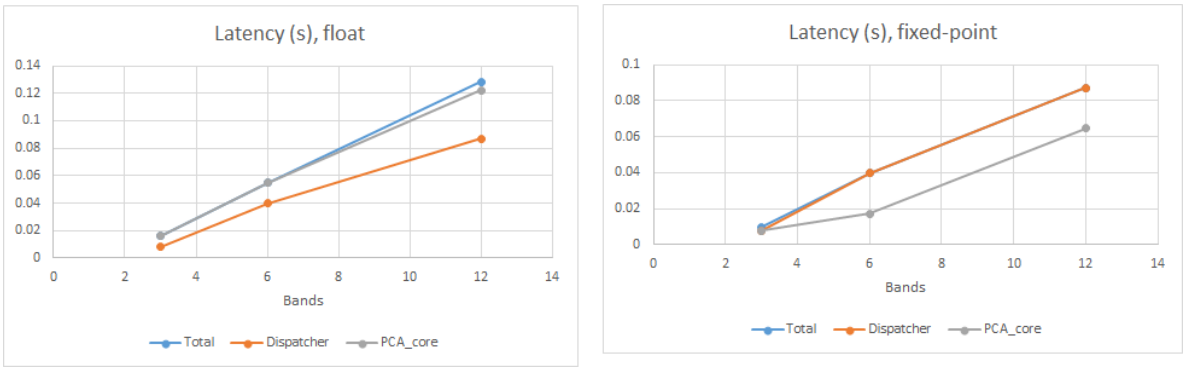

Figure 14. Latency for Zynq7000 with a fixed block size ($B_{max} = 3$), pixels = 300 × 300.

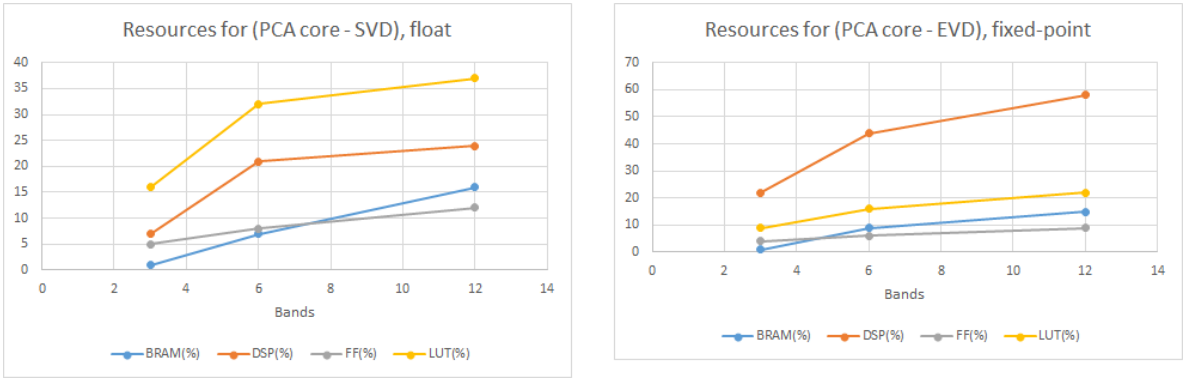

Figure 15. Resource usage for Zynq7000 with a fixed block size ($B_{max} = 3$), pixels = 300 × 300.

5.3. Evaluation on Hyperspectral Images

The hyperspectral image data set is obtained from Purdue Research Foundation and is available online (https://engineering.purdue.edu/~biehl/MultiSpec/hyperspectral.html). It shows a portion of southern Tippecanoe county, Indiana, and comprises 12 bands each of which corresponds to an image of 949 × 220 pixels. We will show that by using PCA, most of the information in the 12 bands is redundant and could be obtained from the first 3 principal components.

The PCA accelerator for this data set is evaluated on the Zynq7000 of the Zedboard for different possible block sizes ($B_{max} = 3, 4$). The HLS estimation of the resource usage for the floating- and fixed-point design is indicated in Table 1. For the floating-point design, the maximum block size is $B_{max} = 3$. In fixed-point design, however, we can use a larger block size ($B_{max} = 4$), which leads to the increase in the resource usage.

Table 1. Resource usage obtained from HLS for HI data set on Zedboard, bands = 12.

	BRAM (%)	DSP (%)	FF (%)	LUT (%)
floating-point ($B_{max} = 3$)	27	57	33	92
fixed-point ($B_{max} = 3$)	23	82	20	75
fixed-point ($B_{max} = 4$)	33	99	20	81

Table 2 shows the latency of different components of the design. According to Table 2, the fixed-point minimum latency is about half of the floating-point latency. in addition, the fixed-point

EVD latency is about 15 times larger than the floating-point SVD latency. However, this does not affect the total latency because the Dispatcher latency in the fixed-point design is higher than the PCA core latency. Therefore, due to the concurrency between Dispatcher and PCA core, the total latency is limited by the Dispatcher. The resource usage for EVD is lower than SVD, so by using the fixed-point EVD we can improve the overall performance because the resources saved by EVD can be used in the rest of the design for more parallelism leading to a lower total latency.

Table 2. Latency (ms) for Zedboard, HI data set, bands = 12.

	Total	Dispatcher	PCA_core	SVD/EVD	Cov	Mean	PU
floating-point (B_{max} = 3)	296.6981	202.0993	282.9185	0.399916	241.1409	13.77959	27.55981
fixed-point (B_{max} = 3)	202.0993	202.0993	141.7858	6.267679	98.75294	9.18632	27.55913
fixed-point (B_{max} = 4)	158.4643	158.4643	91.26137	6.267679	57.4145	6.88974	20.6694

The PCA accelerator resource usage and power consumption in the target hardware are measured by the Vivado software and are shown in Table 3. In addition, the accuracy of the PCA output from FPGA is compared with the MATLAB output by using the Mean Square Error (MSE) metric. MATLAB uses double precision, whereas our FPGA design uses single-precision floating-point as well as fixed-point computations. Although the accuracy of our fixed-point design is reduced, its MSE is still negligible. In contrast, the latency for the fixed-point improves by a factor of 1.8, which shows the efficiency of the fixed-point design.

Table 3. Vivado implementation of PCA accelerator on Zedboard for HI data, bands = 12, accuracy is compared with MATLAB.

	BRAM	DSP	FF	LUT	Power (W)	Accuracy (MSE)
floating-point (B_{max} = 3)	75 (27%)	124 (57%)	29,971 (28%)	27,090 (51%)	2.266	2.08×10^{-7}
fixed-point (B_{max} = 4)	92 (33%)	218 (99%)	18,288 (17%)	20,801 (40%)	2.376	1.1×10^{-3}

The PCA output images generated by the FPGA are visually the same as the MATLAB output. Figure 16 represents the first six outputs of PCA applied to the hyperspectral images data set. The FPGA produces the first 3 principal components that are indicated in the figure as PCA1 to PCA3. As shown in the energy distribution in Figure 17, the first 3 principal components contain almost the entire energy of the input data. The first 3 PCs in Figure 16 correspond to these 3 components. It is evident from Figure 16 that the PCs after the third component do not contain enough information in contrast with the first 3 PCs.

The flexibility of our PCA accelerator allows us to compare it with other state-of-the-art references presenting PCA acceleration with different dimensions and target devices. Table 4 represents the resource usage, frequency and execution time for our FPGA design compared with two other references [15,17]. The input data for all of the references contain 30 rows (pixels) and 16 columns (bands in our design). The first reference uses an HLS approach to design its PCA accelerator on a Zynq FPGA, and the second one uses a manual RTL design in VHDL on a Virtex7 FPGA target. Our HLS design on the same Virtex7 FPGA uses fewer resources as indicated in Table 4. Although the clock frequency of our design is not as high as the previous methods, the total execution time for our design is reduced by a factor 2.3x compared to the same FPGA target.

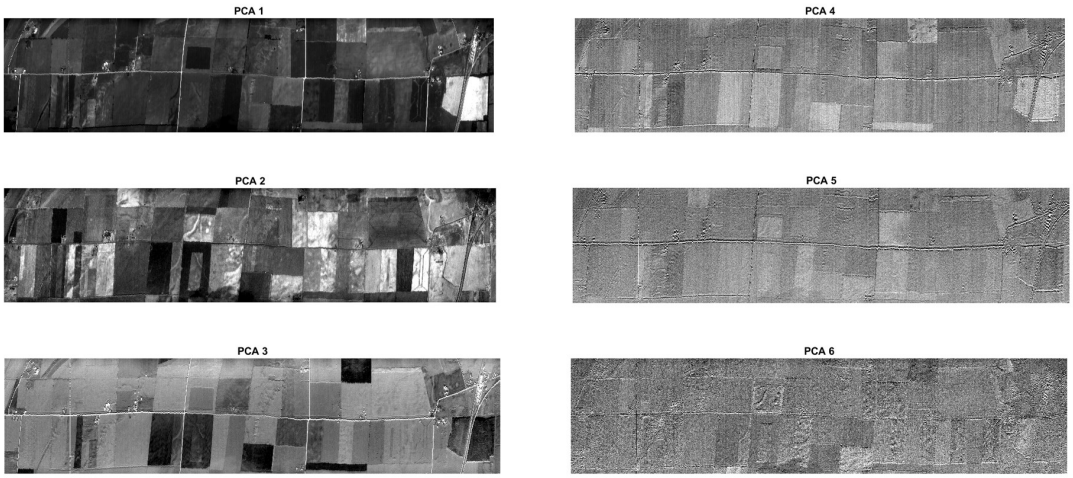

Figure 16. The first 6 principal components of the HI data set. Our PCA accelerator in Zedboard produces the first 3 outputs (PCA1 to PCA3).

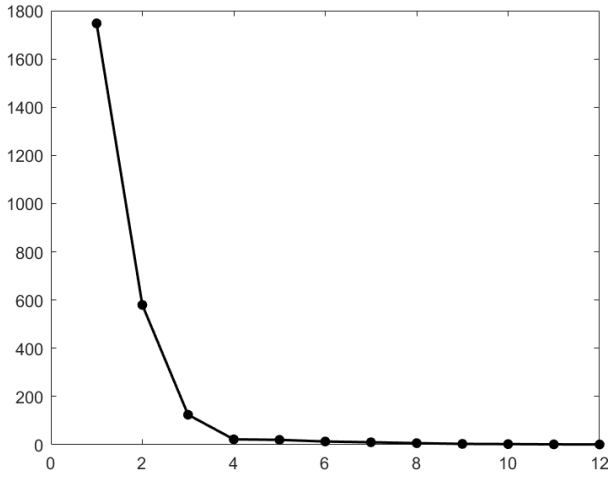

Figure 17. Energy distribution of the eigenvalues for the Hyperspectral Imaging (HI) data set.

Table 4. Comparison of our PCA accelerator with other conventional methods. The input data dimensions are set to 30×16 for all designs.

Work	Device	BRAM	DSP48	FF	LUT	freq (MHz)	Latency (Clock Cycles)	Execution Time (ms)
[17]	Zynq ZC702	6 (2%)	95 (43%)	13,425 (12%)	18,884 (35%)	116	31,707,056	273
[15]	Virtex7 XC7VX485T-2	350 (16%)	2612 (78%)	304,596 (37%)	301,235 (76%)	183	289,352	1.6
This work (B_{max} = 8)	Virtex7 XC7VX485T-2	132 (6%)	385 (13%)	66,790 (10%)	145,220 (47%)	95	64,253	0.675

Table 5 shows the performance of our design compared to other PCA accelerators for spectral images that were also designed in HLS on a Zynq target. In [18] all the PCA functions were designed in HLS except for the EVD unit that was designed in software. A complete hardware design of the PCA algorithm using the standard method for covariance computation (all the bands are streamed at once without blocking) is presented in [20]. Our work uses instead the block-streaming method for covariance computation. The total number of bands is $B = 12$ and the block size in our method

is $B_{max} = 3$. The data representation is floating-point in all of the methods compared in Table 5. As shown in the Table, in our design the DSP and BRAM usage is higher and the FF and LUT usage is lower. Despite the reduction in clock frequency, the total execution time of our design is the minimum (0.44 s) among the three accelerators.

Table 5. Comparison of the proposed PCA hardware design with other High Level Synthesis (HLS)-based accelerators. The dimensions of a spectral image data set ($640 \times 480 \times 12$) is selected for all of the designs.

Work	Execution Time (s)	BRAM (%)	DSP48 (%)	FF (%)	LUT (%)	freq (MHz)
[18] (PCA-SVD)	1.1	12	19	38	73	-
[20]	0.83	9	32	51	94	100
Ours ($B_{max} = 3$)	0.44	27	53	32	90	90

Finally, our FPGA design is compared with a GPU and MPPA implementation of the PCA algorithm [16] for different data dimensions as shown in Table 6. In our design, the target device is the Virtex7 FPGA and the block size is set to $B_{max} = 10$ as the numbers of bands are multiples of 10. For the smaller number of pixels (100×100), our FPGA design outperforms the other two implementations in GPU and MPPA in terms of execution time. For the larger number of pixels, the execution time for our design increases linearly and becomes more than the other designs. It has to be noted, however, that the typical power consumption in MPPAs and GPUs is significantly more than in FPGAs. In the radar chart in Figure 18, four important factors when selecting a hardware platform are considered and their mutual impact is analyzed. These factors are power consumption, latency per pixel, number of bands (input size) and energy. The axes are normalized to the range 0 to 1 and the scale is logarithmic for better visualization.

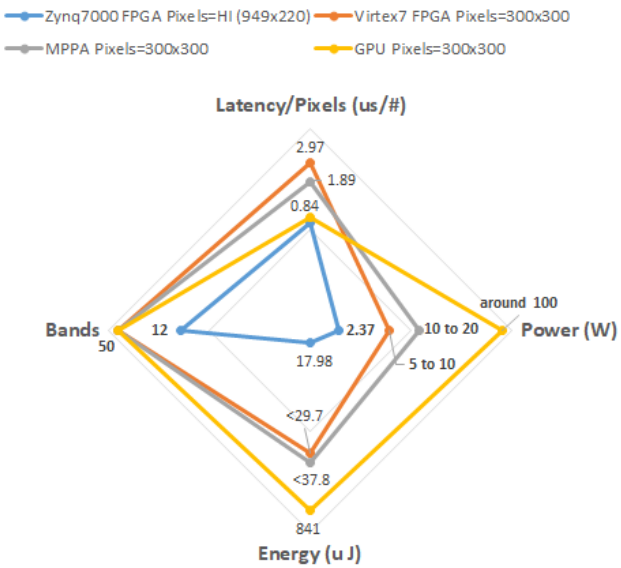

Figure 18. Comparison of different hardware platforms between latency per pixel, power consumption, input size (bands) and energy.

Table 6. Execution time (ms) for the PCA implementation on GPU, Massively Parallel Processing Array (MPPA), and FPGA (our work). The first two designs on GPU and MPPA are from [16].

Dimensions	MPPA	GPU	Ours (FPGA), $B_{max} = 10$
$100 \times 100 \times 50$	140.4	69.28	40.47
$300 \times 300 \times 20$	47.2	70.87	62.37
$300 \times 300 \times 30$	80.1	70.22	121.9
$300 \times 300 \times 50$	170.7	75.74	268.11

As shown in Figure 18, for a small number of bands, a Zynq FPGA has a power consumption of only 2.37 W with a small latency. For larger bands, although GPUs and MPPAs have smaller latency than FPGAs, they consume much more power (especially GPUs). By taking into account the energy consumption that is smaller for FPGAs, one has to select the best hardware based on their needs and use case. Using an FPGA for the PCA accelerator results in a power efficient hardware that can be used for large input sizes without a significant increase in the total latency.

6. Conclusions

In this paper, we proposed a new hardware accelerator for the PCA algorithm on FPGA by introducing a new block-streaming method for computing the internal covariance matrix. By dividing the input data into several blocks of fixed size and reading each block in a specific order, there is no need to stream the entire data at once, which is one of the main problems of resource overuse in the design of PCA accelerators in FPGAs. The proposed PCA accelerator is developed in Vivado HLS tool and several hardware optimization techniques are applied to the same design in HLS to improve the design efficiency. A fixed-point version of our PCA design is also presented, which reduces the PCA latency compared to the floating-point version. Different data dimensions and FPGA targets are considered for hardware evaluation, and a hyperspectral image data set is used to assess the proposed PCA accelerator implemented on Zedboard.

Compared to a similar RTL-based FPGA implementation of PCA using VHDL, our HLS design has a 2.3× speedup in execution time, as well as a significant reduction of the resource consumption. Considering other HLS-based approaches, our design has a maximum of 2.5× speedup. The performance of the proposed FPGA design is compared with similar GPU and MPPA implementations and, according to the results, the execution time changes with data dimensions. For a small number of pixels our FPGA design outperforms GPU and MPPA designs. For a large number of pixels the FPGA implementation remains the most power-efficient one.

Author Contributions: All authors contributed substantially to the paper: conceptualization, M.A.M. and M.R.C.; methodology, M.A.M. and M.R.C.; software, M.A.M.; hardware, M.A.M.; validation, M.A.M. and M.R.C.; writing—original draft preparation, M.A.M.; writing—review and editing, M.R.C.; supervision, M.R.C.; project administration, M.R.C.; funding acquisition, M.R.C. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the EMERALD project funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764479.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

PCA	Principal Component Analysis
PC	Principal Component
MI	Microwave Imaging
HI	Hyperspectral Imaging
HLS	High Level Synthesis
HDL	Hardware Description Language
FPGA	Field Programmable Gate Array
EVD	Eigenvalue Decomposition
SVD	Singular Value Decomposition
MPPA	Massively Parallel Processing Array
Cov	Covariance
PU	Projection Unit
PS	Processing System
PL	Parallel Logic
SoC	System on Chip

References

1. Davis, S.K.; Van Veen, B.D.; Hagness, S.C.; Kelcz, F. Breast Tumor Characterization Based on Ultrawideband Microwave Backscatter. *IEEE Trans. Biomed. Eng.* **2008**, *55*, 237–246. [CrossRef] [CrossRef] [PubMed]
2. Ricci, E.; Di Domenico, S.; Cianca, E.; Rossi, T.; Diomedi, M. PCA-based Artifact Removal Algorithm for Stroke Detection using UWB Radar Imaging. *Med. Biol. Eng. Comput.* **2017**, *55*, 909–921. [CrossRef] [CrossRef] [PubMed]
3. Oliveira, B.; Glavin, M.; Jones, E.; O'Halloran, M.; Conceição, R. Avoiding unnecessary breast biopsies: Clinically-informed 3D breast tumour models for microwave imaging applications. In Proceedings of the IEEE Antennas and Propagation Society International Symposium (APSURSI), Memphis, TN, USA, 6–11 July 2014; pp. 1143–1144. [CrossRef]
4. Gerazov, B.; Conceicao, R.C. Deep learning for tumour classification in homogeneous breast tissue in medical microwave imaging. In Proceedings of the IEEE EUROCON 17th International Conference on Smart Technologies, Ohrid, Macedonia, 6–8 July 2017; pp. 564–569. [CrossRef]
5. Torun, M.U.; Yilmaz, O.; Akansu, A.N. FPGA, GPU, and CPU implementations of Jacobi algorithm for eigenanalysis. *J. Parallel. Distrib. Comput.* **2016**. [CrossRef] [CrossRef]
6. Kasap, S.; Redif, S. Novel Field-Programmable Gate Array Architecture for Computing the Eigenvalue Decomposition of Para-Hermitian Polynomial Matrices. *IEEE Trans. VLSI Syst.* **2014**, *22*, 522–536. [CrossRef] [CrossRef]
7. Wang, X.; Zambreno, J. An FPGA Implementation of the Hestenes-Jacobi Algorithm for Singular Value Decomposition. In Proceedings of the IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, 19–23 May 2014; pp. 220–227. [CrossRef]
8. Shuiping, Z.; Xin, T.; Chengyi, X.; Jinwen, T.; Delie, M. Fast implementation for the Singular Value and Eigenvalue Decomposition based on FPGA. *Chin. J. Electron.* **2017**, *26*, 132–136. [CrossRef]
9. Ma, Y.; Wang, D. Accelerating SVD computation on FPGAs for DSP systems. In Proceedings of the IEEE 13th International Conference on Signal Processing (ICSP), Chengdu, China, 6–10 November 2016; pp. 487–490. [CrossRef]
10. Chen, Y.; Zhan, C.; Jheng, T.; Wu, A. Reconfigurable adaptive Singular Value Decomposition engine design for high-throughput MIMO-OFDM systems. *IEEE Trans. VLSI Syst.* **2013**, *21*, 747–760. [CrossRef] [CrossRef]
11. Athi, M.V.; Zekavat, S.R.; Struthers, A.A. Real-time signal processing of massive sensor arrays via a parallel fast converging SVD algorithm: Latency, throughput, and resource analysis. *IEEE Sens. J.* **2016**, *16*, 2519–2526. [CrossRef] [CrossRef]
12. Perera, D.G.; Li, K.F. Embedded Hardware Solution for Principal Component Analysis. In Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, Victoria, BC, Canada, 23–26 August 2011; pp. 730–735. [CrossRef]
13. Fernandez, D.; Gonzalez, C.; Mozos, D.; Lopez, S. FPGA implementation of the principal component analysis algorithm for dimensionality reduction of hyperspectral images. *J. Real Time Image Process.* **2019**, *16*, 1–12. [CrossRef] [CrossRef]
14. Das, A.; Nguyen, D.; Zambreno, J.; Memik, G.; Choudhary, A. An FPGA-based network intrusion detection architecture. *IEEE Trans. Inf. Forensics Secur.* **2008**, *3*, 118–132. [CrossRef] [CrossRef]
15. Korat, U.A.; Alimohammad, A. A reconfigurable hardware architecture for Principal Component Analysis. *Circ. Syst. Signal Process.* **2019**, *38*, 2097–2113. [CrossRef] [CrossRef]
16. Martel, E.; Lazcano, R.; López, J.; Madroñal, D.; Salvador, R.; López, S.; Juarez, E.; Guerra, R.; Sanz, C.; Sarmiento, R. Implementation of the Principal Component Analysis onto High-Performance Computer Facilities for Hyperspectral Dimensionality Reduction: Results and Comparisons. *Remote Sens.* **2018**, *10*, 864. [CrossRef] [CrossRef]
17. Ali, A.A.S.; Amira, A.; Bensaali, F.; Benammar, M. Hardware PCA for gas identification systems using high Level Synthesis on the Zynq SoC. In Proceedings of the IEEE 20th International Conference on Electronics, Circuits, and Systems (ICECS), Abu Dhabi, United Arab Emirates, 8–11 December 2013; pp. 707–710. [CrossRef]

18. Schellhorn, M.; Notni, G. Optimization of a Principal Component Analysis Implementation on Field-Programmable Gate Arrays (FPGA) for Analysis of Spectral Images. In Proceedings of the Digital Image Computing: Techniques and Applications (DICTA), Canberra, Australia, 10–13 December 2018; pp. 1–6. [CrossRef]
19. Mansoori, M.A.; Casu, M.R. Efficient FPGA Implementation of PCA Algorithm for Large Data using High Level Synthesis. In Proceedings of the 15th Conference on Ph.D Research in Microelectronics and Electronics (PRIME), Lausanne, Switzerland, 15–18 July 2019; pp. 65–68. [CrossRef]
20. Mansoori, M.A.; Casu, M.R. HLS-Based Flexible Hardware Accelerator for PCA Algorithm on a Low-Cost ZYNQ SoC. In Proceedings of the IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC), Helsinki, Finland, 29–30 October 2019; pp. 1–7. [CrossRef]
21. Manolakis, D.; Shaw, G. Detection Algorithms for Hyperspectral Imaging Applications. *IEEE Signal Process. Mag.* **2002**, *19*, 29–43. [CrossRef] [CrossRef]
22. Demmel, J.; Veselić, K. Jacobi's method is more accurate than QR. *SIAM J. Matrix Anal. Appl.* **1992**, *13*, 1204–1245. [CrossRef] [CrossRef]
23. Beilina, L.; Karchevskii, E.; Karchevskii, M. *Numerical Linear Algebra: Theory and Applications*, 1st ed.; Springer International Publishing: Cham, Switzerland, 2017. [CrossRef]

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).