A Scalable Systolic Accelerator for Estimation of the Spectral Correlation Density Function and its FPGA Implementation

XIANGWEI LI, Nanyang Technological University, Singapore
DOUGLAS L. MASKELL, Nanyang Technological University, Singapore
CAROL JINGYI LI, The University of Sydney, Faculty of Engineering, School of Electrical and Information Engineering, Australia
PHILIP H.W. LEONG, The University of Sydney, The University of Sydney Nano Institute, Faculty of Engineering, School of Electrical and Information Engineering, Australia
DAVID BOLAND, The University of Sydney, Faculty of Engineering, School of Electrical and Information Engineering, Australia

The spectral correlation density (SCD) function is the time-averaged correlation of two spectral components, used for analysing periodic signals with time-varying spectral content. Although the analysis is extremely powerful, it has not been widely adopted in real-time applications due to its high computational complexity. In this paper, we present an efficient FPGA implementation of the FFT accumulation method (FAM) for estimating the SCD function and its alpha profile. The implementation uses a linear systolic array with a bi-directional datapath consisting of DSP-based processing elements (PEs) with a dedicated instruction schedule, achieving a PE utilization of 88.2%.

The 128-PE implementation achieves a clock frequency in excess of 530 MHz and consumes 151K LUTs, 151K FFs, 264 BRAMs, 4 URAMs and 1054 DSPs, which is less than 36% of the logic fabric on a Zynq UltraScale+ XCZU28DR-2FFVG1517E RFSoC device. It has a modest 12.5W power consumption and an energy efficiency of 4832 MOPS/W which is 20.6× better than the published state-of-the-art GPU implementation. In terms of throughput, it achieves 15340 windows/s (15340 windows/s × 2048 samples/window = 31.4 MS/s), which is a 4.65× improvement compared to the above-mentioned GPU implementation and 807× compared to an existing hybrid FPGA-GPU implementation.

CCS Concepts: • **Hardware → Reconfigurable logic applications**; **Hardware accelerators**; • **Computer systems organization →** *System on a chip.*

Additional Key Words and Phrases: FPGA, systolic array, spectral correlation density, FFT accumulation method

Authors' addresses: Xiangwei Li, xli045@e.ntu.edu.sg, Nanyang Technological University, 50 Nanyang Avenue, Singapore, 639798; Douglas L. Maskell, Nanyang Technological University, 50 Nanyang Avenue, Singapore, 639798, asdouglas@ntu.edu.sg; Carol Jingyi Li, The University of Sydney, Faculty of Engineering, School of Electrical and Information Engineering, Sydney, New South Wales, Australia., Sydney, Australia, jingyi.li@sydney.edu.au; Philip H.W. Leong, The University of Sydney, The University of Sydney Nano Institute, Faculty of Engineering, School of Electrical and Information Engineering, Sydney, New South Wales, Australia., Sydney, Australia, philip.leong@sydney.edu.au; David Boland, The University of Sydney, Faculty of Engineering, School of Electrical and Information Engineering, Sydney, New South Wales, Australia., Sydney, Australia, david.boland@sydney.edu.au.

1936-7406/2022/6-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Xiangwei Li, Douglas L. Maskell, Carol Jingyi Li, Philip H.W. Leong, and David Boland. 2022. A Scalable Systolic Accelerator for Estimation of the Spectral Correlation Density Function and its FPGA Implementation. *ACM Trans. Reconfig. Technol. Syst.* 1, 1, Article 111 (June 2022), 25 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

A signal exhibits cyclostationarity if and only if the signal is correlated with certain frequency-shifted versions of itself [10]. It is used for the analysis of a wide range of periodic phenomena in applications, such as signal detection and modulation classification; noise analysis of periodic systems; synchronization problems; signal parameter and waveform estimation; channel identification and equalization; autoregressive (AR) and autoregressive moving average (ARMA) modeling; and source separation [13].

A well-known technique to interpret the underlying periodicity of a cyclostationary signal is the spectral correlation density (SCD), also referred to as the cyclic spectral density or spectral correlation function, which describes the cross-spectral density of all pairs of frequency-shifted versions of a time-series. The SCD function represents the time-averaged statistical correlation of two spectral components at frequencies f and $f - \alpha$, as the bandwidth approaches zero [13]. It performs exceptionally well in cognitive radio systems such as modulation classification, under low signal to noise ratio (SNR) conditions [24]. For instance, different modulation types such as BPSK, QPSK and MSK can be easily detected by their distinct SCD functions [10, 22].

We comment that there has been an increasing body of work studying the use of deep learning methods for various signal processing applications, such as spectrum sensing or cognitive radio. To date, these have mostly worked with raw IQ. An excellent survey of this work is provided by Wong et al. [32]. Nevertheless, the performance of deep learning algorithms can often be improved by providing relevant information. With this in mind, our work seeks to develop the ability to generate a key feature with a very low latency. A deep learning algorithm could then realistically use our implementation to calculate this feature, without considerable processing delay, to potentially improve its overall performance.

While the implementation of the SCD function can be done either via time or frequency smoothing, we consider time smoothing as it has been shown to be more computationally efficient in general [25]. One of the most popular time smoothing methods to estimate the SCD is the FFT accumulation method (FAM) [25, 26]. The FAM technique is suitable for hardware implementation due to its parallel FFT-based computations and regular data access patterns. However, the algorithm's diamond-shaped computation pattern and the large amount of output data present a considerable hardware acceleration challenge. The first problem we address through the use of a bi-directional systolic array, the second by following the technique of computing the alpha profile [21]. This technique captures the most significant information from the full SCD function by taking the peak values along the α axis of the bi-frequency (f-α domain) feature map, reducing the two dimensional SCD to a one dimensional vector of alphas.

In this paper, a scalable high-speed FPGA accelerator is proposed for estimating the SCD function and alpha profile using FAM. A linear systolic array is used with programmable processing elements (PEs). The PEs operate on complex-valued data and are optimized for high clock frequency. Inter-PE data dependencies are minimized by maximising intra-PE data cohesion. The proposed FPGA accelerator is evaluated by comparing it to existing hardware accelerator implementations in terms of throughput, area and energy efficiency. The novel contributions of this work are as follows:

- A novel, systolic array for efficient computation of the SCD function and alpha profile using a bi-directional datapath. This allows a sustained PE utilization of 88.2% while that of a single directional datapath is merely 50.4%.

- A highly pipelined, programmable RISC-like load–store architecture with a complex arithmetic ALU based on Xilinx DSP48E2 slices which enables the efficient implementation of the FAM method.
- An optimized microarchitecture and FPGA implementation which maximizes clock frequency through heavy pipelining. Our 128-PE configuration operates at a frequency of 530 MHz on a Zynq UltraScale+ XCZU28DR-2FFVG1517E RFSoC device. An open source FPGA implementation for the systolic array is available at GitHub[1].

The remainder of this paper is organised as follows. In Section 2, the FAM method for cyclostationary signal processing is presented. In Section 3, we analyse the FAM algorithm to exploit the calculation of the computationally intensive kernel in parallel. In Section 4, a bi-directional linear systolic array as an FPGA accelerator for FAM is proposed and its mapping scheme is thoroughly explained. This section also details the implementation of our proposed FPGA accelerator, including the PE microarchitecture and the array of PEs connected via linear interconnect. Section 5 evaluates the proposed FPGA accelerator by comparing it to a state-of-the-art hybrid FPGA-GPU implementation and a state-of-the-art GPU implementation, in terms of throughput and resource utilization. Finally, We draw conclusions and discuss our future work in Section 6.

2 BACKGROUND

2.1 Spectral Correlation Density Function

The time smoothed cyclic cross periodogram [11] of two complex-valued sequences $x(n)$ and $y(n)$ over a time interval of Δt seconds is defined by (1).

$$S_{xy_T}^{\alpha_0}(n, f_0)_{\Delta t} = \frac{1}{T} \left\langle X_T(n, f_1) Y_T^*(n, f_2) \right\rangle_{\Delta t} \tag{1}$$

Here $\langle \cdot \rangle$ is the inner product operation, while $X_T(n, f_1)$ and $Y_T(n, f_2)$ are *complex demodulates* centered at frequency $f_1 = f_0 + \alpha_0/2$ and $f_2 = f_0 - \alpha_0/2$, with α_0 the frequency delay between two complex demodulates. The demodulates, defined by (2), are computed over a windowing function $a(r)$ of length $T = N_p T_s$ seconds from the original time interval Δt, where T_s is the sampling period and N_p is the number of samples.

$$X_T(n, f) = \sum_{r=-N_p/2}^{N_p/2} a(r) x(n-r) e^{-i2\pi f (n-r) T_s} \tag{2}$$

To compute the digital implementation of SCD function of inputs $x(n)$ and $y(n)$, we first compute demodulates for $X_T(n, f_1)$ and $Y_T(n, f_2)$. We then correlate these demodulates using a complex multiplier followed by a low pass filter (LPF) with bandwidth approximately $1/\Delta t$ [25]. Altogether, the SCD estimate at (f_0, α_0) given by (3).

$$S_{xy_T}^{\alpha_0}(n, f_0)_{\Delta t} = \sum_r X_T(r, f_1) Y_T^*(r, f_2) g(n-r) \tag{3}$$

where $g(n)$ is a windowing function with length $\Delta t = N T_s$. For the special case of auto-correlation studied in this paper, $y(n)$ is a time-delayed version of $x(n)$. For a reliable estimate, $\Delta t >> T$.

2.2 Estimating Spectral Correlation Density

While direct application of Equation (3) is computationally inefficient, decimation and the fast Fourier transform (FFT) can be used to reduce the complexity [8]. Decimation, involves reducing the number of complex demodulates computed from N to $P = N/L$ by using an L sample stride. L is set

[1]https://github.com/louislxw/pe_array

as $N_p/4$ to have a good tradeoff between maintaining computational efficiency and minimizing cycle leakage and aliasing [8], producing a new sequence of demodulates $X_T(pL, f)$ for $p = \{0, 1, ..., P-1\}$. To use the FFT, the following substitutions are implemented to Equation (2): $d = N_p/2 - 1, r = d - k, f_m = m/(N_pT_s)$ and $-N_p/2 < m < N_p/2$ [12]. This results in Equation (4), where the summation in brackets can be efficiently computed by the FFT.

$$X_T(pL, f_m) = [\sum_{k=0}^{N_p-1} a(d-k)x(pL-d+k)e^{-i2\pi mk/N_p}]e^{-i2\pi mpL/N_p} \tag{4}$$

The SCD function can then be estimated from the decimated complex demodulates $X_T(rL, f_h)$ and $Y_T(rL, f_l)$ at center frequency of f_h and f_l, as in Equation (5).

$$\hat{S}_x^{\alpha_0}(f_0) = \hat{S}_x^{\alpha_0}(pL, f_{hl})_{\Delta t} = \sum_r X_T(rL, f_h)Y_T^*(rL, f_l)g_d(p-r) \tag{5}$$

where $g_d(r) = g(rL)$. However, a frequency shift $\epsilon = q\Delta\alpha$, where $\Delta\alpha = 1/\Delta t$ for each point estimate $q \in \{1, ..., P\}$, can be introduced into the complex demodulate product in Equation (5). This enables the SCD function to also be efficiently evaluated by a P-point FFT of the complex demodulate product, as in Equation (6). In this equation, we also make the following substitutions: $a_{hl} = f_h - f_l$, $f_0 = f_{hl} = (f_h + f_l)/2, \alpha_0 = a_{hl} + q\Delta\alpha$ and $\Delta\alpha = f_s/P$ [12].

$$\hat{S}_x^{\alpha_0}(f_0) = \hat{S}_x^{a_{hl}+\Delta\alpha}(pL, f_{hl})_{\Delta t} = \sum_r X_T(rL, f_h)Y_T^*(rL, f_l)g_d(p-r)e^{-i2\pi rq/P} \tag{6}$$

2.3 Example of Spectral Correlation Density and Alpha Profile

Figure 1 is an example of a real signal $x(n)$ in OOK modulation, its decimated *complex demodulate* $X_T(rL, f_m)$, the estimated SCD function $\hat{S}_x^{\alpha_0}(f_0)$, and the alpha profile. The alpha profile is the vector of peak values along the α axis of the bi-frequency feature map. This is valuable because the diamond-shaped SCD function is of size $2N \times 2N_p$ for a signal of length N, decimated into P subsequences of length N_p. The alpha profile captures key information with greatly reduced storage requirements compared with the full SCD matrix.

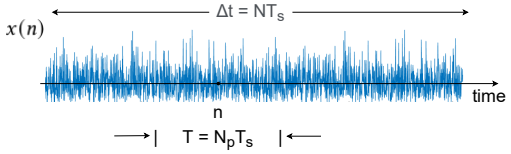
The SCD function can be used to extract the spectrum features, with the alpha profile able to retain critical information sufficient to classify different signals. To demonstrate this, we provide a comparison of these features for two signals from the DeepSig [15] benchmark with different modulations in Figure 2.

2.4 Comparing FFT-based Methods for Estimating SCD Function

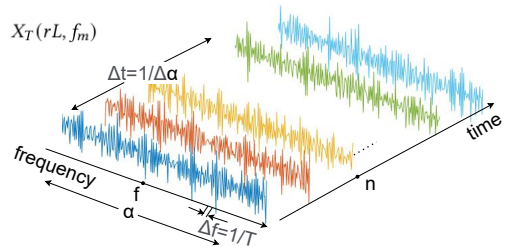
Several methods exist to estimate the SCD function of cyclostationary signals. These include averaged cyclic periodogram (ACP) [1], cyclic modulation spectrum (CMS) [2], FFT accumulation method (FAM) [25], strip spectral correlation algorithm (SSCA) [27] and fast spectral correlation (Fast-SC) [4].

The ACP [1] is an extended "Weighted-Overlapped-Segment-Averaging" [31] method for cyclostationary signals which produces the SCD function with a high resolution. However, the accurate estimation of SCD comes at the cost of substantial computational complexity when the signal length grows. This is mainly due to the massive amount of direct complex multiplications introduced by the traditional implementation of the discrete-time Fourier transform (DTFT).

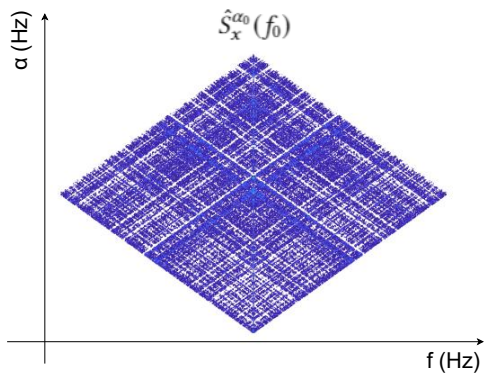
The CMS [2, 3] utilizes the short-time Fourier transform (STFT) and has low complexity. Despite its efficiency, it is a biased SCD estimator and its approximation error increases significantly when the cyclic frequency exceeds the frequency resolution.



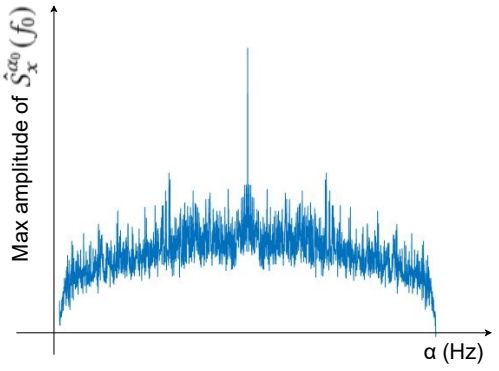
(a) A real signal x(n) with a sample period of T_s.



(b) Decimated complex demodulates of signal x(n).



(c) The SCD function of signal x(n).



(d) Alpha profile: peak values across α axis of SCD.

Fig. 1. The SCD function and alpha profile of OOK signal from DeepSig [15] at SNR = -8 dB.

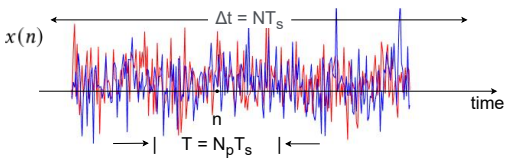
The FAM [25] and SSCA [27] techniques are the most popular time smoothing methods to estimate the SCD function from the decimated complex demodulates. Both are recognized as being computationally efficient algorithms in the specialized literature, with FAM generally requiring a smaller FFT size in the most computationally intensive part.

In recent years, Fast-SC [4] was proposed as an improved version of CMS which estimates the SCD at a higher cyclic frequency resolution, by trading off moderate computational cost. However, the complexity of Fast-SC algorithm increases notably when there is a need to enlarge the maximum range of cyclic frequencies for study [7].

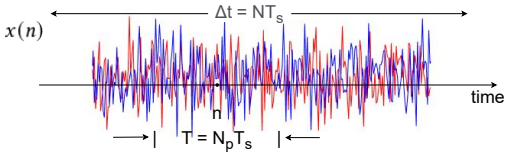
While most of the surveyed approaches (except for ACP) are based on bi-frequency FFT operations, FAM is one of the most favorable algorithms for hardware implementation due to its parallel computations and regular data access patterns. A detailed, algorithmic description of the FAM technique is presented in Section 3.1.

2.5 Existing Implementations of the FAM Method

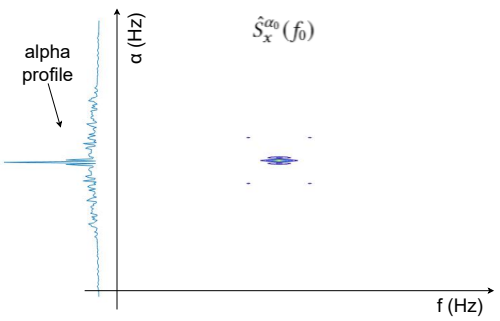
Early research has focused on the serial CPU implementations of the FAM method [9, 26]. However, they were generally written in sequential software languages such as C and MATLAB, and the performance is far from that of real time requirements. To be suitable for real world applications, it is essential that the parallelism of the SCD estimators is exploited. Where the output of the alpha profile could be used as a feature in a broader modulation classification system, the former must operate at rates up to the throughput of the design.



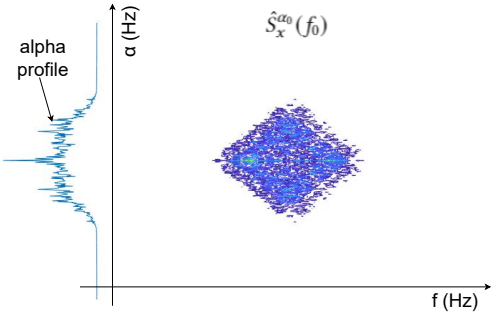
(a) A complex-valued OOK signal.



(b) A complex-valued QPSK signal.



(c) SCD matrix and alpha profile of the OOK signal.



(d) SCD matrix and alpha profile of the QPSK signal.

Fig. 2. A comparison of SCD estimation between OOK and QPSK signals from DeepSig [15] at SNR = 28 dB.

In 2008, Ge et al. [14] designed a parallel FAM algorithm, running on a cell broadband engine of a PlayStation 3. It is comprised of a power processor element (PPE) acting as the controller and eight synergistic processing elements (SPEs) for parallel computing, which runs 7.6× faster than sequential FAM on a general purpose processor. Similarly, GAEA [29] was proposed as a hybrid parallel architecture which supports very long instruction word (VLIW) and single instruction, multiple data (SIMD) instructions. It runs at 350 MHz and requires only 78.8 ms to execute the FAM algorithm for a signal of 32K samples.

Recently, there is a growing interest in developing high-performance hardware implementations of the FAM method. Lee et al. [18] developed a GPU implementation of the FAM method and achieved a speedup of 39× over the serial implementation running on a 2.94 GHz Intel Core 2 CPU. The University of Arizona proposed high throughput GPU implementations [20, 21] and a heterogeneous FPGA-GPU implementation [5] of FAM. While they reported a state-of-the-art throughput of 3300 windows/s on a Tesla K40 GPU, the hybrid FPGA-GPU implementation was significantly less optimised, achieving only 19 windows/s. This was due to the use of a relatively low-end Tegra K1 GPU and a Zynq-7000 FPGA which consumed less than 3% of hardware resource while running at just 140 MHz.

2.6 Systolic Array Processors

The high performance of our design mainly arises through parallel processing of the FAM algorithm in a systolic array. Systolic processor arrays are regular architectures with local interconnection between processing elements (PEs). This design structure facilitates an architecture that operates at a high clock frequency. The main challenge is to ensure the PEs are utilised efficiently. There is seemingly continual research in VLSI and FPGA systolic array processors. For example, this includes an early discussion provided by Kung [17] and more recent FPGA designs for deep learning [30]. Common issues that must be addressed include I/O constraints, memory limitations, pipelining,

challenging compute patterns and efficient PE design. Instead of exploring this entire research area, we highlight examples from the field of solving systems of linear equations that highlight each of these problems and different techniques employed to overcome them. We then discuss how this inspired our own approach.

When designing a systolic array, it is important to ensure that computation overhead is greater than or equal to I/O limitations. For example, in block-based LU factorisation with blocks of size $M \times M$, the number of computations are $O(M^3)$, but the number of clock cycles to load a block from memory to the accelerator is $O(M^2)$. This means the maximum number of parallel operations is $O(M)$. A simple solution, as presented by Zhang et al. [33], is to preload four blocks of a large matrix into on-chip memory, three of which are reused for an entire row. After this double-buffering can be used to overlap loading the next block from memory while computing the current block. This essentially is pipelining the load phase for the next block in an alternative memory so the systolic array can be fully utilised.

Challenging compute patterns occur when an algorithm does not easily fit into a 1D or 2D array. For example, QR-based solution of a system of linear equations typically results in a triangular compute pattern. Folding the dataflow to reuse the PEs of a linear or 2D array is one of the main techniques to address this. However, the chosen folding approach either increases the complexity of the PE or the interconnect. For example, the approach by Rader [23] required a PE that supported CORDIC operations. A more general approach outlining various options to fold an array for QR-based solutions and their trade-offs is presented by Lightbody et al. [19]. An alternative solution for challenging compute patterns is to modify the underlying algorithm that utilises the systolic array. For example, Boland [6] presents a way to restructure LU decomposition with partial pivoting such that it maintains numerical performance, fits onto a systolic array with minimal memory overhead and maintains trivial PEs to maximise clock frequency. However, this method sacrifices some efficiency, reaching only 66%.

In this work, I/O is not an issue because the algorithm, described in Section 3.1, converts a streaming input into a matrix through the use of overlapping windows, essentially performing a serial to parallel conversion. However, we do experience a challenging triangular compute pattern. We considered a folding approach, but decided the additional hardware required was non-trivial. Instead we added a novel modification to the interconnect to support a bi-directional datapath to pipeline successive problems in opposite directions. This achieves a utilisation of 88.2% with minimal overhead.

3 ALGORITHM

3.1 FFT Accumulation Method

In this paper, we focus on the FFT accumulation method (FAM) as it is one of the most computationally efficient methods. The algorithm can be generally decomposed into four steps, i.e., decimation and windowing, first stage FFT, down conversion, SCD matrix and alpha profile generation [5]. There are a few parameters in this algorithm which have been discussed in Section 2. Our design is both scalable and general. It can be readily applied to any parameter setting, subject to available FPGA resources, by changing the PE microcode.

3.1.1 Step 1: Decimation and Windowing. The first step is to decimate the input signal and filter it using a window function, as given in Equation (4). The signal of length N is divided into P channels with sub-sequences of N_p elements and a nonoverlap offset of $L = N_p/4$. After the decimation, a Hamming window is applied to eliminate the artificial high frequency components. The output of this step is referred to as X_W.

3.1.2 Step 2: First Stage FFT. The windowed frames generated from the first step are then applied to a N_p-point fast Fourier transform (FFT), as described by the square brackets in Equation (4). Since the FFT size is the same as that of one frame in X_W, the N_p-point FFT output X_F maintains the size of $N_p \times P$. Figure 3 describes the process of step 1 and step 2 in sequential blocks.

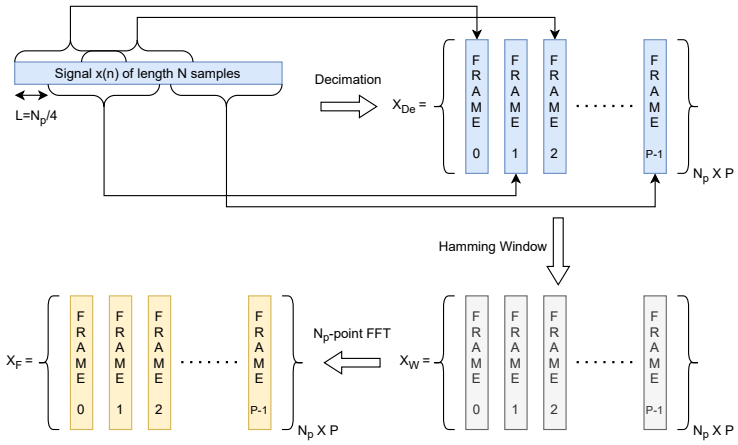


Fig. 3. FAM algorithm step 1-2: decimation, windowing and N_p-point FFT.

3.1.3 Step 3: Down Conversion. Down conversion (frequency shifting) is then performed on X_F to obtain the complex demodulate sequences X_D, which is detailed in Equation (7). To control cycle leakage and aliasing, L is set to $N_p/4$ meaning the exponential function $e^{-i2\pi kmL/N_p}$ can only take the values from (i, -i, 1, -1).

$$X_D[k,m] = X_F[k,m] * e^{-i2\pi kmL/N_p} = X_F[k,m] * [\cos(2\pi kmL/N_p) - i\sin(2\pi kmL/N_p)] \quad (7)$$

3.1.4 Step 4: SCD Matrix and Alpha Profile Generation. The last step of FAM is to calculate the SCD matrix, which corresponds to Equation (6), and the alpha profile. This is the most computational expensive part as the iterative P-point FFT constitutes $\approx 86\%$ of the whole execution time on a

serial CPU implementation [21]. The pseudocode of the SCD matrix and alpha profile computation is described in Algorithm 1.

Algorithm 1: Compute kernel of SCD matrix and alpha profile.

Input: Two matrices $X = X_D$ and $Y = X_D^*$ with a size of $N_p \times P$.
Output: A vector $alpha\_profile$ with a size of $2N \times 1$.
for $i \leftarrow 0$ **to** $N_p - 1$ **by** 1 **do**
  for $j \leftarrow 0$ **to** $N_p - 1$ **by** 1 **do**
    $K[i * N_p + j,\ :] \leftarrow X[i,\ :] * Y[j,\ :]$
    $M[i * N_p + j,\ :] \leftarrow abs(P\text{-point } FFT(K[i * N_p + j,\ :]))$
    $row \leftarrow ((j - i)/N_p + 1) * N$
    $Pa \leftarrow M[i * N_p + j,\ P/4 : (P/2 - 1)]$
    $Pb \leftarrow M[i * N_p + j,\ P/2 : (3P/4 - 1)]$
    $SCD[(row - P/4) : (row + P/4 - 1),\ i + j] \leftarrow \{Pa,\ Pb\}$
  end
end
$alpha\_profile \leftarrow \max(SCD,\ [],\ 2)$

where X_D^* represents for an element-wise complex conjugate of matrix X_D.

The structure of the SCD matrix and alpha profile is shown in Figure 4. As explained in Algorithm 1, half of the iterative P-point FFT outputs (in the middle range) are used to build the SCD matrix, formed by multiple Pa and Pb elements in a staircase fashion, as in Figure 4a. Thus, a huge number of FFT outputs are required to form the full SCD matrix with a size of $2N \times 2N_p$, which imposes a significant memory and communications bottleneck on the implementation making it impractical in a real-time scenario. In order to resolve this issue, a row-wise MAX operation, namely the alpha profile, is applied to the SCD matrix so that only the peak values of each row are used as the final outputs for classification.

According to Algorithm 1, the calculation of the Pa and Pb blocks in the down staircase direction are independent, which means the SCD matrix can be efficiently generated by mapping different blocks in the down staircase direction to different parallel processing elements (PEs). Besides, the PEs should be interconnected with their neighbors to support the distributed computation of the alpha profile in the horizontal view. All these inspirations bring us to an idea of developing a linear systolic array of PEs to support the hybrid dataflow of the SCD matrix and alpha profile.

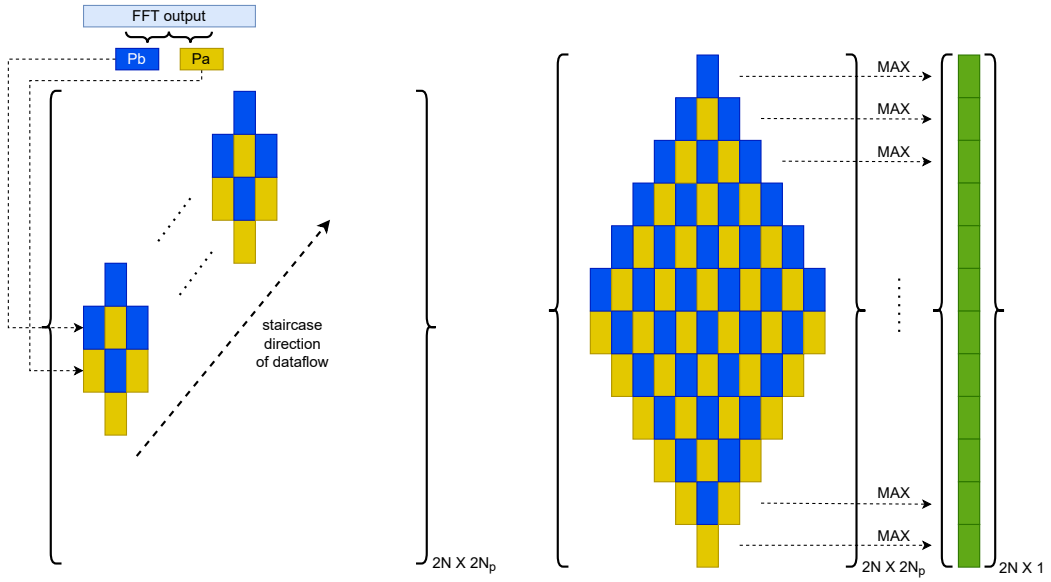
3.2 Exploiting Similarity in SCD

In Section 2 we have shown that the representation of SCD estimation $\hat{S}_x^\alpha(f)$ is a 2-D feature map in f and α axes. It is symmetrical in the bi-frequency dimension [10] as indicated in Equations (8) and (9).

$$\hat{S}_x^\alpha(f) = \hat{S}_x^\alpha(-f) \tag{8}$$

$$\hat{S}_x^{-\alpha}(f) = \hat{S}_x^\alpha(f)^* \tag{9}$$

Thus, it is sufficient to compute just one quadrant of the SCD function $\hat{S}_x^\alpha(f)$ (marked as the shaded triangle in Figure 5). For the alpha profile, only half need be computed as the final output. Thus, the complexity of the FAM method can be reduced by around 75% compared to that of the full SCD computation. The number of individual PEs can also be reduced by 50%.



(a) Staircase SCD matrix formation. (b) Row-wise alpha profile calculation.

Fig. 4. SCD matrix and alpha profile generation.

In the following sections, we focus on the calculation of one quadrant of the SCD function and refer it to as QSCD. We also customize a few parameters of FAM algorithm to match with the literature [5, 21] for the illustration of the microarchitecture in Section 4.2 and evaluation in Section 5. Specifically, in this work $N = 2048$, $N_p = 256$, $L = N_p/4 = 64$ and $P = N/L = 32$.

4 ARCHITECTURE

Due to the straightforward nature of step 1 to step 3 of the FAM algorithm, we implemented these steps in HLS. We then connected this to our optimized systolic array, implemented in RTL, which is described in the following subsection.

4.1 QSCD Mapping Scheme

The methodology for mapping QSCD to the linear systolic array is illustrated in Figure 6. Algorithm 1 is parallelized by mapping the independent complex multiplications (MULs) and FFTs to the corresponding PEs (each PE is represented by a different color in Figure 6).

Initially, each PE has an input of (X_i, Y_i). After the PE computes the MUL and FFT operations for one iteration, it then calculates and transfers the partial alpha results to its adjacent PE (as indicated by the horizontal red dashed line) so that the MAX operation can be split and allocated evenly to each PE. This makes effective use of the systolic array. The PE then changes the input source from (X_i, Y_i) to (X_i, Y_{i+j}) by shifting in the new Y component from the adjacent PE (as indicated by the blue dashed line), upon which the PE proceeds to compute the next iteration and so on. The purpose of the "SHIFT" function is to to shift the Y components from the current PE ($PE[i]$) to the previous $PE[i-1]$ in the shift register array. The "TX" function is responsible for the transmission of partial alpha results from the current PE ($PE[i]$ to the next $PE[i+1]$). In the end, two vectors of length N are calculated by the systolic array (as indicated by the green dashed line) and are compared along the aligned region to generate the final alpha profile.

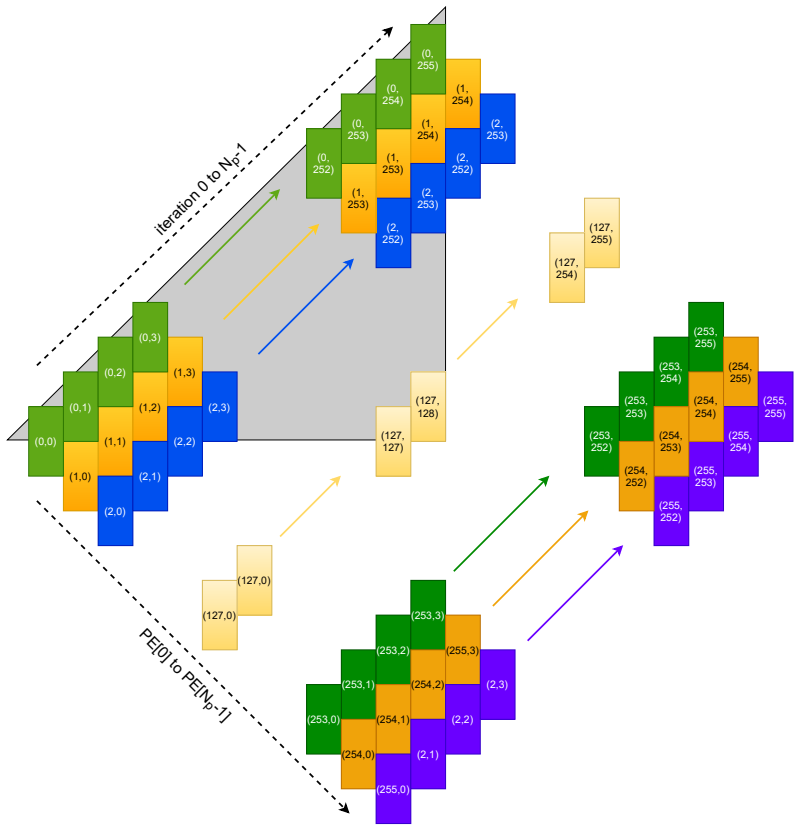


Fig. 5. An example of SCD generated from 256 PEs by processing 256×256 32-element complex demodulates.

Algorithm 2 shows how Algorithm 1 has been modified to exploit the parallelism, showing how the calculations can be performed using a systolic array implemented by a number of PEs, and how the dataflow in between adjacent PEs matches with the mapping scheme in Figure 6. Note that the outer for loop represents the PE index (the parallel architecture) and the inner for loop is the

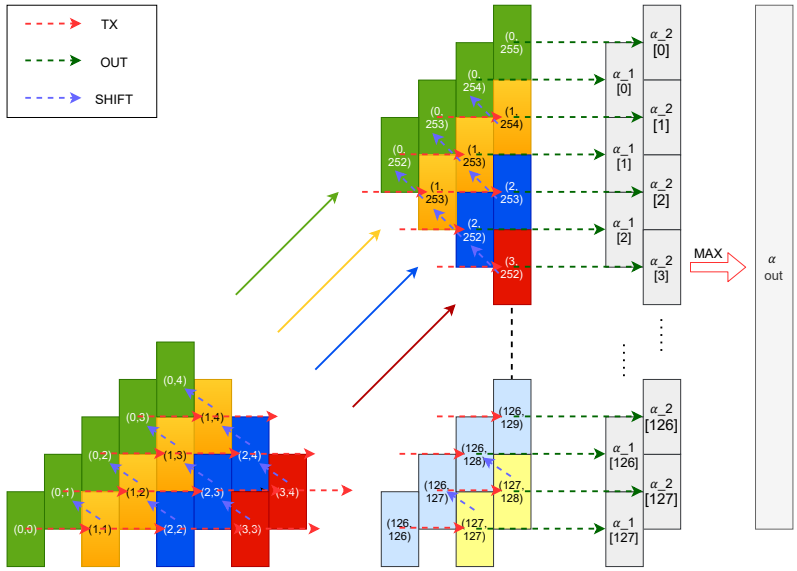


Fig. 6. An example of QSCD generated from 128 PEs by processing a quadrant of 256×256 32-element complex demodulates.

iterative sequential process of an individual PE. The parameterized number of clock cycles required by each sub-process for a single PE is also indicated.

Algorithm 2: Pseudocode for the proposed systolic array.

Input: Two matrices $X = X_D$ and $Y = X_D^*$ with a size of $N_p \times P$.
Output: A vector $alpha\_profile$ with a size of $N \times 1$.
for $i \leftarrow 0$ **to** $N_p/2 - 1$ **by** 1 **do**
  // Process of the ith PE (PE[i])
  IDLE: *wait for input data* // $(2 * P + 1) * i$ cycles
  LOAD: *load* $X[i,:]$ *and* $Y[i,:]$ // $2 * P$ cycles
  COMPUTE(i, i): *compute for* $X[i,:]$ *and* $Y[i,:]$ // $P + P\log_2 P$ cycles
  for $j \leftarrow (i+1)$ **to** $(N_p - 2 - i)$ **by** 1 **do**
    // j times iterative process on PE[i]
    TX: *transmit* $\alpha[i,:]$ *from* $PE[i]$ *to* $PE[i+1]$ // $P/2$ cycles
    SHIFT: $Y[i,:] \leftarrow Y[i+1,:]$ // P cycles
    COMPUTE(i, j): // $P + P\log_2 P$ cycles
  end
  OUT: *output* $\alpha\_1[i,:]$ // $P/2$ cycles
  COMPUTE(i, $N_p - 1 - i$)): // $P + P\log_2 P$ cycles
  OUT: *output* $\alpha\_2[i,:]$ // $P/2$ cycles
end
return $alpha\_profile \leftarrow max\{\alpha\_1, \alpha\_2\}$ // N cycles
Function COMPUTE(i, j):
  $K[i,:] \leftarrow X[i,:] * Y[j,:]$ // P cycles
  $M[i,:] \leftarrow P$-point $FFT(K)$ // $P(\log_2 P - 1/2)$ cycles
  $\alpha[i,:] \leftarrow \max\{M(i, [P/4 : 3P/4 - 1]),\ M(i-1, [P/4 : 3P/4 - 1])\}$ // $P/2$ cycles

4.2 Processing Elements

Processing elements (PEs) perform the fundamental compute operations for the iterative transitions: *TX*, *SHIFT* and *COMPUTE* in the inner loop of Algorithm 2. The datapath for a single PE is a highly pipelined, load–store architecture, as shown in Figure 7. It comprises an instruction memory (IMEM), a data memory (DMEM), a complex arithmetic ALU and a controller. The signals in bold font represent the inputs and outputs of the PE.

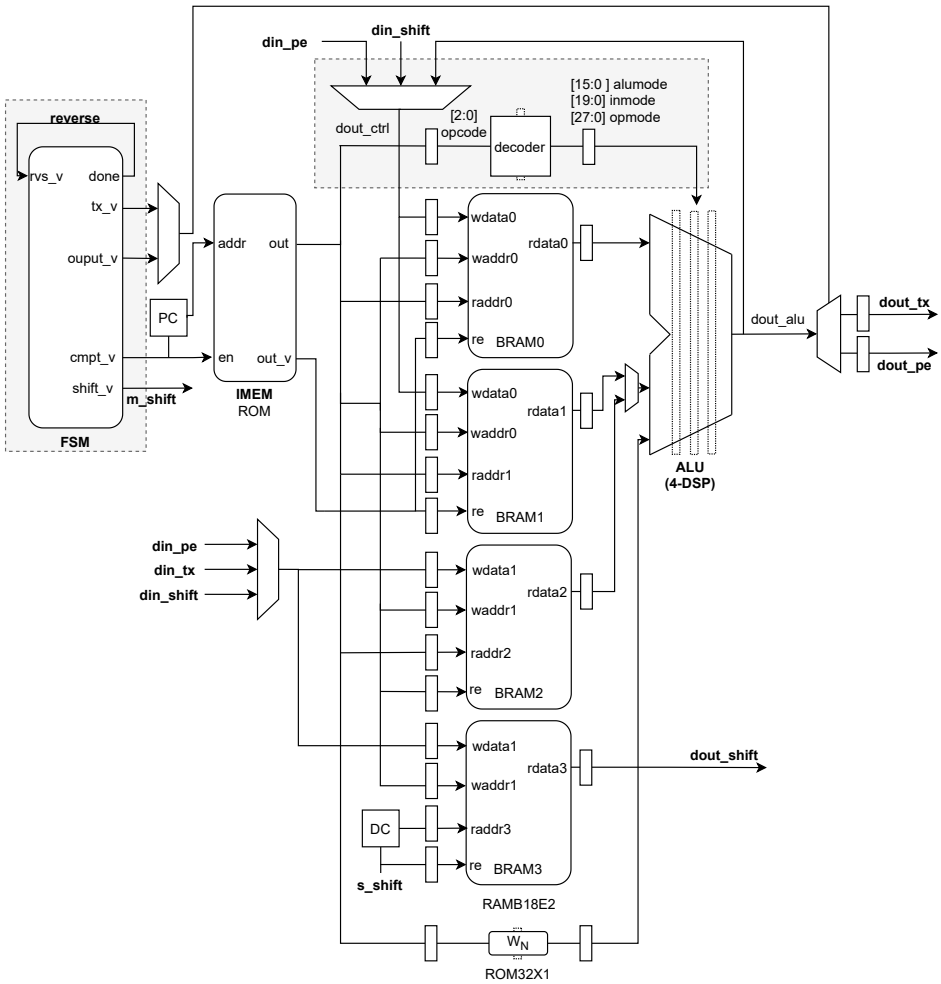


Fig. 7. Microarchitecture of the proposed PE.

4.2.1 Instruction Memory. The instruction memory (IMEM) is implemented using a LUTRAM-based ROM of size 32-bit × 192 deep, which contains all the necessary instructions for one full computation, including the complex multiplication, 32-point FFT and the partial alpha profile. Each 32-bit instruction consists of a 3-bit opcode, a single unused (reserved) bit and a 28-bit address (split into three source address fields and one destination address), as shown in Table 1. Table 2 lists some representative instructions such as the complex multiplication and the butterfly operators for

the radix-2 decimation-in-time (DIT) FFT. Note that the operands in Table 2 are all complex values and each instruction takes one clock cycle for execution. Instructions are reused for each iteration, controlled by the finite state machine (FSM).

Table 1. Instruction format.

Opcode	Reserved	Address			
arithmetic	null	source 3[1]	source 2[2]	source 1[3]	destination[4]
31:29	28	27:24	23:16	15:8	7:0

[1] Read address of ROM32X1 for source 3 in range of [0:15].
[2] Read address of BRAM1 or BRAM2 for source 2 in range of [0:255].
[3] Read address of BRAM0 for source 1 in range of [0:255].
[4] Write address of BRAM0 and BRAM1 for destination in range of [64:255].

Table 2. Instruction examples.

Instruction	Hex representation	Assembly[1]	Operation[2]
Multiplication	32'h80_20_00_40	MUL $64, $32, $0	$R64 = R32 * R0$
Butterfly (up)	32'hA0_50_40_60	MULADD $96, $64, $80, $0	$R96 = R64 + R0 * R80$
Butterfly (down)	32'hC0_50_40_61	MULSUB $97, $64, $80, $0	$R97 = R64 - R0 * R80$

[1] The format of assembly code is $Arithmetic\ RD,\ RS1,\ RS2,\ (RS3)$. $RS3$ is only valid for three-operand instructions.
[2] All the operations are in complex values and the register/memory stores a 32-bit fixed-point complex number by concatenating the 16-bit real part and the 16-bit imaginary part.

4.2.2 Complex Arithmetic ALU. The input signals from the DAC on our target board (ZCU111) are normalized 14-bit fixed-point in-phase and quadrature-phase (I/Q) samples. These are aligned on the 16-bit word boundary and grouped into a single 32-bit complex value. By pre-normalizing them to between -1 and 1, it makes it easy to keep intermediate results within this range by right shifting. For internal operations, we constrain the quantization error to 2^{-16} by using maximum bitwidth during the arithmetic operations and truncating the results in the end.

We develop a customised PE based on a RISC-like load–store architecture, which includes four complex arithmetic instructions (MUL, MAX, $MULADD$, $MULSUB$) are required for the full computation of the kernel, i.e. complex multiplication, the butterfly operators for the radix-2 DIT FFT and the calculation of the maximum of the intermediate alphas.

The complex arithmetic instructions can be decomposed into multiple real-valued multiplications with additional logic units. To achieve this, we integrate four DSP48E2 slices to build a complex ALU and multiplex the sources of the ALU with specific output logic units based on the runtime configuration. While a complex multiplication could be performed using just 3 real-valued multipliers, it requires a pre-adder on both inputs to the third multiplier, which is not supported by the DSP48E2 slice.

Figure 8 gives an example of how our ALU from Figure 7 is configured to implement the $MULADD\ RD,\ RS1,\ RS2,\ RS3$ instruction, which represents the upstream butterfly operator. For the input source, $RS1$ and $RS2$ are two 32-bit complex-valued inputs, while $RS3$ is a 32-bit complex-valued FFT twiddle factor. RD is a 32-bit complex-valued output, whose real part is generated by the upper two DSP blocks and the imaginary part is derived from the lower two DSP blocks. After the multiplication and addition/subtraction operations, a right-shift operation is added to avoid

overflow when converting back to a 16-bit result. The 16-bit real and imaginary parts are then concatenated to reconstruct a 32-bit complex number that will be written back to the data memory.

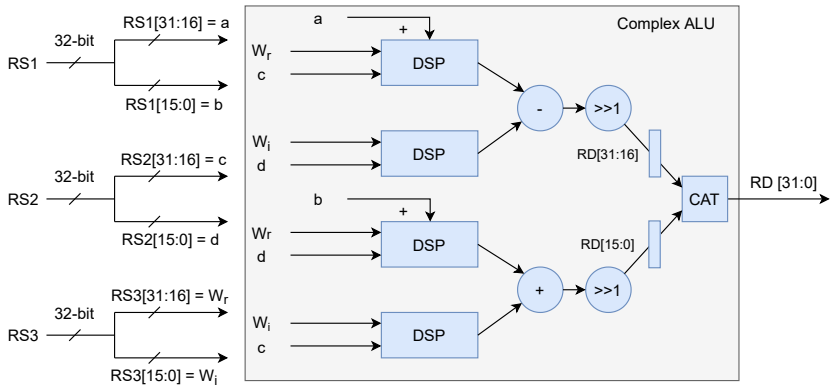


Fig. 8. The complex ALU configuration for a *MULADD* instruction.

4.2.3 Data Memory. The data memory (DMEM) consists of four RAMB18E2 primitives, to support a 2-write, 4-read RAM and 16 ROM32X1 primitives to support a 1-read ROM. The RAM is used to store the 32-bit complex-valued input signals and the intermediate results either from the complex ALU or an adjacent PE, while the ROM is used to store the 32-bit fixed-point FFT twiddle factors.

The Xilinx block RAM (BRAM) can be used as a true dual port (TDP) RAM or a simple dual port (SDP) RAM. The SDP mode is more flexible than the TDP mode when multi-port independent write/read processes are required. As the PE needs to write back the output of the 3-operand complex ALU ($dout\_alu$) to the DMEM, the DMEM needs to support at least 1-write and 3-reads. However, in our design, the partial alpha profile transmitted from the prior PE (din_tx) and the Y_i component shifted to the next PE ($dout\_shift$) could be valid during the operation of the complex ALU. This leads to the requirement for an additional independent write and read. To support this, we configure all the RAMB18E2 primitives as SDP RAMs to support up to four concurrent write/read processes. While this requires two 36Kb BRAMs, the PEs are designed such that each of the four DSP48E2 slices aligns horizontally with an 18K block RAM. This provides optimal connectivity and speed between resources within a PE [16].

4.2.4 Controller. The controller (indicated by the shaded boxes in Figure 7) consists of two parts: a) a finite state machine (FSM) for generating essential signals to determine the state of the PE for a specific iteration index, and b) a decoder which translates the 3-bit opcode into the 64-bit configurations for the four DSP blocks.

Figure 9 shows a Moore FSM with the required 8 states for an arbitrary PE. Each PE starts in the *IDLE* state. When the input signal is valid, the FSM moves to the *START* macro state which is comprised of *LOAD* and *COMPUTE* states. At the *LOAD* state, a 32-bit $\times$ 32 deep block of X and Y values are loaded into the DMEM of the PE. The FSM then moves to the *COMPUTE* state and runs the computation once. Next, there are two possible branches that can be taken, the first is to the *ITERATE* macro state which is an iterative state transition from the *TX* state to the *COMPUTE* state and the other directly jumps to the *END* macro state. The path taken depends on whether the PE has a iterative process or not (all PEs with the exception of $PE[N_p - 1]$ have *ITERATE* cycles). The *OUT* state will be triggered twice, once for the results generated by either the *START* or *ITERATE*

macro and then following the *COMPUTE* state of the *END* macro state. After the second output, the system returns to the *IDLE* state for the next input sequence.

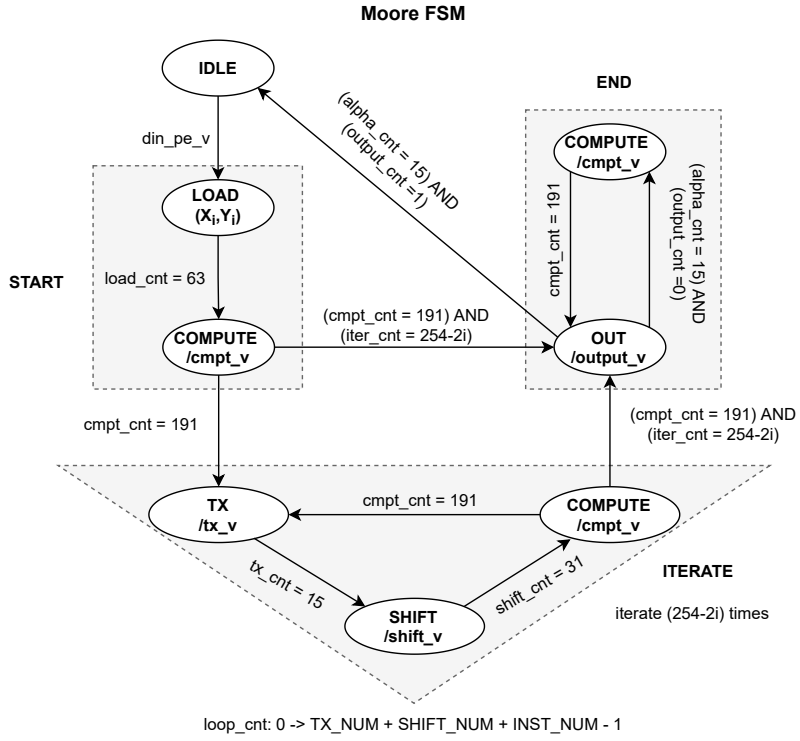


Fig. 9. The FSM of an arbitrary PE.

4.3 Single-direction Datapath Vs. Bi-direction Datapath

An analysis of the systolic array operation using a cycle-aware system flowchart is shown in Figure 10. The execution time can be estimated by determining the number of cycles in this flowchart (or from Algorithm 2). A detailed description of each state and its required number of clock cycles is explained in Table 3, which can be used as the basis for cycle estimation in the system flowchart.

Table 3. The detailed explanation of each state in the FSM.

State	Description	No. of cycles
IDLE	Wait for the input of vector X_i and Y_i	$(2P+1)*i$
LOAD	Load P-element X_i and P-element Y_i in a stream	$2P$
COMPUTE	P element-wise complex multiplication and P-element FFT	$P + P\log_2 P$
TX	Transfer $P/2$ partial alphas from PE[i] to PE[i+1]	$P/2$
SHIFT	Shift in vector Y_{i+1} from PE[i+1] to replace Y_i	P
OUT	Output $P/2$ alpha results	$P/2$

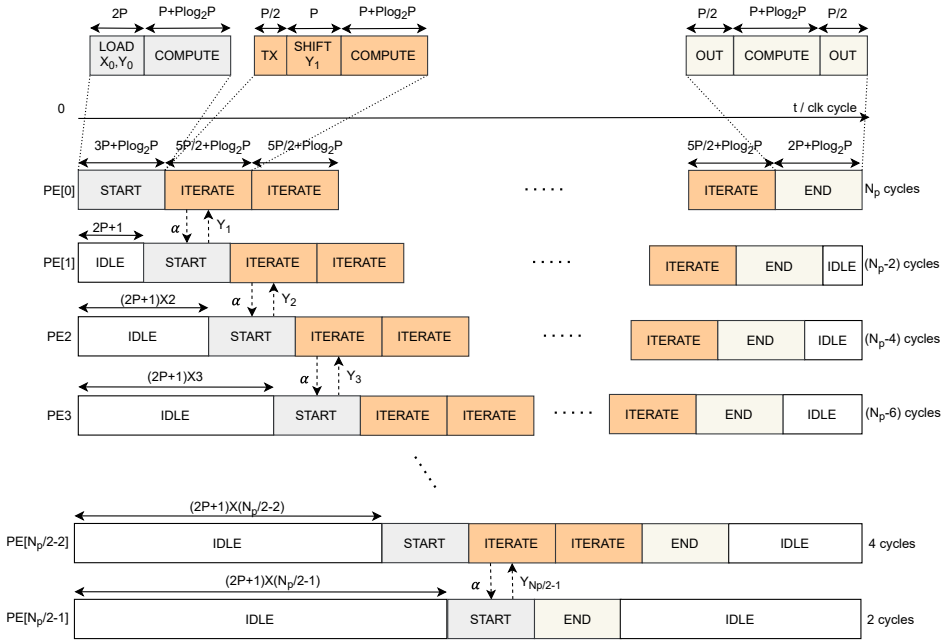


Fig. 10. A cycle-aware system flowchart.

The first PE ($PE[0]$) performs N_p different computational processes corresponding to each of the N_p element sub-sequences. These are: one *START* state ($2P$ clock cycles for the LOAD process and $(P + P\log_2 P)$ clock cycles for the first COMPUTE kernel), $(N_p - 2)$ *ITERATE* states (one *ITERATE* state comprises TX, SHIFT and COMPUTE, requiring $5P/2 + P\log_2 P$ clock cycles) and one *END* state (comprising OUT, COMPUTE and OUT, requiring $2P + P\log_2 P$ clock cycles). For each increase in the PE index, the number of computational processes is reduced by two, with the last PE, $PE[N_p/2 - 1]$, only requiring two computational processes (*START* and *END*, or just $10P + 2P\log_2 P$ clock cycles).

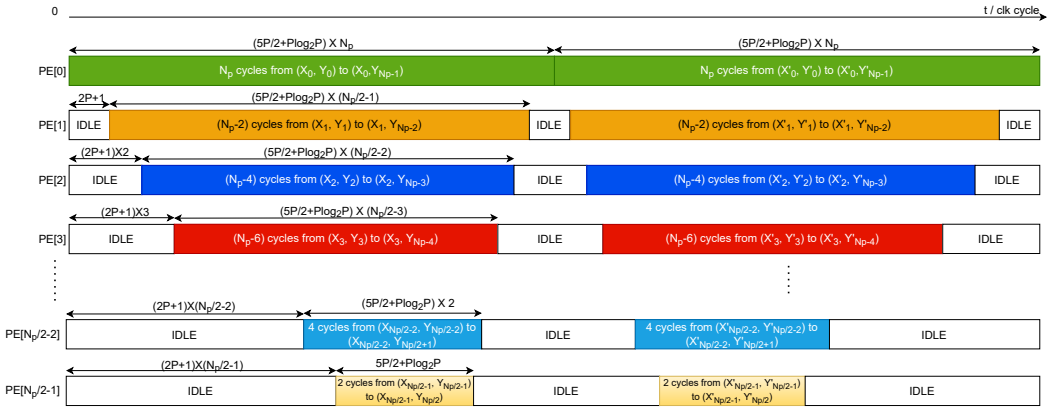
It should be clear that $PE[0]$ requires the most computation time in the system, and has an initiation interval (II) of $(5P/2 + P\log_2 P) * N_p$ clock cycles, whereas the final PE remains idle for most of the time. As a result, while the proposed systolic architecture adopts a fully pipelined implementation of the iterative process, the average PE utilization is just 50.4%. To make better use of the PE resource, we investigate two methods to balance the computation of the different PEs.

One simple solution is to fold QSCD along the anti-diagonal direction and map the additional FFT blocks to the complimentary PEs. In this way, each PE is responsible for $(N_p + 2)/2$ computational processes. While this balances the number of iterative computations across different PEs, the total number of *IDLE* cycles are unchanged and the utilization is unchanged. This is because there is a dependency in the computation that cannot be removed. For instance, $PE[N_p/2 - 1]$ has to wait for $PE[0]$ to complete $(5P/2+P\log_2 P)*(N_p/2-1)-(2P+1)*(N_p/2-1) = (P/2+P\log_2 P-1)*(N_p/2-1)$ clock cycles before it can start the remaining $(N_p/2-1)$ processes, resulting in the same PE utilization as in the original schedule in Figure 10.

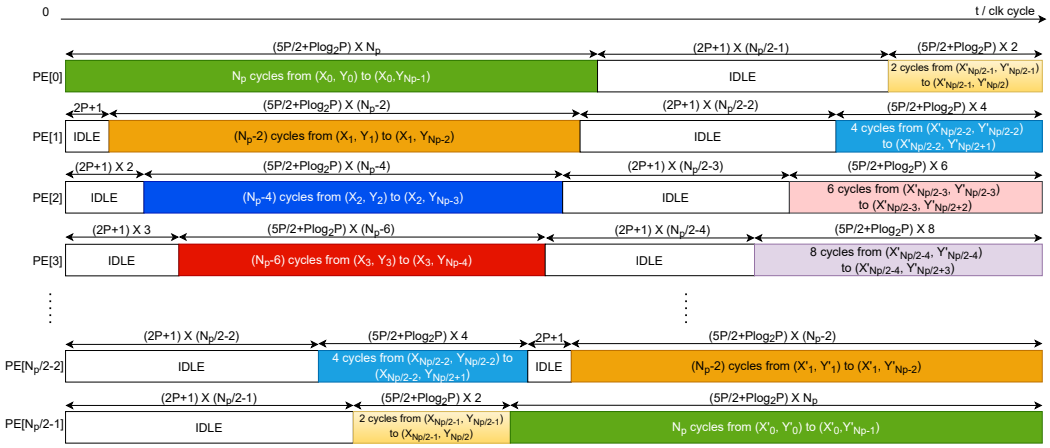
Assuming there is continual input data, a superior approach is to reverse the datapath direction when computing the SCD for independent sequences. As can be seen in Figure 10, $PE[N_p/2 - 1]$ completes its operation, then stays idle waiting for $PE[0]$ to finish. Instead, it can begin computing

the next SCD immediately. This not only balances the workload between PEs, but minimises idle time, resulting in a PE utilization of 88.2%.

We evaluate the performance of the two solutions by estimating the total clock cycles of processing two successive signals in Figure 11. It takes $(5P/2 + P\log_2 P) * (2N_p)$ clock cycles for the unidirectional datapath to complete, however, due to overlapping of computation the bi-directional datapath reduces this to only $(5P/2 + P\log_2 P) * (N_p + 2) + (2P + 1) * (N_p/2 - 1)$ clock cycles. Clearly $(5P/2 + P\log_2 P) >> (2P + 1)$, meaning the bi-directional solution is much more efficient.



(a) A unidirectional datapath for the process of two successive signals. Total cycles: $2(5P/2 + Plog_2P)N_p$



(b) A bi-directional datapath for the process of two successive signals.

Fig. 11. Parameterised flowcharts indicating the required number of clock cycles using different datapaths. Total cycles reduced to: $(5P/2 + Plog_2P)(N_p + 2) + (2P + 1)(N_p/2 - 1)$

4.4 Bi-directional Linear Systolic Array Architecture

To support the approach of Figure 11b, we perform a minor modification of our systolic array to support bi-directional dataflow, as shown in Figure 12.

The physical realisation minimises resource utilisation between adjacent PEs by using two 2-to-1 multiplexers to form the forward and reverse direct connections. The datapath direction for each

PE is then controlled by a *reverse* signal generated by the controller. In this bi-directional system design, there is no data dependency between the different input signals and the reverse datapath is just a mirror of the forward one with the PE utilisation being better balanced. Processing can start immediately after the next sub-sequence has been loaded.

The input data stream, comprising the X_i and Y_i data blocks, is FIFO buffered and connected to the input of each PE by a series of 32-bit registers. The register chain needs to be replicated and reversed to achieve the forward and reverse direction data load. To stream out the $alpha\ profile$, we need to gather the partial alpha results from each PE and combine them. As there is no data overlap, we can propagate the output in a sequential manner by connecting a 2-to-1 multiplexer plus a 32-bit register to the output of each PE. The multiplexers are controlled by the output valid signal ($dout\_pe\_v$) which is also propagated and synchronized with the $alpha\ profile$ via a series of OR logic gates and 1-bit registers. In the end, the output stream and its valid signal are sent back using a standard FIFO, similar to that of the input port. The implementation of the load and output circuits are shown at the top and bottom of Figure 12.

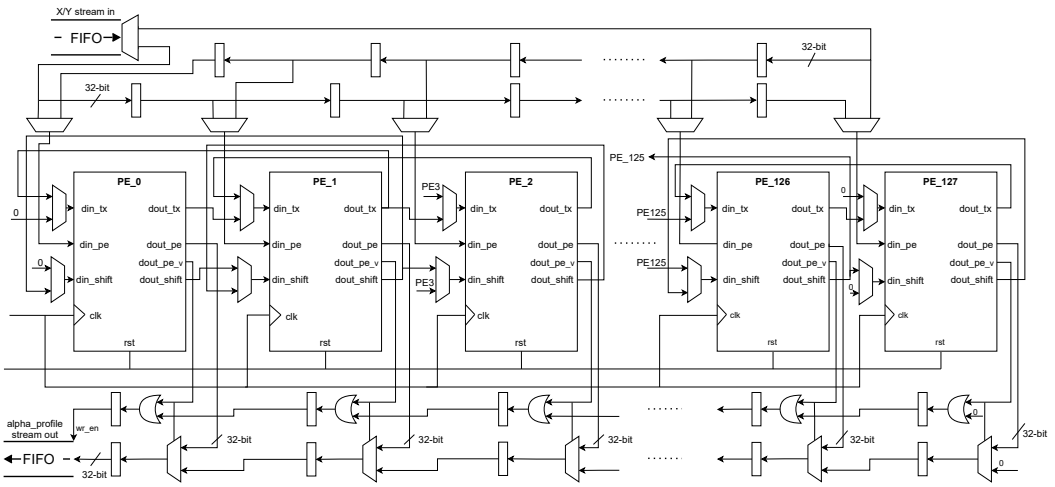


Fig. 12. A linear systolic array with bi-direction datapath support.

5 RESULTS

Most state-of-the-art parallel SCD estimators have been implemented on GPUs [5, 21]. Instead, our proposed architecture is a linear systolic array of processing elements (PEs), with multiple instruction multiple data (MIMD) parallelism. While the description that follows uses specific embedded blocks for a Xilinx UltraScale+ FPGA, we believe that an equivalent design with similar performance could be made on the Intel FPGA architecture. We evaluate the performance of our proposed linear systolic accelerator by comparing with the state-of-the-art hybrid FPGA-GPU implementation [5] and state-of-the-art GPU implementation [21], which both perform the same alpha profile calculation, in terms of resource utilization, throughput and power consumption.

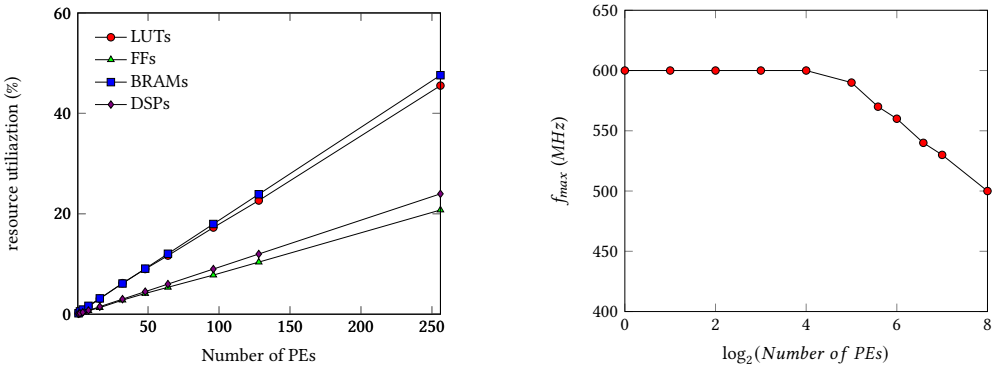
The proposed systolic array is implemented on a Zynq UltraScale+ XCZU28DR-2FFVG1517E RFSoC device using Xilinx Vivado 2020.2. A single PE can achieve a clock frequency of 600 MHz, with a resource usage of 721 LUTs, 615 FFs, 2 BRAMs and 4 DSP blocks. To validate the efficiency of our proposed PE design, a direct HLS-based compute unit is implemented. The HLS-based unit runs at a clock frequency of 500 MHz, with a much larger area overhead of 2894 LUTs, 1973 FFs,

6 BRAMs and 12 DSP block. The linear systolic array is scalable from a single PE to 256 PEs. We have chosen to focus on 128-PE array for direction comparison to the state-of-the-art GPU-based implementations [5, 21].

In terms of resources, the number of logic slices and coarse grained modules (ie. BRAMs and DSPs) grows linearly when the array size increases, as in Figure 13a. For example, the 128-PE bi-directional array consumes around 96K LUTs, 88K FFs, 258 BRAMs and 512 DSP blocks, which is less than 24% of the available resource.

In terms of clock frequency, while it achieves an f_{max} of 600 MHz for configurations from a single PE to 16, the clock frequency gradually drops to 500 MHz as the array size is increased to 256 PEs, as shown in Figure 13b. A sustained f_{max} of 530 MHz is achieved when configuring the systolic array into 128 PEs for comparison with the existing works. Table 4 summarises how the size of the proposed systolic array and the PE memory architecture scale for the FAM method, which is consistent with Algorithm 2.

A break down of the total clock cycles required by the processing of two successive signals is shown in Table 5, which can be also derived from Algorithm 2 with the specific parameters ($N = 2048$, $N_p = 256$, $L = N_p/4 = 64$ and $P = N/L = 32$). It is obvious that the PE utilization of the proposed systolic array is 88.23% as the *IDLE* state only accounts for 11.77% of the total execution time. Since FPGAs are cycle accurate, the number of cycles has been verified by simulation, the operating frequency is calculated post place-and-route, the algorithm requires a straightforward streaming input (it is not I/O constrained), and the FPGA resources are not completely exhausted, the throughput estimates are likely to be highly accurate.



(a) Area overhead.

(b) Clock frequency.

Fig. 13. Array scalability on Zynq UltraScale+ XCZU28DR-2FFVG1517E RFSoC.

Table 4. Proposed systolic array scales with parameters of FAM method.

Signal length	Array size	PE IMEM depth	QSCD size	Stream I/Os
$N = P \times N_p/4$	$N_p/2$	$P + P\log_2 P$	$2N \times 2N_p$	$3N/2$

Table 6 shows a comparison of the FPGA resource usage and operating frequency between the hybrid FPGA-GPU design [5] and our work. Table 6 also shows the total resource, in terms of LUTs, FFs, BRAMs and DSPs, available on the respective FPGA devices. From this table it can be seen that

Table 5. A break down of total clock cycles consumed by different states.

State	No. of clock cycles	Percentage
IDLE	8255	11.77%
LOAD	128	0.18%
COMPUTE	49536	70.62%
TX	4064	5.79%
SHIFT	8128	11.59%
OUT	32	0.05%
Total	70143	100%

the hybrid FPGA-GPU design uses very little FPGA resource as the FPGA is only responsible for a small portion of the full algorithm and it runs at a much lower clock frequency. In contrast, our proposed systolic array achieves the highest clock frequency at 530 MHz, and the FPGA resource usage is better proportioned to the available resource. This is due to the well-designed PEs with a fully pipelined datapath and a lightweight linear interconnect with minimum area overhead.

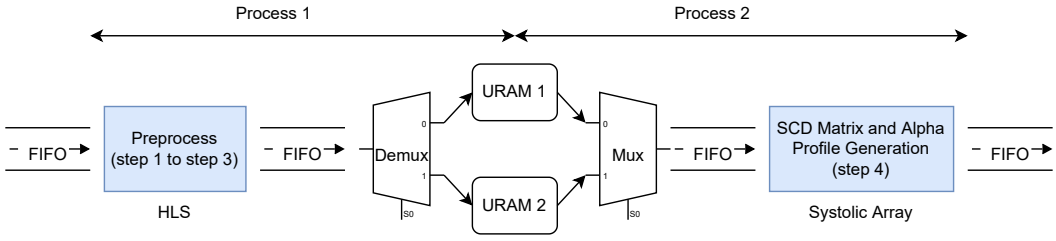
Table 6. Comparison of FPGA resource usage and operating frequency for the same configuration of FAM.

	LUTs	FFs	BRAMs	URAMs	DSPs	f_{max}
Hybrid FPGA-GPU design [5]	69 (0.1%)	153 (0.1%)	4 (2.9%)	0 (0%)	0 (0%)	140
Available on ZedBoard	*53,200*	*106,400*	*140*	*0*	*220*	–
Proposed full system	**150,802 (35.5%)**	**150,824(17.7%)**	**264 (24.4%)**[1]	**4 (5%)**[1]	**1,054 (24.7%)**	**530**
– 128-PE array	96,259 (22.6%)	88,239 (10.4%)	258 (23.9%)	0 (0%)	512 (12.0%)	530
– HLS implementation	54,543 (12.8%)	62,585 (7.3%)	4 (0.4%)	0 (0%)	542 (12.7%)	530
Available on ZCU111	*425,280*	*850,560*	*1,080*	*80*	*4,272*	–

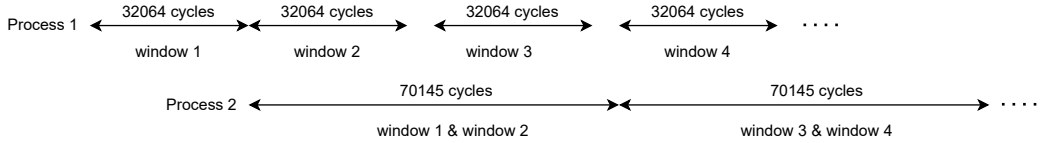
[1] The addtional 2 BRAMs and 4 URAMs are introduced by the buffering circuits between the HLS implementation and 128-PE array.

Table 7 gives a comparison of our work with state-of-the-art GPU-based implementations. All the listed works share the same configuration of FAM and use alpha profile of the SCD/QSCD function as the output. To ensure a fair comparison, the preprocess (step 1 to step 3) of the FAM algorithm is also implemented in Vitis HLS 2020.2 and connected to our proposed systolic array with a double buffering scheme, as shown in Figure 14a. Two UltraRAM (URAM) blocks are used to buffer the adjacent windows from the Preprocess module, ensuring a fully pipelined process in the systolic array. According to Figure 14b, throughput of the proposed architecture is determined by process 2 which is mapped on the systolic array. In this paper we use the term windows/s as it better describes that we are dealing with windowed signals of size 2048 samples/window. This is similar to the use of signals/s in [5, 21]. The proposed systolic array achieves a throughput of 15340 windows/s, which is 4.65× better than the fastest GPU implementation and 807× better than the hybrid FPGA-GPU implementation. While the proposed systolic accelerator may not be the lowest power implementation, its power consumption is significant less than the throughput oriented GPU implementation. It also achieves the best energy efficiency at 4832 million operations per Watt (MOPS/W), which is 20.6× better than the highest throughput GPU implementation. While the latest GPU work [21] adopts a Tesla K40 which was dated to 2013, one could perform a rough performance estimate based on the following (K40: 2880 CUDA cores, base clock 745 MHz. RTX 3080 Ti: 8960 CUDA cores, base clock 1365 MHz). It follows that one could estimate a potential performance improvement by a factor of 5.5×, which would be in line with the performance

achieved on our FPGA, albeit at a maximum power draw of 350W. This implies our FPGA design is still likely to significantly outperform the GPU in performance per watt.



(a) The double buffering scheme between HLS implementation and the systolic array.



(b) The cycles of process 1 (HLS implementation) and process 2 (systolic array).

Fig. 14. Block diagram and timing diagram of the proposed full system.

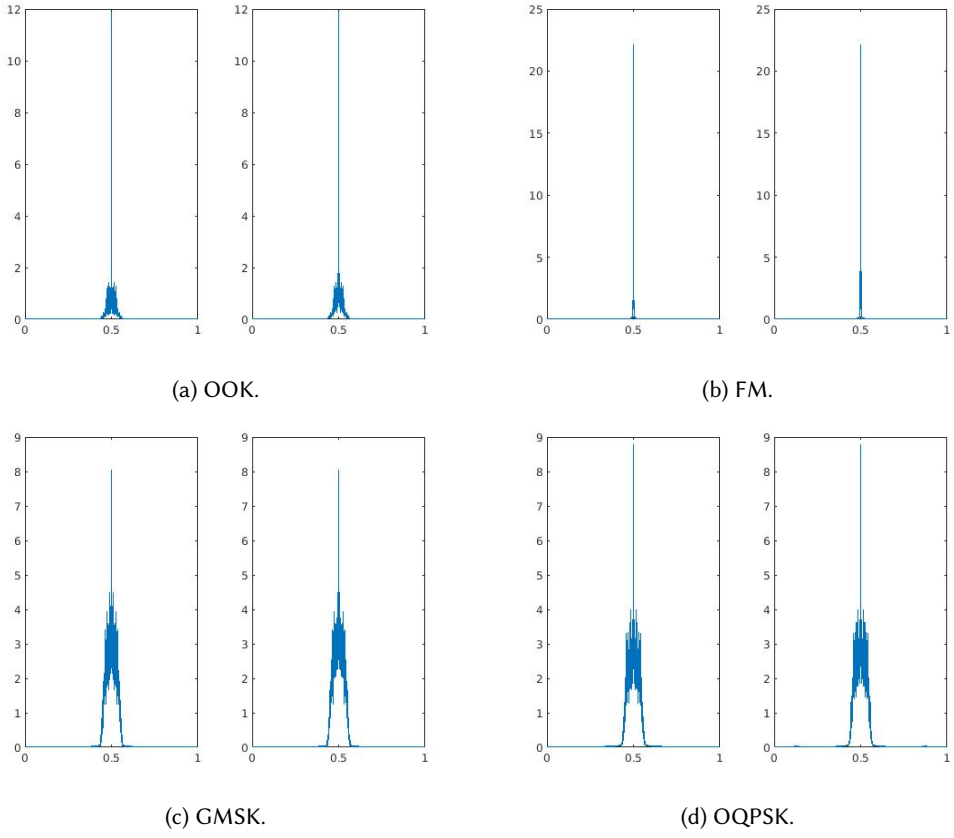
Table 7. Comparison of throughput and power consumption for the same configuration of FAM.

	GPU [5]	GPU [5]	GPU [21]	FPGA+GPU [5]	Ours
Platform	Tegra K1	Tesla K20	Tesla K40	ZedBoard+Tegra K1	ZCU111
Initiation Interval (ms)	111.61	8.98	0.303	50.95	**0.065**
Throughput (windows/s)	9	111	3300	19	**15340**
Speedup	–	12.3	366.7	2.1	**1704.4**
Computational Performance (GOPS)	0.14	1.75	13.0	0.30	**60.4**
Power (W)	3.5	51	55.5[1]	5	12.5[2]
Energy Efficiency (MOPS/W)	40	34	234	60	**4832**

[1] The power consumption is estimated by scaling to the result of [5].
[2] The power consumption is calculated from Vivado.

We evaluate the proposed systolic array across the DeepSig RADIOML 2018.01A dataset [15] and demonstrate the alpha profile results of four different signals generated by an FPGA implementation of the systolic array and those generated by MATLAB in Figure 15. In each subfigure, the plot on the left is double precision floating point output generated by MATLAB and the plot on the right is the 16-bit fixed-point output obtained from the systolic array. Intuitively, the distribution of the alpha profile coming from the fixed-point systolic accelerator is consistent with that of the "golden" MATLAB floating point output for the same modulation type. To evaluate the accuracy of alpha profile with different scales, we calculate the normalized root mean squared error (NRMSE) of each signal in Equation (10) and get an average number of 0.0148.

$$NRMSE = RMSE/[max(alpha) - min(alpha)] \tag{10}$$



(a) OOK. (b) FM.

(c) GMSK. (d) OQPSK.

Fig. 15. The alpha profile of four different signal types.

6 CONCLUSION

In this paper, we presented an optimised implementation of the FAM method to compute the SCD/QSCD function and its alpha profile. A scalable linear systolic array of programmable PEs is proposed as an FPGA accelerator, which operates at a clock frequency of 530 MHz and, through massive parallelism, achieves a sustained PE utilization of 88.2% with a bi-directional datapath. The proposed systolic implementation achieves a significant 807× throughput improvement over a hybrid FPGA-GPU implementation and a speedup of 4.65× over the state-of-the-art GPU implementation. It also attains the best energy efficiency at 4832 MOPS/W, which is 20.6× better than the highest throughput GPU implementation. The alpha profile outputs are accurate and the average NRMSE over all signals is 0.0148.

Though this work is customized for the alpha profile computation of QSCD algorithm, our proposed systolic array can be adapted to different applications such as CNNs by changing the instruction set on the PEs. In future work we plan to generalise the systolic array presented as a similar architecture can be used for other multiply-add intensive problems including: deep neural network inference and training, compressed sensing and signal/image compression. We also plan to study how to integrate this accelerator as a feature extractor within a deep learning

approach for real-time cyclostationary analysis of radio-frequency signals to enhance modulation classification [28].

ACKNOWLEDGMENTS

This research was funded by the Ministry of Education (MOE), Singapore under grant MOE2017-T2-1-002.

REFERENCES

[1] Jérôme Antoni. 2007. Cyclic spectral analysis in practice. *Mechanical Systems and Signal Processing* 21, 2 (2007), 597–630.
[2] Jérôme Antoni. 2009. Cyclostationarity by examples. *Mechanical Systems and Signal Processing* 23, 4 (2009), 987–1036.
[3] Jerome Antoni and David Hanson. 2012. Detection of surface ships from interception of cyclostationary signature with the cyclic modulation coherence. *IEEE Journal of Oceanic Engineering* 37, 3 (2012), 478–493.
[4] Jérôme Antoni, Ge Xin, and Nacer Hamzaoui. 2017. Fast computation of the spectral correlation. *Mechanical Systems and Signal Processing* 92 (2017), 248–277.
[5] Nilangshu Bidyanta, G Vannhoy, M Hirzallah, A Akoglu, B Ryu, and T Bose. 2015. GPU and FPGA based architecture design for real-time signal classification. In *Proceedings of the 2015 Wireless Innovation Forum Conference on Wireless Communications Technologies and Software Defined Radio (WInnComm'15)*. Springer, San Diego, CA, 70–79.
[6] David Boland. 2016. Reducing memory requirements for high-performance and numerically stable gaussian elimination. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 244–253.
[7] P Borghesani and J Antoni. 2018. A faster algorithm for the calculation of the fast spectral correlation. *Mechanical Systems and Signal Processing* 111 (2018), 113–118.
[8] William A Brown and Herschel H Loomis. 1993. Digital implementations of spectral correlation analyzers. *IEEE Transactions on signal processing* 41, 2 (1993), 703–720.
[9] Evandro L Da Costa. 1996. *Detection and Identification of Cyclostationary Signals.* Technical Report. NAVAL POST-GRADUATE SCHOOL MONTEREY CA.
[10] William A Gardner. 1986. The spectral correlation theory of cyclostationary time-series. *Signal processing* 11, 1 (1986), 13–36.
[11] William A Gardner. 1989. *Statistical Spectral Analysis: A Nonprobabilistic Theory.* Prentice-Hall, Englewood Cliffs, NJ.
[12] William A Gardner. 1994. *Cyclostationarity in communications and signal processing.* IEEE Press, New York.
[13] William A Gardner, Antonio Napolitano, and Luigi Paura. 2006. Cyclostationarity: Half a century of research. *Signal processing* 86, 4 (2006), 639–697.
[14] Feng Ge and Charles W Bostian. 2008. A parallel computing based spectrum sensing approach for signal detection under conditions of low snr and rayleigh multipath fading. In *2008 3rd IEEE Symposium on New Frontiers in Dynamic Spectrum Access Networks*. IEEE, Chicago, IL, 1–10.
[15] DeepSig Inc. 2018. RF Datasets For Machine Learning. https://www.deepsig.ai/datasets.
[16] Xilinx Inc. 2020. UltraScale Architecture DSP Slice User Guide.
[17] Sun Yuan Kung. 1988. VLSI array processors. *Englewood Cliffs* (1988).
[18] Chu-Han Lee, Chia-Jen Chang, and Sao-Jie Chen. 2012. Parallelization of spectrum sensing algorithms using graphic processing units. In *CSQRWC 2012*. IEEE, New Taipei, Taiwan, 35–39.
[19] Gaye Lightbody, Roger Woods, and Richard Walke. 2003. Design of a parameterizable silicon intellectual property core for QR-based RLS filtering. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11, 4 (2003), 659–678.
[20] Scott Marshall, Garrett Vanhoy, Ali Akoglu, Tamal Bose, and Bo Ryu. 2018. GPU based quarter spectral correlation density function. In *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, Porto, Portugal, 88–93.
[21] Scott Marshall, Garrett Vanhoy, Ali Akoglu, Tamal Bose, and Bo Ryu. 2020. GPGPU based parallel implementation of spectral correlation density function. *Journal of Signal Processing Systems* 92, 1 (2020), 71–93.
[22] Alexandru Martian, Bogdan Tudor Sandu, Octavian Fratu, Ion Marghescu, and Razvan Craciunescu. 2014. Spectrum sensing based on spectral correlation for cognitive radio systems. In *2014 4th International Conference on Wireless Communications, Vehicular Technology, Information Theory and Aerospace & Electronic Systems (VITAE)*. IEEE, 1–4.
[23] Charles M Rader. 1996. VLSI systolic arrays for adaptive nulling [radar]. *IEEE Signal Processing Magazine* 13, 4 (1996), 29–49.
[24] Barathram Ramkumar. 2009. Automatic modulation classification for cognitive radios using cyclic feature detection. *IEEE Circuits and Systems Magazine* 9, 2 (2009), 27–45.
[25] Randy S Roberts, William A Brown, and Herschel H Loomis. 1991. Computationally efficient algorithms for cyclic spectral analysis. *IEEE Signal Processing Magazine* 8, 2 (1991), 38–49.

[26] Steven R Schnur. 2009. *Identification and classification of OFDM based signals using preamble correlation and cyclostationary feature extraction.* Technical Report. NAVAL POSTGRADUATE SCHOOL MONTEREY CA.

[27] Dorde C Simic and JR Simic. 1999. The strip spectral correlation algorithm for spectral correlation estimation of digitally modulated signals. In *4th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services. TELSIKS'99 (Cat. No. 99EX365)*, Vol. 1. IEEE, Nis, Yugoslavia, 277–280.

[28] Stephen Tridgell, David Boland, Philip HW Leong, Ryan Kastner, Alireza Khodamoradi, and Siddhartha. 2020. Real-time Automatic Modulation Classification using RFSoC. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, New Orleans, LA, 82–89.

[29] Shixian Wang, Botao Zhang, Hengzhu Liu, and Lunguo Xie. 2010. Parallelized cyclostationary feature detection on a software defined radio processor. In *2010 International Symposium on Signals, Systems and Electronics*, Vol. 1. IEEE, Nanjing, China, 1–4.

[30] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[31] Peter Welch. 1967. The use of fast Fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms. *IEEE Transactions on Audio and Electroacoustics* 15, 2 (1967), 70–73.

[32] Lauren J Wong, William H Clark IV, Bryse Flowers, R Michael Buehrer, Alan J Michaels, and William C Headley. 2020. The RFML Ecosystem: A Look at the Unique Challenges of Applying Deep Learning to Radio Frequency Applications. *arXiv preprint arXiv:2010.00432* (2020).

[33] Wei Zhang, Vaughn Betz, and Jonathan Rose. 2012. Portable and scalable FPGA-based acceleration of a direct linear system solver. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 5, 1 (2012), 1–26.