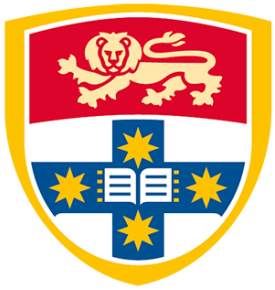
High-Speed Implementations of Spectral Correlation Analysis Techniques

CAROL JINGYI LI

M.Eng



THE UNIVERSITY OF SYDNEY

Supervisor: Prof. Philip H.W. Leong
Associate Supervisor: Dr. David Boland

A thesis submitted in fulfilment of
the requirements for the degree of
Doctor of Philosophy (PhD)

School of Electrical and Computer Engineering
Faculty of Engineering
The University of Sydney
Australia

24 April 2025

Declaration

I, *Miss Jingyi Li*, declare that this thesis is submitted to fulfill the requirements for the conferral of the degree *Doctor of Philosophy (PhD)*, from the University of Sydney, is wholly my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualifications at any other academic institution.

Jingyi Li, 24 April 2025

Abstract

Spectral correlation density (SCD) is widely used to characterize cyclostationary signals, but its high computational requirements pose significant challenges. Although the fast Fourier transform (FFT) enables efficient methods such as the FFT accumulation method (FAM) and strip spectral correlation analyzer (SSCA), their real-time adoption remains limited. Therefore, optimizing these methods for computational efficiency is essential to balance accuracy and efficiency. This work focuses on reducing wordlength, leveraging parallel hardware architectures, and minimizing computational complexity.

This study first analyzes the relationship between wordlength and signal-to-quantization-noise ratio (SQNR) in fixed-point SCD implementation, using a canonical FAM-based estimator with both fixed- and mixed-point arithmetic. High performance is achieved on the AMD/Xilinx Zynq UltraScale+ RFSoC ZCU111 platform by exploiting spatial parallelism, pipelining, I/O optimization, and algorithmic symmetry.

Next, an SSCA implementation is proposed for large datasets on the AMD/Xilinx Versal VCK5000 platform. It utilizes parallelism across the programmable logic (PL), double data rate memory controller (DDRMC), and artificial intelligence engine (AIE), and combines very-long instruction word (VLIW) and single-instruction multiple-data (SIMD) architectures. The PL handles data transfer between the DDRMC and AIE, ensuring seamless communication. This architecture maximizes hardware efficiency and throughput for large-scale processing.

Lastly, the sparse strip spectral correlation analyzer (S^3CA) is introduced, employing the sparse fast Fourier transform (SFFT) to leverage the sparsity of the cyclic spectrum in practical signals. It computes a sparse, downsampled channel-data product (CDP) and applies a modified SFFT to estimate the SCD efficiently. This approach reduces computation and enhances scalability, enabling real-time spectral analysis of large datasets.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Professor Philip H.W. Leong, whose expertise, patience, and unwavering support have guided me throughout my PhD journey. His invaluable advice and insights have been instrumental in shaping this work, and working alongside him has been an immensely rewarding experience. I feel truly fortunate to have been his student.

I am also incredibly thankful to my co-supervisor, David Boland, for his valuable guidance and insightful advice over the years. I would like to extend my gratitude to Richard Rademacher, Craig T. Jin, Chad M. Spooner, and Xiangwei (Louis) Li for their helpful discussions, valuable advice, and ongoing encouragement.

My heartfelt thanks go to my colleagues from the Computer Engineering Lab (CEL), Xueyuan Liu, Huiyuan (June) Sun, Binglei Lou, and Wenjie Zhou, for their friendship, stimulating discussions, and for creating an environment filled with laughter and mutual support. It has been a privilege to share this journey with such a dedicated group of researchers.

I am also grateful to the Research Training Program (RTP) for providing the financial support that made this research possible. I'd also like to thank Barry Flower, Philip H.W. Leong, and Stephen Tridgell for the opportunity to undertake an internship at CruxML, supported by the Defence Innovation Network (DIN) internship program. Further, thanks go to Richard Rademacher for helping to review this thesis.

I would also like to acknowledge the support of AMD/Xilinx and, in particular, their University Program. Joe Peng, Joshua Lu, and Wen Chen donated a Versal VCK5000 board and were very helpful in answering many of my questions.

I also thank the dissertation examiners, Professor Ali Akoglu from the University of Arizona and Professor S. C. Chan from the University of Hong Kong, whose insightful feedback greatly strengthened this study.

Lastly, I thank my parents for their tremendous support throughout my PhD and my entire life, especially during the challenging times of COVID. Their patience, love, and constant encouragement have been invaluable.

Authorship Attribution Statement

This thesis contains material that has been previously published or prepared for publication during my PhD under the supervision of Professor Philip H.W. Leong. The ideas and preparation behind each publication, primarily my own work under the oversight of my supervisors and with all assistance that I received, stated and acknowledged below:

- Professor Philip H.W. Leong provided overall research direction and guidance throughout my PhD.
- Chapter 3. Discussions assisted the quantization error analysis and hardware implementation aspects with David Boland and Xiangwei Li. I measured the quantization error analysis for FAM and verified the results. Additionally, I designed the fastest state-of-the-art field-programmable gate array (FPGA) implementation for FAM.
- Chapter 4. The concept of implementing SSCA on the AMD/Xilinx VCK5000 was suggested by Professor Philip H.W. Leong. I developed the AIE implementation for strip spectral correlation analyzer utilizing a decomposition FFT (SSCA_2DFFT) and designed the PL to manage the dataflow between the DDRMC and AIE.
- Chapter 5. The idea of S^3CA originated from Professor Philip H.W. Leong and was further developed with the assistance of David Boland, Richard Rademacher, Craig T. Jin, and Chad M. Spooner. I designed the S^3CA to meet the input requirement of SFFT, focusing specifically on computing only the necessary sparse inputs to reduce the computational overhead. The implementation ensures sufficient randomness while maintaining controllability. Additionally, I developed the C/C++ implementation to evaluate the performance compared with SSCA for large datasets ranging from 2^{16} to 2^{24}.
- The writing and presentation of each publication were improved through discussion and feedback from all co-authors.

In addition to the statements above, in cases where I am not the corresponding author of a published item, the corresponding author has granted permission to include the published material.

Jingyi Li, 24 April 2025

As supervisor for the candidature upon which this thesis is based, I can confirm that the authorship attribution statements above are correct.

Philip H.W. Leong, 24 April 2025

Contents

List of Symbols

T_s	sampling period
N	number of samples in a window
$x(t)$	continues time-series
$x(n)$	discrete time-series
$\hat{x}(f)$	Fourier transform of x
N_P	$2^{\log_2(T_s/\Delta f)}$
Δf	frequency resolution
$\Delta\alpha$	cycle frequency resolution
G_*	gain of sections ($*$ refers to section)
$P_{i.s}$	power of input signal
L	$N_P/4$ in FAM, 1 in SSCA and S^{3}CA
P	$2^{\log_2(T_s/\Delta\alpha/L)} = \lfloor N/L \rfloor$
m_1	$\log_2(N_P)$
m_2	$\log_2(P)$
m_3	$\log_2(N)$
A_1	$2 - \frac{m_1+1.5}{N_P}$
A_2	$2 - \frac{m_2+1.5}{P}$
A_3	$2 - \frac{m_3+1.5}{N}$
B_1	$\frac{2^{m_1}}{6} - 1$
B_2	$\frac{2^{m_2}}{6} - 1$
B_3	$\frac{2^{m_3}}{6} - 1$
X_T	complex demodulate
X_g	channel-data product
q_1	normalization coefficient for the first FFT
q_2	normalization coefficient for the conjugate multiplication
q_3	normalization coefficient for the second FFT
$\mathbf{B}$	Number of Buckets
$\mathbb{B}$	Bandwidth

List of Figures

List of Tables

CHAPTER 1

Introduction

A time series is said to be *cyclostationary* if its probability distribution varies periodically with time. Cyclostationary time series analyses are suitable for a wide range of periodic phenomena in signal processing, including characterization of modulation types; noise analysis of periodic time-variant linear systems; synchronization problems; parameter and waveform estimation; channel identification and equalization; signal detection and classification; autoregressive (AR) and autoregressive moving average (ARMA) modeling and prediction; and source separation [31, 57, 48]. A signal exhibits cyclostationarity if and only if the signal is correlated with certain frequency-shifted versions of itself [30]. Cyclostationary analysis often involves computing the spectral correlation density (SCD), also called the cyclic spectral density or spectral correlation function. This function is the idealized temporal cross-correlation between all pairs of narrowband spectral-component time series and reflects the correlation distribution of the signal in terms of both spectral frequency and cycle frequency.

Based on the SCD algorithm, the computation of the entire cyclic spectrum (CS) is considerably more complex and time-consuming than the computation of the conventional power spectral density (PSD). The estimation of correlation between spectral components of signals, as opposed to merely computing the spectral components themselves, renders the SCD a computationally complex mechanism. This increased complexity arises primarily from the potentially large number of correlation computations required.

The estimation of SCD algorithm for the efficient algorithm is in two main subgroups: the frequency-smoothing (FS) algorithms and the time-smoothing (TS) algorithms. If we want to compute the entire CS for blind estimation, TS algorithms result in more computationally efficient algorithms than those involving FS [58].

1.1 Motivation

Although the SCD method reveals extensive information about cyclostationary processes even under low signal-to-noise ratio (SNR) conditions, the high computational complexity of the method poses challenges for real-time applications, particularly due to the large input sizes often involved. Although the fast Fourier transform (FFT) improves the computational efficiency of TS algorithms—such as the FFT accumulation method (FAM) and the strip spectral correlation analyzer (SSCA)—achieving real-time performance remains challenging. Consequently, there has been significant interest in developing high-performance implementations of the SCD method to detect and classify cyclostationary signals using central processing units (CPUs), graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and artificial intelligence engines (AIEs).

In real-time applications, FPGA-based SCD accelerators have gained popularity due to their utilization of on-chip high-speed arithmetic primitives in digital signal processing (DSP) and memory blocks, along with their higher power efficiency, making them suitable for remote, portable, real-time applications. However, achieving high resource utilization efficiency and placement to attain maximum speed remain critical challenges for researchers in this field.

Signal-to-quantization-noise ratio (SQNR) and speed are critical factors in the implementation of real-time SCD algorithms. Achieving the balance between these factors is challenging due to high computational complexity and power requirements. Current literature and implementations have gaps that need addressing. Specifically, the problems to be solved are:

Theoretical Analysis of SQNR for Changing Precision:
When applying SCD algorithms in hardware, one general way to reduce computational complexity and increase speed is to shift from floating to fixed-point precision. The fixed-point precision can also reduce the amount of intermediate memory required by utilizing shorter wordlength, such as reducing from 24 bits to 12 bits. However, this reduction in precision introduces quantization errors, which can degrade the SQNR of the output. The tradeoff between SQNR and precision becomes a significant design consideration. In the design of the real-time SCD implementation, it is necessary to develop a detailed theoretical

and analytical model of the relationship between the different accuracies and precision of the different parts of the algorithm. Such an analysis will help determine the optimal balance for different conditions, ensuring that the algorithm runs as quickly as possible while maintaining a level of SQNR that is within acceptable limits.

Custom FPGA Data Paths for Improved Parallelism and Speed:
Traditional implementations of SCD algorithms on CPUs and GPUs are constrained by the fixed architectures of these platforms. In contrast, FPGAs offers a flexible architecture that can be customized for specific applications. This flexibility allows for the design of custom data paths that can enhance parallelism and improve computational speed.

A study is needed to explore how custom FPGAs data paths can be designed to maximize the efficient use of FPGAs resources. This involves understanding the specific ways in which FPGAs customization can enhance the performance of SCD algorithms. By leveraging the on-chip high-speed arithmetic primitives present in DSP and memory blocks of FPGAs, it is possible to achieve significant improvements in speed and parallelism. Understanding these design principles will help in developing FPGA-based SCD accelerators that outperform their CPU and GPU counterparts.

The AMD/Xilinx Versal ACAP architecture, described in merges general-purpose CPUs, programmable logic (PL), and AIE processors optimized for AI and machine learning optimization. With 400 AI Engine processors executing at 1 GHz, capable of delivering 8 MACs/cycles for 32-bit floating-point data, the implementation of SCD on the Versal ACAP-using multiple AIEs and collaborating with PL and CPU-can lead to more efficient SCD accelerations.

Impact of Sparsity Assumption on Speed:
In SCD estimation algorithms, fixed-precision is often used uniformly throughout the design [46]. However, this approach may not be optimal for all applications. Introducing a sparsity assumption can potentially improve speed. The sparsity assumption posits that the data or the signal of interest is sparse, meaning that it contains a significant number of zero or near-zero elements. This assumption can be leveraged to reduce computational complexity. Recent

developments, such as the sparse fast Fourier transform (SFFT) designed by MIT [42], offer promising techniques for reducing computation time while focusing on the most significant variables. In the context of SCD algorithms, the symbol rate and carrier frequency are the most significant elements after integration, the rest of the variables are near-zero elements. A study is necessary to examine how to combine the FFT with SSCA, as the SFFT only requires parts of input compared with FFT and we can customize the SSCA only support those input data to SFFT to reduce the computational complexity of SSCA.

1.2 Aims

The primary objective of this research was thus to develop a scalable, high-speed solution for cyclostationary analysis using FPGAs. The proposed solution involves designing an FPGA-based accelerator that leverages on-chip arithmetic and memory blocks, alongside AIEs to enhance performance and manage computational complexity. This research bridges the gap between theoretical cyclostationary analysis and practical, real-time application, enhancing the capability of radio frequency (RF) signal processing under constraints of speed and accuracy.

The specific aims of this work were:

(1) Develop new techniques to theoretically analyze quantization error and hence exploit the trade-offs between SQNR and resource utilization.
(2) Develop novel spatial architectures for parallel estimation of the SCD and implement them on FPGAs.
(3) Develop SCD estimation techniques with reduced time complexity by exploiting sparsity.

1.3 Contributions

Our contributions include a theoretical model that enables analysis of the trade-offs between SQNR and precision, the exploration of customized FPGAs data paths, and taking advantage

of sparsity to improve computational complexity. Additionally, we provide open-source code for each method[1] [2] [3].

Quantization Error Analysis: [44]

The first analytical SQNR model for fixed-point implementations of the FAM for estimating SCD, enabling better tradeoffs between precision and area. Based on this model, discuss the quantitative comparison of multiple wordlength assignment strategies and verify with hardware implementations.

FAM Implementation on FPGAs: [44]

Design an architecture for implementing FAM on FPGA with high parallelism and mixed precision to be used throughout. This high-level synthesis (HLS) implementation achieves the highest reported throughput and power performance for the FAM techniques with a sparse matrix output to minimize accelerator-to-host bandwidth.

SSCA Implementation on the Versal AI Engine:

Design the first implementation of SSCA on the Versal ACAP platform, utilizing very-long instruction word (VLIW) and single-instruction multiple-data (SIMD) vector processors for acceleration. The AIE's parallel processing capabilities enhance SSCA performance significantly. The design integrates communication between the double data rate memory controller (DDRMC), PL, and AIE. The DDRMC manages high-speed data access, while the PL coordinates dataflow and preprocessing tasks. Together, these components optimize memory handling and computational efficiency, enabling a high-performance SSCA solution that leverages the advantages of Versal's architecture to traditional CPU- and GPU-based implementations.

The sparse strip spectral correlation analyzer (S^3CA): [45]

Developed a novel S^3CA algorithm based on the SFFT. The S^3CA approach involves computing a sparse, downsampled channel-data product, which is then passed to a modified SFFT implementation to obtain the spectral density. The computational complexity of the SSCA

[1] https://github.com/Jingyi-li/FAM_Synthesis.git
[2] https://github.com/Jingyi-li/SSCA_Implementation.git
[3] https://github.com/Jingyi-li/S3CA.git

is reduced from $O(NN_P(\log N_P + \log N))$ to $O(N_P \log N_P \log N \sqrt[3]{N\kappa^2 \log N})$ (refer to Chapter 5).

Publications arising from this research were:

- Chapter 3: [44] Carol Jingyi Li, Xiangwei Li, Binglei Lou, Craig T. Jin, David Boland, and Philip H.W. Leong. 2023. Fixed-point FPGA Implementation of the FFT Accumulation Method for Real-time Cyclostationary Analysis. ACM Trans. Reconfigurable Technol. Syst. 16, 3, Article 41 (September 2023), 28 pages. https://doi.org/10.1145/3567429.
- Chapter 3: [46] 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. Reconfigurable Technol. Syst. 16, 1, Article 9 (March 2023), 24 pages. https://doi.org/10.1145/3546181.
- Chapter 5: [45] Carol Jingyi Li, Richard Rademacher, David Boland, Craig T. Jin, Chad M. Spooner and Philip H.W. Leong, "S^3CA: A Sparse Strip Spectral Correlation Analyzer," in IEEE Signal Processing Letters, vol. 31, pp. 646-650, 2024, doi: 10.1109/LSP.2024.3364062.

Taken together, this work contributes to SCD estimation via techniques that are not only faster and more accurate but are also more adaptable to the specific needs of different applications.

1.4 Thesis Structure

This dissertation details the design and implementation processes associated with the research. It is organized as follows:

- Chapter 2 provides a literature review, introducing the background of cyclostationary processes and the state-of-the-art in algorithm derivation and implementation.

- Chapter 3 details the implementation of FAM on the AMD/Xilinx Zynq UltraScale+ RFSoC ZCU111 platform with quantization error analysis.
- Chapter 4 discusses the implementation of SSCA on the AMD/Xilinx VCK5000 platform.
- Chapter 5 introduces a new algorithm, S^3CA, which accelerates the SSCA using SFFT to achieve a more efficient algorithm.
- Chapter 6 discusses the conclusions of this work.

CHAPTER 2

Literature Review

This chapter introduces the background of SCD, including two frequently utilized algorithms: FAM and SSCA. It also provides a literature review on the derivation of these algorithms and their benefits and limitations. In the implementation, low precision reduces resource requirements and computational complexity; however, this tradeoff can negatively impact the quantization error of the outputs. Additionally, this chapter discusses error analysis for data subject to quantization, focusing on roundoff and truncation errors arising from multiplication and addition operations. It further examines the SQNR in the context of the decimation-in-time (DIT) and decimation-in-frequency (DIF) FFTs algorithms, which is the major part of the SCD algorithm.

2.1 Cyclostationary

A signal as *cyclostationary* of order n (in the wide sense) if and only if there is some nth-order nonlinear transformation of the signal that will generate finite-strength additive sine-wave components [29]. In other words, a time series is considered *cyclostationary* if its probability distribution varies periodically over time. The property of cyclostationarity can generally be exploited to enhance the reliability of information gleaned from data sets if the signal is corrupted.

If a time series is given by $x(t) = a\cos(2\pi\alpha t + \theta)$, where $\alpha \neq 0$, it contains a finite additive sinusoidal component with frequency α. The spectral parameter $\hat{x}(\alpha) \triangleq \lim_{T\to\infty} \frac{1}{T} \int_{-T/2}^{T/2} x(t)e^{-j2\pi\alpha t}\, dt$ is non-zero, indicating the presence of a strong frequency component at α. The spectral density of $x(t)$ shows spectral lines at frequencies α and $-\alpha$.

Therefore, the time-series $x(t)$ exhibits *first-order periodicity* characteristic of cyclostationarity at frequency α.

If $x(t)$ contains a non-zero value of parameter

$$R_x^\alpha(\tau) \triangleq \lim_{T\to\infty} \frac{1}{T} \int_{-T/2}^{T/2} x(t+\frac{1}{2}\tau)x(t-\frac{1}{2}\tau)e^{-j2\pi\alpha t}dt \tag{2.1}$$

at frequency $\alpha \neq 0$ as a function of τ, $x(t)$ is a time-series exhibiting *second-order periodicity*. The autocorrelation R_x^α can also be interpreted as the cross-correlation between two complex-valued frequency-shifted versions of the time-series $x(t)$, denoted as $u(t) \triangleq x(t)e^{-j\pi\alpha t}$ and $v(t) \triangleq x(t)e^{j\pi\alpha t}$. Thus Equation (2.1) becomes:

$$R_x^\alpha(\tau) \equiv R_{uv}(\tau) \triangleq \lim_{T\to\infty} \frac{1}{T} \int_{-T/2}^{T/2} u(t+\frac{1}{2}\tau)v^*(t-\frac{1}{2}\tau). \tag{2.2}$$

The Fourier transform of the R_x^α is the cross-spectral density S_x^α of $u(t)$ and $v(t)$ and defined as

$$S_{uv}^\alpha(f) \triangleq S_x^\alpha(f) = \int_\infty^\infty R_x^\alpha(\tau)e^{-j2\pi f\tau}d\tau, \tag{2.3}$$

which is also the SCD of the two spectral components of $x(t)$ with frequencies $f + \frac{1}{2}\alpha$ and $f - \frac{1}{2}\alpha$. Thus, the S_x^α is also defined as:

$$S_x^\alpha(f) = \lim_{T\to\infty}\lim_{\Delta t\to\infty} S_{uv_T}(t,f)_{\Delta t} = \lim_{T\to\infty}\lim_{\Delta t\to\infty} \frac{1}{\Delta t}\int_{-\Delta t/2}^{\Delta t/2} \frac{1}{T} U_T(t+\zeta,f)V_T^*(t+\zeta,f)d\zeta, \tag{2.4}$$

in which the $U_T(t,f) \triangleq X_T(t, f+\frac{1}{2}\alpha)$ and $V_T(t,f) \triangleq X_T(t, f-\frac{1}{2}\alpha)$.

$X_T(t,f)$ presents the complex demodulates of $x(t)$ and is defined as:

$$X_T(t,f) \triangleq \int_{t-T/2}^{t+T/2} x(\zeta)e^{-j2\pi f\zeta}d\zeta. \tag{2.5}$$

In other words, $X_T(t,f)$ is the complex envelope of the narrow bandpass component of $x(t)$ with center frequency f and approximate bandwidth $\frac{1}{T}$.

2.2 Spectral Correlation Analyzers

In cyclic spectral analysis, algorithms compute the correlation between the spectral components of two real or complex-valued signals. This process is computationally intensive, requiring numerous Fourier transforms, which highlights the growing demand for more efficient algorithms [59]. The complexity poses challenges, even for hardware implementations using discrete-time versions. Therefore, improving computational efficiency in estimating the spectral correlation is crucial.

The discrete-time versions of Equation (2.3) and Equation (2.1) become:

$$S_x^\alpha(f) \triangleq \sum_{k=-\infty}^{\infty} R_x^\alpha(k) e^{-j2\pi fk} \tag{2.6}$$

and

$$R_x^\alpha(k) \triangleq \lim_{N\to\infty} \frac{1}{2N+1} \sum_{n=-N}^{N} [x(n+k)e^{-j\pi\alpha(n+k)}][x(n)e^{j\pi\alpha n}]^*, \tag{2.7}$$

where $S_x^\alpha(f)$ represents the cross spectrum of two frequency-shifted signals $x(n)e^{-j\pi\alpha n}$ and $x(n)e^{j\pi\alpha n}$. The sampling interval is denoted as T_s, and the input signal duration is $\Delta t = NT_s$, which is ideally infinite. The cycle frequency resolution provided by SCD is $\Delta\alpha = 1/\Delta t$. The discrete set of spectrum frequency $f_k = k\Delta f$, where Δf is the frequency resolution of the analyzer. Thus, the value of $S_x^\alpha(f)$ is located in the frequency/cycle-frequency plane, also known as the bifrequency plane, with resolutions of Δf and $\Delta\alpha$. In practical applications, handling an infinite input signal duration is not feasible, making the computational complexity of a naive implementation of $S_x^\alpha(f)$ effectively unbounded and impractical. To address this, N is assumed to be large, resulting in $\Delta t\Delta f \gg 1$, which allows for an estimate to be made. An approximation using very long input data N (matching the size in Equation (2.7)), with k ranges from $-\frac{N}{2}$ to $\frac{N}{2}$ can be employed to estimate the result. The computational complexity of this approximate version is $O(N^2)$.

Two common techniques are used to estimate the SCD in cyclic spectral analysis algorithms: average in frequency (frequency-smoothing), and the average in time (time-smoothing).

2.2.1 Frequency Smoothing Algorithms

The frequency-smoothing algorithm computes the set of demodulates by first calculating the frequency spectrum and then down-converting the input signal segments of length Δt. Consequently, the input sequence $x(n)$ is transformed as follows [58]:

$$X_{\Delta t}(k) = \sum_{n=0}^{N-1} a(n)x(n)e^{-j2\pi kn/N}, \tag{2.8}$$

where $-N/2 \leq k \leq N/2 - 1$ and $a(n)$ represents a data tapering window of length N. The same process is applied to the input sequence $y(n)$ to compute $Y_{\Delta t}(k)$. The frequency bins of $X_{\Delta t}(k)$ and $Y_{\Delta t}(k)$ are then cross-multiplied, and the resulting products are averaged to achieve frequency-smoothing with a window size of M:

$$S^{\alpha}_{xy_{\Delta t}}(f)_{\Delta f} = \frac{1}{M} \sum_{k=\lfloor -M/2 \rfloor}^{\lfloor M/2 \rfloor - 1} X_{\Delta t}(f + \lfloor \frac{\alpha}{2} \rfloor + k(\frac{f_s}{N}))Y^{*}_{\Delta t}(f - \lceil \frac{\alpha}{2} \rceil + k(\frac{f_s}{N})). \tag{2.9}$$

In this equation, the frequency resolution of $S^{\alpha}_{xy_{\Delta t}}(f)_{\Delta f}$ is given by $\Delta f = M(f_s/N)$, the cycle frequency resolution is $\Delta\alpha = f_s/N$, and $S^{\alpha}_{xy_{\Delta t}}(f)_{\Delta f}$ is smoothed using a window size of M.

If an FFT is employed to compute the Fourier spectrum, the computational complexity for calculating $X_{\Delta t}(k)$ is $O(N \log_2 N)$. When estimating only a single point on the bifrequency plane, the computational complexity is $O(M)$. However, if all points on the bifrequency plane are estimated, the computational complexity escalates to $O(N^2)$.

2.2.2 Time Smoothing Algorithms

In contrast to the frequency-smoothing algorithm, the time-smoothing algorithm estimates the cyclic spectrum within a block. Figure 2.1 illustrates the basic implementation of the discrete-time smoothed cyclic cross periodogram, where a data tapering window of length T seconds slides over the data for a time span of Δt seconds, then computes the cyclic cross periodogram by averaging the product over a sliding time interval of approximately Δt

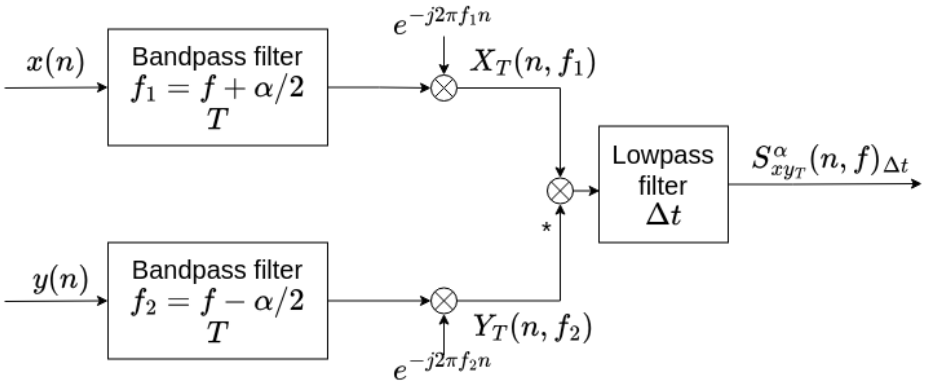


FIGURE 2.1. Implementation of the time-smoothed cyclic cross periodogram

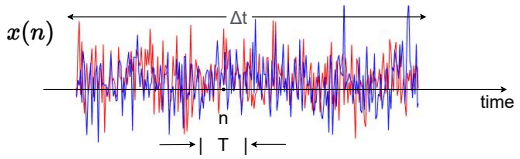
samples. For a fixed spectrum frequency f and cycle frequency α, the SCD method function performs as the time-smoothing technique of cross-spectral analysis.

The conventional time-smoothed cyclic cross periodogram is given by:

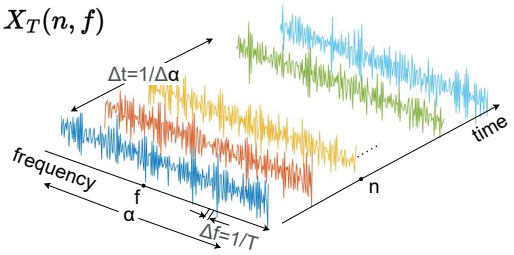
$$S^{\alpha}_{xy_T}(n, f)_{\Delta t} = \frac{1}{T}\langle X_T(n, f + \alpha/2) Y^*_T(n, f - \alpha/2)\rangle_{\Delta t}. \tag{2.10}$$

The cyclic cross periodogram is the correlation between the spectral components $X_T(n, f + \alpha/2)$ and $Y_T(n, f - \alpha/2)$ obtained by passing inputs $x(n)$ and $y(n)$ through the narrowband bandpass filters centered at frequencies $f + \alpha/2$ and $f - \alpha/2$, respectively. Similarly, in a practical implementation, the input data length is finite value Δt; a data tapering window (e.g., Hamming, Chebyshev, etc) of length T seconds is applied to the complex demodulates, sliding across the entire window of input data to capture the spectral correlation effectively. To achieve a reliable estimate, Δt must be significantly greater than T.

In Figure 2.1, the length of the input signal is $N = \Delta t/f_s$ as f_s is the sampling frequency ($T_s = 1/f_s$ is the sampling interval). The tapering window length of complex demodulates $N_P = T/f_s$.



(A) A complex-valued signal $x(n)$ with a sample period of $T_s = 1/f_s$



(B) Complex demodulates of signal x(n)

FIGURE 2.2. Complex demodulate of a single complex-valued signal

To compute the spectral cross-correlation $S^{\alpha}_{xy_T}(n, f)$, the complex demodulates can be mathematically expressed as

$$X_T(n, f) = \sum_{-N_P/2}^{N_P/2-1} a(r)x(n + r)e^{-j2\pi f(n+r)T_s} \tag{2.11}$$

where $a(r)$ is a data tapering window. The frequency resolution of complex demodulates is $\Delta f = \Delta a = f_s/N_P$. Figure 2.2 illustrates the output of the complex demodulate of a complex-valued signal.

Then based on Figure 2.1, the output of two complex demodulates are correlated and followed by a lowpass filter with a memory length of $\Delta t = NT_s$ seconds. The complex demodulate frequencies $f_1 = f + \alpha/2$ and $f_2 = f - \alpha/2$ are related to the spectrum frequency f and the cycle frequency α. In other words, $f = (f_1 + f_2)/2$ and $\alpha = f_1 - f_2$. Then Equation (2.10) becomes:

$$S^{\alpha_0}_{xy_T}(n, f_0)_{\Delta}t = \sum_r X_T(r, f_1)Y^*_T(n, f_2)g(n + r). \tag{2.12}$$

The computational complexity of a complex demodulate measured is $O(NN_P + NN_P^2)$, and of Equation (2.12) via FFT is $O(NN_P^2 + N_P^2(N/2)\log_2 N)$.

Figure. 2.3 shows the estimated SCD function $S^{\alpha_0}_{xy_T}(n, f_0)_{\Delta t}$ of a direct-sequence spread-spectrum (DSSS) binary phase-shift keying (BPSK) signal with 10 dB SNR, and the corresponding alpha profile which captures the maximal SCD values along the spectral frequency axis for each cyclic frequency α, as defined in [25]:

$$\text{alpha profile} = \max_f(S^{\alpha_0}_{xy_T}(n, f)_{\Delta t}). \tag{2.13}$$

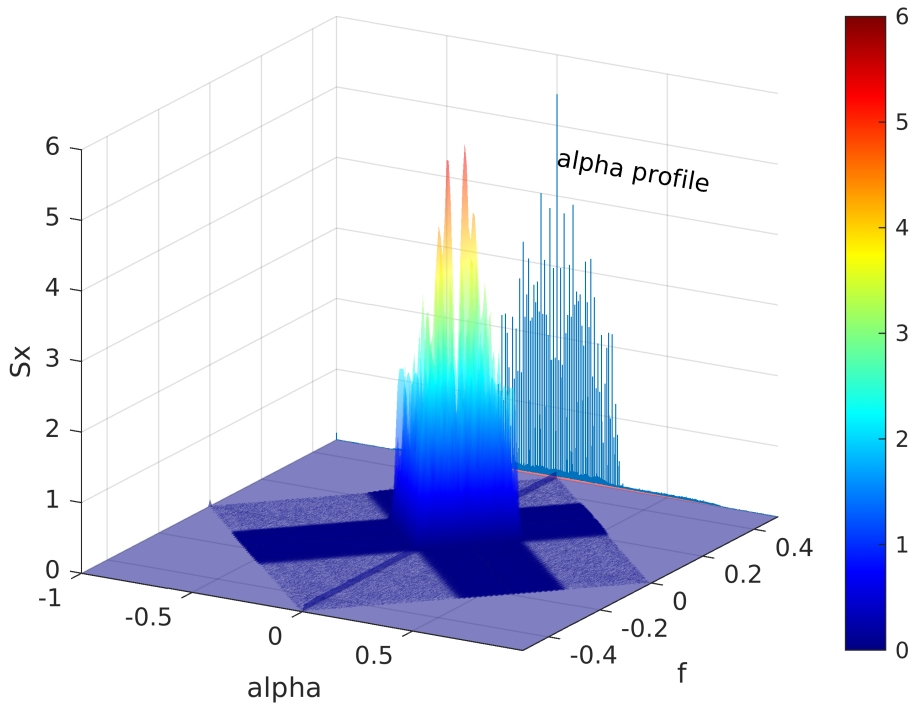


FIGURE 2.3. The SCD function and alpha profile of DSSS BPSK signal

The purpose of the alpha profile is to reduce the data volume caused by high α resolution. By retaining only the peak SCD value per α, the dimensionality and storage bandwidth are cut while still retaining the cyclic frequency information in the SCD output.

The frequency-smoothing algorithms compute a single point estimate at a time, whereas the time-smoothing algorithms calculate point estimates of the cyclic spectrum in blocks, making them more computationally efficient for blind applications where the entire bifrequency plane is needed; therefore, this thesis focuses on time-smoothing and will explore further methods to reduce the computational complexity in the commonly used FAM and SSCA algorithms within this approach.

2.2.3 Computational Efficiency

Fourier Transform: The computational efficiency of time-smoothing algorithms can be significantly enhanced by utilizing a computationally efficient Fourier transform to perform the

necessary summations. In Equation (2.11) and Equation (2.12), a Fourier transform enables efficient computation of time-smoothing and the complex demodulates. By manipulating Equation (2.11) into the following form:

$$X_T(n,f) = [\sum_{r=-N_P/2}^{N_P/2-1} a(r)x(n+r)e^{-j2\pi frT_s}]e^{-j2\pi fnT_s}, \tag{2.14}$$

the summation in the brackets can be efficiently computed using an N_P-point FFT, where the exponential coefficient in Equation (2.11) changes from $e^{-j2\pi f(n+r)T_s}$ to $e^{-j2\pi frT_s}$ in the N_P-point FFT; the subsequent exponential coefficient, $e^{-j2\pi fnT_s}$, then corrects the output of the N_P-point FFT to match the complex demodulates.

To achieve time-smoothing using the Fourier transform, a frequency shift ϵ is introduced to the complex demodulate product sequence before smoothing. Consequently, Equation (2.12) is modified as follows:

$$S_{xy_T}^{\alpha_0}(n,f_0)_\Delta t = \sum_r X_T(r,f_1)Y_T^*(r,f_2)g(n+r)e^{-j2\pi\epsilon rT_s} \tag{2.15}$$

where $|\epsilon| < \Delta a$. Additionally, the cycle frequency parameter α_0 has been redefined to $f_1 - f_2 + \epsilon$.

If $\epsilon = q\Delta\alpha$, Equation (2.15) becomes

$$S_{xy_T}^{\alpha_0}(n,f_0)_\Delta t = \sum_r X_T(r,f_1)Y_T^*(r,f_2)g(n+r)e^{-j2\pi rq/N}, \tag{2.16}$$

which means the sum in Equation (2.16) can be efficiently evaluated by an N-point FFT. Based on this computation, the estimation of N point product is centered at $(f_0, f_1 - f_2)$. The qth point causes a variable frequency resolution of $\Delta f = \Delta a - |q|f_s/N$, and a constant cycle frequency resolution of $\Delta\alpha = f_s/N$.

Decimation: The computational efficiency of the time-smoothing algorithm can also be enhanced by decimating the outputs of the bandpass filters using an appropriate factor. The input data is sampled at intervals of L samples to the filter, which results in processing only $P = N/L$ samples per point estimate, thereby reducing the overall computational complexity of the algorithm by a factor of L. After decimation, the output sampling rate decreases to

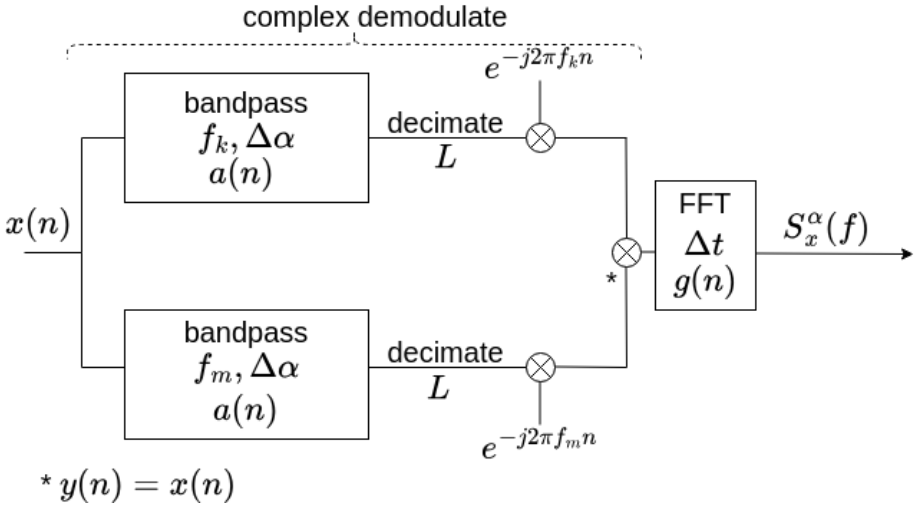


FIGURE 2.4. The FFT accumulation method

f_s/L with the filter output window being N_P samples long. The computational complexity decreases as the factor L increases, and the likelihood of cycle leakage also increases. This leakage becomes significant when L equals N_P, and aliasing begins to occur once L surpasses N_P. The decimation version of Equation (2.11) and Equation (2.12) become:

$$X_T(pL, f) = \sum_{r=-N_P/2}^{N_P/2-1} a(r)x(pL + r)e^{-j2\pi f(pL+r)T_s} \tag{2.17}$$

and

$$S_{xy_T}^{\alpha_0}(pL, f_0)_\Delta t = \sum_r X_T(rL, f_1)Y_T^*(rL, f_2)g_c(pL + rL) \tag{2.18}$$

where $n = pL$ and g_c is a comb filter with the bandwidth of f_s/L Hz in cycle frequency. The determination of appropriate values for L should consider the cycle leakage and cycle aliasing alongside the reduced computational complexity. In SSCA, the sampling rate of $X_T(n, f_*)$ cannot be decimated as it should match the sampling rate with $y(n)$. Thus, the determination of appropriate values for L will be discussed only in the context of the FAM method.

2.2.4 The FFT Accumulation Method

The FAM incorporates similar computationally efficient techniques as those discussed in Section 2.2.3. Figure 2.4 illustrates the signal flow for the FAM method, where the first task

is to compute the complex demodulates, X_T and Y_T (in Equation (2.20)), and substituting $y(n) = x(n)$. Channelization is performed using an N_P-point FFT that is applied to the data in steps of L samples. The outputs of the FFT are then downconverted to the baseband to obtain decimated complex demodulate sequences. Based on Equation (2.14) and Equation (2.17), the complex demodulates are computed as follows:

$$\begin{aligned} X_T(pL, f_m) &= [\sum_{r=-N_P/2}^{N_P/2-1} a(r)x(pL+r)e^{-j2\pi f_m r T_s}]e^{-j2\pi f_m pLT_s} \\ &= \mathtt{FFT}_{N_P}[a(r)x(pL+r)]e^{-j2\pi f_m pLT_s}, \end{aligned} \tag{2.19}$$

where the downconversion coefficient is $e^{-j2\pi f_m pLT_s}$ used to correct the FFT output to complex demodulate sequences. The center frequency f_m of each complex demodulates $X_T(pL, f_m)$ is $f_m = mf_s$, in which the m is in the range of $[-N_P/2, N_P/2 - 1]$.

After the complex demodulates of two input signals are computed, conjugate product sequences $X_T(pL, f_*)Y_T^*(pL, f_*)$ are formed, and a P-point FFT to compute the time-smoothing ($P = N/L$). Based on Equation (2.16) and Equation (2.18), the time-smoothed estimate becomes:

$$\begin{aligned} S_{xy_T}^{\alpha_0}(pL, f_{kl})_\Delta t &= \sum_r X_T(rL, f_k)Y_T^*(rL, f_l)g_d(p+r)e^{-j2\pi rq/P} \\ &= \mathtt{FFT}_P[X_T(rL, f_k)Y_T^*(rL, f_l)g_d(p+r)], \end{aligned} \tag{2.20}$$

where $\alpha_0 = f_k - f_l + q\Delta\alpha$, $f_{kl} = (f_k + f_l)/2$, k and l are in the range of $[-N_P/2, N_P/2 - 1]$, and $g_d(r) = g_c(rL)$. Thus, Equation (2.20) can be efficient evaluated by a P-point FFT.

In FAM, the value of L is crucial, as it has to balance reducing computational complexity and avoiding severe cycle frequency leakage. Reference [18] recommends $L = N_P/4$. This recommendation is based on the relationship between the number of P-point FFT and L, given by $P = N/L$. A larger L reduces the computational complexity of FAM significantly. However, to avoid the possibility of severe cycle frequency leakage, the L should be less than $N_P/2$. Choosing the $L = N_P/4$ has an additional benefit: the downconversion coefficient simplifies to $e^{-j\pi mp/2} = \{j, -j, 1, -1\}$, eliminating the need for multiplication in the downconversion section. The computational complexity of FAM is $O(N_P P(\log_2 N_P +$

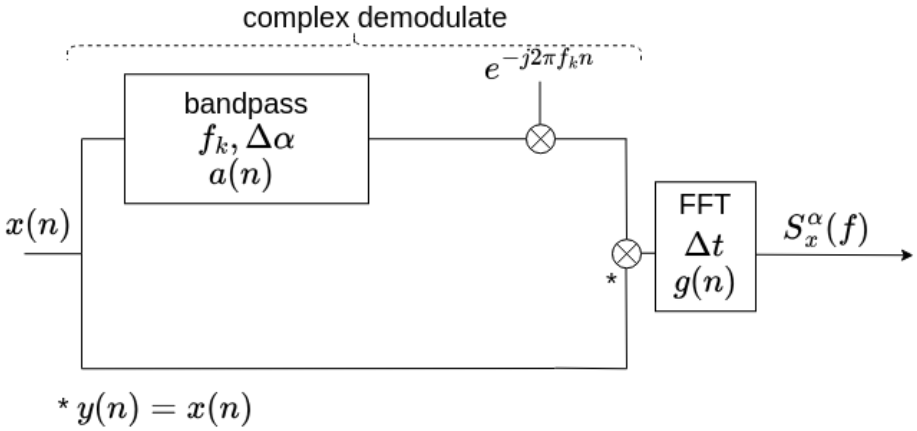


FIGURE 2.5. The strip spectral correlation analyzer

$N_P \log_2 P)$). If minimizing computation time is crucial, L can be increased to $N_P/2$. Further derivations and implementation details are presented in Section 3.1.

2.2.5 The Strip Spectral Correlation Analyzer

The SSCA is another FFT-based time-smoothing algorithm that computes the correlation products between spectral and temporal components. Figure 2.5 shows the dataflow of SSCA. Comparing with the Figure 2.4, the difference between the SSCA and FAM is that instead of multiplying the complex demodulate $X_T(n, f_k)$ with $Y_T^*(n, f_k)$, it is multiplied by $y^*(n)$. The channelizer decimation factor L is set to 1 to match the sampling rate of the two terms. Thus the complex demodulate is same as Equation (2.14) and the equation of SSCA is

$$S_{xy_T}^{\alpha_0}(n, f_0)_\Delta t = \sum_r X_T(r, f_k) y^*(r) g(n-r) e^{-j2\pi rq/N}, \tag{2.21}$$

where $\alpha_0 = f_k + q\Delta\alpha$ and $f_0 = \frac{f_k - q\Delta\alpha}{2}$. Equation (2.21) can be efficiently evaluated with an N-point FFT. The products will lie along the frequency-skewed family of lines $\alpha = 2f_k - 2f$ to result in a strip for each f_k. The SSCA provides a highly desirable feature for the estimator as it allows uniform output in Δf and $\Delta t \Delta f$. The computational complexity of SSCA is $O(NN_P \log_2 N_P + N_P N \log_2 N)$. Additional derivations and a deeper exploration of the SSCA implementation are provided in Section 4.1.

2.2.6 Other SCD Estimation Algorithms

In recent years, researchers have developed new approaches for estimating spectral correlation that build upon conventional SCD algorithms.

Antoni et al. introduced the cyclic modulation spectrum (CMS), which relies on an envelope-like procedure applied to a discrete short-time Fourier transform (STFT), followed by a subsequent series of discrete Fourier transform (DFT) operations along the remaining time axis [7, 13]. The CMS offers advantages such as simplicity, ease of interpretation, and, importantly, computational efficiency. However, a key limitation of this method is its inability to detect cycle frequencies beyond the maximum cycle frequency, α_{max}, relative to the spectral resolution, Δf [14].

Another approach is the averaged cyclic periodogram (ACP), which involves calculating a discrete-time correlation between the signal's DFT and a version shifted by a specific cycle frequency (α) [6]. The spectral correlation is computed separately for different segments of the signal, and the results are averaged to reduce variance. However, compared to the CMS, the implementation of ACP suffers from a significant loss of computational efficiency as the input length increases. This inefficiency is primarily due to the large number of complex multiplications required for the α-shift of the DFT [14].

To address some of these challenges, Antoni et al. proposed the fast spectral correlation (FSC) algorithm, which is an efficient method for calculating spectral correlation [8]. The FSC algorithm works by decomposing the products of STFT lines around a specific spectral frequency to derive the spectral correlation. The method incorporates amplitude and phase corrections to enhance estimation accuracy. However, its application is restricted due to the block shift requirement, which confines its use to cases with low cycle frequency ranges. Additionally, as the cycle frequency range increases, FSC requires high memory which is unsuitable for platforms with limited memory.

To offer a more computationally efficient solution, Borghesani and Antoni proposed an alternative method that approximates the ACP, offering a reduction in computational effort but at the cost of doubling the memory requirement [14].

Jaafar K. Alsalaet derived a new method, the fast averaged cyclic periodogram (FACP), which precisely calculates the ACP. It reduces computational cost and overcomes memory limitations by leveraging the frequency-shifting property of the Fourier transform [1]. Although the FACP is highly effective in vibration analysis, it requires greater computational complexity when dealing with large cycle frequencies compared to FAM.

2.3 Quantization Error Analysis

Quantization is the process of mapping a continuous range of values into a finite set of discrete levels, which occurs because digital systems have limited bit resolution, meaning certain values cannot be represented precisely and must be approximated by truncating or rounding to their nearest available unit. The mismatch between the accurate value and its approximate unit is known as a quantization error. Quantization error analysis is essential before implementing SCD algorithms in hardware because a tradeoff between resource utilization and estimation accuracy can be obtained via quantization. By analyzing these errors, we can predict their impact on system performance, make informed decisions about resource allocation, and aid in designing the implementation. This analysis is crucial for optimizing the system to achieve the best possible balance between computational efficiency and accuracy.

2.3.1 Quantization Noise Model

This thesis represents signals as B-bit two's complement fractions

$$a = -a_{B-1} + \sum_{i=0}^{B-2} a_i 2^{i-(B-1)}, \tag{2.22}$$

where $\forall i, a_i \in \{0, 1\}$, and range $a \in [-1, 1)$. In this representation, the most significant bit determines the sign, and the remaining $F = B - 1$ bits are used for the fraction.

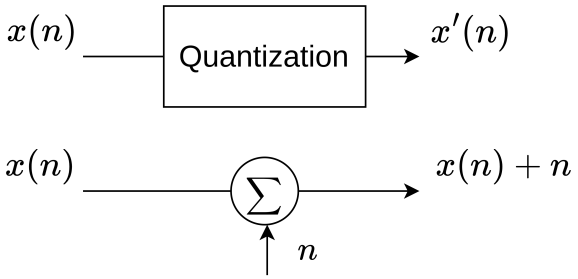


FIGURE 2.6. Quantization noise model ($x'(n) = x(n) + n$)

Oppenheim et al. [54] and Widrow et al. [68] have developed a statistical quantization analysis, illustrated in Figure 2.6. All operations where a signal x is quantized to x' by quantizer Q are modeled by an additive noise source n, which is assumed to be uncorrelated, uniformly distributed $\in [-2^{-F-1}, 2^{-F-1})$ and with variance $\sigma^2 = \frac{(2^{-F})^2}{12}$.

For the analysis in this dissertation, the computations only involve a series of multiplication and addition operations. Therefore, it is necessary to account for the quantization errors introduced by these blocks in order to deduce a quantitative analysis for the whole system.

2.3.1.1 Roundoff

For two's complement fractions, the number of fraction bits for the product result is $2F$ which is rounded to F bits to match the wordlength. In this case, the quantization error is $\frac{(2^{-F})^2}{12} - \frac{(2^{-2F})^2}{12}$ and is approximated as

$$\sigma_m^2 = \frac{(2^{-F})^2}{12}. \tag{2.23}$$

Moreover, for the multiplication of two complex values, four real multiplications contribute to the variance [64, 54]:

$$\sigma_{cc}^2 = 4\sigma_m^2 = \frac{(2^{-F})^2}{3}. \tag{2.24}$$

For a real number multiplied with a complex number, it is

$$\sigma_{rc}^2 = 2\sigma_m^2 = \frac{(2^{-F})^2}{6}. \tag{2.25}$$

2.3.1.2 Truncation

Quantization errors are also introduced by addition. An F-bit addition, in general, has an $F+1$ bit result with no quantization error. To scale this back to F-bits without overflow requires a right shift which introduces a noise term with variance [54, 68]

$$\sigma_{ad}^2 = \frac{(2^{-F})^2}{8}. \tag{2.26}$$

As the addition of two complex numbers computes the real and imaginary components separately, the variance [54, 68] becomes:

$$\sigma_{ac}^2 = 2\sigma_{ad}^2 = \frac{(2^{-F})^2}{4}. \tag{2.27}$$

2.3.1.3 Quantization Error for Complex Multiplication

W. Schlecker et al. derive the quantization error for the multiplication result of two quantized signals (in real number) [60] as

$$SNR = \frac{\sigma_{x1}^2\sigma_{x2}^2}{\sigma_{x1}^2\sigma_{e2}^2 + \sigma_{x2}^2\sigma_{e1}^2 + \sigma_{e1}^2\sigma_{e2}^2 + \sigma_{eq}^2} \tag{2.28}$$

where σ_x^2, σ_e^2, and σ_{eq}^2 are the variance of input signal, input error, and the error generated by multiplication.

2.3.1.4 Quantization Error Analysis for the Fixed-Point Fast Fourier Transform

The FFT [16] employs a butterfly structure and Figure 2.7 illustrates the noise model of the DIT Radix-2 FFT and the DIF Radix-2 FFT for different quantization scheme [64], where N is the number of points of the FFT. This involves $m = \log_2 N$ stages. The noise model introduces round off (σ_{cc}^2) and truncation (σ_{ac}^2) terms to the system. For the kth ($1 < k < m$) stage, the input is $x_{k-1} + \sigma_{k-1}^2$ and the output is $x_k + \sigma_k^2$, with $[a]$ and $[b]$ denoting the upper and lower part of the butterfly structure.

There are two different schemes for computing the quantization error analysis of FFT algorithm. Scheme 1, similar to the reference [54], introduces the roundoff error after the multiplication and incorporates an attenuation factor of $1/2$ before the addition at each stage to avoid overflow. Scheme 2 is a variant of Scheme 1, assuming a 1-bit overflow occurs in each addition operation of the FFT computation.

Radix-2 DIF Scheme 1:

To avoid overflow, an attenuation factor of $1/2$ is introduced in each stage. Thus, the variance of signal and noise have to be multiplied by $1/4$ before introducing the truncation noise term. Therefore, the variance is

$$\sigma_k^2[a] = (\sigma_{k-1}^2[a] + \sigma_{k-1}^2[b])\frac{1}{4} + \sigma_{ac1}^2 + \sigma_{cc}^2 \tag{2.29a}$$

$$\sigma_k^2[b] = (\sigma_{k-1}^2[a] + \sigma_{k-1}^2[b])\frac{1}{4} + \sigma_{ac2}^2 + \sigma_{cc}^2. \tag{2.29b}$$

In summary, the quantization noise arising from the FFT operation is modeled as the sum of the roundoff (σ_R^2) and truncation (σ_T^2) components. As $N = 2^m$ the variance of those two noise terms generated from each butterfly and passed to the final stage are

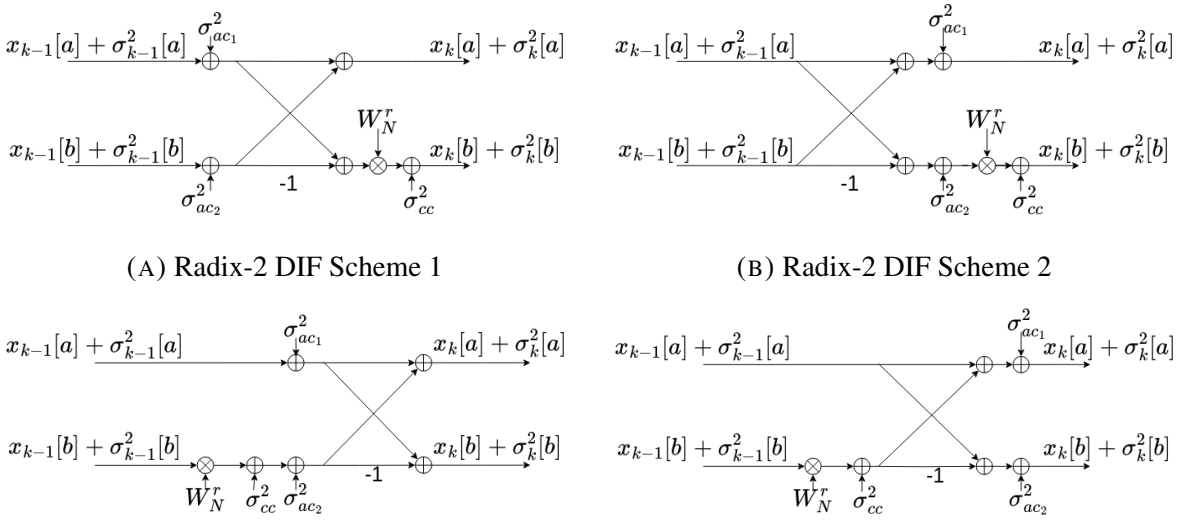
$$\begin{aligned}\sigma_R^2 &= \sigma_{cc}^2(\frac{N}{2})(\frac{1}{N})(\frac{N}{2}\frac{1}{4^{m-1}} + \frac{N}{2^2}\frac{1}{4^{m-2}} + ... + \frac{N}{2^m}\frac{1}{4^{m-m}}) \\ &= \sigma_{cc}^2(1 - \frac{1}{2^m})\end{aligned} \tag{2.30}$$

and

$$\begin{aligned}\sigma_T^2 &= \sigma_{ac}^2(N\frac{1}{4^{m-1}} + \frac{N}{2}\frac{1}{4^{m-2}} + ... + \frac{N}{2^{m-1}}\frac{1}{4^{m-m}}) \\ &= 4\sigma_{ac}^2(1 - \frac{1}{2^m}).\end{aligned} \tag{2.31}$$

Thus, the output noise variance is

$$\sigma_E^2 = \sigma_T^2 + \sigma_R^2 = \frac{4}{3}2^{-2F}(1 - \frac{1}{2^m}). \tag{2.32}$$



(A) Radix-2 DIF Scheme 1

(B) Radix-2 DIF Scheme 2

(C) Radix-2 DIT Scheme 1

(D) Radix-2 DIT Scheme 2

FIGURE 2.7. Quantization noise model of butterfly structure for different FFT algorithms and scheme (k^{th} Stage, 1<k<m)

However, when $w \in \{1, -1, j, -j\}$, the error from multiplication will be zero. Therefore, the last two stages of DIF do not introduce noise. The variance of those noises is computed as:

$$\begin{aligned}\sigma_w^2 &= \frac{\sigma_{cc}^2}{2}(2\frac{N}{2}\frac{1}{4^{m-1}} + 2^2\frac{N}{2^2}\frac{1}{4^{m-2}} + ... + 2^{m-1}\frac{N}{2^{m-1}}\frac{1}{4} + \frac{N}{2}\frac{N}{2^m}) \\ &= \sigma_{cc}^2(\frac{5}{6} - \frac{4}{3N^2}).\end{aligned} \tag{2.33}$$

After extracting Equation (2.33) from the Equation (2.32), the final expression of FFT quantization noise becomes

$$\sigma_F^2 = \sigma_E^2 - \sigma_w^2 = \frac{2^{-2F}}{3}(\frac{19}{6} - \frac{4}{N}\frac{4}{3N^2}). \tag{2.34}$$

Radix-2 DIT Scheme 2:

Similar to the variance in each stage of Radix-2 DIF, the variance of Radix-2 DIT is

$$\sigma_k^2[a] = (\sigma_{k-1}^2[a] + \sigma_{k-1}^2[b] + \sigma_{cc}^2)\frac{1}{4} + \sigma_{ac1}^2 \tag{2.35a}$$

$$\sigma_k^2[b] = (\sigma_{k-1}^2[a] + \sigma_{k-1}^2[b] + \sigma_{cc}^2)\frac{1}{4} + \sigma_{ac2}^2. \tag{2.35b}$$

The variance of roundoff and truncation generated from each butterfly and passed to the final stage are

$$\begin{aligned}\sigma_R^2 &= \sigma_{cc}^2(\frac{N}{2})(\frac{1}{N})(N\frac{1}{4^m} + \frac{N}{2}\frac{1}{4^{m-1}} + ... + \frac{N}{2^{m-1}}\frac{1}{4}) \\ &= \frac{1}{2}\sigma_{cc}^2(1 - \frac{1}{2^m})\end{aligned} \tag{2.36}$$

and

$$\begin{aligned}\sigma_T^2 &= \sigma_{ac}^2(\frac{N}{2}\frac{1}{4^{m-1}} + \frac{N}{4}\frac{1}{4^{m-2}} + ... + \frac{N}{2^m}\frac{1}{4^{m-m}}) \\ &= 2\sigma_{ac}^2(1 - \frac{1}{2^m}).\end{aligned} \tag{2.37}$$

The output noise variance is

$$\sigma_E^2 = \sigma_T^2 + \sigma_R^2 = \frac{2}{3}2^{-2F}(1 - \frac{1}{2^m}). \tag{2.38}$$

Remove those noises which are supposed to be zero and the final expression of FFT quantization noise variance becomes

$$\sigma_{DIT.S2}^2 = \frac{2^{-2F}}{3}(2 - \frac{m + 1.5}{N}). \tag{2.39}$$

2.4 FFT Algorithms and Architectures

The DFT underpins modern spectral analysis. Its naive evaluation requires $O(N^2)$ complex operations, rendering large transforms prohibitive on resource-constrained platforms. The FFT, anticipated by Gauss and brought to prominence by Cooley and Tukey in 1965, reduces this cost to $O(N \log_2 N)$, thereby enabling ubiquitous deployment in audio, image, radar, biomedical, industrial monitoring, and, more recently, deep-learning accelerators that implement large-kernel convolutions in the frequency domain [20, 43, 55].

2.4.1 Algorithmic Derivatives

Over six decades of research have produced a rich taxonomy of FFT variants, each exploiting mathematical symmetries or bespoke data layouts to trade arithmetic complexity, memory traffic, and control overhead for specific application needs [15, 24, 65]:

- **Radix-2 FFT.** The canonical Cooley–Tukey DIT/DIF factorizations recursively decompose an N-point DFT into two $N/2$-point sub-DFTs offering the simplest control flow and twiddle-factor schedule [65].
- **Higher-radix FFTs.** Radix-4, -8, and -16 variants further reduce the number of complex multiplications at the expense of more elaborate butterflies and twiddle tables, often preferred in vector processors and FPGA pipelines where wide data paths are inexpensive [65].
- **Split-radix FFT.** Combining radix-2 and -4 decompositions, the split-radix algorithm attains the lowest known arithmetic count for power-of-two lengths without incurring irregular memory accesses [67].
- **Mixed-radix and prime-factor FFTs.** When N contains small co-prime factors, mixed-radix FFT or Good–Thomas prime-factor algorithms eliminate zero-padding and facilitate efficient multidimensional factorizations [34, 62, 65]. A more general approach, known as the common factor map (CFM) decomposition FFT, extends the mixed-radix methodology by incorporating customizable factor mappings tailored to arbitrary radix combinations [34]. We utilize and explain the CFM decomposition of FFT in Section 4.1.3.
- **Winograd short-length FFTs.** Winograd's convolution reformulation minimizes multiplications for very small N and serves as the base-case kernel in many hierarchical FFT libraries [69].
- **Multidimensional FFTs.** For images, volumetric data, and MIMO channel estimation, separable row–column strategies or cache-oblivious Morton-order traversals generalize 1D FFT kernels while preserving locality [65].
- **SFFT.** For signals that are sparse in the frequency domain, the SFFT uses randomized sampling, filtering, and hashing to achieve sub-linear complexity, recovering only

the dominant frequency bins. It is especially useful when only a small number of significant frequency components need to be recovered [38]. Further details are provided in Section 5.1.

- **Approximate FFT.** To reduce power and silicon area, approximate transforms prune butterflies or substitute exact multipliers with quantized operators, yielding near-lossless accuracy for edge and low-power DSP nodes [10].
- **Streaming FFT.** Streaming variants of the FFT support real-time processing of continuous signals by applying overlap-save or overlap-add techniques. They enable convolution and spectral estimation in systems with finite memory, such as software-defined radios and embedded spectrum analyzers [55].

This work begins with the canonical radix-2 Cooley-Tukey FFT in Section 3, establishing a baseline SCD implementation. Section 4 and 5 then extend the design to a CFM decomposition FFT and a SFFT, respectively, to address large window sizes. Although these variants yield significant speed and resource benefits, many other FFT formulations could be leveraged to meet different application constraints. Investigating such alternatives offers a promising avenue for future work.

2.4.2 Hardware Realizations

Algorithmic innovations have been mirrored by architecture-aware implementations that expose algorithmic parallelism while respecting hardware constraints:

- **CPUs.** FFTW dynamically explores a space of radix, split-radix, and mixed-radix plans to select the fastest configuration for a given problem size and cache hierarchy [26]. Although highly portable, its sophisticated planner and runtime twiddle generation incur non-trivial memory overheads.
- **GPUs.** Graphics processors offer massive data-parallel throughput ideally matched to FFT butterflies. NVIDIA's CUFFT delivers batched, multi-dimensional, and streamed transforms through hand-tuned CUDA kernels and autoselected decomposition plans, supporting real-time, high-throughput signal workloads [22].

- **FPGAs.** Fine-grained parallelism and deep pipelining make FPGAs attractive for low-latency, energy-efficient FFTs. Available support spans everything from "push-button" IP blocks to highly specialized research cores. For turnkey integration, commercial vendor IP such as the Xilinx LogiCORE FFT IP [40] and Intel FFT IP [21] offer fully parameterizable designs. Open-source and auto-generated options include Xilinx Vitis DSP Library FFT kernels [2], the OpenCores Versatile FFT [70], and SPIRAL-generated FFT libraries [23]. Beyond these general-purpose solutions, there are also other implementations for specific applications like: high-throughput pipeline architectures [33], million-point FFT [56], SFFT cores[49], approximate FFT [47], short-time FFT [11], and multidimensional FFT [5].

In summary, the convergence of algorithmic diversity and architecture-specific optimizations provides a versatile toolkit that can be tuned—from sparse to streaming variants and from CPU libraries to FPGA IP—for the distinct performance, power, and accuracy envelopes demanded by contemporary signal-processing systems.

2.5 Summary

This chapter has reviewed the theoretical background and practical implementation of cyclostationary signal processing algorithms, focusing on FAM and SSCA methods. It has also explored recent developments aimed at reducing computational complexity and memory usage, discussing both their benefits and limitations.

Additionally, the chapter also reviews the analysis of quantization errors in fixed-precision data, focusing on roundoff and truncation errors that arise during multiplication and addition, essential for understanding the tradeoffs involved in implementing these algorithms on digital platforms. Furthermore, it evaluates the SQNR of the DIT and DIF FFT, which are critical components of SCD algorithms.

CHAPTER 3

FFT Accumulation Method Implementation on FPGAs

The SCD is an important tool for extracting statistical features from signals, particularly in the field of communications [31]. By leveraging its integral properties, the SCD enhances the reliability of information derived from data sets, even in the presence of corrupted signals. Achieving real-time or near real-time signal processing in communications is crucial, making the acceleration of SCD essential. In Chapter 2.1, it is evident that extracting the cyclostationary features requires high computational complexity due to the correlation between the two spectra. This chapter aims to accelerate one of the methods of SCD, which is the FAM.

Although the SCD method reveals rich information about cyclostationary processes even under low signal-to-noise ratio conditions, its high computational complexity makes it difficult to apply in real-time applications. Thus there has been interest in developing high-performance implementations of the FAM method to detect and classify cyclostationary signals on CPUs [61], GPUs [52], and FPGAs [12, 46].

To maximize performance, fixed-point implementations of signal processing techniques should be considered as they are more computationally efficient than floating-point implementations and can lead to improved hardware efficiency at the cost of a quantization error. This chapter analyzes the relationship between wordlength and SQNR in fixed-point implementations of the SCD function. A canonical SCD estimation algorithm, the FAM using fixed-point arithmetic is developed. Closed-form expressions for SQNR are derived and compared at different wordlengths.

An open-source[1], scalable high-speed FPGA-based SCD accelerator, utilizing on-chip high-speed arithmetic primitives present in the DSP and memory blocks is presented. The design is synthesized from a C description using HLS tools [53], allowing the calculations to be verified and performance metrics such as speed and performance to be determined. Previous work in references [59, 52, 18] use floating-point calculations and are unable to achieve the performance of this proposed design.

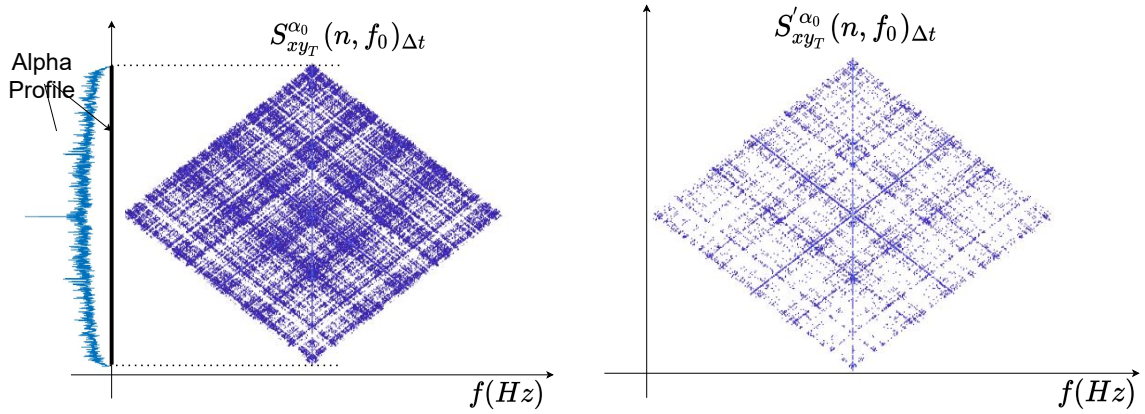
The contributions of this chapter are:

- The first analytical SQNR model for fixed-point implementations of the FAM technique for estimating SCD, enabling aggressive tradeoffs between precision and area.
- A quantitative comparison of two wordlength assignment strategies, FAM_M1, which employs a fixed wordlength throughout the data path, and FAM_M2 with mixed precision.
- A parallel architecture for computing the SCD using fixed-point arithmetic with the FAM_M1 and FAM_M2 wordlength assignments. The architecture is highly parallel and allows mixed precisions to be used throughout.
- An HLS implementation of the architecture which, to the best of our knowledge, achieves the highest reported throughput and power performance for the FAM technique. It employs our SQNR model to minimize resource requirements through careful precision optimization and sparse matrix output to minimize accelerator-to-host bandwidth.

3.1 Background

The section builds upon Section 2.2.4, providing a more detailed description of the FAM.

[1] https://github.com/Jingyi-li/FAM_Synthesis.git



(A) The SCD function of signal $x(n)$ (B) Sparse SCD with only 0.457% non-zero values

FIGURE 3.1. The SCD function and sparse SCD of OOK signal from DeepSig [39] at SNR = -8 dB

3.1.1 The FAM Technique

Due to the parallel FFT-based computation and regular data access patterns, the FAM technique is a commonly used estimator for the SCD, particularly for a small number of cycle frequencies. In Section 2.2.4, the description of the SCD function follows that of Roberts et al. [59] and Brown et al [18]. The FAM first computes the discrete-time *complex demodulate* of a continuous time as Equation (2.19), then we correlate demodulates $X_T(n, f_k)$ with $Y_T(n, f_l)$ separated by α_0 ($f_k = f_0 + \alpha_0/2$, $f_l = f_0 - \alpha_0/2$) over the time window $\Delta t = NT_s$ using a complex multiplier followed by a low pass filter (LPF) with bandwidth approximately $1/\Delta t$ to achieve the SCD function as Equation (2.20). Figure 3.1 shows an on-off keying (OOK) modulated in-phase and quadrature (I/Q) signal $x(n)$ with *complex demodulate* $X_T(rL, f_m)$, an estimated SCD function $S^{\alpha_0}_{xy_T}(n, f_0)_{\Delta t}$, the alpha profile and the sparse SCD $S'^{\alpha_0}_{xy_T}(n, f_0)_{\Delta t}$.

Using Equation (2.13), the alpha profile reduces the data dimensionality from three to two. Implementing the method in [52] further compresses storage, retaining only $2N$ values instead of $2N \times N_P$.

The sparse SCD matrix is formed from the full SCD matrix by setting values smaller than a threshold value ($Tred$) to zero. It captures the critical information and greatly reduces storage

requirements compared to the full SCD matrix. The sparse SCD is defined as:

$$S'^{\alpha_0}_{xy_T}(n, f_0)_{\Delta t} = \begin{cases} S^{\alpha_0}_{xy_T}(n, f_0)_{\Delta t} & \text{if } (S^{\alpha_0}_{xy_T}(n, f_0)_{\Delta t} \geq Tred) \\ 0 & \text{otherwise} \end{cases} . \tag{3.1}$$

3.1.2 FAM Algorithm

Based on Equation (2.19) and Equation (2.20), the FAM algorithm can be implemented in the following steps.

Step 1 Data collection and in a step of $L = N_P/4$ samples:

The size of input window samples is $N + N_P$, and then convert to a block of $P \times N_P$ two-dimensional data in step of $L = N_P/4$ samples ($P = N/L = 4N/N_P$).

$$x_b(pL, r) = x(pL + r),\ r = 0, 1, ..., N_P - 1,\ p = 0, 1, ..., P - 1 \tag{3.2}$$

Step 2 Compute P N_P-point FFTs of each row of x_b:

Define $a(r)$ for $r = 1, 2, ..., N_P$ to be an N_P-point FFT data-tapering window (e.g. rectangular, Hamming, Chebyshev, Kaiser ...).

$$x_T(p, k) = \texttt{FFTS}_{N_P}\{a(r)x_b(pL, r)\},\ k = -N_P/2, ..., N_P/2 - 1 \tag{3.3}$$

The $FFTS_{N_P}$ is the N_P-point FFT operation centered at the frequency $k = 0$.

Step 3 Downconvert the FFT output to baseband:

$$X_T(p, k) = x_T(p, k)e^{-j\pi pk/2} \tag{3.4}$$

Step 4 Compute the weighted product from Equation (2.20):
Define $g(p)$ for $p = 0, 1, ..., P - 1$ to be a P-point data-tapering window (e.g. as above but different shape).
For $k_1, k_2 = -N_P/2, ..., N_P/2 - 1$:

$$X_g(p, k_{12}) = X_T(p, k_1)X^*_T(p, k_2)g(p) \tag{3.5}$$

Step 5 Compute the spectral correlation functions:

For $q = -P/2, ..., P/2 - 1$, $k_1, k_2 = -N_P/2, ..., N_P/2$, $n = 1, ..., P$ and $m = 1, ..., (N_P)^2$

$$S_x(q, k_{12}) = \texttt{FFTS}_P\{X_g(n, m)\} \tag{3.6}$$

where $\texttt{FFTS}_P$ is the P-point FFT operation.

Step 6 Map $S_x(qL, k_{12})$ to $S_x^{\alpha}(f)$:

The mapping is performed using the equations

$$f = \frac{k_1 + k_2}{2N_P}$$

$$\alpha = \frac{k_1 - k_2}{N_P} + \frac{q}{PL}$$

where f and α are normalized with respect to $f_s = 1$, which means $-0.5 \leq f \leq 0.5$ and $-1 \leq \alpha \leq 1$.

3.1.3 FPGA Implementation

FPGA implementation can be designed using very high speed integrated circuit hardware description language (VHDL), Verilog, and HLS. VHDL and Verilog, which were introduced in the 1980s, provide register-transfer-level (RTL) abstractions. Electronic design automation (EDA) tools convert these RTL specifications into a digital circuit model and generate detailed specifications for the device implementing the digital circuit.

HLS emerged as another step of higher abstraction level, allowing designers to focus on broader architectural issues instead of individual registers and cycle-to-cycle operations. HLS supports C/C++ as the input language, making it simpler and more user-friendly with a lower learning curve. Designers can add specific information to the program using directives (e.g., #pragma) to guide the tool in creating the most efficient design. The HLS process then synthesizes an RTL hardware design that can be further processed through the hardware design flow.

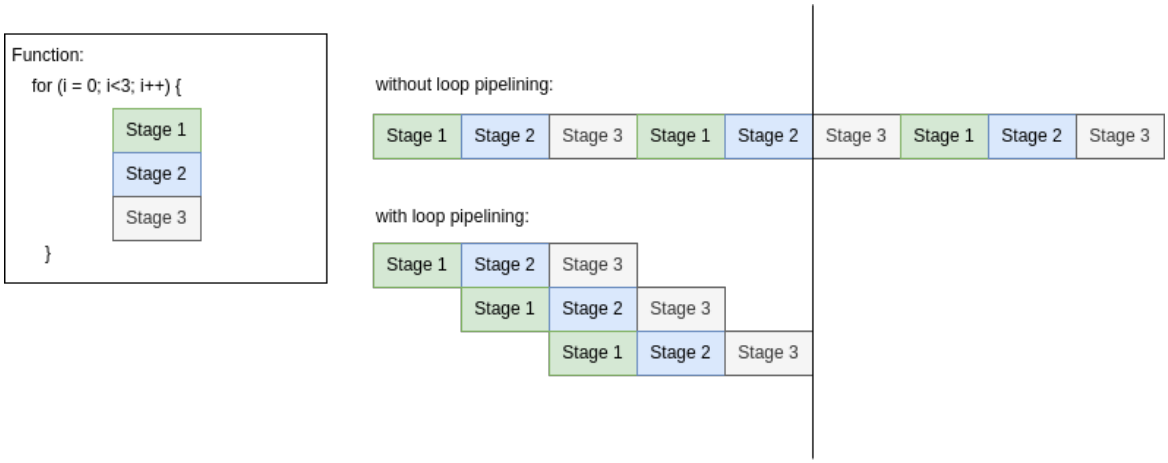


FIGURE 3.2. Loop pipeline

There are three main #pragma directives in HLS to enhance implementation efficiency and throughput: unroll, pipeline, and dataflow.

Parallel: By default, loops in C/C++ are implemented sequentially, meaning the synthesis tool generates logic for a single iteration, which is executed repeatedly in sequence by the RTL design. In the Vitis HLS tool, the UNROLL pragma allows for some or all iterations of loops to be executed in parallel by creating multiple copies of the loop body in the RTL design. This transformation enables loops to improve data access speed and throughput, effectively enhancing overall performance. However, this method increases resource utilization, as multiple hardware copies of the loop body require additional logic and memory, which can lead to area constraints and power consumption challenges in the FPGA.

Pipeline: In the Vitis HLS tool, designers can implement a function or loop to execute operations concurrently, reducing the initiation interval (II) using the PIPELINE pragma. As illustrated in Figure 3.2, assuming each stage requires one cycle to complete, the II of each loop in the function is initially 3, requiring 8 cycles to perform the last output write for 3 iterations. However, with loop pipelining, only 4 cycles are needed to perform the last write, saving 4 cycles and enhancing overall efficiency. This optimization, however, comes at the cost of increased resource utilization, as pipelining requires additional registers, logic, and potentially larger routing resources to handle concurrent operations effectively.

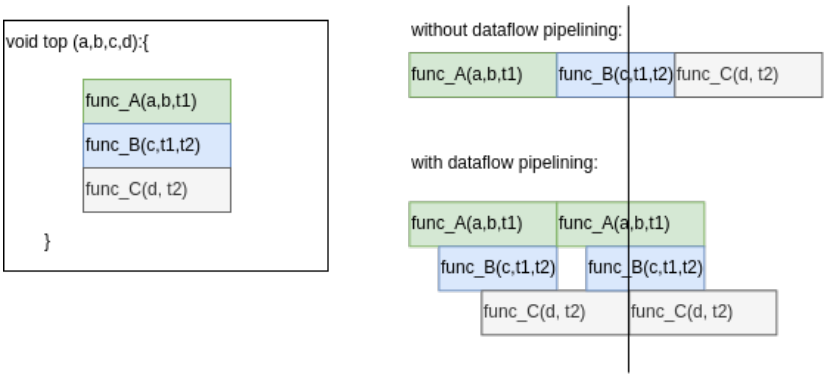


FIGURE 3.3. Dataflow pipeline

Dataflow: The DATAFLOW pragma in Vitis HLS is similar to PIPELINE in that it enables pipelining at the task level, allowing functions and loops to overlap their operations. However, data dependencies between different tasks can impact the flow, as subsequent functions can only access arrays that have finished being written or read in the previous task. DATAFLOW optimization allows operations in a function or loop to start running before the previous function or loop has completed all operations, effectively enabling concurrent execution and improving overall throughput. When using the DATAFLOW pragma, careful attention must be given to synchronization mechanisms to avoid dynamic hazards or unintended stalls, ensuring efficient and correct execution.

When the DATAFLOW pragma is specified, the HLS tool creates channels like ping pong RAMs or FIFOs to support dataflow between sequential functions or loops which enables consumer functions or loops to start before producer functions or loops complete and reduces latency and improves RTL throughput in task-level.

3.2 FAM Quantization Error Analysis

This section computes the quantization error analysis for the fixed precision FAM algorithm, referred to as FAM_M1. (The following section covers a mixed precision FAM algorithm, referred to as FAM_M2.) The tapping-window function $a(n)$ is the Hamming Window [63] in complex demodulate, the $g(n)$ is a square window in the second FFT and **decimation-in-time** Radix-2 FFT in Scheme 2 for both the integration. Figure 3.4 illustrates the signal flow

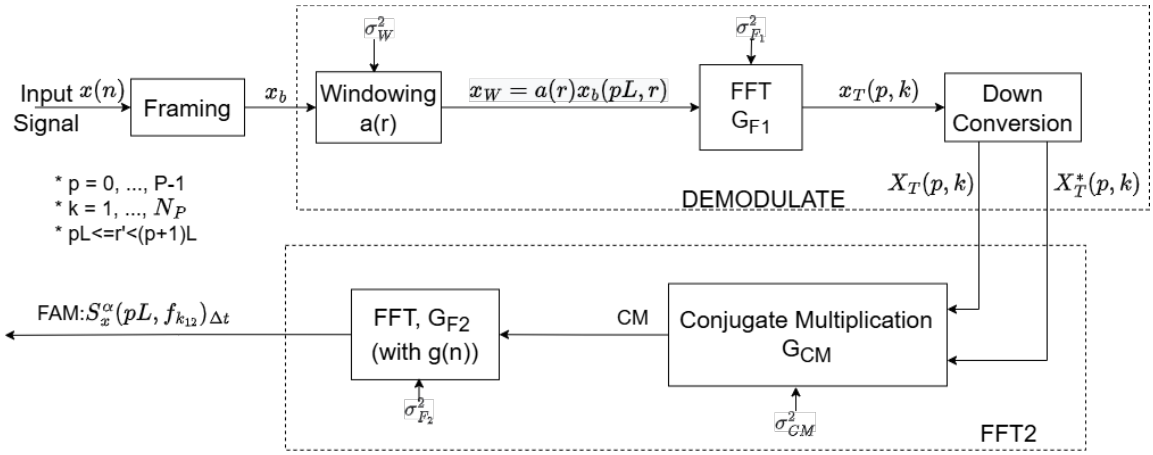


FIGURE 3.4. SCD signal flow graph for FAM_M1 (fixed precision)

TABLE 3.1. Summary of gain and quantization error for each block of FAM_M1 and FAM_M2

Blocks		Gain	Block Error	Output Error	Output Signal	Integer bits
FAM_M1 FAM_M2	Framing	1	-	-	P_s	1
	Windowing	$G_W = \frac{1}{1.59^2}$ [17]	$\sigma^2_w = \sigma^2_{rc}$	$\sigma^2_{N.W} = \sigma^2_{rc}$	$P_{S.W} = \frac{P_s}{1.59^2}$	1
	Down Conversion	1	-	-	-	1
FAM_M1	First FFT	$G_{F_1} = 1/N_P$	$\sigma^2_{F1} = \sigma^2_{F\_M1}$	$\sigma^2_{N.F1} = \sigma^2_{N.W}/N_P + \sigma^2_{F1}$	$P_{S.F1} = P_{S.W}/N_P$	1
	Conjugate Multiply	G_{CM}	$\sigma^2_{CM} = \sigma^2_{cc}$	$\sigma^2_{N.CM}$	$P_{S.CM}$	1
	Second FFT	$G_{F_2} = 1/P$	$\sigma^2_{F2} = \sigma^2_{F\_M1}$	$\sigma^2_{N.F2} = \sigma^2_{N.W}/P + \sigma^2_{F2}$	$P_{S.F2} = P_{S.CM}/P$	1
FAM_M2	First FFT	$G_{F_1} = N_P$	$\sigma^2_{F1} = \sigma^2_{F\_M2}$	$\sigma^2_{N.F1} = \sigma^2_{N.W}N_P + \sigma^2_{F1}$	$P_{S.F1} = P_{S.W}N_P$	$log_2 N_P + 1$
	Normalization 1	$G_{Norm_1} = q_1^2$	$\sigma^2_{q1} = \sigma^2_{rc}$	$\sigma^2_{N.q1} = \sigma^2_{N.F1}q_1^2 + \sigma^2_{q1}$	$P_{S.q1} = P_{S.F1}q_1^2$	1
	Conjugate Multiply	G_{CM}	$\sigma^2_{CM} = \sigma^2_{cc}$	$\sigma^2_{N.CM}$	$P_{S.CM}$	2
	Normalization 2	$G_{Norm_2} = q_2^2$	$\sigma^2_{q2} = \sigma^2_{rc}$	$\sigma^2_{N.q2} = \sigma^2_{N.CM}q_2^2 + \sigma^2_{q2}$	$P_{S.q2} = P_{S.CM}q_2^2$	1
	Second FFT	$G_{F_2} = P$	$\sigma^2_{F2} = \sigma^2_{F\_M2}$	$\sigma^2_{N.F2} = \sigma^2_{N.q2}P + \sigma^2_{F2}$	$P_{S.F2} = P_{S.q2}P$	$log_2 P + 1$
	Normalization 3	$G_{Norm_3} = q_3^2$	$\sigma^2_{q3} = \sigma^2_{rc}$	$\sigma^2_{N.q3} = \sigma^2_{N.F2}q_3^2 + \sigma^2_{q3}$	$P_{S.q3} = P_{S.F2}q_3^2$	1

diagram for FAM_M1. The gain and variance introduced by each block in Figure 3.4 are calculated using the equations from the previous subsection and summarized in Table 3.1. In the figure, the noise introduced in each block is the block error (σ^2_*), the output error ($\sigma^2_{N.*}$) is the sum of the block error and the output error from previous block multiplied by the gain of the current block, and the power of the signal passing out of each block is the output signal ($P_{S.*}$) where $*$ refers to a particular block in the signal flow.

Framing + Windowing:

The Framing block rearranges the data sequence $x(n)$ into P segments, becomes $x_b(pL, r)$, defined as $x(pL + 0 : pL + N_P - 1)$ where $p = 0, ..., P - 1$ and $r = 0, ..., N_P - 1$. The power of the input signal is P_s. For the Windowing block, $x_W = a(r)x_b(pL, r)$ in Equation (3.3)

shows the multiplication between the inputs and the Hamming Window [63] yielding a rounding error ($\sigma_w^2 = \sigma_{rc}^2$). In addition, the power correlation factor of the Hamming Window is 1.59 ($\approx 1/(RMS(Hamming\,Window))$) where RMS is root-mean-square. The signal power after window section becomes $P_{S.W} = P_s/1.59^2$ So, there is approximately a 4 dB ($\approx 10log_{10}(1.59^2)$) reduction in SQNR after windowing.

The First Fast Fourier Transform:
The DIT FFT blocks are employed to implement Equation (3.3). The noise power is given by Equation (2.34) where N_P is the points of FFT, $m_1 = \log_2(N_P)$ denotes the stages of the FFT. The gain of this block is $1/N_P$, meaning that the power of the signal and variance of the noise from the previous window section passing through the FFT section are attenuated by this gain. Based on Equation (2.39), the variance of the noise generated from the FFT is $\sigma_{F1}^2 = \sigma_{DIT.S2}^2 = \frac{2^{-2F}}{3}[2 - \frac{m_1+1.5}{N}]$.

Down Conversion:
To implement the complex demodulate, the FFT output needs to be down-converted. To control cycle leakage and aliasing, L is set to $N_P/4$ and therefore the phase correction, $e^{-j2\pi mpL/N_P}$, can only take the values $(j, -j, 1, -1)$ and does not introduce quantization error or gain to either signal or noise from previous sections.

Conjugate Multiplication:
The function of this block is to compute a complex dot product in Equation (3.5) and the underlying computation is to multiply an input with its conjugate. The error analysis assumes the signals are all independent. Hence, the derivation of signal power after complex conjugate multiplication based on Equation (2.28) becomes

$$P_{S.CM}^2 = P_{s1}^2 P_{s2}^2 \tag{3.7}$$

and the variance of the output noise becomes

$$\sigma_{N.CM}^2 = P_{s1}^2\sigma_{s2}^2 + P_{s2}^2\sigma_{s1}^2 + \sigma_{s1}^2\sigma_{s2}^2 + \sigma_{CM}^2 \tag{3.8}$$

where σ_{s1}^2 and σ_{s2}^2 denote the input noise sources, P_{s1} and P_{s2} are the power of two input signals, and $\sigma_{CM}^2 = \sigma_{cc}^2$ is the quantization error introduced by the complex multiplication. We assume those two signals are independent but have the same input signal and noise power ($P_{S.S} = P_{s1} = P_{s2}$ and $\sigma_{N.S}^2 = \sigma_{s1}^2 = \sigma_{s2}^2$). Thus, Equation (3.8) can be simplified to $\sigma_{N.CM}^2 = 2P_{S.S}^2\sigma_{N.S}^2 + \sigma_{N.S}^4 + \sigma_{CM}^2$ where $\sigma_{N.S}^4$ is approximated as zero due to its high order. The power of the output signal, $P_{S.CM}$, is $P_{S.S}^2$.

The Second Fast Fourier Transform:

The second FFT is of size P. Similar to the first FFT, the noise power of the Second FFT is given by Equation (2.39) where P is FFT size and $m_2 = log_2(P)$ is the number of stages. The gain of this block is $1/P$.

3.2.1 SQNR Noise Model for FAM_M1

The FAM_M1 model employs a fixed wordlength for all signals. This is achieved by truncating multiplications and right-shifting additions so that the result remains in the range $[-1, 1)$.

Combining all the noise terms in Figure 3.4, the output noise variance (σ^2) of FAM_M1 can be reduced to the following form

$$\begin{aligned}\sigma_{FAM\_M1}^2 &= [(\sigma_W^2 G_{F_1} + \sigma_{F_1}^2)G_{CM} + \sigma_{CM}^2]G_{F_2} + \sigma_{F_2}^2 \\ &= W_W 2^{-2F_W} + W_{F_1} 2^{-2F_1} + W_{CM} 2^{-2F_{CM}} + W_{F_2} 2^{-2F_2} \\ &= \sum_{\#\in\{W,F_1,CM,F_2\}} W_\# 2^{-2F_\#}\end{aligned} \tag{3.9}$$

where $W_\#$ and $F_\#$ are the parameters and the number of fraction bits of each section. The power of the output signal is

$$P_{FAM\_M1} = P_{i.s} G_W G_{F_1} G_{CM} G_{F_2} \tag{3.10}$$

where $P_{i.s}$ is the power of the input signal. The detailed derivation for the FAM_M1 is given in Appendix A1.1.

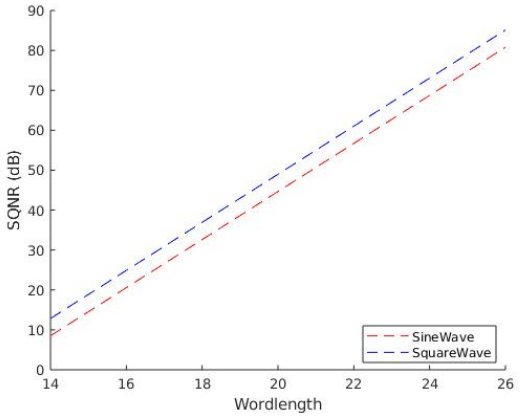


FIGURE 3.5. SQNR performance for the FAM_M1 method at different wordlengths

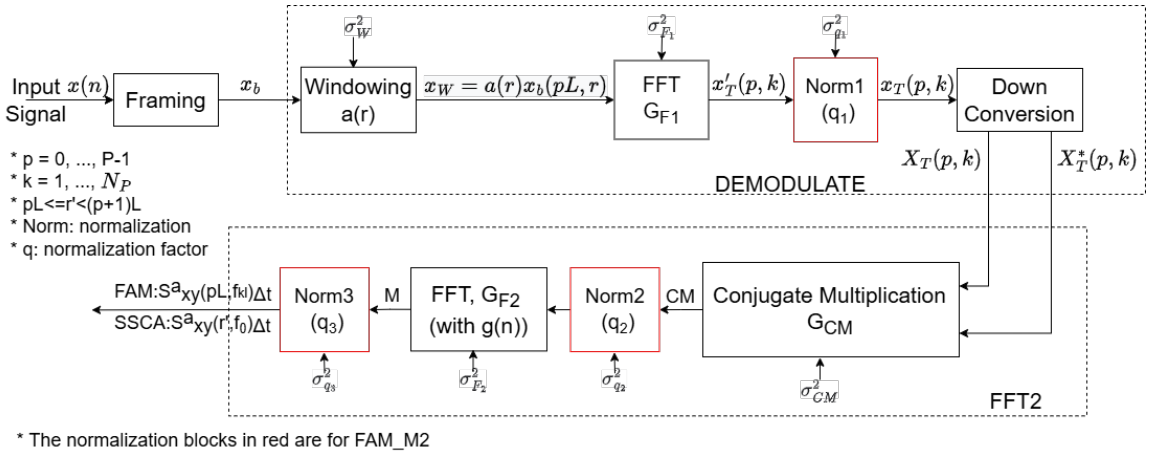


FIGURE 3.6. SCD signal flow graph for FAM_M2 (mixed precision)

The SQNR for the FAM_M1 method is

$$SQNR = 10\log_{10}(\frac{P_{FAM\_M1}}{\sigma^2_{FAM\_M1}}). \tag{3.11}$$

Figure 3.5 shows the SQNR analysis using Equation (3.11) for both sine wave and square wave input using the FAM_M1. A low SQNR results from scaling of each addition operation through a right shift (division by 2) which is necessary to avoid overflow.

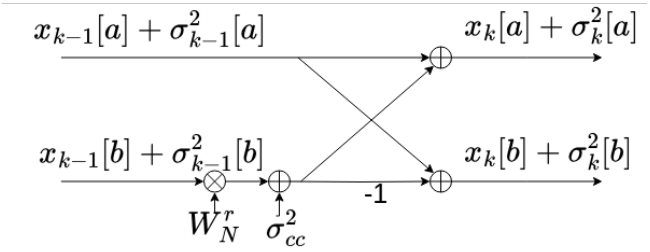


FIGURE 3.7. Quantization noise model of butterfly structure of Radix-2 DIT with only the rounding error (k^{th} Stage, 1<k<m)

3.3 Improving FAM SQNR

Since quantization noise is determined by the number of fractional bits, each right shift in the FAM_M1 model degrades the SQNR. Therefore, we introduce a new FAM_M2 model, which uses mixed precision to improve the SQNR by increasing the number of bits per addition and rescaling to avoid overflow [35]. Figure 3.6 describes the signal flow diagram for FAM_M2 which incorporates the normalization blocks, highlighted in red boxes, to restrict values back to B bits. Consequently, this design is a mixed precision one, and only rounding errors are introduced. Since overflow cannot occur in either FFT or the conjugate multiplications, the Framing, Windowing, and Down Conversion blocks remain unchanged from FAM_M1.

The First Fast Fourier Transform + Normalization 1:
Figure 3.7 illustrates the new noise model for DIT Radix-2 FFT, which has only roundoff noise, accompanied by a normalization to adjust the weight of the signal after the first FFT, conjugate multiplication and second FFT. The other settings are the same as in FAM_M1.

To compute the new noise model for the FFT, the variance for both the lower and upper parts of the k^{th} butterfly become

$$\sigma^2_k[a] = \sigma^2_k[b] = \sigma^2_{k-1}[a] + \sigma^2_{k-1}[b] + \sigma^2_{cc} \tag{3.12}$$

In the k^{th} stage, $N_P/2$ points are multiplied with twiddle factors, introducing an error that doubles when passing through each of the $m - k + 1$ subsequent stages. Thus, the variance

of output of the FFT can be computed from Equation (3.12) to

$$\begin{aligned}\sigma^2_{E\_M2} &= \sigma^2_{cc}\frac{N_P}{2}\frac{1}{N_P}(2^m + 2^{m-1} + ... + 2) \\ &= (2^m - 1)\sigma^2_{cc}.\end{aligned} \tag{3.13}$$

Removing the quantization error of the twiddle factor $\in \{j, -j, 1, -1\}$ from the Equation (3.13), the variational expression for the FFT quantization error is

$$\sigma^2_{F\_M2} = (\frac{2^m}{6} - 1)\frac{2^{-2F}}{3}. \tag{3.14}$$

The noise power of the new system is given by Equation (3.14). Because additions in each FFT stage require an additional integer bit to avoid overflow, the complex output is scaled via the (real) normalization factor q_1 so the real and imaginary parts are $\in [-1, 1)$. This introduces variance σ^2_{q1}.

Conjugate Multiplication + Normalization 2:

The computation part is the same as the previous method. The normalized factor for this part is q_2, which rescales to produce a variance σ^2_{q2}.

The Second Fast Fourier Transform + Normalization 3:

The second FFT is P points to estimate the sum with q_3 normalization factor and variance σ^2_{q3}.

3.3.1 SQNR Noise Model for FAM_M2

Combining all the noise and gain values of Figure 3.6 and using the definitions in Table 3.1, the output noise variance and power of output signal can be written as

$$\begin{aligned}\sigma^2_{FAM\_M2} &= [((((\sigma^2_W G_{F_1} + \sigma^2_{F_1})G_{Norm_1} + \sigma^2_{q_1})G_{CM} + \sigma^2_{CM})G_{Norm_2} + \sigma^2_{q_2})G_{F_2} \\ &\quad + \sigma^2_{F_2}]G_{Norm_3} + \sigma^2_{q_3} \\ &= W_W 2^{-2F_W} + W_{F_1} 2^{-2F_1} + W_{CM} 2^{-2F_{CM}} + W_{F_2} 2^{-2F_2} \\ &= \sum_{\#\in\{W,F1,CM,F2\}} W_\# 2^{-2F_\#}\end{aligned} \tag{3.15}$$

and

$$P_{FAM\_M2} = P_{i.s} G_W G_{F_1} q_1^2 G_{CM} q_2^2 G_{F_2} q_3^2. \tag{3.16}$$

The detailed derivation for the FAM_M2 is given in Appendix A1.2.

3.4 Implementation

The FAM algorithm can be decomposed into three sections: DEMODULATE, FFT2, and Sparse SCD. Figure 3.4 indicates the details of the first two parts, and the Sparse SCD will be presented in Section 3.4.3. This section first shows a naive baseline implementation, followed by a second design where computational efficiency is maximized through parallelism. Finally, a technique for I/O bandwidth reduction which improves system-level performance is described. An HLS-based parallel architecture is designed and integrated into a Zynq processing system on the Xilinx ZCU111 board, on top of which a Jupyter notebook is developed to visualize the results.

C/C++-based synthesis via Vivado HLS was chosen in preference to a register transfer language (RTL) design flow such as VHDL or Verilog because it allows non-FPGA experts to modify the code, directly supports fixed-point types, and has high design productivity.

The `ap_fixed` [41] bit-accurate fixed-point library, greatly facilitated the comparison of our theoretical SQNR models with simulations, and hardware implementations in Verilog could be generated from the same source code. In this library, a fixed-point data type is represented using the C++ template `ap_[u]fixed<W,I,Q,O,N>`, where W is the wordlength in bits, I the number of integer bits, Q the quantization mode, O the overflow mode and N is the number of saturation bits in the overflow wrap mode [41].

3.4.1 Baseline Implementation

An implementation based on Figure 3.6 was first developed. The system accepts a window of input data and calculates all the outputs for each block, and the results are streamed to

Algorithm 1: Baseline implementation

Input: Stream in.
Output: Stream out.
pragma HLS DATAFLOW
$xin[0:P-1, 0:N_P-1] \leftarrow$ Framing$(in.read())$ ▷ `Framing`
$x_w[0:P-1,:] \leftarrow xin[0:P-1,:] * a[:]$ ▷ `Windowing`
$x_T[0:P-1,:] \leftarrow \mathbf{N_P}$-dimensional FFT$(x_w[0:P-1,:])$ ▷ `First FFT`
$X_T[0:P-1,:] \leftarrow$ Down Conversion$(x_T[0:P-1,:])$ ▷ `Down Conversion`

for $i \leftarrow 0$ **to** $N_P - 1$ **by** *1* **do** ▷ `Conjugate Multiplication`
  for $j \leftarrow 0$ **to** $N_P - 1$ **by** *1* **do**
    $CM[:, i * N_P + j] \leftarrow X_T[:, i] * conj(X_T[:, j])$
  end
end

$M[:, 0 : N_P * N_P - 1] \leftarrow$ **P**-dimensional FFT$(CM[:, 0 : N_P * N_P - 1])$ ▷ `Second FFT`
$P_a[:,:] \leftarrow M[P/2 : (3P/4 - 1), 0 : N_P * N_P - 1]$
$P_b[:,:] \leftarrow M[P/4 : (P/2 - 1), 0 : N_P * N_P - 1]$
$out.write() \leftarrow \{P_a, P_b\}$
return *out*

subsequent blocks. The parallelism between blocks is achieved via DATAFLOW pragmas in the HLS description. The pseudocode for the baseline implementation is listed in Algorithm 1. The blocks operate independently in a pipelined manner, so the throughput is equal to the throughput of the block with the highest II. The Second FFT part requires $O(N_P^2)$ operations and hence is the computational bottleneck.

3.4.2 Computation Optimization

To optimize the design we employ spatial parallelism by instantiating DSTRIDE parallel units for the `DEMODULATE` computation and FSTRIDE parallel units for `FFT2`. We note that `DEMODULATE` (framing, windowing, N_P-point FFT and down-conversion) is computed row-wise, whereas `FFT2` requires column-wise inputs (Algorithm 1). This makes their boundary a natural place for a pipeline stage.

Algorithm 2: Merging of Algorithm 1

Input: Stream in.
Output: Stream out.
pragma HLS DATAFLOW
$x_b[0:P-1, 0:N_P-1] \leftarrow$ Framing$(in.read())$ ▷ `Framing`

for $i \leftarrow 0$ **to** $P-1$ **by** *1* **do** ▷ `DEMODULATE`
  $x_w[:] \leftarrow x_b[i,:] * a[:]$ ▷ `Windowing`
  $x_T[:] \leftarrow$ $\mathbf{N_P}$-dimensional FFT$(x_w[:])$ ▷ `First FFT`
  $X_T[i,:] \leftarrow$ Down Conversion$(x_T[:])$ ▷ `Down Conversion`
end

for $i \leftarrow 0$ **to** N_P-1 **by** *1* **do** ▷ `FFT2`
  for $j \leftarrow 0$ **to** N_P-1 **by** *1* **do**
    $CM[:, i*N_P+j] \leftarrow X_T[:,i] * conj(X_T[:,j])$
    $M[:] \leftarrow$ $\mathbf{P}$-dimensional FFT$(CM[:, i*N_P+j])$
    $P_a[:] \leftarrow M[P/2 : (3P/4-1)]$
    $P_b[:] \leftarrow M[P/4 : (P/2-1)]$
    $out.write() \leftarrow \{P_a, P_b\}$
  end
end
return out

Our overall strategy is to create a pipeline where the `DEMODULATE` and `FFT2` stages have a similar throughput interval. In Algorithm 2, we merge related blocks in the same loop so that we can circulate arrays with N_P items in complex demodulation, or pass arrays with P items in the second part of the loop. Streams are characterized by reading or writing once in each loop, and the HLS PARTITION pragma is used to parallelize array accesses.

The pseudocode for computing the `DEMODULATE` and `FFT2` stages is presented as Algorithm 3. Referring to Figure 3.4, the x_w, x_T, and X_T variables represent the output arrays of windowing, first FFT, and down conversion steps (described in Section 3.2) respectively. We implement the FFT for the `DEMODULATE` block using the Xilinx FFT library. The FFT IP core library computes the unscaled fixed point precision DIT FFT and rounds to the specified wordlength after the butterfly, which means the rounding error comes from the real and imaginary part of the complex result.

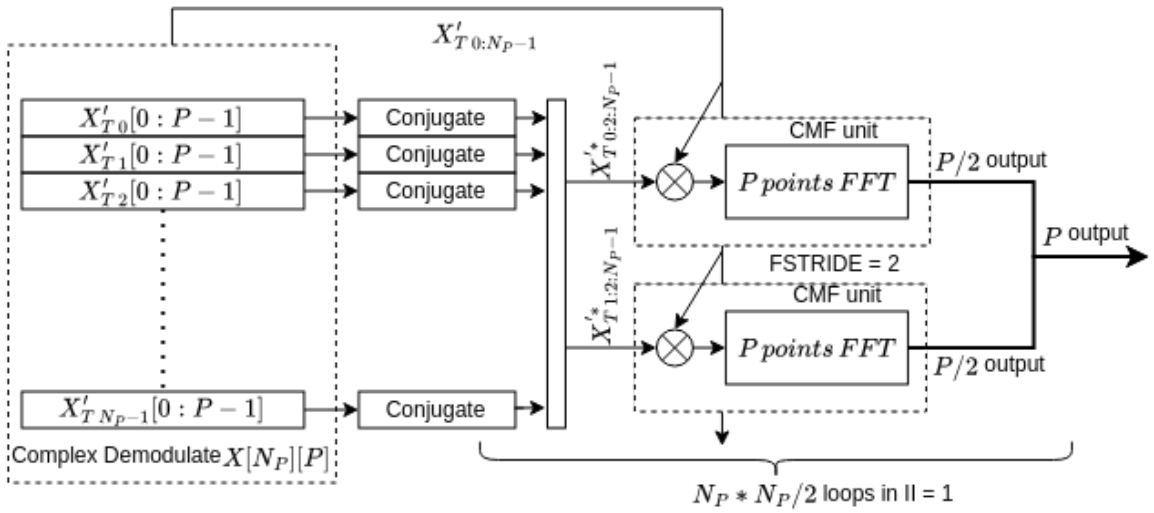


FIGURE 3.8. SCD signal flow graph (CM+FFT) based on HLS design with FSTRIDE=2 (Part 2)

The `DEMODULATE` computation requires less resources than the following `FFT2` part, and so compile-time parameters are introduced to balance the throughput interval of each stage. In Algorithm 3, DSTRIDE controls the degree of parallelism. The `save_in` function is used for buffering. The final stage of `DEMODULATE`, `array_reorder`, is transferring the matrix from the size of $DSTRIDE \times (P/DSTRIDE) \times N_P$ to $N_P \times P$ and reordering in preparation for `FFT2`.

Figure 3.8 illustrates the dataflow for the `FFT2` part of the FAM implementation. The initial stages prepare data for parallel Conjugate Multiplication + FFT (CMF) units (Figure 3.9). FSTRIDE determines the degree of parallelism used in the CMF units. A total of FSTRIDE×CMF units are operated in parallel, with each CMF unit optimized for minimal initiation interval. Together this has $execution\,time = N_P^2 II_{CMF}/FSTRIDE$. For high performance, we wish to have $II_{CMF} = 1$. This can be achieved under the following conditions represented in Figure 3.9:

- an array and its conjugate should be passed to the CMF unit each cycle;
- arrays must use PARTITION to read or write values each cycle;
- the inner for loops must be UNROLLed;

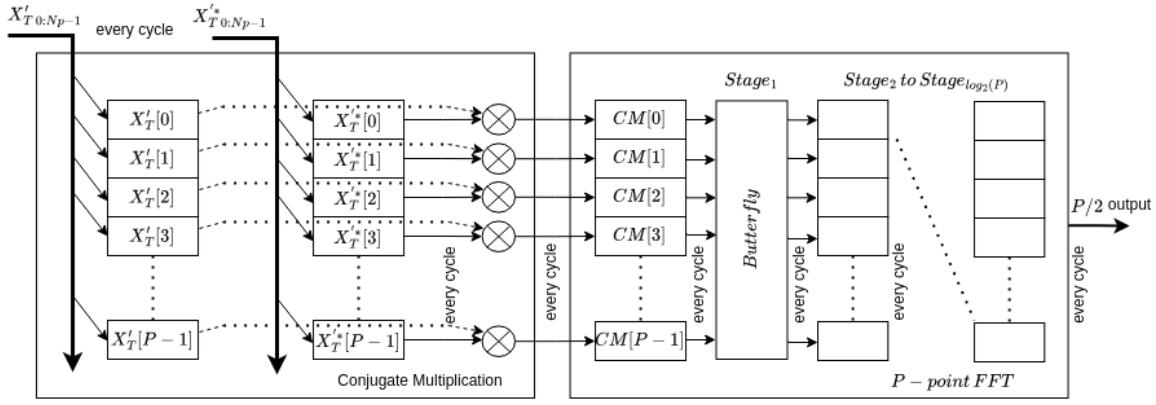


FIGURE 3.9. CMF unit dataflow

- the computation of each stage and the buffers for the result in the P-point FFT must all be independent;
- the P-point FFT executes all butterflies in parallel.

The `FFT2` part of Algorithm 3 shows the pseudocode for Figure 3.8. Before passing the data to the CMF unit, we split and save the conjugate values into FSTRIDE matrices of size $N_P/FSTRIDE \times P$ and PARTITION as $dim = 2$ (the second dimension is partitioned). The data is then transferred and run into the FSTRIDE$\times$ CMF units synchronously. To accept the coming data set in the next cycle, the CMF is fully expanded by UNROLL in a for loop to compute the complex multiplication, and the result is stored in a new CM array. This is then passed to the P-point FFT, and similarly, after each stage of the FFT, the results are saved and passed in a new array while all the butterfly functions in the for loop are fully expanded and computed.

The `FFT2` requires a $Plog_2(P)/2$ butterfly operation and, as illustrated in Figure 3.8, requires one complex multiplication (two real multiplications and four real additions) and two complex additions (four real additions). In `FFT2`, executing all butterflies in parallel will cause high computational complexity. For the twiddle factor whose value is $(j, -j, 1, -1)$, the butterfly calculation can be done by addition only. Therefore, we replace the first two stages of the FFT with pure addition, so that the complex multiplication of $2/log2(P)$ can be reduced.

Algorithm 3: Optimization through spatial parallelism

Input: Stream in.
Output: An array of stream Out with a size of $FSTRIDE \times P/2$.
pragma HLS DATAFLOW

▷ DEMODULATE

$xin[:][:,:] \leftarrow$ save_in$(in.read())$

▷ The size of xin is $[DSTRIDE][P/DSTRIDE, N_P]$

for $i \leftarrow 0$ **to** $P/DSTRIDE - 1$ **do**
  for $n \leftarrow 0$ **to** $DSTRIDE - 1$ **do**
    # pragma HLS UNROLL
    Preprocess:
    {
    # pragma HLS DATAFLOW
    $x_w[n][i,:] \leftarrow xin[n][i,:] * a[:]$
    $X[n][i,:] \leftarrow$ $\mathbf{N_P}$-dimensional FFT$(x_w[n][i,:])$
    $Y[n][i,:] \leftarrow$ Down Conversion$(X[n][i,:])$
    }
  end
end
$X_T[:, 0:P-1] \leftarrow$ array_reorder$(Y[0:DSTRIDE-1][0:P/DSTRIDE-1,:])$

▷ FFT2

for $n \leftarrow 1$ **to** $FSTRIDE$ **do**
  # pragma HLS ARRAY_PARTITION variable = $X_{n.conj}$ complete dim = 2
  $X_{n.conj}[:,:] \leftarrow Conjugate(X_T[n : FSTRIDE : N_P - 1, :])$
end
for $i \leftarrow 0$ **to** $N_P - 1$ **by** 1 **do**
  $X_{temp}[:] \leftarrow X_T[i, :]$
  for $j \leftarrow 0$ **to** $N_P - 1$ **by** $FSTRIDE$ **do**
    # pragma HLS PIPELINE = 1
    for $n \leftarrow 0$ **to** $FSTRIDE - 1$ **by** 1 **do**
      # pragma HLS UNROLL
      CMF unit:
      {
      $CM[:] \leftarrow X_{temp}[:] * X_{n.conj}[j + n, :]$
      $M[:] \leftarrow$ **P**-dimensional FFT$(CM[:])$
      $P_a[:] \leftarrow M[P/2 : (3P/4 - 1)]$
      $P_b[:] \leftarrow M[P/4 : (P/2 - 1)]$
      $Out[n * (P/2) : (n + 1) * (P/2)].write() \leftarrow \{P_a[:], P_b[:]\}$
      }
    end
  end
end
return $stream\,Out[0 : FSTRIDE * P/2]$

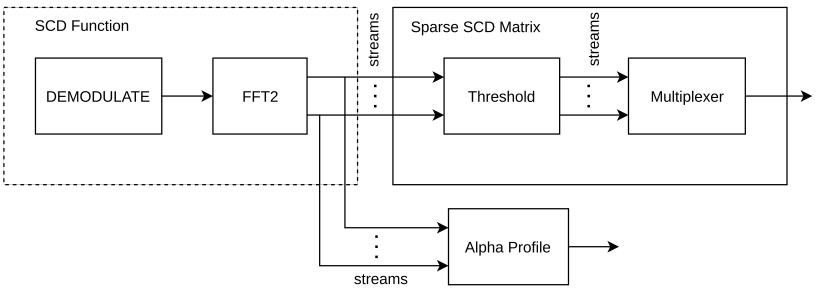


FIGURE 3.10. The SCD function is reduced either via thresholding or computing the alpha profile.

TABLE 3.2. Parameters chosen for our design. We set $L = N_P/4$ and $P = N/L$.

Parameters	N	N_P	P	L	$DSTRIDE$	$FSTRIDE$
value	2,048	256	32	64	1 or 4	2

3.4.3 I/O Optimization

As illustrated in Figure 3.10, the architecture first computes the entire SCD matrix (dotted box) using the approach just described. The number of streams between SCD function to sparse SCD matrix section in Figure 3.10 is $FSTRIDE \times P/2$. The result is a dense matrix, but this chapter is only interested in entries with high correlation, those values being less than 1% of the complete matrix. The right-hand box illustrates a block to create a sparse SCD matrix. It thresholds multiple parallel streams and then multiplexes them into a single one. The bottom box also illustrates the alpha profile, which is not implemented but could be easily included following the implementing method in [52], which selects, for each α, the maximum magnitude in the SCD matrix and projects it onto the alpha axis via Equation (2.13) [52, 12, 46]. This collapse reduces the output from $2N \times N_P$ samples to $2N$, a compression factor of $1/N_P$. The downside is that detection performance becomes tied to the chosen N_P. The sparse SCD matrix alternative retains the full plane, dropping only coefficients whose magnitude is below a custom-set threshold. Therefore, it preserves an additional dimension of SCD output while avoiding the storage of near-zero values.

Both returning a sparse matrix and generating the alpha profile reduce the amount of data transferred from the accelerator to the host system. This is fortunate because the SCD matrix

has dimension $2N \times 2N_P$, e.g., for the example of Figure 3.1(A), the value of N_P and N are set in Table 3.2, the output has 524,288 values. In comparison, the sparse matrix format in Figure 3.1(B) has only 2,396 values but is able to capture the features of interest.

This implementation of the FAM method just described outputs $FSTRIDE \times P/2$ parallel streams, each having a width of $N_P^2/FSTRIDE$ values. As the number of CMF cells increases, the number of output streams increases, which requires more I/O transactions. Algorithm 4 describes how this implementation filters and outputs the target value with associated position information and eventually merges the multiple data streams into a single one. After the threshold, a converter function combines the nonzero value and its coordinates into an INT64 data type which is streamed to the Multiplex function, which combines data from the parallel streams to produce a stream of outputs that includes value and the label information (frequency label (*flabel*) and cycle frequency label (*alabel*)).

3.4.4 Exploiting Symmetry

The Section 3.1.1 has shown that the representation of the SCD estimation $\hat{S}_x^{\alpha}(f)$ is a 2-D feature map with the f and α axes. It is symmetrical in the bi-frequency dimension [30] as indicated in Equations (3.17) and (3.18).

$$\hat{S}_x^{\alpha}(f) = \hat{S}_x^{\alpha}(-f) \tag{3.17}$$

$$\hat{S}_x^{-\alpha}(f) = \hat{S}_x^{\alpha}(f)^* \tag{3.18}$$

Thus, it is sufficient to compute a quarter of the SCD matrix $\hat{S}_x^{\alpha}(f)$, reducing computation by 25%. Figure 3.11 shows an example for $N_P = 256$ and $N = 2,048$ (the i and j symbols are from the `FFT2` section of Algorithm 3). Note that only Pa and Pb contribute to the SCD matrix, represented by the light and dark colors of the same color. For each color in Figure 3.11, different rows of X_T are represented, and for the same color from left to right j

[1]Algorithm 4: for $FSTRIDE = 2$;

Algorithm 4: Sparse SCD

▷ `Threshold`

Input: FSTRIDE array stream $In_{1,...,FSTRIDE}$, each of size of $P/2$ and sequence order $countn$.
Output: FSTRIDE array stream $Out_{1,...,FSTRIDE}$.
for $j \leftarrow 1$ **to** $FSTRIDE$ **by** 1 **do**
  # pragma HLS UNROLL
  for $i \leftarrow 0$ **to** $P/2-1$ **by** 1 **do**
    # pragma HLS UNROLL
    if $In_j[i] > THRESHOLD$ **then**
      $alabel \leftarrow alpha\_label(countn)$
      $flabel \leftarrow frequency\_label(countn)$
      $Out_j[i] \leftarrow (INT64)pack\{In_j[i], alabel, flabel\}$
    end
    countn = countn+1
  end
end
return $stream\,(Out_{1,...,FSTRIDE}[0 : P/2])$

▷ `Multiplex`

Input: FSTRIDE array stream $In_{1,...,FSTRIDE}$, each of size $P/2$.
Output: Three streams $value$, $alabel$ and $flabel$.
for $j \leftarrow 1$ **to** $FSTRIDE$ **by** 1 **do**
  for $i \leftarrow 0$ **to** $P/2-1$ **by** 1 **do**
    $\{value, alabel, flabel\} \leftarrow unpack(In_j[i])$
  end
end
return $stream\,(value, alabel, flabel)$

increases from 1 to N_P representing the rows of the conjugate matrix, which are arranged in the SCD matrix in the staggered order shown in Figure 3.11. Moreover, in Figure 3.11, the shaded area is the quarter SCD, and it is obtained by changing $i \in [0:1:N_P-1]$ to $i \in [0:1:N_P/2-1]$ and $j \in [0:1:N_P-1]$ to $j \in [i:1:N_P/2-1-i]$ in Algorithm 3.

3.4.5 Cycle Count Summary

After applying the optimizations described above, the pipelining scheme for the `DEMODULATE` and `FFT2` blocks is illustrated in Figure 3.12. Since the `FFT2` stage is the computational bottleneck, we design the II of `FFT2` to meet throughput requirements and then ensure the

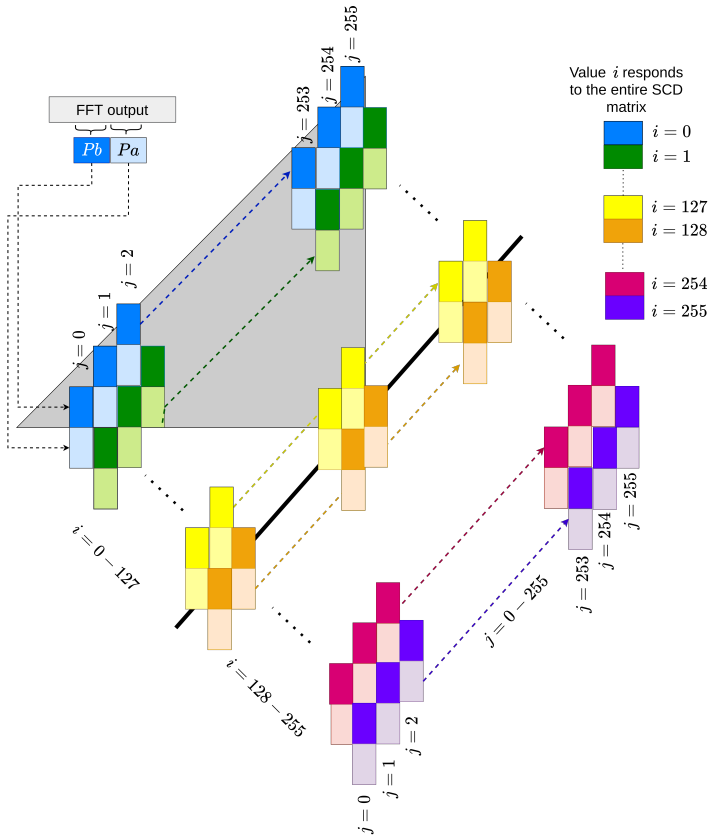


FIGURE 3.11. An example of an SCD matrix (symbol i and j are from Algorithm 3 `FFT2`)

throughput interval of `DEMODULATE` ($II_{DEMODULATE}$) is less than or equal to II_{FFT2}. We apply a DATAFLOW pragma so that the II of each block is equal to the maximum throughput interval over the sub-blocks multiplied by the number of iterations. For the parameters in Table 3.2, the II_{FFT2} of the full and quarter SCD matrix computations are 32,768 and 8,192 respectively. From synthesis reports, the II of `save_in`, `preprocess`, and `save_out` are 1, 875 and 4. Therefore, the initiation intervals of the sub-blocks of `DEMODULATE` are $II_{save\_in} = N_P * P = 8,192$, $II_{Preprocess} = max\{II_W, II_{IP}, II_{DC}\} * P/DSTRIDE = 874*32/DSTRIDE = 27,968/DSTRIDE$, $II_{save\_out} = N_P * P = 8,192$. Thus to match II_{FFT2}, DSTRIDE should be set to 1 and 4 for computing the full and quarter SCD matrix.

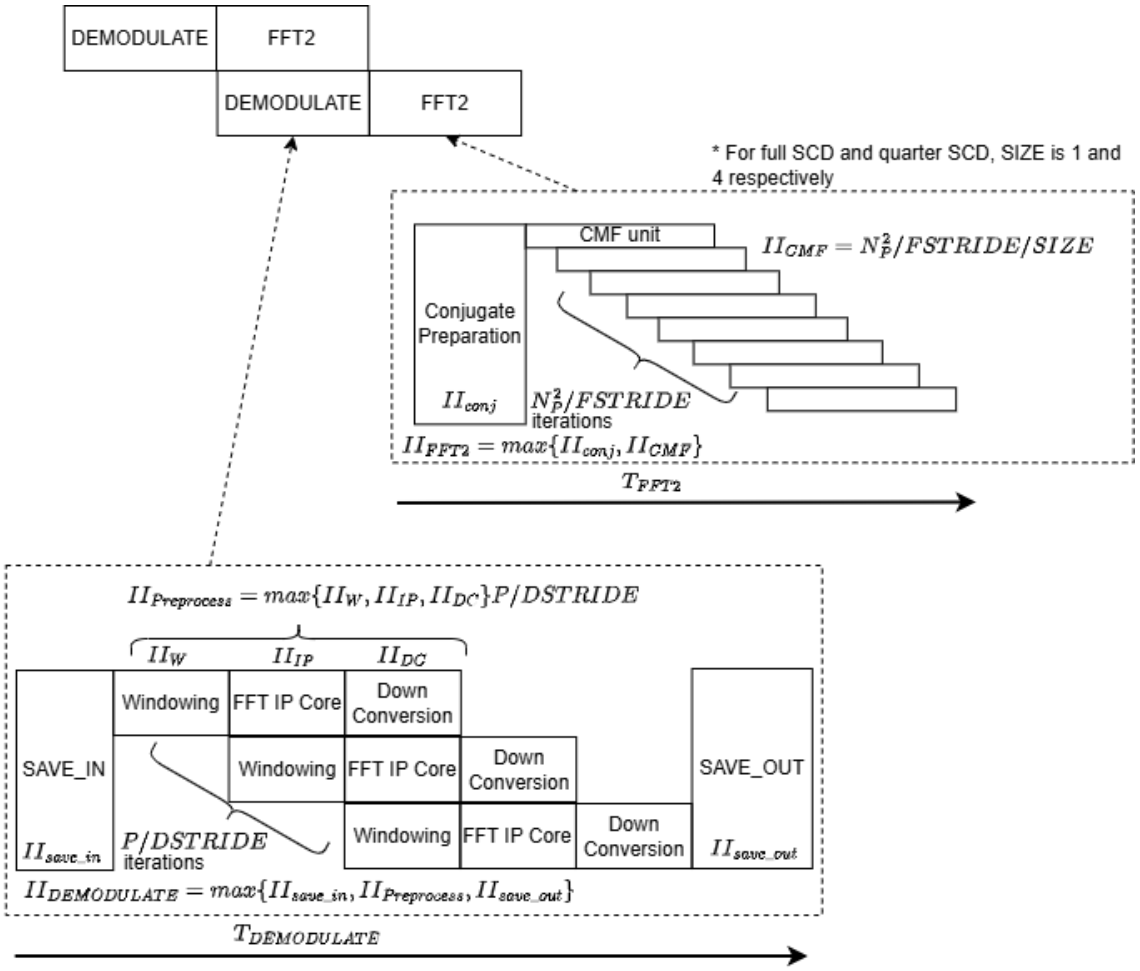


FIGURE 3.12. A cycle-aware system flowchart with pipeline stage details

3.5 Results

The IP blocks for FAM_M1 and FAM_M2 are created separately using the Vivado HLS 2020.1 High-level synthesis tool. Then Vivado 2020.1 was used to generate bitstreams which were tested on a Xilinx ZCU111 RFSoC board which uses a Zynq UltraScale+ XCZU28DR-2FFVG1517E device. We verified the accuracy between the computational expressions of the two methods FAM_M1 and FAM_M2, and the actual operation and compared a range of information regarding SQNR and resource and energy consumption. Although the ZCU111 is larger than needed for just the IP blocks, it was chosen so we could accommodate designs with larger wordlengths and more parallelism. Moreover, future work will integrate the FAM core with high-speed ADCs and deep learning. The implementation is parameterized, so arbitrary

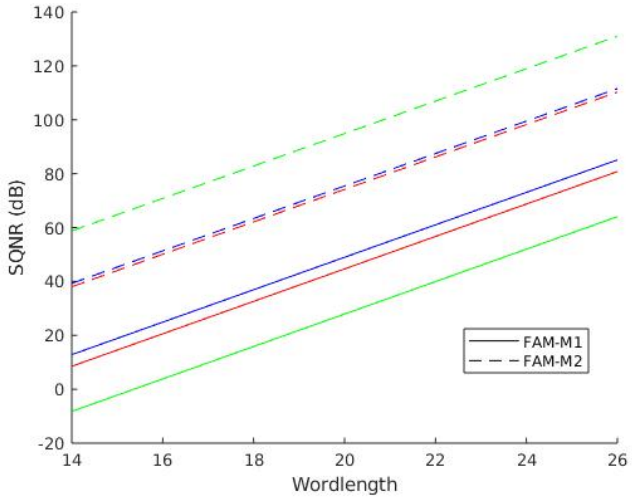


FIGURE 3.13. SQNR performance for the FAM method in our models FAM_M1 and FAM_M2 (theory) at different wordlengths B ($F = B - 1$) (Red: Sine Wave; Blue: Square Wave; Green: Deepsig)

values are supplied. In our experiments, FAM parameters matching the literature [12, 52] were used and these are summarized in Table 3.2. The choice of parallelization parameters is $FSTRIDE = 2$ and $DSTRIDE = 1$ (Full SCD) or $DSTRIDE = 4$ (Quarter SCD).

3.5.1 SQNR

Bit-accurate simulations were made through a direct implementation of the FAM_M1 and FAM_M2 algorithms in C. The `ap_fixed` type in Vivado HLS was used for fixed point arithmetic. The simulations were compared with the mathematical derivations using sine-waves, square-waves, and samples from the DeepSig RADIOML 2018.01A dataset [39]. On FPGA devices, wordlengths up to 18 bits are supported by the embedded DSP blocks, and additional bits can be implemented using the programmable logic, so experiments were focused around this and higher values.

Figure 3.13 shows the SQNR for different uniform wordlengths for Deepsig, Sine Wave, and Square Wave signals using the different methods (FAM_M1 and FAM_M2). The FAM_M2 techniques have considerably improved SQNR compared with FAM_M1. For each signal and method, it can be seen in Figure 3.13 that there is a 6 dB improvement in the SQNR with

each additional bit because $2^{-2(F-1)} = 4 \times 2^{-2F}$. The traces in Figure 3.13 are input-signal dependent because Equation (3.10) and Equation (3.16) depend on the input power. Note that for different input signals, the normalization involves different scale factors, which affects a direct comparison with FAM_M2.

SQNR vs Number of Bits: One of the benefits of the proposed approach is that it enables the best bit allocation achieving the highest SQNR to be determined. Noting that Equation (3.9) and Equation (3.15) are simple sums of products and the SQNR is given by Equation (3.11), the proposed approach to minimizing this expression results in each block providing an equal contribution to σ^2. Given the $W_{\#}$ values, and assigning F bits to the `FFT2` stage, we make each sum term in Equation (3.9) and Equation (3.15) equal via the allocation in Table 3.3.

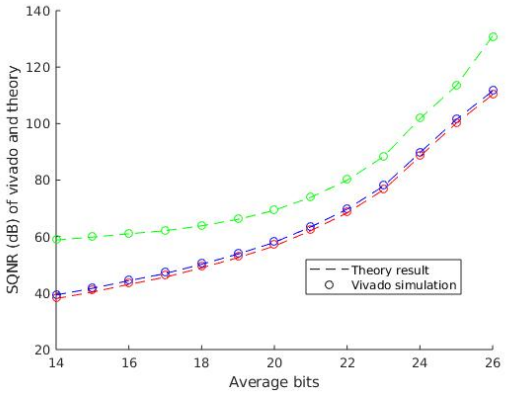
TABLE 3.3. Reduce wordlength with less impact on SQNR

	Methods	Window	FFT1	CM	FFT2
DeepSig	FAM_M1	$F-13$	$F-8$	$F-3$	F
	FAM_M2	$F-1$	$F-1$	F	F
SineWave	FAM_M1	$F-11$	$F-6$	$F-2$	F
SquareWave	FAM_M2	$F-6$	$F-4$	$F-1$	F

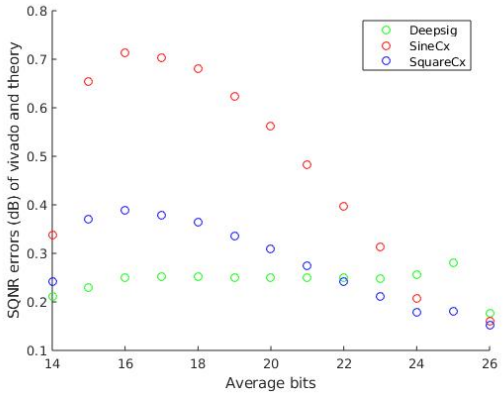
TABLE 3.4. Bit allocation for best SQNR using exhaustive search ($F_{sum} = 72$ Signal: DeepSig) vs Uniform

Methods	Uniform	Window	FFT1	CM	FFT2	SQNR
FAM_M1	No	11 (24-13)	16 (24-8)	21 (24-3)	24	46.04
	Yes	18	18	18	18	15.87
FAM_M2	No	17 (19-2)	17 (19-2)	19	19	84.29
	Yes	18	18	18	18	82.85

Table 3.4 illustrates the potential benefits of using a non-uniform wordlength throughout the computation. This example is run on the Deepsig input, with the optimal bitwidth allocation computed by an exhaustive search over all of the possible bit allocations, for a fixed number of total bits $F_{sum} = 72$. We can see that using a non-uniform number of bits enables one to achieve a higher SQNR. Note that exhaustive search results have reached similar bitwidth configurations to our formulas from Table 3.3; the minor difference being that the formula for FAM_M2 is not designed for a maximum of $F_{sum} = 72$ total bits.



(A) The comparison of theory and Vivado simulation



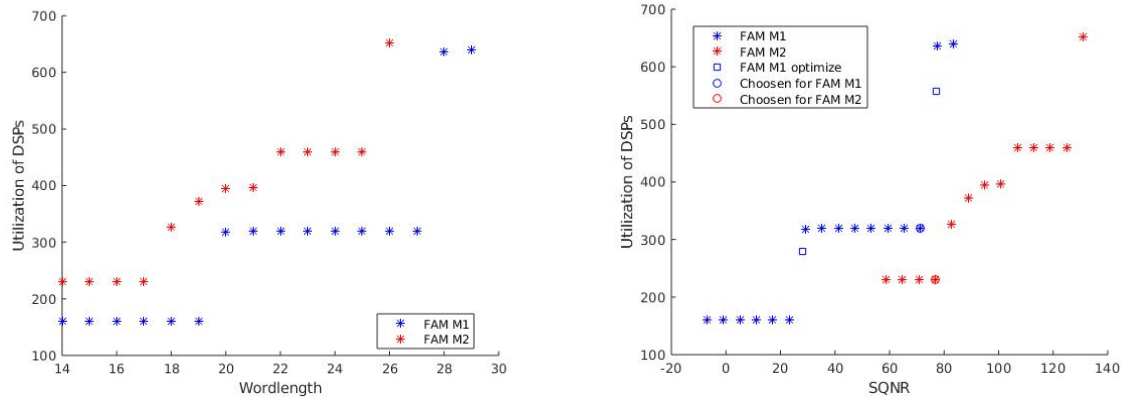
(B) Absolute error between theory and simulation

FIGURE 3.14. Simulation result in average bits FAM_M2 (Red: Sine Wave; Blue: Square Wave; Green: Deepsig)

3.5.2 Vivado HLS Simulation

Bit-accurate simulations using Vivado_HLS were used to verify the theoretical results of Section 3.3. The designs are described in C, compiled, and executed to obtain a bit-exact result. Since the theory directly calculates SQNR from the SCD parameters, it is orders of magnitude faster than simulation using HLS.

Figure 3.14(A) shows the average SQNR with non-uniform accuracy for FAM_M2 for theoretical and HLS simulations with different input signals, where the average number of bits $= (B_W + B_1 + B_{CM} + B_2)/4$ ($B_\# \in [14, 26]$ in steps of 4). Figure 3.14(B) shows the difference between the theoretical and Vivado simulations in Figure 3.14(A). The quantization error analysis aims to evaluate the general error behavior across different signal precisions. We assume that errors are introduced after each addition and multiplication operation. However, for a Sine Wave signal, most of the data, except for the spectral components, approach zero after the first FFT stage. As a result, the difference between the theory and simulation is larger than the Square Wave. The result shows the models described in Section 3.2.1 and Section 3.3.1 can serve as an accurate estimate of the lower bound for SQNR. We further verified the models with other parameter settings (N_P, L, and P) and found similar correspondence between simulation and theory.



(A) Different wordlength (B) Different SQNR

FIGURE 3.15. FAM methods for DSP utilization at different conditions

Given that the bit-accurate simulations match the theoretical results, the methodology described in Section 3.5.1 can be used to search for non-uniform bits allocation to optimize performance for a given resource budget.

The Vivado_HLS synthesis reports the LUTs and DSPs utilization for each design. The DSP48E2 slice contains a 27 bit by 18 bit two's complement multiplier. In Figure 3.15(A), the DSPs utilization of FAM_M1 doubles when the wordlength is increased from 19 to 20 bits. This is because in this number system, 1 integer bit is used for the sign and the remaining 18 bits are used as fraction bits. For FAM_M2, as we assume that there are enough integer bits for the FFT sections and conjugate multiplication, the DSP utilization increases as the number of bits increases. In Figure 3.15(A), at the same wordlength, FAM_M2 requires more DSPs than the FAM_M1. Hence in Figure 3.15(B), at the same SQNR, FAM_M2 consumes fewer DSPs. Furthermore, a non-uniform allocation can be searched based on the area model to balance DSP usage with the SQNR.

3.5.3 FPGA Implementation

The performance of the design is evaluated by comparing it with the state-of-the-art hybrid FPGA-GPU implementation [12] and the state-of-the-art GPU implementation [52], both of which perform alpha-profile calculations in terms of resource utilization, throughput, and

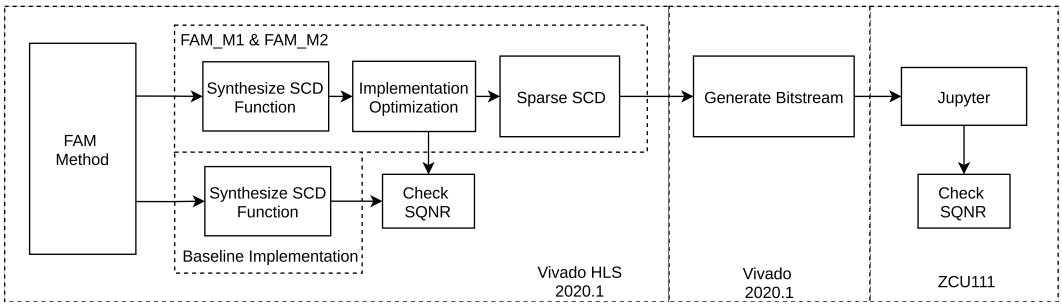


FIGURE 3.16. Verification flow

power consumption. The alpha profile (Figure 3.1(A)) is a one dimensional output, but our sparse SCD (Figure 3.1(B)), includes the frequency axis and, as explained in Section 3.4.3, provides richer information.

The verification process is shown in Figure 3.16. Based on the parameter settings in the previous section, we first apply the baseline implementation of the FAM method, then derive the FAM_M1 and FAM_M2 methods with the implementation optimization mentioned in Section 3.4.2 and Section 3.4.3 in Vivado HLS. Then the SQNR of the full `DEMODULATE+FFT2` was checked using a bit-accurate C simulation. Then, we generate an IP block for each of the two FAM methods with sparse SCD. Bitstreams are then generated in Vivado and tested via a Jupyter Notebook, which controls the execution of this FAM accelerator on the FPGA board. The output of the Jupyter Notebook is compared with a floating point. This chapter chooses 16 and 24 bits as the wordlength for the design to trigger the doubling of the number of DSPs required for implementation (see Figure 3.15(A)). In terms of implementation, the data transfer between IP blocks will use AXI, which supports wordlengths in 16 bit multiples.

Table 3.5 gives a comparison of FPGA resource usage and operating frequency between an FPGA-GPU hybrid design [12], a recent FPGA Verilog design by Li et al. [46], the baseline implementation, and our optimized implementation. It can be seen that the hybrid FPGA-GPU design uses very few FPGA resources since the FPGA is responsible for only a small part of the overall algorithm and it runs at a much lower clock frequency. While Li et al. [46] achieved a high clock frequency, it has higher resource utilization for the same wordlength. Even though our baseline implementation achieves a maximum clock frequency of 330 MHz, its initiation interval is 29 M cycles, so it takes 87.88 ms to process a window. In contrast, our

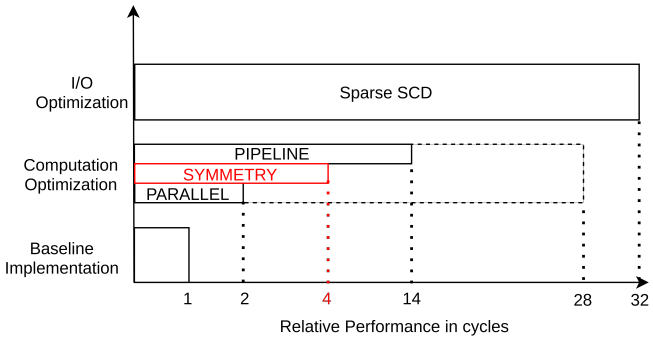


FIGURE 3.17. Speedup breakdown for full-size SCD compared to baseline implementation in cycles. The total speedup is the product of all these optimizations.

TABLE 3.5. Comparison of FPGA resource usage and operating frequency

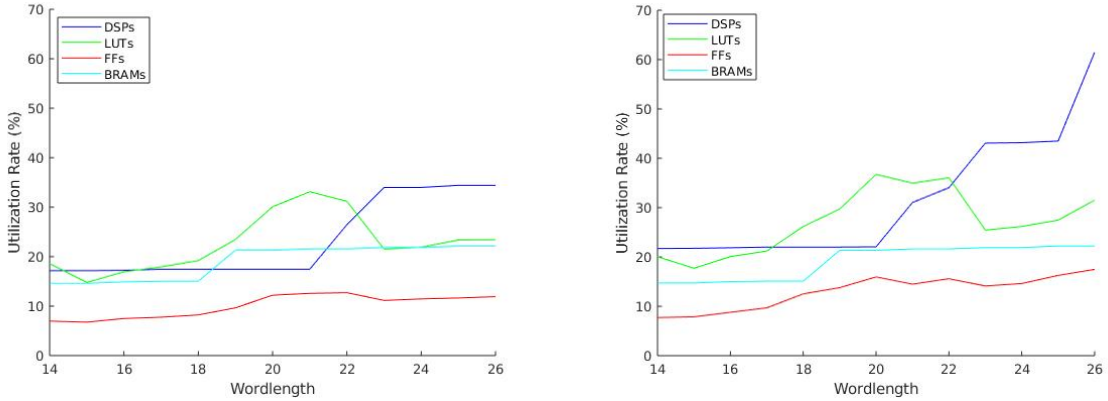
		wordlength	LUTs	FFs	BRAMs	DSPs	Power (W)	SQNR (dB)	Fmax
Full SCD	Hybrid FPGA-GPU design [12]	-	69 (0.1%)	153 (0.1%)	4 (2.9%)	0 (0%)	-	-	140MHz
	Avaliable on ZedBoard	-	53,200	106,400	140	220	-	-	-
Full SCD	Baseline implementation	16 bit	38,345 (9.0%)	46,557 (5.5%)	23 (2.1%)	226 (5.3%)	2.91[2]	71.05(70.88)	330MHz
	Optimized FAM_M1	16 bit	72,140 (17.0%)	63,699 (7.5%)	161 (14.9%)	736 (17.2%)	6.00[2]	5.41(3.83)[1]	200MHz
	Optimized FAM_M2	16 bit	85,050 (20.0%)	74,722 (8.8%)	162 (15.0%)	934 (21.9%)	6.10[2]	71.05(70.88)[1]	200MHz
Quarter SCD	Li et al. [46]	16 bit	150,802 (35.5%)	150,824 (17.7%)	264 (24.4%)	1,054 (24.7%)	12.5[2]	-	530MHz
	Optimized FAM_M2	16 bit	97,603 (23.0%)	89,477 (10.5%)	177 (16.4%)	1,048 (24.5%)	7.73[2]	71.05(70.88)[1]	200MHz
	Resources on ZCU111	-	425,280	850,560	1,080	4,272	-	-	-

[1] The theory values are in parentheses
[2] Total on-chip power estimated by Vivado

optimized design achieves only a clock frequency of 200 MHz, but it takes only 33k clock cycles and has an execution time of 0.165 ms per window.

The optimization steps described in Section 3.4 include pipelining, parallelism, I/O, and symmetry. Figure 3.17 shows a bar chart with the performance gain achieved by each optimization. Since the computational bottleneck is `FFT2`, we set FSTRIDE to 2, doubling its performance. Pipelining minimizes the II and achieves a speedup of 14. The I/O optimization parallelizes this stage with a speedup of 32 (value of P). Therefore the total speedup for the full-size SCD is the product of all of these factors, $2 \times 14 \times 32 = 896$. Finally, symmetry as shown in Figure 3.17 allows this design to avoid computing the full SCD, reducing computing for the Quarter SCD by a factor of 4, and improving the speedup to 3,584.

The FAM_M2 is also compared with FAM_M1. In Table 3.5, FAM_M2 achieves a higher SQNR for the same wordlength, but requires more resources. Thus, in Figure 3.18 the implementation resource utilization is plotted from 14 bits to 26 bits, showing that even at 24 bits FAM_M1 utilizes more resources than FAM_M2 at 16 bis but still has a lower



(A) FAM_M1 utilization rate (B) FAM_M2 utilization rate

FIGURE 3.18. Comparison of FPGA resource utilization between FAM_M1 and FAM_M2 (wordlengths from 14 bits to 26 bits)

SQNR (53.32 dB in Figure 3.13), this indicates that the improvement in SQNR for FAM_M2 outweighs the additional resources. We also explored 16 bit, a half-precision floating point that uses 5 exponent bits and 10 fractional bits. This achieved an SQNR of 60 dB with $4\times$ higher DSP utilization than the optimized 16 bit FAM_M2.

Referring to the `FFT2` block in Figure 3.6, we denote the number of real-valued multiply-accumulate (MAC) operations required per window for the conjugate multiplication, norm2, `FFT2` and norm3 blocks as O_{CM}, O_{n2}, O_{FFT2}, and O_{n3} respectively. Since these components account for the majority of FAM operations, we estimate the total number of MACs (N_{MAC}) as (Article 6 of reference [29]):

$$N_{MAC} \approx O_{CM} + O_{n2} + O_{FFT2} + O_{n3} \tag{3.19}$$

$$\approx 4N_P^2P + 2N_P^2P + 2N_P^2P\log_2 P + 2N_P^2P. \tag{3.20}$$

Counting a MAC as 2 operations, number of cycles, $N_{cycles} = N_P^2$, and clock frequency $f_{clk} = 200\ MHz$, we estimate the numerical performance in billion operations per second to be

$$GOPS = \frac{2N_{MAC} \times f_{clk}}{N_{cycles}} \tag{3.21}$$

$$\approx 460. \tag{3.22}$$

TABLE 3.6. Comparison of throughput and power consumption for the same configuration of FAM

	Full SCD[1]			Full SCD[2]	Quarter SCD[1]		Quarter SCD[2]
	GPU [12]	GPU [12]	FPGA+GPU [12]	Optimized	GPU [52]	FPGA [46]	Optimized
Platform	Tegra K1	Tesla K20	ZedBoard+Tegra K1	ZCU111	Tesla K40	ZCU111	ZCU111
Initiation Interval (ms)	111.61	8.98	50.95	**0.164**	0.303	0.065	**0.041**
Throughput (MS/s)	0.018	0.228	0.040	**12.5**	6.8	31.5	**50**
Speedup	1	12.3	2.1	**677.6**	366.7	1,704.4	**2,710**
Computational Performance (GOPS)	0.14	1.75	0.30	**460**	13.0	60.4	**460**
Power (W)	3.5	51	5	**35(6.1)**[5]	55.5[3]	12.5[4]	**37(7.7)**[5]
Energy (mJ)	390.64	457.98	254.75	**1.00**[6]	16.82	0.81	**0.32**[6]
Signal-to-quantization noise ratio (dB)	-	-	-	**73**[7]	-	-	**73**[7]

[1] Output is the alpha profile.
[2] Output is the sparse SCD.
[3] Power consumption is estimated by scaling to the result of [12].
[4] Power consumption is reported by Vivado report_power [46].
[5] The system power of entire ZCU111 board (power consumption is reported by Vivado report_power).
[6] Energy is calculated using Vivado report_power value.
[7] An example of FAM_M2 using 16 bits. The system supports quantization error analysis for custom wordlengths.

When computing one-quarter of the SCD, the $GOPS$ do not change as N_{cycles} and N_{MAC} are both reduced by the same amount. Furthermore, the designs in references [12, 46, 52] referenced in Table 3.6 calculate the alpha profile by first computing a full-sized or quarter SCD matrix.

We also measured the CPU performance of the FAM_M2 design using single-precision floating point arithmetic. The GNU gcc compiler was used with "-Wall –std=c++14 -O3" parameters which gave the best performance. On an Intel Core i7-9700 operating at 3.00GHz with 8 cores; Memory: 32 GB; and System: Ubuntu 18.04.5 LTS, a profile confirmed that 93% of the time was spent in `FFT2` (15 ms).

Power consumption was measured using an Ecoflow River Pro inverter. The Table 3.6 reports the system power (power consumption measured at the AC power supply for the ZCU111 transformer) and total FPGA power (dynamic + static) as reported by the Vivado report_power command in parentheses. Figure 3.19 shows the EcoFlow AC power supply powering the ZCU111. It is unclear to the authors whether the power consumption figures in reference [12] refer to system power or dynamic power, and power consumption is not reported in [52].

Li et al. [46] extrapolated Nvidia K40 performance (2,880 CUDA cores, 745 MHz clock) to a more recent RTX 3080 Ti GPU (8,960 CUDA cores, 1,365 MHz clock) and estimated a potential performance improvement of 5.5x [46]. The HLS design can achieve this performance.

FIGURE 3.19. Measuring the power consumption of the ZCU111 via AC power supply

TABLE 3.7. Performance of two FAM methods running on FPGAs in 16 bits for different sizes of SCD matrices

	Size of SCD	SCD Function Time (ms)	Interface Delay (ms)	Execution Time (ms)	SQNR (dB)	Error[1]
FAM_M2	Full	0.164	0.161	0.266	73.41	$2^{-13.7}$
	Quarter	0.041	0.055	0.088	73.66	2^{-14}

[1] The max error compared with floating point.

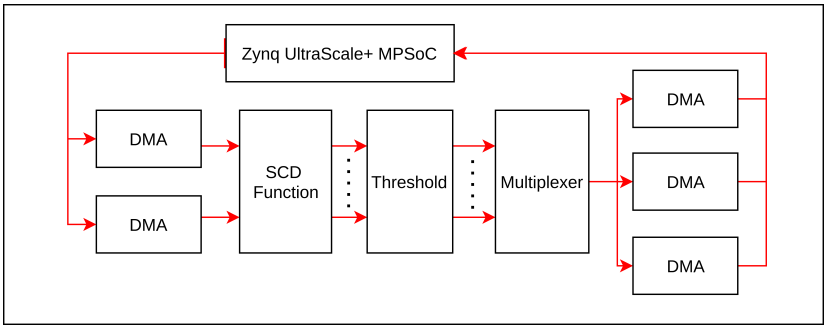


FIGURE 3.20. Interface delay

However, the RTX 3080 Ti has a maximum power consumption of 350W so we believe this design will be more energy efficient.

Table 3.7 lists the execution time measured on the ZCU111 FPGA board. The execution time is the average time over 100 windows of input data and is larger than the FAM time due to data transfer overheads. Figure 3.20 is a block diagram illustrating the interface delay of the system. To estimate the SCD function execution time, the threshold and multiplexer blocks were removed, keeping only the data transfer. In Table 3.7, it can be seen that the total execution time is close to the sum of SCD time and data transfer delay. The error of

the sparse SCD output is also calculated, and the maximum error compared to floating-point results for 16 bit data (15 fraction bits) was $2^{-13.7}$ and 2^{-14} for full-size or quarter-size sparse SCDs. The sparse SCD matrix was verified with the expected value.

3.6 Summary

The work described in this chapter derives explicit expressions to estimate SQNR for the FAM technique in fixed precision (FAM_M1) and mixed precision (FAM_M2). This enables an understanding of how different blocks contribute to overall SQNR, and area-precision tradeoffs to be navigated in an analytical manner. Based on the quantization error analysis, the FAM_M2 significantly improves SQNR with a minor increase in DSP resources (16 bit wordlength) and is hence preferable. The simulations confirm that the analytic result matches the bit-exact simulation to within 1 dB.

An HLS-based FPGA design is implemented on a Xilinx Zynq UltraScale+ XCZU28DR-2FFVG1517E RFSoC with the FAM_M1 and FAM_M2 quantization schemes. Using less than 25% of the available LUT and BRAM resources on the device, it consumes 7.7 W total on-chip power and has a power efficiency of 12.4 GOPS/W, which is an order of magnitude improvement over an Nvidia Tesla K40 GPUs implementation [52]. In terms of throughput, it achieves 50 MS/sec, which is a speedup of 1.6 over a recent optimized FPGAs implementation. High performance was achieved by exploiting spatial parallelism, pipelining, I/O optimization, and symmetry. Together, these techniques enable a design with state-of-the-art throughput and energy consumption.

The quantization error analysis is also significant for SSCA, with the detailed derivation provided in Appendix A2. Since SSCA is a time-smoothing method as FAM, its implementation on the same FPGA platform follows a similar approach. Therefore, we prefer to explore the implementation of SSCA on a different architecture. This work can also be extended to block floating point and other number systems. It can be integrated with data acquisition and deep neural networks for prediction in real-time applications.

CHAPTER 4

Versal-Based Implementation of a Strip Spectral Correlation Analyzer

---

Recall that if the probability distribution of a time series exhibits periodic variations, the series is considered as *cyclostationary* [31, 30]. Cyclostationary signals display periodic statistical properties, which can be characterized using the SCD. The SCD captures the idealized temporal cross-correlation between all pairs of narrowband spectral components, providing a comprehensive representation of the signal's correlation as a function of spectral and cycle frequency.

Since the 1990s, computationally efficient algorithms for estimating the SCD have been studied [32, 59, 6]. Building upon the FFT, Roberts et al. introduced the FAM and the SSCA. The SSCA is widely used due to its computational efficiency and uniform frequency resolution [59, 8].

This chapter studies the real-time implementation of the SSCA, accelerated by the AIE and FPGA, on the AMD/Xilinx VCK5000 Gen4x8 QDMA Versal ACAP platform, which is a highly integrated system-on-chip (SoC) that unifies CPUs, DSPs, I/O, RAM control, and PL into a single device. The implementation is available as open-source on GitHub [1]. The aim is to optimize performance and resource utilization, enabling efficient cyclostationary analysis across the entire bifrequency plane. This accelerates signal processing by utilizing the parallel architecture of the AIE.

The main contributions of this chapter are:

[1] https://github.com/Jingyi-li/SSCA_Implementation.git

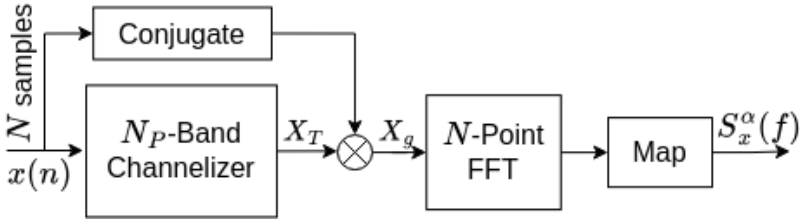


FIGURE 4.1. The strip spectral correlation analyzer signal flow

- A comprehensive design methodology for implementing the SSCA on the Versal heterogeneous platform, leveraging AIE acceleration and optimizing communication between the DDRMC and AIE through the PL for seamless data management.
- The development of a novel SSCA implementation utilizes a decomposed FFT for the second FFT stage to better accommodate hardware constraints. This approach adjusts the SSCA structure to minimize intermediate matrix storage by incorporating a naive decomposed FFT, resulting in a highly efficient processing structure that is well-suited for large-scale inputs.
- The implementation demonstrates a significant performance improvement, achieving a 2.36x speedup compared to GPU implementations and a 99.1x speedup compared to CPU implementations. Additionally, it showcases optimized utilization of the Versal platform resources, including AIE, PL, and DDR memory, effectively balancing computational throughput and resource efficiency.

The rest of this chapter is structured as follows: Section 4.1 provides background on the conventional implementation of SSCA on the VCK5000. Section 4.2 introduces a derivation of SSCA based on the decomposed FFT. Section 4.3 discusses the floating-point implementation of SSCA with support for large input datasets. Section 4.4 presents the experimental results, followed by a summary in Section 4.5. This chapter focuses on the floating-point implementation of SSCA. Thus, the quantization error analysis of fixed-point and mixed-point precision are covered in Appendix A2.

4.1 Background

This section expands on the concepts introduced in Section 2.2.5, offering a more comprehensive and detailed explanation of the SSCA.

4.1.1 Strip Spectral Correlation Analyzer

The SSCA description and implementation are based on April's derivation as presented in reference [9]. As shown in Figure 4.1, the initial step involves computing the *complex demodulate*, X_T, at frequency f, from the discrete-time input values $x(n) \in \mathbb{C}$,

$$X_T(n,f) = \underbrace{[\sum_{r=-N_P/2}^{N_P/2-1} a(r)x(n+r)e^{-i2\pi f r T_s}]}_{N_P-\text{point FFT}} \underbrace{e^{-i2\pi f n T_s}}_{\text{down conversion}} \tag{4.1}$$

where n is a sample index, $a(r)$ is a length $T = N_P T_s$ data tapering window function, T_s is the sampling period and N_P is the number of samples [9].

Next, the complex demodulate X_T is multiplied by the conjugate input $x^*(n)$ [19] and windowed to produce the channel-data product (CDP) for $k \in [-N_P/2, N_P/2 - 1]$.

$$X_g(n+m,k) = X_T(n+m, f_k)x^*(n+m)g(m) \tag{4.2}$$

where the * operator is a complex conjugate, $g(m)$ is a length $\Delta t = NT_s$ windowing function, and $m \in [-N/2, N/2 - 1]$. The center frequencies of X_T are set to $f_k = k(f_s/N_P)$ for $f_s = 1/T_s$.

Finally, the N-point FFT of each of the N_P CDP values is computed resulting in the SCD estimate

$$S_X^{f_k+q\Delta\alpha}(\frac{f_k}{2} - q\frac{\Delta\alpha}{2})_{\Delta t} = \sum_{m=-N/2}^{N/2-1} X_g(n+m,k)e^{-i2\pi qm/N} \tag{4.3}$$

where cycle frequency $\alpha = f_k + q\Delta\alpha$, $\Delta\alpha = f_s/N$, $q \in [-N/2, N/2-1]$, and $f = (f_k - q\Delta\alpha)/2$ [9, 29]. In the implementation, both f and α are normalized based on $f_s = 1$, which maps the $S_X^\alpha(f)$ to a range $f \in [-0.5, 0.5]$ and $\alpha \in [-1, 1]$.

4.1.2 SSCA Implementation

Based on the Equation (4.1) and Equation (4.3), the SSCA algorithm can be implemented in the following steps.

Step 1 Data collection:
The size of input window samples is $N + N_P$, then converted to a block of $N \times N_P$ two-dimensional data.

$$x_b(n, r) = x(n + r),\ r = 0, 1, ..., N_P - 1,\ n = 0, 1, ..., N - 1 \tag{4.4}$$

Step 2 Compute N N_P-point FFTs of x_b:
The $a(r)$ is the N_P-point data-tapering window, same in FAM. The computation is:

$$x_T(n, k) = \texttt{FFTS}_{N_P}\{a(r)x(n, r)\} \tag{4.5}$$

where $n = 0, 1, ..., N - 1$, $r = 0, 1, ..., N_P - 1$, $k = -N_P/2, ..., N_P/2 - 1$, and $\texttt{FFTS}_{N_P}$ is the N_P-point centered FFT operation.

Step 3 Downconvert the FFT output to baseband:

$$X_T(n, k) = x_T(n, k)e^{-j2\pi nk/N_P},\ n = 0, 1, ..., N - 1,\ k = -N_P/2, ..., N_P/2 - 1 \tag{4.6}$$

Step 4 Compute the weighted product from Equation (2.21):
Define $g(n)$ for $n = 0, 1, ..., N - 1$ to be a N-point data-tapering window (e.g. same choice as FAM). The weight product X_g is computed as:

$$X_g(n, k) = X_T(n, k)x^*(n + N_P/2)g(n) \tag{4.7}$$

where $n = 0, 1, ..., N - 1$ and $k = -N_P/2, ..., N_P/2 - 1$.

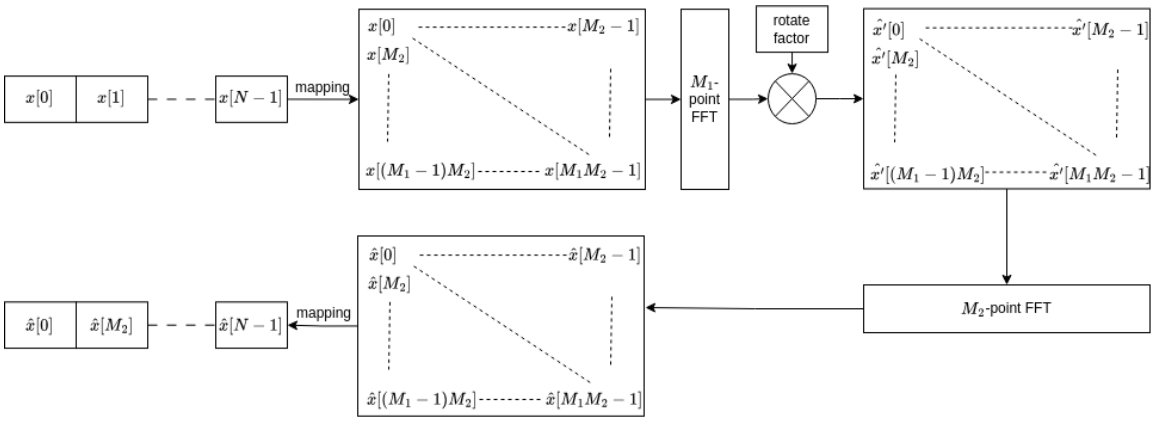


FIGURE 4.2. Computing N-point DFT by M_1 and M_2-point FFT ($N = M_1M_2$)

Step 5 Compute N_P N-point FFTs for spectral correlation function:

The $S_x(q,k)$ can be computed as:

$$S_x(q,k) = \texttt{FFTS}_N\{X_g(n,k)\} \tag{4.8}$$

where $q = -N/2, ..., N/2-1$, $k = -N_P/2, ..., N_P/2-1$, $n = 0, 1, ..., N-1$, and $\texttt{FFTS}_N$ is the N-point centered FFT operation.

Step 6 Map $S_x(q,k)$ to $S_x^\alpha(f)$:

Use the following formulas for mapping:

$$\begin{aligned} f &= \frac{k}{2N_P} - \frac{q}{2N} \\ \alpha &= \frac{k}{N_P} + \frac{q}{N} \end{aligned} \tag{4.9}$$

where f and α are normalized with respect to f_s = 1, which means $-0.5 \leq f \leq 0.5$ and $-1 \leq \alpha \leq 1$.

4.1.3 Common Factor Map Decomposition FFT

For very large N-point FFTs, the complete transform can exceed the on-chip memory and compute resources available. A practical solution is to partition the computation while remaining mathematically equivalent to the full transform. I. J. Good [34] showed that

a one-dimensional N-point DFT can be expressed as a multi-dimensional transform with a common factor mapping. For example, an N-point DFT can be computed as a two-dimensional DFT. In this case, the spectral computation involves a matrix DFT, where an $N = M_1 M_2$-point DFT is computed using M_1- and M_2-point DFTs with a complex multiplication stage.

The DFT of the N-point input signal $x(n)$ is given by:

$$\hat{x}(k) = \frac{1}{N} \sum_{n=0}^{N-1} x(n) e^{-j2\pi \frac{nk}{N}}, \quad 0 \leq k \leq N-1. \tag{4.10}$$

The input $x(n)$ can be decomposed into an M_1 times M_2 matrix as:

$$x(m_1, m_2) = x(M_2 m_1 + m_2), \tag{4.11}$$

for $0 \leq m_1 \leq M_1 - 1$ and $0 \leq m_2 \leq M_2 - 1$. The output transform sequence is recovered by the mapping:

$$\hat{x}(M_1 m_2' + m_1') = \hat{x}(m_1', m_2'), \tag{4.12}$$

for $0 \leq m_1' \leq M_1 - 1$ and $0 \leq m_2' \leq M_2 - 1$. The N-point DFT from Equation (4.10) can be computed using an $M_1 M_2$-point matrix DFT as:

$$\begin{aligned} \hat{x}(m_1', m_2') &= \sum_{m_2=0}^{M_2-1} \sum_{m_1=0}^{M_1-1} x(m_1, m_2) e^{-j2\pi \frac{(M_2 m_1 + m_2)(M_1 m_2' + m_1')}{M_1 M_2}} \\ &= \underbrace{\sum_{m_2=0}^{M_2-1} \underbrace{\{ \sum_{m_1=0}^{M_1-1} x(m_1, m_2) e^{-j2\pi \frac{m_1 m_1'}{M_1}} \}}_{M_1-\text{point DFT on } m_2\text{th column}} \underbrace{e^{-j2\pi \frac{m_2 m_1'}{M_2 M_1}}}_{\text{rotate factor}} e^{-j2\pi \frac{m_2 m_2'}{M_2}}}_{M_2-\text{point DFT on } m_1\text{th row}} \end{aligned} \tag{4.13}$$

Figure 4.2 illustrates the dataflow for computing an N-point DFT using M_1- and M_2-point FFTs. The input signal is first reshaped from a $N \times 1$ vector to an $M_1 \times M_2$ matrix. This reshaped matrix is then processed by applying an M_1-point FFT to each column, followed by a complex multiplication with the rotation factor defined in Equation (4.13). The resulting product is then passed through an M_2-point FFT for each row, and finally, the matrix output is mapped to an $N \times 1$ spectrum output.

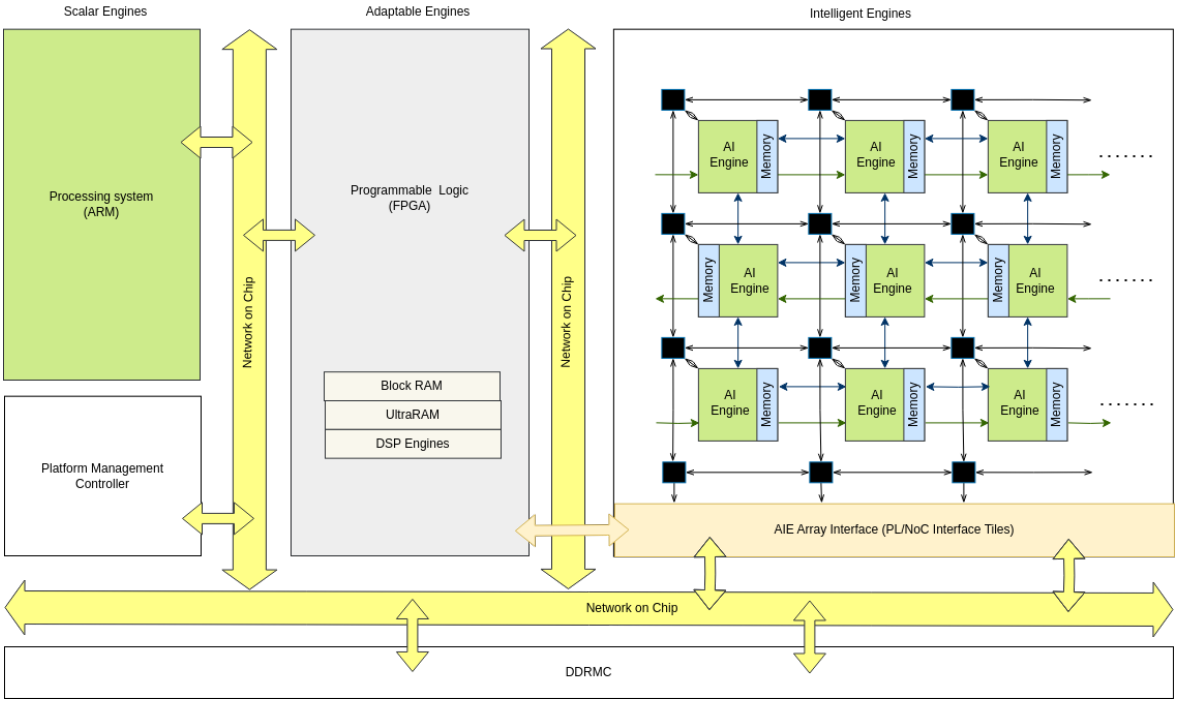


FIGURE 4.3. AMD/Xilinx Versal ACAP Architecture

4.1.4 Versal ACAP Architecture Overview

In this section, we provide a brief introduction to the system architecture of the AMD/Xilinx Versal ACAP, using the VCK5000 as an example. This is followed by an overview of the structure of a single AIE and the connections between different components: AIEs↔AIEs, PL↔AIEs, and DDRMC↔PL, respectively.

Versal ACAP Architecture of VCK5000:

The AMD/Xilinx VCK5000 Versal development card utilizes AMD's 7nm Versal™ adaptive SoC architecture, specifically designed for AIE development using Vitis and AI inference development with partner solutions [66]. This board is equipped with the Versal AI Core XCVC1902-2MSEVSVA2197 Adaptive SoC (VC1902), which features 400 AIE cores. Each core consists of VLIW and SIMD vector processors capable of operating at up to 1 GHz [3].

Figure 4.3 presents the overall architecture of the Versal Card, emphasizing the AIE array on the right [3]. The PL in the VCK5000 can be customized to meet specific application

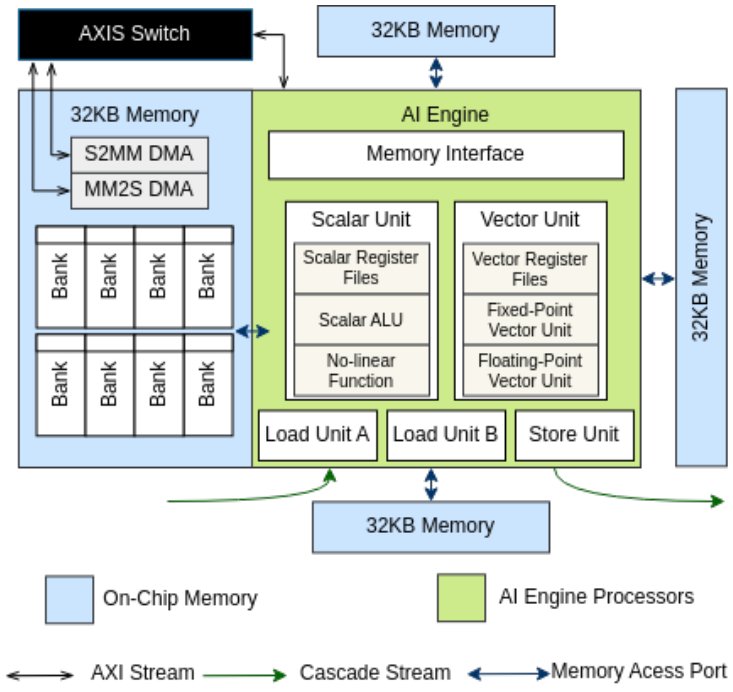


FIGURE 4.4. Versal ACAP structure of AIE

requirements, incorporating DSP capabilities for integration. Additionally, the board incorporates an ARM-based host for handling general-purpose processing tasks. The AIE cores are programmable using C/C++, while the host can be programmed in C/C++ or Python. The PL can be developed with both RTL and C/C++ code utilizing HLS.

All three components, AIE, host, and PL, work independently, interconnected through I/O peripherals like PCIe and DRAM controllers, into a heterogeneous SoC supported by a network-on-chip (NoC). The VCK5000 also includes four DDR4 off-chip memory modules, each providing a peak bandwidth of 25.6 GB/s.

The Architecture of a Single AIE:

Figure 4.4 shows the architecture of a single AIE, a key processing element in the AIE array of the Versal ACAP. This structure comprises the AIE core, on-chip memory, and interconnections with other AIEs to support collaborative tasks. Each AIE is a specialized processor optimized for high-performance parallel computations, which includes a memory interface, scalar and vector units, two load units, a store unit, and an instruction fetch and decode unit. The AIE cores employ VLIW instructions, enabling multiple operations to

execute in parallel within a single clock cycle. The SIMD architecture allows a single instruction to operate on multiple data elements simultaneously, supporting up to 8, 32, and 128 MAC operations per cycle for FP32, INT16, and INT8 data types, respectively. The two load units and one store unit access the data memory through the address generation unit (AGU) with a latency of one cycle.

Each AIE processor has 32KB of local data memory, consisting of eight memory banks, and can access up to 96KB of neighboring on-chip memory, including the other three ports simultaneously, without bank conflicts. The AIE can transfer data to adjacent AIEs using shared memory in a ping-pong buffer scheme. Additionally, the AIE supports a direct stream interface (cascade stream and AXI stream) for data transfer with other AIEs. Unlike the on-chip memory access, the cascade connection offers a 384 bit fine-grained streaming mechanism for transferring data between accumulator registers in neighboring AIEs via unidirectional connections that alternate in different rows. For non-adjacent AIEs, data transfer occurs through the AXIS stream network, managed by a switch box with a 32 bit width per wire, providing two input and two output connections for each AIE.

Data Transmission within the VCK5000:

The AXI4-Stream interfaces are the primary method for data transmission between the AIE and PL, as well as between the DDRMC and PL. In the VC1902 device, there are 50 columns of AIE array interface tiles, of which only 39 are available to the PL interface via the programmable logic input/ouputs (PLIOs). This includes six streams from AIE to PL and eight streams from PL to AIE. On the PL domain, each interface tile has eight 64 bit input channels and six 64 bit output channels running at the PL clock frequency. To match the speed of the AIE, the bit width is intended to be increased to 128 bit with the halved number of channels. On the AIE domain, there are eight 32 bit input channels and six 32 bit output channels per AIE array interface tile.

The PL can transfer data with the DDRMC through the AXI NoC IP, which supports both the AXI memory-mapped protocol and the AXI4-Stream protocol. The *sp* option in the connectivity section of the system configuration file can be used to specify the connection [4].

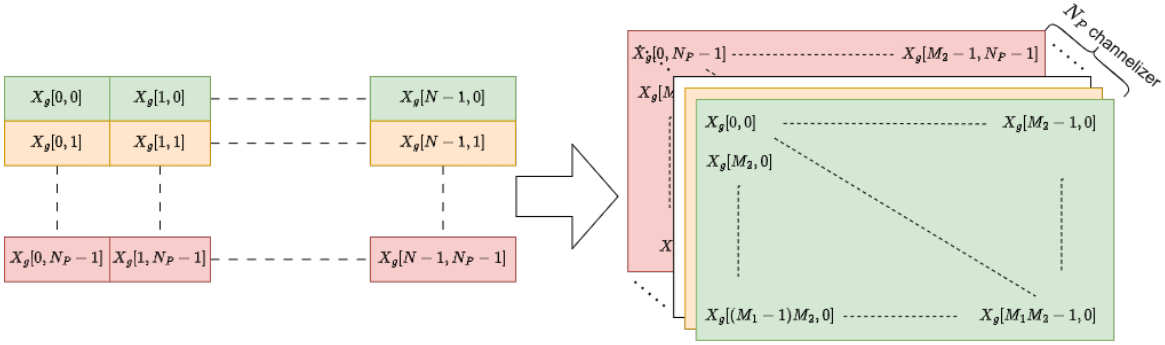


FIGURE 4.5. Reshape X_g from size $[N \times N_P]$ to $[M_1 \times M_2 \times N_P]$, where $N = M_1 M_2$.

4.2 Method

In this chapter, we design a large-scale SSCA that supports an input size of $N = 2^{20}$, which corresponds to 1 million points, with $N_P = 64$ channelizers. As described in Section 4.1.2, given an input size of N and N_P channelizers, the intermediate output CDP matrix is $[N \times N_P]$. This matrix is then processed using an N-point FFT for each channelizer to obtain the final SCD output.

However, the VCK5000 hardware cannot execute the complete 1-million-point FFT on the AIE in a single pass. Therefore, it is necessary to modify the SSCA structure to ensure compatibility with the VCK5000's constraints.

4.2.1 SSCA_2DFFT

The strip spectral correlation analyzer utilizing a decomposition FFT (SSCA_2DFFT) is a novel structure designed to compute the SSCA using a common factor map decomposition FFT (2DFFT) while adjusting the CDP computation order to match the required input sequence for the 2DFFT, as described in Equation (4.13). The implementation eliminates the need for large intermediate matrices and avoids external memory access. The naive approach replaces the N-point FFT in each channelizer with the decomposed FFT described in Section 4.1.3. Figure 4.5 illustrates how to reshape X_g for the decomposed FFT. The N_P channelizer computations are performed simultaneously in X_g and can be executed in parallel with the

decomposed FFT. Therefore, the N_P decomposed FFTs can be treated as a whole, computing X_g for the decomposed FFT as needed.

The SSCA_2DFFT is processed in two stages. **Stage 1** computes M_1-point FFTs for each of the M_1 rows of X_g, repeating this process for N_P channelizers across all M_2 columns. X_g is computed only once when needed during Stage 1. The results are then multiplied by a rotation factor and stored. **Stage 2** involves computing the M_2-point FFT for each row and channelizer using the values from Stage 1.

To achieve this, Equation (4.1) becomes:

$$\begin{aligned} X_T^{2D}(m_1, m_2, f) &= X_T(m_1 M_2 + m_2, f) \\ &= \underbrace{\left[\sum_{r=-N_P/2}^{N_P/2-1} a(r) x(m_1 M_2 + m_2 + r) e^{-j2\pi f r T_s} \right]}_{N_P\text{-point FFT}} \underbrace{e^{-j2\pi f (m_1 M_2 + m_2) T_s}}_{\text{down conversion}} \end{aligned} \tag{4.14}$$

where n is replaced with $n = m_1 M_2 + m_2$, for $0 \leq m_1 \leq M_1 - 1$, $0 \leq m_2 \leq M_2 - 1$, and $N = M_1 M_2$. If M_2 is divided by N_P, the down conversion can be simplified to $e^{-i2\pi f m_2 T_s}$ for all iterations of m_1.

After multiplying with the conjugate input $x^*(m_1 M_2 + m_2)$, the CDP can be expressed as:

$$\begin{aligned} X_g^{2D}(m_1, m_2, k) &= X_g(m_1 M_2 + m_2, k) \\ &= X_T^{2D}(m_1, m_2, f_k) x^*((m_1 M_2 + m_2) + m) g(m), \end{aligned} \tag{4.15}$$

for $k = -N_P/2, \ldots, N_P/2 - 1$ and $f_k = k(f_s/N_P)$.

The final SCD can then be estimated by computing the 2DFFT for the N_P CDP values as:

$$S_x^{2D}(m_1', m_2', k)_{\Delta t} = \textbf{Stage 2}\{\textbf{Stage 1}\{X_g^{2D}(m_1, m_2, k)\}\}, \tag{4.16}$$

in which the **Stage 1** is

$$S_{x.s1}^{2D}(m_1', m_2, k)_{\Delta t} = \underbrace{\sum_{m_1=0}^{M_1-1} X_g^{2D}(m_1, m_2, k) e^{-j2\pi \frac{m_1 m_1'}{M_1}}}_{M_1\text{-point FFT}} \underbrace{e^{-j2\pi \frac{m_2 m_1'}{M_2 M_1}}}_{\text{rotate factor}} \tag{4.17}$$

and the **Stage 2** is

$$S_x^{2D}(m_1', m_2', k)_{\Delta t} = \underbrace{\sum_{m_2=0}^{M_2-1} S_{x.s1}^{2D}(m_1', m_2, k)_{\Delta t} e^{-j2\pi \frac{m_2 m_2'}{M_2}}}_{M_2\text{-point FFT}}, \tag{4.18}$$

for $0 \leq m_1' \leq M_1 - 1$ and $0 \leq m_2' \leq M_2 - 1$.

Finally, $S_x^{2D}(m_1, m_2, k)$ is mapped to $S_x^{\alpha}(f)$ using the following relationships:

$$f = \frac{k}{2N_P} - \frac{M_1 m_2' + m_1' - N/2}{2N}$$

$$\alpha = \frac{k}{N_P} + \frac{M_1 m_2' + m_1' - N/2}{N}$$

where f and α are normalized with respect to $f_s = 1$, resulting in $-0.5 \leq f \leq 0.5$ and $-1 \leq \alpha \leq 1$.

4.2.2 SSCA_2DFFT Implementation

Based on Section 4.1.2 and Section 4.2.1, the SSCA_2DFFT algorithm can be implemented through the following steps:

Step 1 Data collection:

The input window samples, of size $N + N_P$, are converted into a 3D matrix with dimensions $M_1 \times M_2 \times N_P$, where $N = M_1 M_2$. M_2 is chosen such that it is divisible by N_P:

$$x_b(m_1, m_2, r) = x(m_1 * M_2 + m_2 + r), \tag{4.19}$$

for $r = 0, 1, ..., N_P - 1$, $m_1 = 0, 1, ..., M_1 - 1$ and $m_2 = 0, 1, ..., M_2 - 1$.

Step 2 Compute M_1 N_P-point FFTs for $x_b(:, m_2, :)$ for each m_2:

$$x_T(m_1, m_2, k) = \texttt{FFTS}_{N_P}\{a(r)x(m_1, m_2, r)\} \tag{4.20}$$

where $n = 0, 1, \ldots, N - 1$, $r = 0, 1, \ldots, N_P - 1$, $k = -N_P/2, \ldots, N_P/2 - 1$, and $\texttt{FFTS}_{N_P}$ is the N_P-point FFT operation.

Step 3 Downconvert the FFT output to baseband:

$$X_T(m_1, m_2, k) = x_T(m_1, m_2, k)e^{-j2\pi m_2 k/N_P}, \tag{4.21}$$

where $e^{-j2\pi(m_1 * M_2 + m_2)k/N_P}$ is simplified to $e^{-j2\pi m_2 k/N_P}$ since M_2 is divisible by N_P, and $k = -N_P/2, \ldots, N_P/2 - 1$.

Step 4 The weight product X_g is computed as:

$$X_g(m_1, m_2, k) = X_T(m_1, m_2, k)x^*(m_2 M_2 + m_2 + N_P/2)g(m_2 M_2 + m_2). \tag{4.22}$$

Step 5 Compute N_P M_1-point FFTs of $X_g(:, m_2, :)$ and multiply by rotate factors in Equation (4.16). For each m_2, computes:

$$X_{s1}(m_1, m_2, k) = \texttt{FFTS}_{M_1}\{X_g(m_1, m_2, k)\} \times \texttt{rotate\_factors} \tag{4.23}$$

where $\texttt{FFTS}_{M_1}$ is the M_1-point FFT operation.

Step 6 Repeat Step 1 to Step 5 for m_2 from 0 to $M_2 - 1$.

Step 7 Compute $M_1 N_P$ M_2-point FFTs to obtain the SCD:

$$S_x(m_1, m_2, k) = \texttt{FFTS}_{M_2}\{X_{s_1}(m_1, m_2, k)\} \tag{4.24}$$

for $\texttt{FFTS}_{M_2}$ is the M_2-point FFT operation.

Step 8 Map $S_x(m_1, m_2, k)$ to $S_x^\alpha(f)$ with a recenter for $N \times N_P$ matrix output.

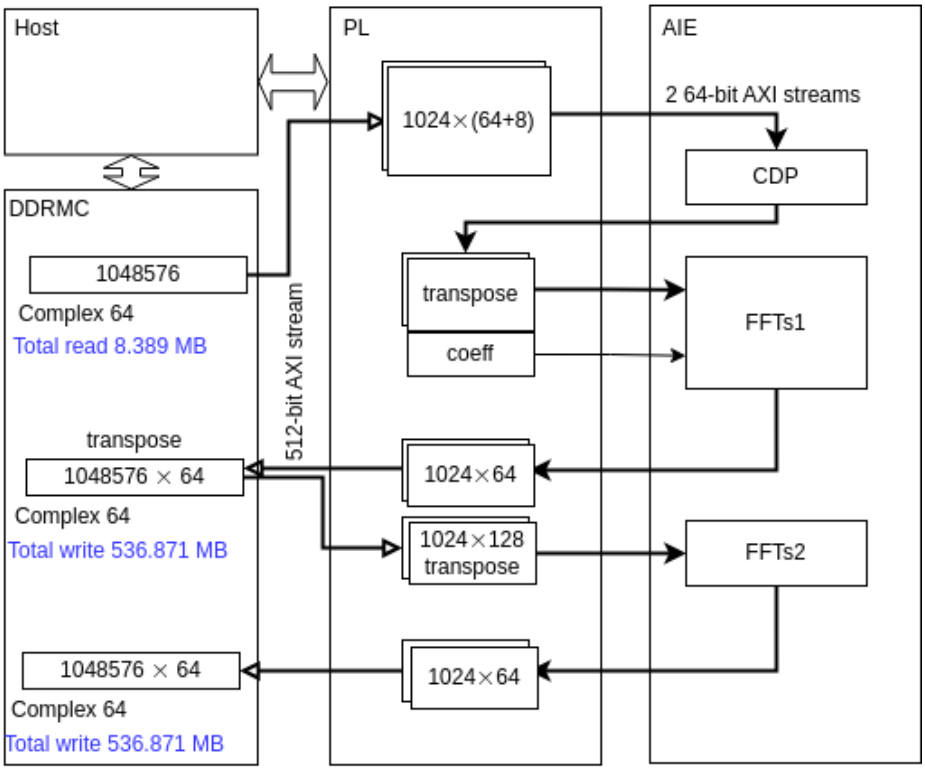


FIGURE 4.6. Dataflow of SSCA_2DFFT on Versal

4.2.3 Methodology for Estimating AIE Tile Requirements

The VCK5000 platform offers 23.9 MB of on-chip SRAM and four 4 GB off-chip DDR. Since an intermediate matrix in single-precision complex format requires $8 \times N \times N_P$ bytes, off-chip memory is needed if $N \times N_P > 2^{20}$.

Each AIE tile features eight single-port data memory banks (32KB total), enough for two-step 1K (1,024-element) single-precision complex computations via a ping-pong buffer. Our design uses N_P channelizers ranging from 2^5 to 2^8, and a second FFT window size N from 2^{12} to 2^{20}. Because each tile processes a 1K array, the AIE tiles required for computing X_g (Equation (4.2)) is

$$\mathbb{A}_{CDP} = 1 + \lceil \log_2(N_P)/2 \rceil, \tag{4.25}$$

where $\mathbb{A}$ denotes the number of AIE tiles. The first tile performs the downconversion and conjugate multiplication, while the remaining tiles implement the N_P-point FFT using a

pipelined radix-2 structure. The window function is merged into the first FFT stage. Up to $2^{10}/\log_2(N_P)$ windows of CDP can operate in parallel within those $\mathbb{A}_{CDP}$ AIE tiles.

For the large-point FFT—the N-point FFT is decomposed into M_1 and $M_2 = N/M_1$. Thus, the number of AIE tiles required is:

$$\mathbb{A}_{2DFFT} = \lceil \log_2(M_1)/2 \rceil + 1 + \lceil \log_2(M_2)/2 \rceil, \tag{4.26}$$

where $M_1,\ M_2 < 1K$. Additionally, one extra tile is allocated for multiplication with the rotation factor. The design computes $1K/M_1$ instances of M_1-point FFT and $1K/M_2$ instances of M_2-point FFT in parallel.

Thus, the total AIE tiles needed for SSCA_2DFFT is

$$\mathbb{A}_{SSCA\_2DFFT} = \mathbb{A}_{CDP} + \mathbb{A}_{2DFFT}. \tag{4.27}$$

Replicating $\mathbb{A}_{SSCA\_2DFFT}$ modules enables further parallelization.

4.3 Implementation of the SSCA on Versal

We designed the implementation for $N = 2^{20}$, $N_P = 64$, and $M_1 = M_2 = 1024$. Figure 4.6 illustrates the implementation of a large-size SSCA on the Versal platform (VCK5000). The Versal PL supports a custom program to manage the input dataflow from the DDRMC to the AIE, as well as to facilitate data transfer between AIE kernels.

The PL initially passes a group of data to the CDP kernel (`CDP`), then transposes the output and sends it to `FFTs1` includes the M_1-point FFT and complex multiplies with the rotation factors. The 2DFFT process is designed to first complete all M_1-point FFTs for M_2 iterations, followed by the M_2-point FFTs (`FFTs2`) for M_1 iterations. Unfortunately, the intermediate matrix used to store the output of the M_1-point FFT is too large to fit within the available PL memory, requiring storage in the DDR memory.

Algorithm 5: `AIE` section pseudocode

Function `CDP` ($datain\_even, datain\_odd$)**:** ▷ Channel Data Product
- $data\_win$, $data\_fft$, $data\_dc$, $data\_out = zeros(M_1, 1)$; ▷ $M_1 = 16N_P$
- $xc = zeros(16, 1)$;
- static int $itr = 0$;
- $data\_win, xc \leftarrow$ Window$(chebwin[N_P], datain_even, datain_odd)$;
- $data\_fft \leftarrow \mathbf{N_P}$-dimensional FFT$(data_win)$; ▷ Step 2
- $data\_dc \leftarrow$ Down_Conversion$(data_fft,$ int$(itr/N_P))$; ▷ Step 3
- $data\_out = data\_dc \times xc$; ▷ Step 4
- $itr = (itr == (M_2 * N_P))?\, 0 :\ (itr + 1)$;
- **return** $data\_out$;

Function `FFTs1` ($datain\_even, datain\_odd, tow$)**:** ▷ 2DFFT stage 1
- $data\_fft$, $data\_out$, $rotate\_factor = zeros(M_1, 1)$;
- **if** $itr\%N_P == 0$ **then**
  - $rotate\_factor \leftarrow Compute\_rotate\_factor(tow)$;
- $data\_fft \leftarrow \mathbf{M_1}$-dimensional FFT$(data_even,\ data_odd)$; ▷ Step 5
- $data\_out = data\_fft \times rotate\_factor$;
- $itr = (itr == (M_2 * N_P))?\, 0 :\ (itr + 1)$;
- **return** $data\_out$;

Function `FFTs2` ($datain\_even, datain\_odd, tow$)**:** ▷ 2DFFT stage 2
- $data\_out = zeros(M_2, 1)$;
- $data\_out \leftarrow \mathbf{M_2}$-dimensional FFT$(data_even,\ data_odd)$; ▷ Step 7
- **return** $data\_out$;

To address this, the PL writes the output of the `FFTs1` operation sequentially to the DDRMC and later reads it for the `FFTs2` operation, applying a stride to achieve the necessary transpose. The detailed design and implementation are described in this section.

4.3.1 Architecture Design on AIE

In this implementation, with $M_1 = M_2 = 1024$, to match the data size between kernels, the `CDP` is designed for 1024 input data, which is 16 sets of $N_P = 64$ samples, and the window index follows the input index requirements of `FFTs1`, which is controlled in PL. The algorithm for the portion executed in the AIE is described in Algorithm 5, which follows the description provided in Section 4.2.2.

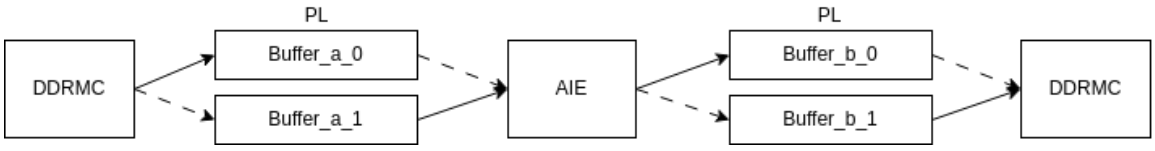


FIGURE 4.7. Ping-pong buffer

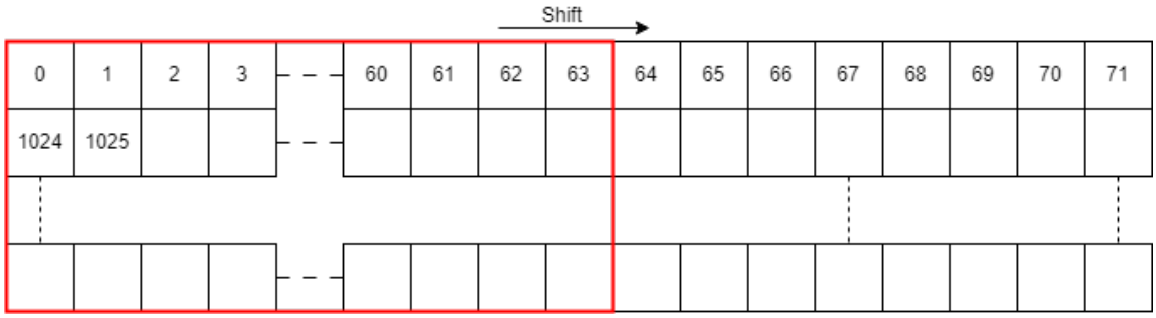


FIGURE 4.8. Input buffer for CDP (size of $M_1 \times (N_P + 8)$)

4.3.2 Design Strategies on PL

The PL section manages data transfer between DDRMC and AIE. As illustrated in Figure 4.7, a ping-pong buffer approach maximizes communication efficiency. While the PL loads data from DDRMC to Buffer_a_1, the AIE processes the data already stored in Buffer_a_0, reducing idle time and improving overall efficiency.

Input Buffer for `CDP`:

The connection between DDRMC and PL uses a 512 bit data bus. Since the input data for AIE consists of single complex values of 64 bits, the PL loads $LANE = 8$ consecutive complex values ($512 = 8 \times 64$ bits) into the buffer each cycle. The PL then manages the data transfer efficiently to ensure continuous and timely delivery to the AIE.

The Figure 4.8 and Algorithm 6 show how PL efficiently loads data sets and passes them to AIE. The buffer size is $M_1 \times (N_P + LANE)$, thus it supports $LANE \times N_P$ iterations to send data to `CDP`.

Transpose Buffer for AIE and FFTs1:

To match the input requirements of `FFTs1`, a transpose operation is required for the output of `CDP`, where the matrix size is $[M_1 \times N_P]$. Due to the large matrix size, this transposition

Algorithm 6: Input for `CDP`

Function `LoadA` $(memin, idx)$**:** ▷ `Load to buffer`
- $buffA = zeros(M_1, N_P + LANE)$;
- **for** $i \leftarrow 0$ **to** $M_1 - 1$ **do**
  - **for** $j \leftarrow 0$ **to** $N_P/LANE$ **do** ▷ $LANE = 512/64$
    - $buffA[i, LANEj : LANE(j+1) - 1] = memin(j + M_1 i/LANE + idx)$;
- **return** $buffA$;

Function `SendA` $(buffA, data\_odd, data\_even, idx)$**:** ▷ `Send to AIE`
- **for** $i \leftarrow 0$ **to** $LANE - 1$ **do**
  - **for** $j \leftarrow 0$ **to** $N_P - 1$ **do**
    - $data\_even.write(buffA[\frac{M_1}{N_P}j : 2 : \frac{M_1}{N_P}(j+1) - 1, i : i + N_P - 1])$;
    - $data\_odd.write(buffA[\frac{M_1}{N_P}j + 1 : 2 : \frac{M_1}{N_P}(j+1) - 1, i : i + N_P - 1])$;

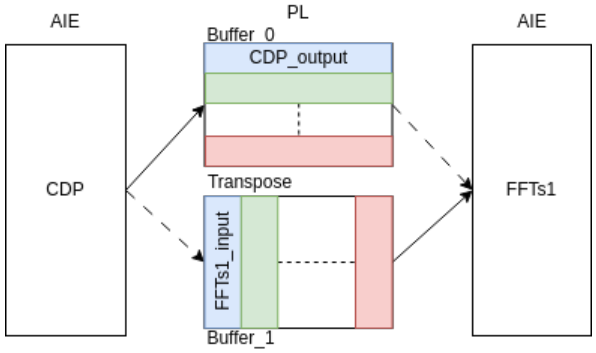


FIGURE 4.9. Transposing the matrix for ping-pong buffer

is performed in the PL. A ping-pong buffer is utilized to balance data loading from `CDP` and transferring it to `FFTs1`, as depicted in Figure 4.9.

Optimizing Transpose Loading from DDR to PL:

The input buffer for `FFTs2` needs to be loaded in transpose order, as the data saved from `FFTs1` is stored sequentially. However, accessing data with a stride reduces the access speed of DDRMC. To mitigate this, the buffer size in PL is increased to accelerate the loading process. For the matrix size as $[M_2 \times N_P]$, continuous access for N_P data is efficient, but accessing data with a stride of every $M_1 N_P$/LANE values takes more cycles to access in DDRMC. Thus, instead of loading $[M_2 \times N_P]$ data to the PL, we load $[M_2 \times CN_P]$ for every C iteration, effectively reducing the access time by a factor of C.

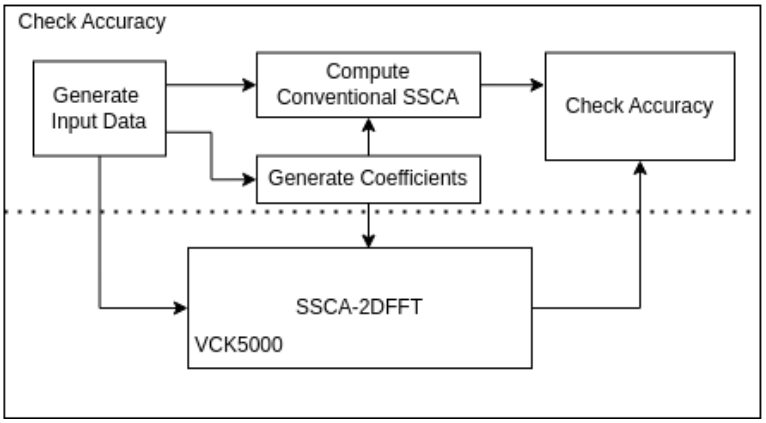


FIGURE 4.10. Check the accuracy

4.4 Result

In this section, we implement the dataflow illustrated in Figure 4.6 on the VCK5000 board, with the AIE running at 1 GHz and PL running at 312.5 MHz. We then compare its performance with the conventional SSCA implementation running on an Intel(R) Xeon(R) Silver 4208 CPU at 2.10 GHz and NVIDIA GeForce RTX 2060 at 1.47 GHz, on Ubuntu 20.04.6 LTS. Additionally, we verify the accuracy of the results against the output of the conventional SSCA. In the conventional SSCA, N is 1,048,576 and N_P is 64. In the SSCA_2DFFT, N_P is the same as the conventional SSCA, which is 64, but the second FFT is replaced from N-points to $N = M_1 \times M_2$, where $M_1 = M_2 = 1,024$.

4.4.1 Accuracy

Accuracy was tested using a DSSS BPSK signal with 10 dB SNR, processing gain of 31, chip rate 0.25 and sample rate normalized to 1, resulting in cycle frequencies that are multiples of the data rate (0.25/31). Figure 4.10 illustrates the progress of accuracy checking. To compare the output of SSCA_2DFFT with conventional SSCA, the system converts the complex floating-point data into two separate INT32 numbers for the real and imaginary components, which are then passed to the DDRMC. The system also computes the coefficients for SSCA_2DFFT, including the initialization of FFT twiddle factors, window coefficients, 2DFFT rotation factors, and exponent coefficients in downconversion using the matched

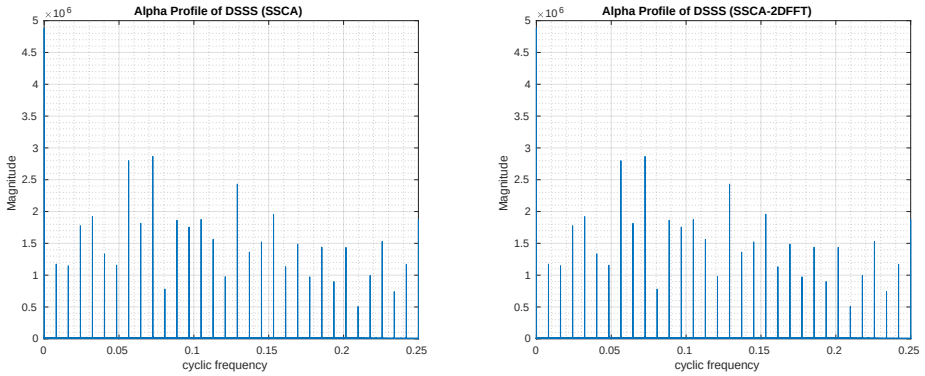


FIGURE 4.11. Alpha profile for conventional SSCA and SSCA_2DFFT

TABLE 4.1. Utilization in VCK5000

	PL					AIE Array	
	Register	LUT	LUTasMEM	BRAM	URAM	AIE tile	PLIO
Total resources	1,739,432	860,336	446,367	933	463	400	-
SSCA_2DFFT_PL	15,475 (0.89%)	11,824 (1.37%)	1,575 (0.35%)	349 (37.41%)	192 (41.47%)	15 (3.75%)	13

coefficients as the conventional SSCA computation. The average relative error between the output of the conventional SSCA and SSCA_2DFFT is 1.08e-6. Figure 4.11 shows the alpha profile of SSCA and SSCA_2DFFT computed based on Equation (2.13) with only half of the profile plotted due to symmetry.

4.4.2 Utilization

The `CDP` module includes four AIE cores that compute M_1/N_P sets of N_P data in each iteration, utilizing a ping-pong buffering scheme to pass data between cores efficiently. In the `FFTs1` and `FFTs2` modules, five AIE cores are used to compute the M_1-point FFT, with an additional core in `FFTs1` for the computation of rotation factors. Moreover, the stream connections utilize 20 out of 450 available links, and 13 PLIO connections are used for interfacing inputs and outputs with the PL.

Table 4.1 shows the utilization of resources in the PL section, indicating a significant consumption of BRAM and URAM, mainly for intermediate ping-pong matrix between the DDRMC and AIE components.

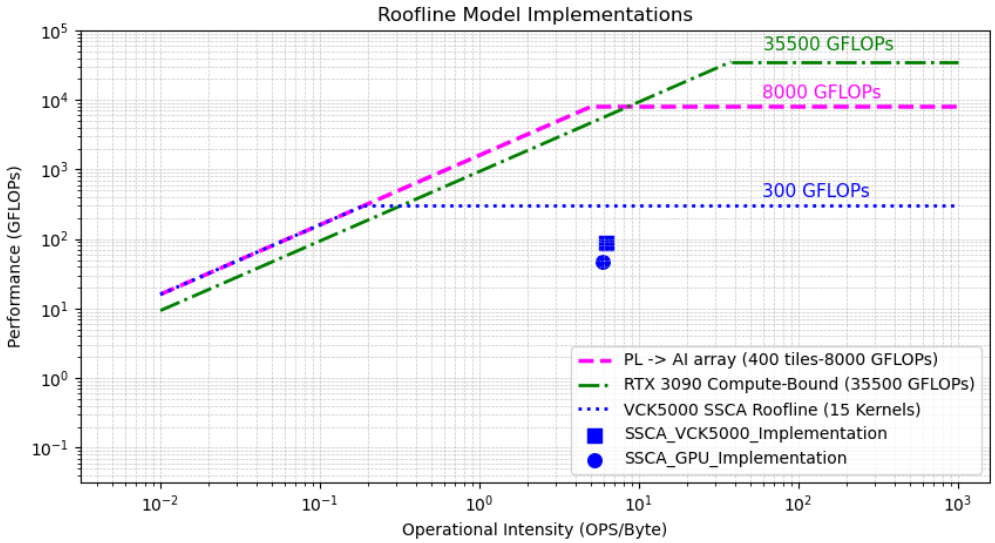


FIGURE 4.12. Rooflines of SSCA implementations

TABLE 4.2. Execution time (SSCA with $N = 2^{20}$ and $N_P = 64$)

	Execution Time	Speedup vs CPU
CPU	11.3 s	1
GPU	269 ms	42
VCK5000	114 ms	99.12

In the SSCA implementation, the `CDP` module requires 4 AIE tiles that compute M_1/N_P sets of N_P data in each iteration, and this utilises a ping-pong buffering scheme to exchange data between tiles. In the `FFTs1` and `FFTs2` modules, 5 AIE tiles are used to compute the M_1-point FFT, with an additional tile in `FFTs1` for the computation of rotation factors. This totals 15 AIE tiles (Table 4.1), matching Equation (4.27).

4.4.3 Performance

The code running in CPU was compiled using g++ version 9.4.0 with the "-O2" optimization flag. The GPU code was compiled using nvcc, with the host compiler set to g++. The compilation targeted CUDA architecture "sm_70", using C++17 standard with the "-O3" optimization flag. Additionally, the code linked against the cuFFT library to support efficient FFT operations [22]. Table 4.2 presents the execution times for the CPU, GPU, and VCK5000

platforms. The VCK5000 achieved a speedup of 99.12x compared to the CPU and 2.36x compared to the GPU.

As shown in the roofline plot of Figure 4.12, the SSCA implementation reaches 88.30 GFLOPs, achieving 37% of its 15-tile peak (240 GFLOPs). The performance of SSCA is constrained by the bandwidth between PL and DDRMC. In our system, communication with 15 AIE tiles saturates this available bandwidth. With increased off-chip bandwidth, additional AIE tiles could be utilised to further enhance performance. On the RTX 3090, the SSCA implementations is memory-bound, achieving 46.39 GFLOPs, also far below the GPU's ceiling of 35 TFLOPs.

We measure power consumption on the VCK5000 and GPU using the "xbutil" and "nvidia-smi" command-line tools, respectively. For SSCA, the VCK5000 consumes 8 W, on top of an idle power of 23 W. The GPU requires 103 W with an idle of 33 W. Consequently, compared to GPU, the VCK5000 achieves a 24.5x higher for SSCA.

4.5 Summary

This chapter presents the implementation of SSCA_2DFFT on the Versal ACAP, an efficient approach that enhances the traditional SSCA architecture. By replacing the second large-point FFT with a vectorized FFT and restructuring the computation of CDP to accommodate the vectorized FFT's input, the implementation significantly optimizes performance. This approach removes the need for a large intermediate matrix and avoids accessing the DDR between the `CDP` and `FFTs1`, instead storing the intermediate data directly in the PL. The result is a 99.12x speedup compared to a CPU and a 2.36x improvement over a GPU implementation of conventional SSCA.

CHAPTER 5

S^{3}CA: A Sparse Strip Spectral Correlation Analyzer

The SCD is widely used to characterize cyclostationary signals, and the SSCA is commonly used to estimate the SCD. Although the SSCA utilizes the FFT for computational efficiency, its real-time implementation still poses challenges as large input sizes are often involved. This chapter presents a S^{3}CA based on the SFFT. The S^{3}CA approach involves computing a sparse, downsampled CDP, which is then passed to a modified SFFT implementation to obtain the spectral density.

The SSCA method was considered to be limited to smaller-size signals due to its large memory requirements [59, 18]. For fine resolution and sufficient integration for the cycle frequency α, SSCA also requires a large input signal window. Additionally, SSCA focuses on computing the cycle frequency, which is only a small portion of the total output, leading to inefficiencies when only the cycle frequency is of interest. There is considerable interest in developing a novel method capable of scanning the entire frequency/cycle-frequency plane while estimating the cycle frequency with reduced memory requirements and computational complexity.

Figure 5.1 is a signal flow diagram for the SSCA. In both the channelizer and FFT blocks, the primary computational complexity involves executing FFTs: N_P-point FFTs for the former and N-point for the latter. In practice, the value of N is commonly set within the range of 2^{16} to 2^{24}. N_P represents the number of channelizer bands and is typically chosen from 2^5 to 2^9. The cyclic spectrum is sparse in cycle frequency f for all known practical digital signal types [31]. It is continuous in spectral frequency for each cycle frequency exhibited by the signal. When the cycle frequencies are unknown in advance of processing, the entire frequency/cycle-frequency plane must be computed and searched over to find the significant peaks.

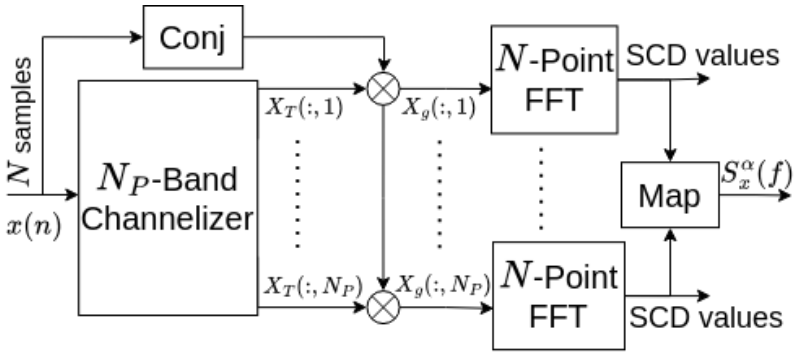


FIGURE 5.1. The strip spectral correlation analyzer signal flow

The SFFT is a recent algorithm designed for efficiently computing a FFT where the frequency domain is approximately κ-sparse, meaning κ coefficients are non-zero [38, 37]. This chapter presents the S^{3}CA, which enables fast and accurate estimation of the SCD. The implementation is available as an open-source[1]. It is particularly useful for real-time applications involving large signal sizes, as computation and memory requirements are reduced.

The main contributions of this chapter are:

- An algorithm, based on the SFFT, that reduces the computational complexity of the SSCA from $O(NN_P(\log N_P + \log N))$ to $O(N_P \log N_P \log N \sqrt[3]{N\kappa^2 \log N})$.
- An additional optimization in which only a subset of channelizer outputs are computed and stored. This reduces space complexity of an intermediate matrix from $O(N \times N_P)$ to $O(\log N \sqrt[3]{N\kappa^2 \log N} \times N_P)$.
- A comparison of execution time and sparsity between the S^{3}CA and an SSCA using FFTW version 3.3.10 [27].

The remainder of this chapter is organized as follows. The Section 5.1 provides the background on the sparse fast Fourier transform. Section 5.2 describes the sparse strip spectral correlation analyzer. Section 5.3 presents the experimental results, and a summary is drawn in Section 5.4.

[1] https://github.com/Jingyi-li/S3CA.git

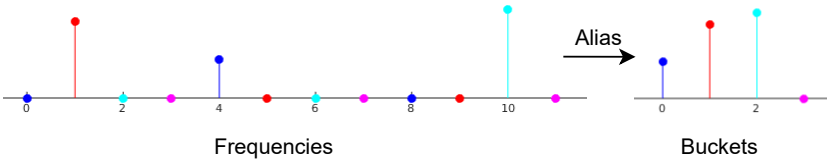


FIGURE 5.2. Bucketization with aliasing filter ($\sigma = 3$)

5.1 Background

In this section, we present the fundamental theory of SFFT [36]. Although this technique could be applied to any of the versions in reference [36], the following description in this chapter refers to SFFT 2.0.

5.1.1 Sparse Fast Fourier Transform

For an input $u \in \mathbb{C}^N$, the notation $\hat{u} \in \mathbb{C}^N$ is used for its FFT. The SFFT $\hat{u}'$ is an approximation to $\hat{u}$ and assumed κ-sparse. Reference [36] proposes a number of different SFFT algorithms [42].

Almost all sparse Fourier transform algorithms are composed of *Frequency Bucketization*, *Frequency Estimation*, and *Collision Resolution* steps.

Frequency Bucketization:
Frequency Bucketization hashes the Fourier coefficients of $\hat{u}$ into a limited number of buckets and sums the values within each bucket. Since $\hat{u}$ is assumed to be sparse, only a few buckets are expected to contain non-zero values, which are of primary interest.

Normally, a filter is chosen to suppress the Fourier coefficients, which hashes the Fourier coefficients into buckets using a small number of input time samples. As illustrated in Figure 5.3, the aliasing filter exhibits a spike-train structure with a period of σ, and its frequency domain characteristics involve the summation of Fourier coefficients that are equally spaced by N/σ. For example, if b is the subsampled version of u (e.g $b(i) = u(i * \sigma)$)

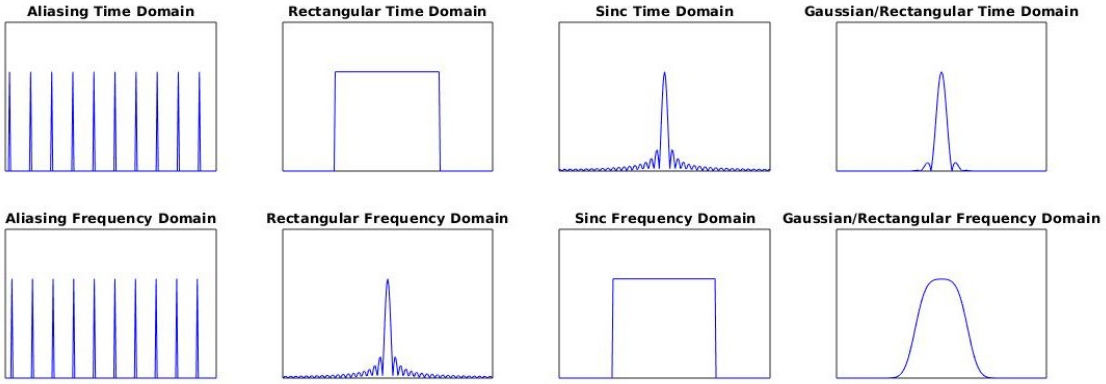


FIGURE 5.3. Filters used for frequency bucketization

where σ is a subsampling factor that divides N, an aliased version of $\hat{u}$ is computed as:

$$\hat{b}(i) = \sum_{m=0}^{\sigma-1} \hat{u}(i + m(N/\sigma)) \tag{5.1}$$

In Figure 5.2, after the aliasing filter, the frequencies are equally spaced by an interval $\mathbf{B} = N/\sigma$ and hash to the same buckets, which can be efficiently computed with a $\mathbf{B}$-point FFT taking $O(\mathbf{B}log(\mathbf{B}))$ time. However, the aliasing filter is not well-suited for advanced randomization techniques, and consequently, its effectiveness can only be demonstrated for average-case input signals rather than for worst-case scenarios. [36]

The ideal filter for *Frequency Bucketization* should employ only a minimal number of input time samples to hash the frequency coefficients into their corresponding buckets effectively. As illustrated in Figure 5.3, both the rectangular and sinc filters are considered ideal within their respective domains: the rectangular filter in the time domain and the sinc filter in the frequency domain. However, in the frequency domain, the rectangular filter exhibits polynomial decay, causing "leakage" of frequency coefficients between buckets, similar to the leakage observed with the sinc filter in the time domain. Thus, an optimal filter for *Frequency Bucketization* would have a profile resembling a rectangular shape in the frequency domain while still using a minimal number of time samples.

Figure 5.3 also presents a flat window filter, which is constructed by multiplying a Gaussian function with a sinc function in the time domain. The leakage between buckets for this

filter is negligible due to the rapid exponential decay of the Gaussian function in both time and frequency domains. Consequently, the resulting hash function for such a filter can be expressed as $h(f) = \lceil f/(N/\mathbf{B}) \rceil$.

Frequncy Estimation:

The concept of *frequency estimation* involves determining the position f and corresponding values $\hat{u}(f)$ of the non-zero frequency coefficients. Since $\hat{u}$ is sparse, most buckets likely contain one non-zero frequency coefficient at most, with only a few containing multiple non-zero coefficients. The *frequency estimation* focuses on identifying buckets with a single non-zero frequency coefficient and then estimating both its value $\hat{u}(f)$ and its associated position f. In cases where multiple non-zero coefficients are present in a single bucket, a *collision resolution* process is employed to detect and resolve such collisions, which is described below.

The non-zero frequency coefficient $\hat{b}(i) = \hat{u}(f)$ can be directly obtained from the bucket's value if no collision occurs. However, determining the exact frequency position f is complicated by aliasing, which causes multiple frequencies to map to the same bucket. To address this, the phase-rotation property of the Fourier transform is used to compute f. As noted in [50], a time shift τ in the input signal results in a corresponding phase rotation in the frequency domain, changing the value in the bucket from $\hat{b}(i) = \hat{u}(f)$ to $\hat{b}^{(\tau)}(i) = \hat{u}(f) \cdot e^{2\pi j f \tau}$. It is crucial to keep τ as small as possible to prevent phase wrapping, which occurs when large values of f cause the phase to wrap around every 2π.

Collision Resolution:

In the process of *collision resolution*, there are two primary steps: first, detecting whether a collision has occurred in a bucket, and second, resolving it. This principle also applies to the phase-rotation property when dealing with multiple values in the same bucket, such as two frequency collisions in the same bucket, the value is $\hat{b}(i) = \hat{u}(f_1) + \hat{u}(f_2)$. When a time shift τ is introduced to the input signal, the bucket's value transforms into $\hat{b}^{(\tau)}(i) = \hat{u}(f_1) \cdot e^{j2\pi f_1 \tau} + \hat{u}(f_2) \cdot e^{j2\pi f_2 \tau}$. As a result, the magnitude of the bucket changes due to the colliding frequencies rotating by different phases. By examining the change in the magnitude of the frequency, we can determine whether a collision has occurred.

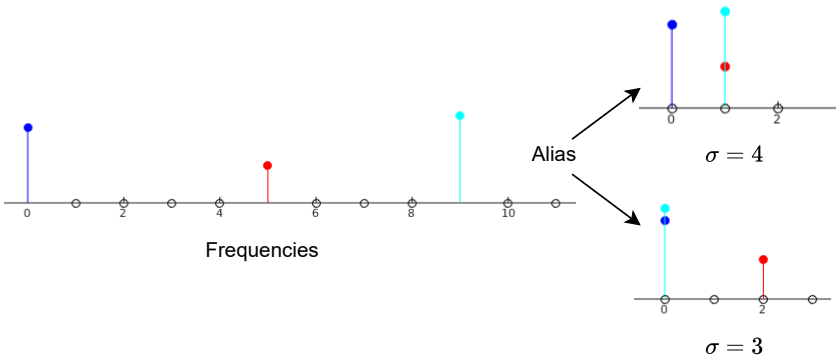


FIGURE 5.4. Collision resolution with co-prime filters

One effective method for resolving collisions is to bucketize the spectrum multiple times using aliasing filters with co-prime sampling rates. According to the Chinese Remainder Theorem, these co-prime aliasing filters ensure that any two frequencies that collide in one bucketization will not collide again in subsequent bucketizations. This approach leverages the uniqueness of remainders when frequencies are mapped into buckets, allowing for accurate resolution of collisions across multiple bucketizations.

For instance, as illustrated in Figure 5.4, consider a scenario with three non-zero frequency coefficients in the frequency domain. With a subsampling factor of four, the blue frequency is aliased into a separate bucket, while the clay and red frequencies are aliased into the same bucket, causing a collision. By changing the subsampling factor to three, which is co-prime to four, the blue and clay frequencies are aliased into the same bucket. Since the blue frequency was identified in the previous stage, its value can be subtracted from the bucket, isolating the clay frequency. Through this process, the spectrum can be successfully recovered despite initial collisions.

5.1.2 SFFT 2.0

The SFFT 2.0 algorithm applies two randomized inner loops to obtain a high probability of achieving an error bound: 1) *Frequency bucketization* involves using a random hash function to hash the κ non-zero Fourier coefficients of $\hat{u}$ into a small number of buckets, and 2)

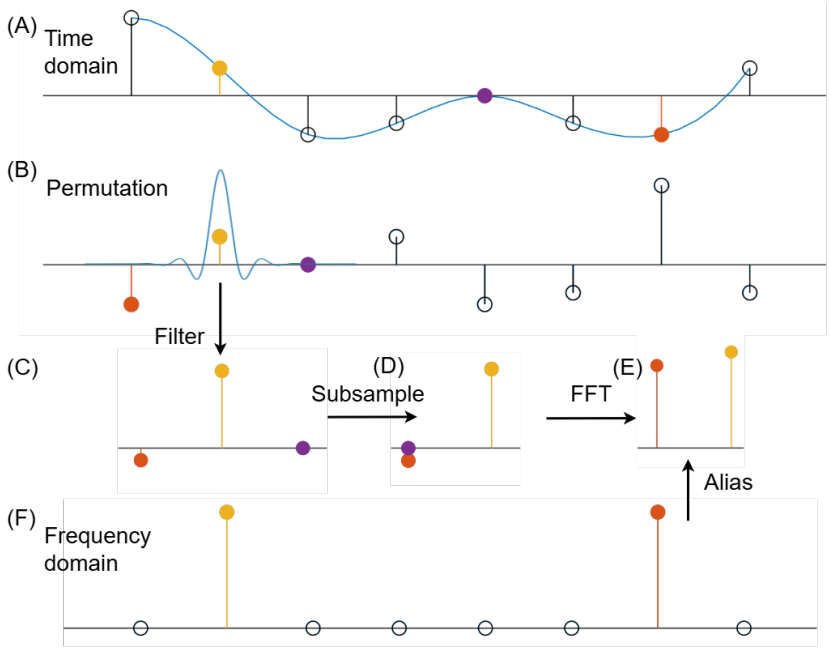


FIGURE 5.5. An example of the SFFT with $N = 8$, $\sigma = 3$, $\tau = 6$, $w = 3$, $\mathbf{B} = 2$, and $\kappa = 2$. (A) shows the input signal u, $N = 8$; (B) is the permuted u ($\mathbf{P}_{3,6}u$); (C) after filtering with G to restrict the time domain length of $\mathbf{P}_{3,6}u$ to 3; (D) subsampled $\mathbf{P}_{3,6}u$ to 2 buckets to get v; (E) $\hat{v}$, the FFT of v; (F) The 2-sparse approximation of $\hat{u}$ that is bucketized into subfigure (E).

Frequency estimation finds the frequency locations of non-zero Fourier coefficients and their corresponding magnitudes. Information obtained from the two inner loops is combined in an outer loop to form the final output.

Figure 5.5(A) to (E) illustrates the steps involved in frequency bucketization (FB). Let $\mathbf{B}$ be the number of buckets and is an integer that divides N; σ an integer invertible mod N; and κ the number of non-zero Fourier coefficients desired in the output. Figure 5.5(B) is the permuted frequency spectrum, achieved via the time domain permutation operator $\mathbf{P}_{\sigma,\tau}$, $\tau \in [0, N-1]$. If $(\mathbf{P}_{\sigma,\tau}u)(i) = u((\sigma i + \tau) \mod N)$, then $\widehat{(\mathbf{P}_{\sigma,\tau}u)}(\sigma i) = \hat{u}(i)e^{-j2\pi\tau}$ [38].

Figure 5.5(C) represents the output of a w-dimensional filter function G, which is restricted to a subset of the input in both the time and frequency domain. In this work, a Dolph-Chebyshev function [63] is used, which has little leakage between buckets, and this restricts the time-domain region of interest to $w = O(\mathbf{B} \log \frac{N}{\delta})$ coordinates (δ is the maximum ripple

Algorithm 7: `FB` pseudocode

Function `FB` $(u, \sigma, \tau, w, \mathbf{B}, G, N)$ **:** ▷ `Frequency Bucketization`

  for $i \leftarrow 0$ **to** $w - 1$ **do**

    $v[i \mod \mathbf{B}] \mathrel{+}= u[(i \cdot \sigma + \tau) \mod N] \cdot G[i]$;

  $\hat{v} \leftarrow \mathbf{B}$-dimensional FFT$(v)$;

  return $\hat{v}$;

in the passband or stopband), and performs bandpass filtering in the frequency domain [38].[2] Figure 5.5(D) to (E) shows that the subsampled FFT $\hat{v} = \hat{u}(iN/\mathbf{B})$ of an N-dimensional vector u can be computed via the $\mathbf{B}$-point FFT of $v = \sum_{j=0}^{N/\mathbf{B}-1} u(i + \mathbf{B}j)$ for $i \in [0, \mathbf{B} - 1]$ [38].

Algorithm 7 provides the pseudo-code of *Frequency Bucketization*, which computes the sum of the frequency coefficients and returns the corresponding values depicted in Figure 5.5(A) to (E). This process constitutes the core of the *location loops* and *estimation loops*. *Location loops* involve multiple iterations of `FB` with randomly chosen values of σ and τ (where σ is odd and $\tau \in [0, N-1]$). For a given value of d in the location loops, only $d\kappa N/\mathbf{B}$ coordinates from the list of candidate coordinates I are retained, which are expected to have a high probability of being "good". These "good" probability coordinates are those that are likely to appear in more than one iteration of the location loops. To identify candidate coordinates with a high likelihood of being among the κ non-zero coordinates, the top $d\kappa N/\mathbf{B}$ candidate coordinates are selected from the set I based on their frequency of appearance across multiple iterations, as these coordinates have a significant probability of being correct. *Estimation loops* use multiple iterations of `FB` to exactly determine the coefficients $\hat{u}_I$ for the given set of coordinates I by reversing the filter applications in `FB`.

Additional `FB` loops are applied using the *Mansour filter* described in [51] to further restrict the locations of the large coefficients in SFFT 2.0. In this version, SFFT 2.0 employs `FB` with a parameter M, where M divides N, and outputs a subset $T \subset [0 : M - 1]$ containing the 2κ largest frequency coefficients. It is then assumed that all "large" coefficients j satisfy the

[2] The support of the G filter, i.e. the coordinates of the non-zero coefficients, is limited to the interval $[-(w-1)/2, (w-1)/2]$, and computations outside of this interval are removed.

Algorithm 8: Modified SFFT 2.0 pseudocode

Function SFFT_2.0 $(u, \kappa, \mathbf{B}, \mathbf{L}, M, G, d, N, \Sigma, \Upsilon)$ **:**

- **for** $r \leftarrow 0$ **to** $\mathbf{L}-1$ **do** ▷ Mansour filter
  - $\hat{v} \leftarrow \mathtt{FB}\,(u, \frac{N}{M}, \Upsilon_{(2,r)}, M, M, \mathit{ones}(M,1), N)$;
  - $T_r \leftarrow$ indices of 2κ largest elements of $\hat{v}$;
  - ▷ $T_r \subset [0, \mathbf{B}-1]$
- $T \leftarrow T_0 \cup \cdots \cup T_{\mathbf{L}-1}$;
- **for** $r \leftarrow 0$ **to** $\mathbf{L}-1$ **do** ▷ location loop
  - $\hat{v} \leftarrow \mathtt{FB}\,(u, \Sigma_{(0,r)}, \Upsilon_{(0,r)}, w, \mathbf{B}, G, N)$;
  - $J \leftarrow$ indices of $d\kappa$ largest elements of $\hat{v}$;
  - $I_r \leftarrow \{i \in [0, N-1] \mid h_\sigma(i) \in J, i \mod M \in T\}$;
  - ▷ $h_\sigma(i) = \mathtt{round}(\Sigma_{(0,r)} i\mathbf{B}/N)$
- $I \leftarrow I_0 \cup \cdots \cup I_{\mathbf{L}-1}$;
- $I' \leftarrow i$ values that occur frequently in sets I;
- **for** $r \leftarrow 0$ **to** $\mathbf{L}-1$ **do** ▷ estimation loop
  - $\hat{v} \leftarrow \mathtt{FB}\,(u, \Sigma_{(2,r)}, \Upsilon_{(2,r)}, w, \mathbf{B}, G, N)$;
  - $\hat{u}^r_{I'} \leftarrow$ estimate frequency spectrum from $\hat{v}, I'$;
- $\hat{u}'_i = \text{median}(\{\hat{u}^r_i \mid i \in I'\})$;
- **return** $\hat{u}'$;

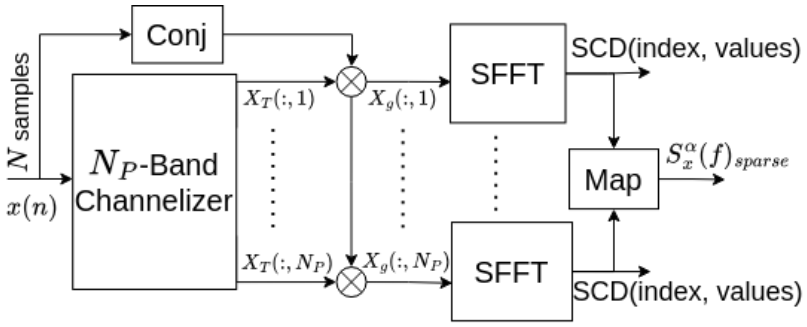


FIGURE 5.6. The naive sparse strip spectral correlation analyzer implementation replace the N-point FFT with the SFFT.

condition that $j \mod M$ in T, which is highly efficient and ensures that there is no leakage at all.

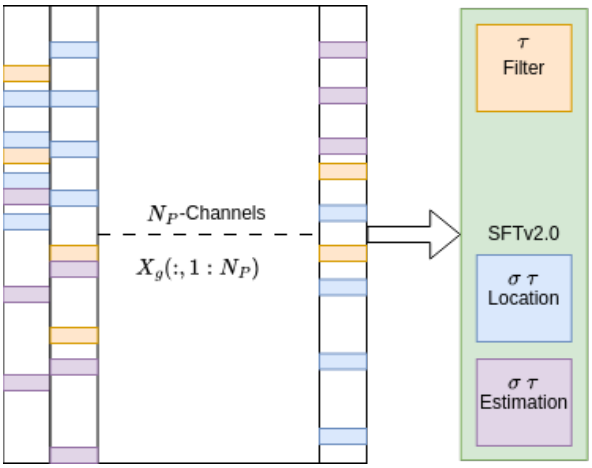


FIGURE 5.7. Vector X_g representing the input for an N_P-channel SFFT

5.2 S^{3}CA Algorithm

This section presents the S^{3}CA technique. A naive S^{3}CA implementation can be implemented by simply replacing the N_P N-point FFTs with SFFTs in Equation (4.3), with the input to the k$^{\text{th}}$ FFT being the $X_g(:, k)$ vector. This is shown in Figure 5.6 and Figure 5.7, which involves computing the entire matrix X_g, but most values are not used.

However, as described in the previous section, the `FB` step within each SFFT only requires w inputs, based on $\mathbf{P}_{\sigma,\tau}$, where σ an integer invertible mod N, and $\tau \in [0, N-1]$, are both drawn from a random distribution. In Algorithm 7, the indices of the w inputs form a set $W = \{i * \sigma + \tau \mod N \mid i \in [0, \ldots, w-1]\}$. In SFFT 2.0, the *Frequency bucketization* and *Frequency estimation* steps both involve iterations of `FB` processing. The union of all sets, $W'_i = W_0 \cup W_1 \cup \ldots$, represents the indices required for the SFFT to compute κ non-zero frequency coefficients for channelizer i via iterations of `FB`. The sparse FFT processing of each channelizer operates independently; however, if σ and τ are randomly selected in `FB` for N_P channelizers, the indices X_g must prepare are $W' = W'_0 \cup W'_1 \cup \cdots \cup W'_{N_P}$. In Figure 5.8, if all the channelizers choose the random σ and τ for iterations of `FB` across different channelizers, the set of indices that X_g needs to prepare is reduced to $W'_0 = W'_1 = \cdots = W'_{N_P-1}$, leading to a shared and compact set of indices $W' = W'_0$.

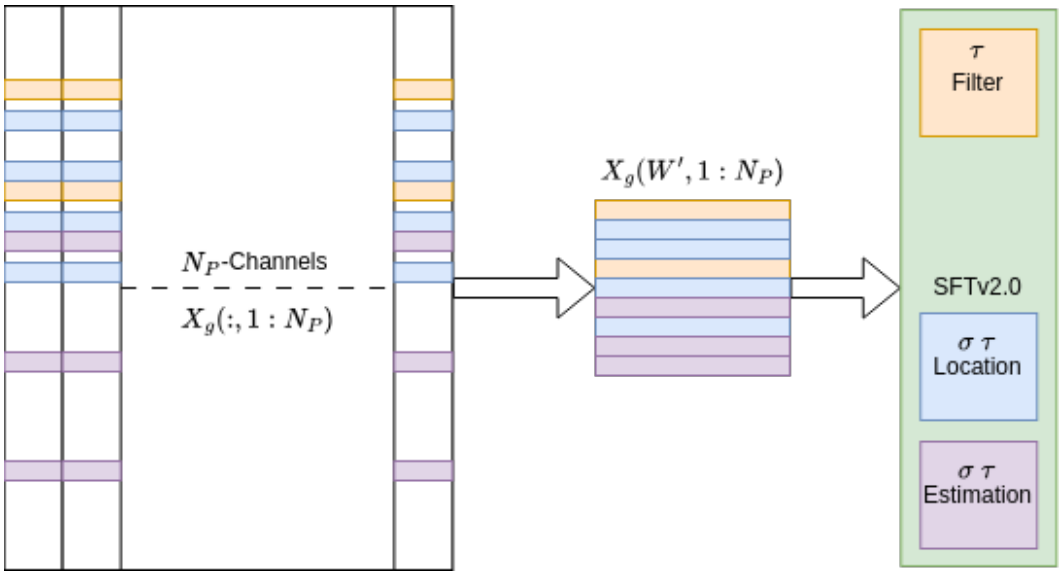


FIGURE 5.8. Vector $X_g(W', 1 : N_P)$ serving as the input for the N_P-channels SFFT with identical parameters applied across all channels

This strategy significantly reduces the input data preparation required by X_g, since values can be computed and shared across all channelizers simultaneously. To bound the computational and memory requirments, the maximum size of W_i' is bounded by $|W_i'| \leq wL$, where L is the number of FB iterations per channelizer. The total number of unique input indices across all channelizers satisfies $|W'| \leq \min(wLN_P, N)$.

To achieve this, we describe a procedure COMPIDX, which precomputes a subset of indices W' for the channelizer, and corresponding arrays Σ of σ and Υ of τ for the SFFT. The channelizer now only computes the outputs $X_T(W', k)$ instead of $X_T(n, k)$, then the CDP, $X_g' = X_g(W', k)$, using Equation (4.2). All CDP outputs are then used by the subsequent N_p N-point SFFTs. The output of S^{3}CA is a sparse matrix and only returns non-zero values and corresponding location information. An equivalent approach is using lazy evaluation to avoid computing unnecessary inputs to the SFFT.

Algorithm 8 shows how we modified FB to use the precomputed σ and τ, with the SFFT updated to make use of this function. Algorithm 9 gives the pseudocode for COMPIDX and S^{3}CA. x is the input signal with length of N, and N_P is the number of channelizers. COMPIDX, which is the dashed block in Figure 5.9, randomly selects σ and τ required by

Algorithm 9: S^{3}CA pseudocode

Procedure COMPIDX$(\mathbf{L}, w, \mathbf{B}, N)$**:** ▷ Compute Indices for X'_g

$\Upsilon \leftarrow zeros(3, \mathbf{L})$, $\Sigma \leftarrow zeros(2, \mathbf{L})$;

for $r = 0$ **to** $\mathbf{L} - 1$ **do**

$\Upsilon_{(2,r)} \leftarrow$ uniform$(0, \mathbf{B} - 1)$;

$\Upsilon_{(0,r)}, \Upsilon_{(1,r)} \leftarrow$ uniform$(0, N - 1)$;

$\Sigma_{(0,r)}, \Sigma_{(1,r)} \leftarrow 2*$ uniform$(0, N/2 - 1) + 1$;

$W_0^r \leftarrow \{i * \Sigma_{(0,r)} + \Upsilon_{(0,r)} \bmod N \mid i \in [0, w - 1]\}$;

$W_1^r \leftarrow \{i * \Sigma_{(1,r)} + \Upsilon_{(1,r)} \bmod N \mid i \in [0, w - 1]\}$;

$W_2^r \leftarrow \{i * N/\mathbf{B} + \Upsilon_{(2,r)} \mid i \in [0, \mathbf{B} - 1]\}$;

$W' \leftarrow \{W_j^r \mid j \in \{0, 1, 2\}, r \in [0, \mathbf{L} - 1]\}$;

return W', Σ, Υ;

Procedure S^{3}CA$(x, N, N_P, \mathbf{L}, w, \mathbf{B}, N, G)$**:**

$W', \Sigma, \Upsilon \leftarrow$ COMPIDX$(\mathbf{L}, w, \mathbf{B}, N)$;

$X'_g \leftarrow X_g(W', k)$; ▷ Equation (4.2), $k \in [-\frac{N_P}{2}, \frac{N_P}{2} - 1]$

for $k = -\frac{N_P}{2}$ **to** $\frac{N_P}{2} - 1$ **do**

$\hat{u}'_k \leftarrow$ SFFT_2.0$(X'_g, \kappa, \mathbf{B}, \mathbf{L}, G, d, N, \Sigma, \Upsilon)$;

$value, \alpha, f \leftarrow$ map$(\hat{u}')$;

return $value, \alpha, f$;

our modified FB to compute W for each new input window. It then returns the set W' of all required indices, the array Σ of σ and the array Υ of τ.[3] In Figure 5.9, each channelizer performs an independent N-point FFT. Consequently, in our implementation, in each of the different FB calls, the same σ and τ values are used for all k and the w inputs are $X_g(W, k)$. This necessitates a modified SFFT that can accommodate the shared σ and τ.

Table 5.1 compares the computational complexity of SSCA and S^{3}CA. Referring to Figure 5.1 the SSCA channelizer requires a total of N evaluations of Equation (4.2); and the FFT block N_P evaluations of Equation (4.3) (using the FFT). In contrast for the S^{3}CA channelizer, the required number of N_p-point FFT evaluations is equal to the sampling complexity of the SFFT, $N_{SFFT} = O(\log N \sqrt[3]{N\kappa^2 \log N})$ [36].

[3] In Algorithm 8 and 9, we present the loop value $\mathbf{L}$ and buckets value $\mathbf{B}$ for simplicity; performance can be improved with different values of $\mathbf{L}$ and $\mathbf{B}$ for the three for loops in SFFT.

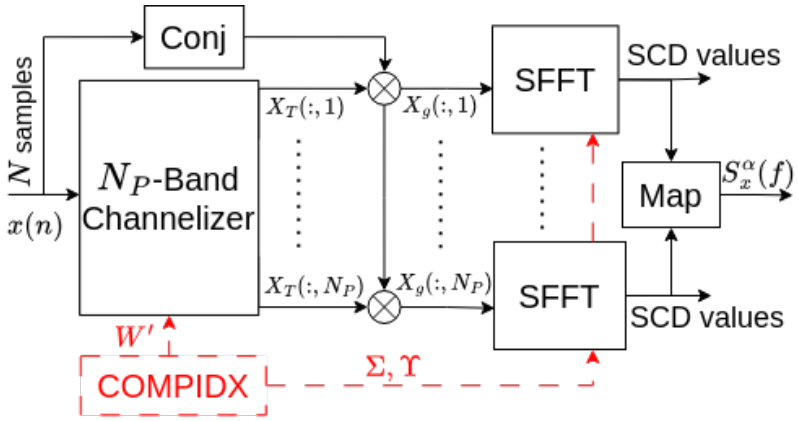


FIGURE 5.9. The S^{3}CA technique accelerates the SSCA via: (1) the COMP-IDX block that evaluates a subset of the inputs and (2) replacing the N-point FFT with the SFFT.

TABLE 5.1. Comparison of computational complexity between SSCA and S^{3}CA

	SSCA	S^{3}CA
Channelizer	$O(NN_P \log N_P)$	$O(N_{SFFT}N_P \log N_P)$
$N_p\times$ FFT	$O(N_P N \log N)$	$O(N_P N_{SFFT})$
$S_X^\alpha(f)$	$O(NN_P(\log N_P + \log N))$	$O(N_{SFFT}N_P \log N_P)$

5.3 Results

We implemented the SSCA and S^{3}CA using the C programming language and the FFTW library [28]. Experiments were conducted using Ubuntu 20.04.6 LTS on an Intel(R) Xeon(R) Silver 4208 CPU running at 2.10 GHz with 256 GB of memory. All the code was compiled using g++ version 9.4.0 with the "-O2" optimization flag, as recommended in Reference [42].

5.3.1 Accuracy

To evaluate the accuracy of our designed S^{3}CA compared to the conventional SSCA, we choose a simple digital modulation scheme, BPSK, to verify whether S^{3}CA can detect peaks in the cycle frequency similar to SSCA. Additionally, We use DSSS BPSK to check S^{3}CA's ability to identify multiple peaks. The results are presented below.

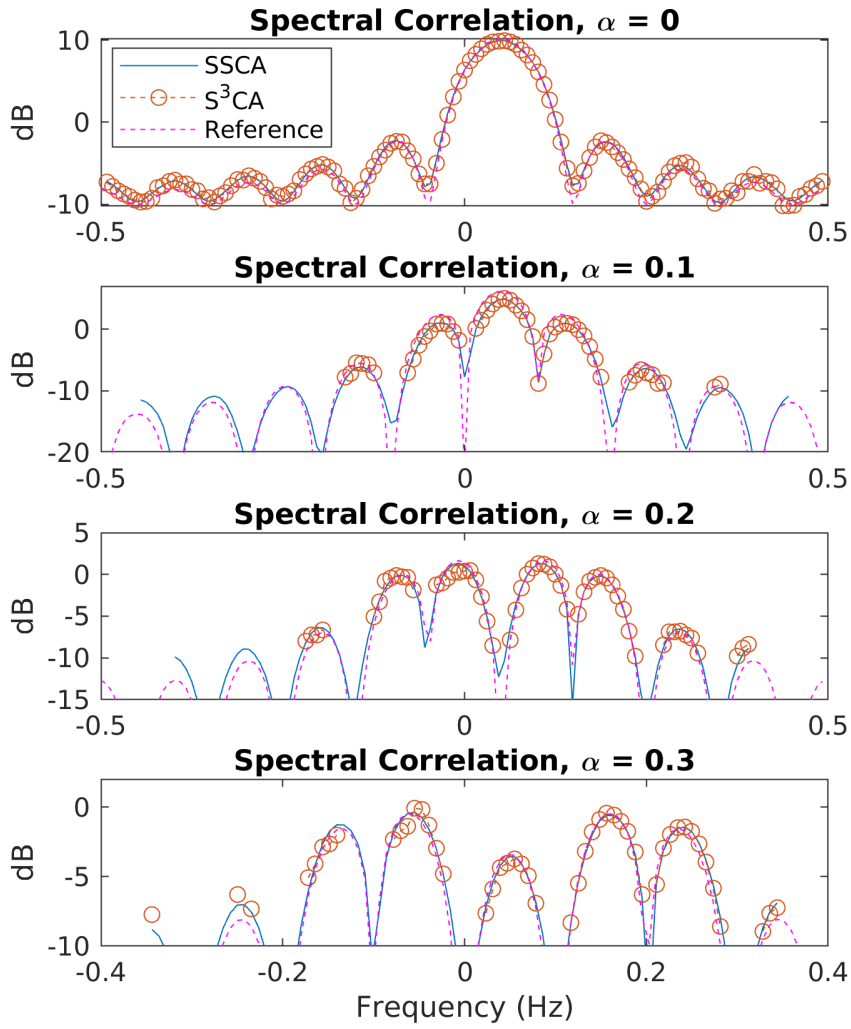


FIGURE 5.10. Convertional SSCA, reference and S^{3}CA at four different cycle frequencies. All three are very similar.

Verification with BPSK Signal:

Figure 5.10 is validated through the use of a noise-free BPSK signal with a symbol rate of 0.1 and a carrier offset of 0.05. We compare the cycle frequency at 0, 0.1, 0.2, and 0.3 with reference to theoretical expectations. In Figure 5.10, the result of conventional strip spectral correlation analyzer is denoted as SSCA, and S^{3}CA represents the sparse strip spectral correlation analyzer. The reference is the theoretical expectations at each cycle frequency.[4] In

[4]Detailed on the cyclostationary.blog (https://cyclostationary.blog/2016/02/24/second-order-estimator-verification-guide/).

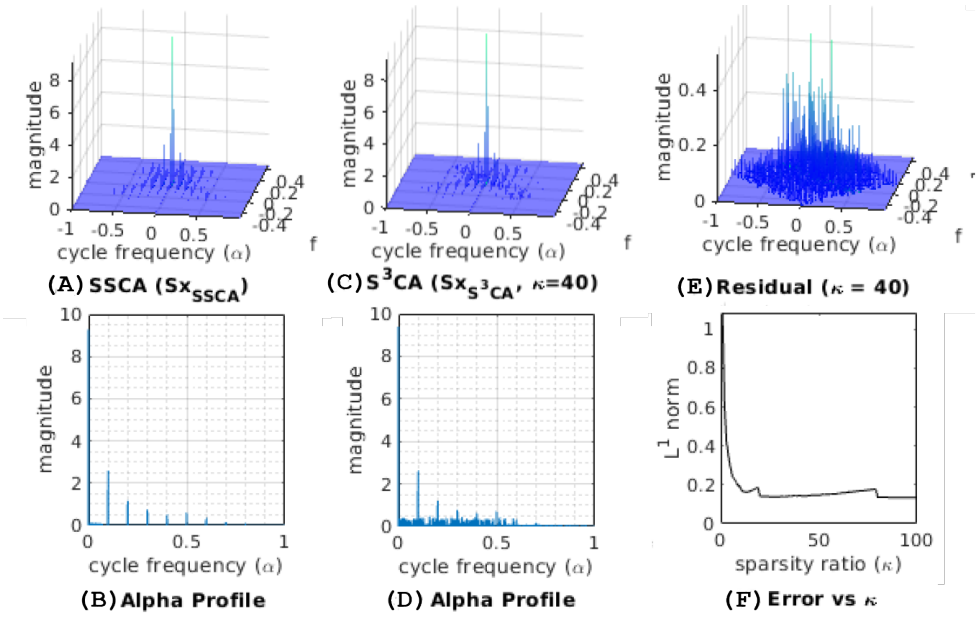


FIGURE 5.11. SCD estimates and alpha profiles using SSCA (A) and (B), and S^{3}CA (C) and (D), their residual (E), and $\mathbf{L}^1$-norm of the residue for different κ (F).

the implementation, we set N to 2^{16} and N_P to 2^6 for both SSCA and S^{3}CA, and κ to 40 for S^{3}CA. The rest of the parameters for S^{3}CA are selected in the same way as in the paper.

For more details, we present the outputs in dB, and the output of S^{3}CA successfully captures the peaks of the SSCA at each cycle frequency.

SCD Estimation of BPSK Signal:

We compare the SCD estimation of a BPSK signal with a 10 dB SNR and the same symbol rate and carrier offset as the previous example. We increase N to 2^{20} for both SSCA and S^{3}CA, and keep N_P and κ unchanged. Figure 5.11(A) shows a 3-D plot of the largest κN_P magnitude SSCA outputs, $S_{X_{SSCA}}$, with its alpha profile corresponding to the largest alpha value over all frequencies below. Figure 5.11(B) shows the S^{3}CA output, $S_{X_{S^3CA}}$, with sparsity parameter $\kappa = 40$, and its alpha profile in Figure 5.11(D). In Figure 5.11(C), the residual with the average $\mathbf{L}^1$-norm of the residue below in Figure 5.11(F).

The symbol rate of 0.1 is easily discernible from the alpha profile presented in Figure 5.11.

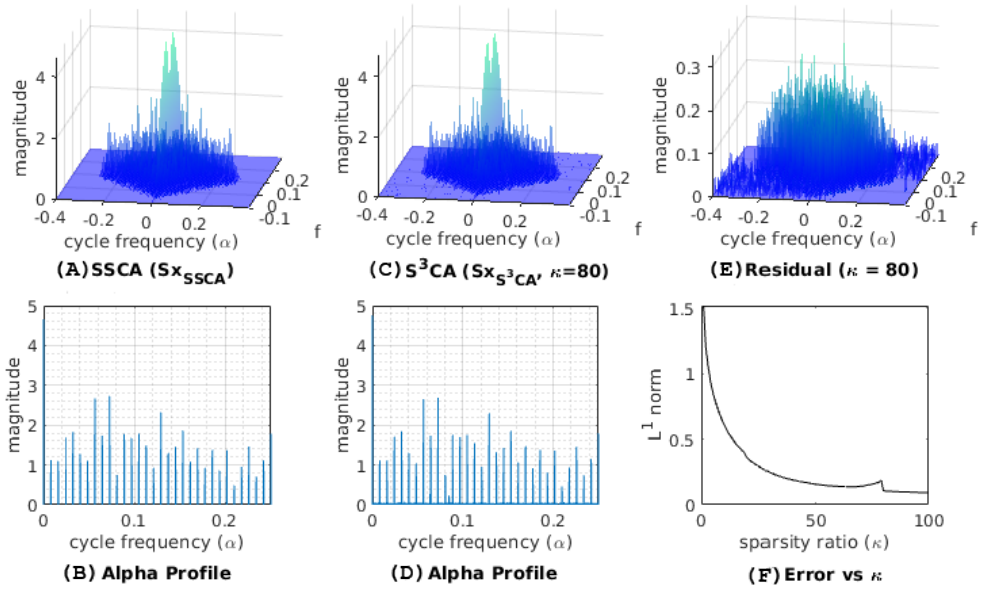


FIGURE 5.12. SCD estimates and alpha profiles using SSCA (A) and (B), and S^{3}CA (C) and (D), their residual (E), and $\mathbf{L}^1$-norm of the residue for different κ (F).

SCD Estimation of DSSS Signal:

We also test accuracy using a DSSS BPSK signal with 10 dB SNR, processing gain of 31, chip rate 0.25, and sample rate normalized to 1, in which case the cycle frequencies are multiples of the data rate (0.25/31). The Figure 5.12 shows the SCD estimates with $N = 2^{20}$ and $N_P = 2^6$. We configure the remaining parameters of S^{3}CA in accordance with the default parameters outlined in the SFFT library [38]. Figure 5.12(A) shows a 3-D plot of the largest κN_P magnitude SSCA outputs, $S_{X_{SSCA}}$, with its alpha profile corresponding to the largest alpha value over all frequencies below. Due to symmetry, the non-redundant interval of normalized cycle frequency, α, is in $[0, 1]$. To highlight the important details, the display area of the alpha profile is restricted to $[0, 0.25]$. Figure 5.12(B) shows the S^{3}CA output, $S_{X_{S^3CA}}$, with sparsity parameter $\kappa = 80$, and its alpha profile in Figure 5.12(D). In Figure 5.12(C), the residual, $r = S_{X_{SSCA}} - S_{X_{S^3CA}}$, is shown together with the average $\mathbf{L}^1$-norm of the residue, $\sum_i |r_i|/(\kappa N_P)$, below in Figure 5.12(F). Again, good correspondence between the SSCA and S^{3}CA is observed.

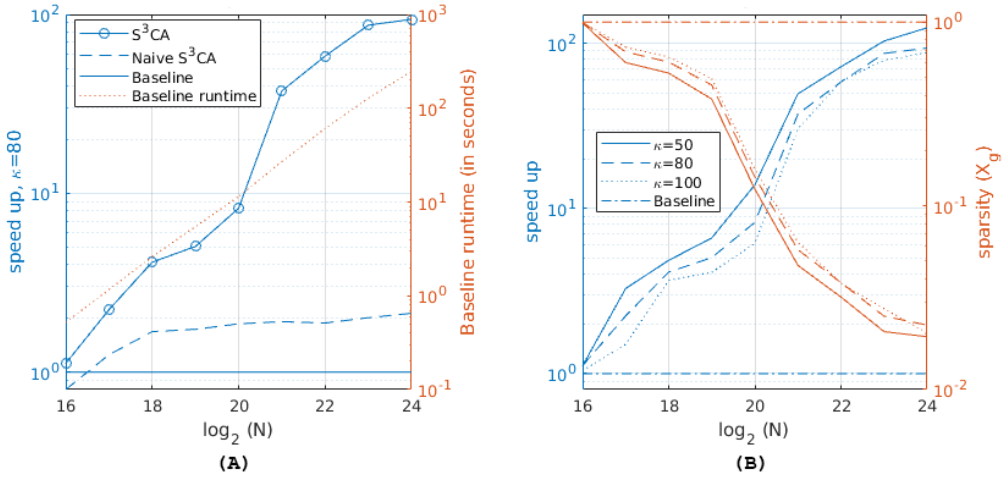


FIGURE 5.13. (A) Speedup of the naive S^{3}CA and S^{3}CA compared with the conventional SSCA. (B) Speedup and X'_g sparsity of S^{3}CA for different values of κ.

5.3.2 Speedup and Storage Optimization

The baseline in Figure 5.13 is the conventional SSCA. Figure 5.13(A) compares the speedup achieved by replacing the FFT with SFFT in SSCA, labeled as naive S^{3}CA, and the speedup obtained by S^{3}CA, for input window sizes from 2^{16} to 2^{24}. For an input size of 2^{24}, the S^{3}CA achieves a speedup that surpasses a factor of 90 when κ is 80 and more than 100 when κ is 50. The naive S^{3}CA achieves a more modest speedup of 2. The baseline runtime on our test computer is also provided. The sparsity of X'_g is $\mathbf{S} = |W'|/N$, where $|.|$ denotes the number of indices in W', hence the storage savings over the full X_g is approximately $1 - \mathbf{S}$. Figure 5.13(B) shows the speedup and the sparsity ratio of S^{3}CA for different κ. The output of the SSCA has $N \times N_P$ values, whereas the output of S^{3}CA only has $\kappa \times N_P$ values.

5.4 Summary

In this chapter, we presented a novel S^{3}CA method that utilizes the SFFT to achieve significant acceleration over the conventional SSCA, particularly for digital radio signals that are always sparse in cycle frequency. The speedup achieved was more than 30 for input windows of

2 million samples and 100 for input windows of 16 million samples. Our S^{3}CA avoids unnecessary computations and employs a sparse CDP matrix to reduce memory requirements.

CHAPTER 6

Conclusion

This work explored high-speed spectral correlation density estimators for cyclostationary analysis, focusing on two primary time-smoothing spectral correlation analyzers: the FFT accumulation method (FAM) and strip spectral correlation analyzer (SSCA). Specifically, this research addressed the Aims in Section 1.2 as follows:

- Aim 1 by analyzing quantization errors for both FAM (Chapter 3) and SSCA (Appendix A) techniques.
- Aim 2 was addressed by implementing the FAM on an FPGA ZCU111 platform (discussed in Chapter 3) and the SSCA on an AMD VCK5000 platform (discussed in Chapter 4).
- Aim 3 with a new algorithm sparse strip spectral correlation analyzer to enhance efficiency by targeting non-zero cyclic frequencies (discussed in Chapter 5).

Initially, the quantization error analysis of the FAM was conducted to address the challenges of hardware implementation using fixed-point precision. The FAM was implemented using various fixed-point precision levels, and the observed quantization errors closely matched the computed results. Subsequently, the FAM was implemented on the FPGA ZCU111 platform using an HLS-based design. The implementation was optimized through several techniques, including pipelining, parallelism, dataflow, I/O optimization, and symmetry, to accelerate execution. The quantization error analysis is crucial for understanding the contribution of fixed-point arithmetic to overall quantization error and optimizing computational complexity and resource utilization. The study demonstrated that mixed precision and normalization

techniques significantly reduced quantization errors compared to fixed precision alone. Additionally, this FPGA implementation of FAM achieved the fastest execution time compared to existing state-of-the-art approaches. The same implementation strategies can be applied to the SSCA on the same platform to optimize its performance.

Secondly, the implementation of SSCA on the AMD/Xilinx VCK5000 platform highlights both the potential and the challenges of utilizing high-performance hardware for processing large datasets. The PL effectively handles critical tasks such as data loading, transposition, and FFT computation. However, the bandwidth limitation of 25.6 GB/s per DDR4 module, in stark contrast to the 1 TB/s bandwidth between the PL and AIE, significantly impacts overall performance. The Versal ACAP architecture excels in supporting repetitive and highly parallel computations, because AIE for high-frequency operations and VLIM and SIMD capabilities for efficient parallel processing, addressing the bottlenecks in data transfer between the DDRMC and PL is essential. Two methods can be employed to address this issue while avoiding reliance on the DDRMC. The first approach involves using a smaller input window for SSCA, and the second involves optimizing the computation of a large input window to reduce intermediate memory requirements.

To address the challenges associated with handling large datasets, the S^3CA was developed to efficiently compute cyclostationary features by targeting non-zero cyclic frequencies. This approach leveraged the inherent sparsity in practical digital signals, significantly reducing computational complexity, improving resource utilization, and enhancing I/O performance. The S^3CA demonstrated better performance as the system size increased and sparsity became more pronounced, achieving speedups of over 30 for 2 million sample windows and over 100 for 16 million sample windows.

Overall, this study has shown that optimizing spectral correlation density estimators for cyclostationary analysis on hardware platforms can achieve substantial gains in efficiency and performance, particularly by leveraging sparsity and utilizing advanced optimization techniques. This thesis significantly advances the real-time implementation of SCD. It establishes a foundation for future cyclostationary signal analysis applications, such as real-time automatic modulation classification and RF signal detection.

6.1 Future Outlook

There are several promising directions for future research in hardware-accelerated cyclostationary analysis. First, adopting fixed-point precision for SSCA and S^3CA could significantly reduce resource utilization and allow for a thorough quantization error analysis, similar to what was done for FAM. This would help optimize resource allocation for different stages of SSCA and S^3CA, enabling efficient implementation.

Another potential direction involves implementing real-time S^3CA on hardware platforms, such as AI Engines or FPGAs. Leveraging on-chip memory, including BRAM and URAM, could mitigate the bandwidth limitations between DDRMC and PL components, leading to improved performance. Implementing S^3CA on hardware would enable real-time processing, making it more applicable to practical communication systems.

Additionally, a comprehensive performance comparison between the hardware implementations of FAM, SSCA, and S^3CA would provide further insights into their strengths and limitations under various conditions. Such a comparison could help determine the most suitable approach for specific applications, depending on factors like data size, sparsity, and resource availability.

Finally, a tool that can automatically generate implementations of these methods with customized size and precision is worth designing. This generator would provide theoretical reports of quantization error and resource requirements, allowing users to efficiently evaluate different configurations before committing to a full hardware synthesis. By enabling early performance estimates and quantization error analysis, such a generator could significantly reduce the overall design time, helping to identify the optimal configuration and saving time compared to lengthy hardware synthesis processes. This approach would facilitate rapid prototyping and decision-making and balance accuracy, resource usage, and performance easier.

APPENDIX A

Quantization Error Analysis Expression for SCD Function

This appendix gives a derivation of the output noise and signal variance for the FAM and SSCA method using Fixed Precision (FAM_M1, SSCA_M1) and Mixed Precision (FAM_M2, SSCA_M2) arithmetic. The definitions of the symbols in the expression are listed in the front (List of Symbols).

A1 Quantization Error Analysis of FAM

The σ^2_{FAM} is the variance of the output noise of FAM algorithm and P_{FAM} is the variance (power) of the output signal. In this appendix, the expressions of the output noise variance are simplified to the format of Equation (3.9).

A1.1 FAM_M1 - Fixed Precision Model

In this section, σ^2_W, $\sigma^2_{F_1}$, σ^2_{CM} and $\sigma^2_{F_2}$ are the noise variance generated by the quantization in windowing, first FFT, conjugate multiplication and second FFT section respectively. Similarly, G_{F_1}, G_{CM} and G_{F_2} are the gains of each section, used to amplify the noise and signal passed through. Details of the calculation of those parameters are described in Section 2.3.1 and Section 3.2.1.

$$\begin{aligned}\sigma^2_{FAM\_M1} &= ((\sigma^2_W G_{F_1} + \sigma^2_{F_1})G_{CM} + \sigma^2_{CM})G_{F_2} + \sigma^2_{F_2} \\ &= ((2\sigma^2_{s.CM}(\sigma^2_W \frac{1}{N_P} + \sigma^2_{F1}) + (\sigma^2_W \frac{1}{N_P} + \sigma^2_{F1})^2) + \sigma^2_r)\frac{1}{P} + \sigma^2_{F2} \\ &= (\frac{2P_{i.s}}{N_P 1.59^2}(\frac{2^{-2F_W}}{6N_P} + \frac{2^{-2F_1}}{3}(2 - \frac{m_1 + 1.5}{N_P})) \\ &+ (\frac{2^{-2F_W}}{6N_P} + \frac{2^{-2F_1}}{3}(2 - \frac{m_1 + 1.5}{N_P}))^2 + \frac{2^{-2F_{CM}}}{3})\frac{1}{P} + \frac{2^{-2F_2}}{3}(2 - \frac{m_2 + 1.5}{P}) \\ &\approx \frac{2P_{i.s}}{6N_P^2 P 1.59^2}2^{-2F_W} + \frac{2P_{i.s}(A_1)}{3N_P P 1.59^2}2^{-2F_1} + \frac{1}{3P}2^{-2F_{CM}} + \frac{A_2}{3}2^{-2F_2} \\ &= W_W 2^{-2F_W} + W_{F_1}2^{-2F_1} + W_{CM}2^{-2F_{CM}} + W_{F_2}2^{-2F_2}\end{aligned} \tag{A.1}$$

$$P_{FAM\_M1} = (\frac{P_{i.s}}{N_P 1.59^2})^2 \frac{1}{P} \tag{A.2}$$

A1.2 FAM_M2 - Mixed Precision Model

Expressions for the output noise and signal variance for the FAM_M2 method are given below.

$$\begin{aligned}\sigma^2_{FAM\_M2} &= (((((\sigma^2_W G_{F_1} + \sigma^2_{F_1})G_{Norm_1} + \\ &\sigma^2_{q_1})G_{CM} + \sigma^2_{CM})G_{Norm_2} + \sigma^2_{q_2})G_{F_2} + \sigma^2_{F_2})G_{Norm_3} + \sigma^2_{q_3} \\ &\approx \frac{N_P^2 P P_{i.s} q_1^4 q_2^2 q_3^2}{3 * 1.59^2}2^{-2F_W} + \frac{2P_{i.s}N_P P q_1^2 q_2^2 q_3^2}{3 * 1.59^2}(q_1^2 B_1 + 0.5)2^{-2F_1} \\ &+ \frac{P q_3^2(q_2^2 + 0.5)}{3}2^{-2F_{CM}} + \frac{q_3^2 B_2 + 0.5}{3}2^{-2F_2} \\ &= W_W 2^{-2F_W} + W_{F_1}2^{-2F_1} + W_{CM}2^{-2F_{CM}} + W_{F_2}2^{-2F_2}\end{aligned} \tag{A.3}$$

$$P_{FAM\_M2} = (\frac{P_{i.s}N_P}{1.59^2}q_1^2)^2 q_2^2 q_3^2 P \tag{A.4}$$

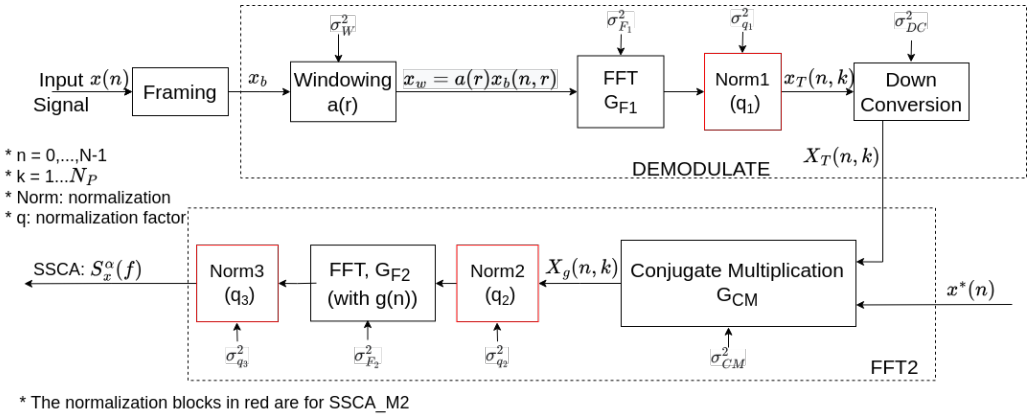


FIGURE A.1. SCD signal flow graph for SSCA_M1 (fixed precision) in black and SSCA_M2 (mixed precision) techniques

A2 Quantization Error Analysis of SSCA

The quantization error analysis for the SSCA method and FAM method differ in that, for the complex demodulate, the parameter L is set to 1, and instead of multiplying by the conjugate complex demodulate, it involves the product with the conjugate $x(n)$. Additionally, quantization errors are introduced during downconversion due to the multiplication with exponential factors in Equation (4.1). In Figure A.1, σ^2_W, $\sigma^2_{F_1}$, σ^2_{DC}, σ^2_{CM} and $\sigma^2_{F_2}$ represent the noise variance generated by quantization in windowing, first FFT, downconversion, conjugate multiplication and second FFT, respectively. Similarly, G_{F_1}, G_{CM}, and G_{F_2} represent the gains of each section, which amplify both noise and signal. The SSCA_M1 and SSCA_M2 utilize similar design methods and symbol definitions as described in Section 3.2.1 and Section 3.3, respectively, as shown in Figure A.1. A detailed explanation of the symbols used in the expressions can be found at the beginning of the thesis in the List of Symbols. This appendix presents the derivation of quantization error analysis for the SSCA method under different models.

A2.1 SSCA_M1 - Fixed Precision Model

$$\begin{aligned}\sigma^2_{SSCA\_M1} &= ((\sigma^2_W * G_{F_1} + \sigma^2_{F_1} + \sigma^2_{DC})G_{CM} + \sigma^2_{CM})G_{F_2} + \sigma^2_{F_2} \\ &= (P_{i.s}(\sigma^2_W \frac{1}{N_P} + \sigma^2_{F1} + \sigma^2_{DC}) + \sigma^2_r) * \frac{1}{N} + \sigma^2_{F2} \\ &= \frac{2^{-2F_W} P_{i.s}}{6N_P N} + \frac{2^{-2F_1} P_{i.s}}{3N}(2 - \frac{m_1 + 1.5}{N_P}) + \frac{2^{-2F_{CM}}}{3N} + \frac{2^{-2F_{DC}} P_{i.s}}{3N} \\ &+ \frac{2^{-2F_2}}{3}(2 - \frac{m_3 + 1.5}{N}) \\ &= \frac{P_{i.s}}{6N_P N} 2^{-2F_W} + \frac{P_{i.s} A_1}{3N} 2^{-2F_1} + \frac{P_{i.s}}{3N} 2^{-2F_{DC}} + \frac{1}{3N} 2^{-2F_{CM}} + \frac{A_3}{3} 2^{-2F_2} \\ &= W_W 2^{-2F_W} + W_{F_1} 2^{-2F_1} + W_{DC} 2^{-2F_{DC}} + W_{CM} 2^{-2F_{CM}} + W_{F_2} 2^{-2F_2}\end{aligned} \tag{A.5}$$

$$P_{SSCA\_M1} = \frac{P^2_{i.s}}{1.59^2 N_P N} \tag{A.6}$$

A2.2 SSCA_M2 - Mixed Precision Model

$$\begin{aligned}\sigma^2_{SSCA\_M2} &= (((((\sigma^2_W * G_{F_1} + \sigma^2_{F_1})G_{Norm_1} + \sigma^2_{DC} + \sigma^2_{q_1}) * G_{CM} + \sigma^2_{CM})G_{Norm_2} \\ &+ \sigma^2_{q_2})G_{F_2} + \sigma^2_{F_2})G_{Norm_3} + \sigma^2_{q_3} \\ &= \frac{N_P N q_1^2 q_2^2 q_3^2 P_{i.s}}{6} 2^{-2F_W} + \frac{N q_2^2 q_3^2 P_{i.s}}{3}(q_1^2 * B_1 + 0.5) 2^{-2F_1} \\ &+ \frac{N q_3^2 q_2^2 P_{i.s}}{3} 2^{-2F_{DC}} + \frac{N q_3^2 (q_2^2 + 0.5)}{3} 2^{-2F_{CM}} + \frac{q_3^2 * B_3 + 0.5}{3} 2^{-2F_2} \\ &= W_W 2^{-2F_W} + W_{F_1} 2^{-2F_1} + W_{DC} 2^{-2F_{DC}} + W_{CM} 2^{-2F_{CM}} + W_{F_2} 2^{-2F_2}\end{aligned} \tag{A.7}$$

$$P_{SSCA\_M2} = \frac{P^2_{i.s}}{1.59^2} * q_1^2 q_2^2 q_3^2 N_P N \tag{A.8}$$

A3 Simulation Result

The simulations were performed by directly implementing the FAM and SSCA algorithms in C. Fixed-point arithmetic was achieved using the `ap_fixe` type in Vivado HLS. The simulation results were compared with our mathematical derivations using Sine Wave, Square

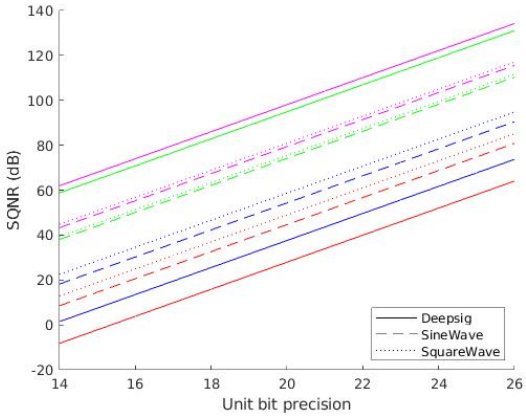
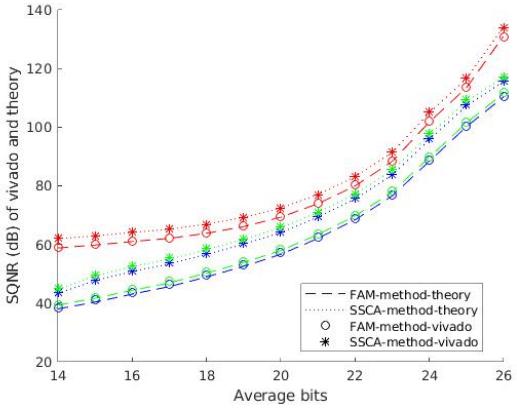


FIGURE A.2. SQNR performance for the FAM and the SSCA method in our models (theory) at different precisions B ($F = B - 1$) (Red: FAM_M1; Blue: SSCA_M1; Green: FAM_M2; Magenta: SSCA_M2)

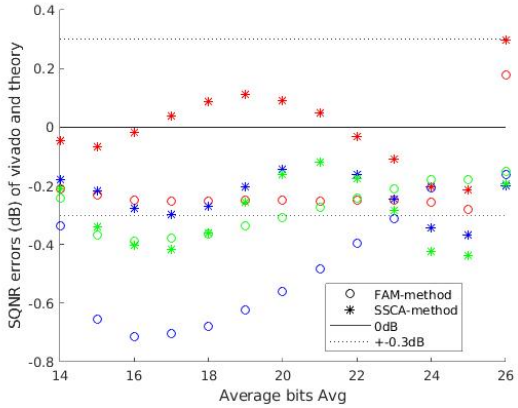
Wave, and samples from the DeepSig RADIO ML 2018.01A dataset [39]. On FPGA devices, the embedded DSP blocks support precision up to 18 bits, with additional bits achievable through programmable logic, so experiments focused on this range and higher values.

The input size was 2048 points, with the following SCD parameters: $N_P = 256$, $L = 64$, $P = 32$ (for FAM) or $N = 2048$ (for SSCA).

Figure A.2 shows the SQNR for different uniform bit precisions for the Deepsig, Sine Wave, and Square Wave signals across the methods discussed. The M2 techniques show a significant improvement in SQNR compared to M1. Figure A.3(A) presents the average SQNR for FAM_M2 and SSCA_M2, comparing non-uniform precision results from theoretical calculations and HLS simulations for different input signals. The average number of bits is calculated as $(B_W + B_1 + B_{CM} + B_2)/4$, where B_* ranges from 14 to 26. Figure A.3(B) illustrates the difference between the theoretical estimates and Vivado simulation results shown in Figure A.3(A). The findings indicate that our formulae provide an accurate estimate of the lower bound for SQNR, with different signals showing similar or higher values. The rightmost examples, which have positive error values (DeepSigFAM and DeepSigSSCA), exhibit very small differences (0.3 dB).



(A) The comparison of theory and Vivado simulation in average bits (Red: DeepSig; Green: Squarewave; Blue: Sinewave)



(B) Error between theory and simulation in average bits (Red: DeepSig; Green: Squarewave; Blue: Sinewave)

FIGURE A.3. Comparison between theoretical calculations and Vivado simulations

Abbreviations

2DFFT: common factor map decomposition FFT. 72, 73, 77, 81

ACP: averaged cyclic periodogram. 19, 20

AGU: address generation unit. 71

AIE: artificial intelligence engine. iii, vi, 2–5, 63, 64, 69–72, 76–79, 81–84, 104

AR: autoregressive. 1

ARMA: autoregressive moving average. 1

BPSK: binary phase-shift keying. 13, 81, 97–100

CDP: channel-data product. iii, xiv, 65, 72, 73, 77–79, 84, 85, 102

CFM: common factor map. 26, 27

CMS: cyclic modulation spectrum. 19

CPU: central processing unit. 2, 3, 28, 29, 63, 64, 84

CS: cyclic spectrum. 1

DDRMC: double data rate memory controller. iii, vi, 5, 64, 69, 71, 77–82, 84, 104, 105

DFT: discrete Fourier transform. xiv, 19, 25, 26, 67, 68

DIF: decimation-in-frequency. 8, 22–24, 26, 28

DIT: decimation-in-time. 8, 22, 24, 26, 28, 35, 37, 40

DSP: digital signal processing. xiv, 2, 3, 27, 30, 56, 63, 70

DSSS: direct-sequence spread-spectrum. 13, 81, 97, 100

EDA: electronic design automation. 33

FACP: fast averaged cyclic periodogram. 20

FAM: FFT accumulation method. iii, vi, xiii–xvi, 2, 5, 7, 8, 14, 16–18, 20, 28–32, 35, 40, 53, 56, 57, 60–63, 66, 103–106, 108, 110

FB: frequency bucketization. 91, 92, 94–96

FFT: fast Fourier transform. iii, xiii–xv, 2, 4, 8, 11, 13, 15, 17, 18, 22–28, 32, 33, 35, 37, 38, 40, 41, 43, 44, 46, 56, 63–68, 72, 73, 75–77, 81–87, 91–94, 96, 97, 101, 104

FPGA: field-programmable gate array. vi, xiv, xvi, 2–5, 26, 28–30, 33, 34, 53, 57–63, 104, 110

FS: frequency-smoothing. 1, 10, 11, 14

FSC: fast spectral correlation. 19

GPU: graphics processing unit. 2, 29, 62, 64, 84

HLS: high-level synthesis. 5, 30, 33, 34, 70

II: initiation interval. 34, 43, 50, 51, 57, 58

MAC: multiply-accumulate. 59, 71

NoC: network-on-chip. 70

PL: programmable logic. iii, vi, 3, 5, 63, 64, 69–71, 77–82, 84, 104, 105

PLIO: programmable logic input/ouput. 71, 82

PSD: power spectral density. 1

RF: radio frequency. 4

RTL: register-transfer-level. 33

S^{3}CA: sparse strip spectral correlation analyzer. iii, vi, xv, xvi, 5, 7, 85, 86, 93–105

SCD: spectral correlation density. iii, xiii–xvi, 1–6, 8–10, 12–14, 19, 20, 27–32, 36, 39, 42, 45, 48–51, 55, 57, 58, 60–63, 72, 73, 75, 85, 86, 99, 100, 104, 108, 110

SFFT: sparse fast Fourier transform. iii, vi, xiv, xv, 4, 5, 7, 26–28, 85–87, 90–97, 100, 101

SIMD: single-instruction multiple-data. iii, 5, 69, 71

SNR: signal-to-noise ratio. 2, 13, 81, 99, 100

SoC: system-on-chip. 63

SQNR: signal-to-quantization-noise ratio. iii, xiii, xv, xvi, 2–5, 8, 28–30, 37, 39, 40, 42, 52–59, 62, 110

SSCA: strip spectral correlation analyzer. iii, vi, xiii–xvi, 2, 4, 5, 7, 8, 14, 16, 18, 28, 62–66, 72, 77, 81–86, 96–101, 103–106, 108–110

SSCA_2DFFT: strip spectral correlation analyzer utilizing a decomposition FFT. vi, xiv, 72–74, 76, 77, 81, 82, 84

STFT: short-time Fourier transform. 19

TS: time-smoothing. 1, 2, 10–12, 14, 15, 17, 18, 62

VHDL: very high speed integrated circuit hardware description language. 33

VLIW: very-long instruction word. iii, 5, 69, 70

Bibliography

[1] Jaafar K. Alsalaet. 'Fast Averaged Cyclic Periodogram method to compute spectral correlation and coherence'. In: *ISA Transactions* 129 (2022), pp. 609–630. ISSN: 0019-0578. DOI: https://doi.org/10.1016/j.isatra.2022.01.029. URL: https://www.sciencedirect.com/science/article/pii/S0019057822000441.

[2] AMD Inc. *Vitis DSP Library: Fast Fourier Transform (FFT)*. AMD Adaptive and Embedded Computing Group. Nov. 2024. URL: https://github.com/Xilinx/Vitis_Libraries/tree/v2024.2_rel/dsp (visited on 22/04/2025).

[3] AMD Xilinx. *AM009 Versal AI Engine*. Versal ACAP AI Engine Architecture Manual. 2021. URL: https://www.xilinx.com.

[4] AMD Xilinx. *UG1393 Application Acceleration Development*. Vitis Unified Software Platform Documentation. 2022. URL: https://docs.amd.com/r/en-US/ug1393-vitis-application-acceleration.

[5] Roberto Ammendola and Pierpaolo Loreti. 'Design and Evaluation of a Scalable Engine for 3D-FFT Computation in an FPGA Cluster'. In: *International Journal on Advanced Science, Engineering and Information Technology* 9.2 (Apr. 2019), pp. 677–684. DOI: 10.18517/ijaseit.9.2.8308. URL: https://doi.org/10.18517/ijaseit.9.2.8308.

[6] Jérôme Antoni. 'Cyclic spectral analysis in practice'. In: *Mechanical Systems and Signal Processing* 21.2 (2007), pp. 597–630. ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2006.08.007. URL: https://www.sciencedirect.com/science/article/pii/S0888327006001816.

[7] Jérôme Antoni. 'Cyclostationarity by examples'. In: *Mechanical Systems and Signal Processing* 23.4 (2009), pp. 987–1036. ISSN: 0888-3270. DOI: https://doi.org/

10.1016/j.ymssp.2008.10.010. URL: https://www.sciencedirect.com/science/article/pii/S0888327008002690.

[8] Jérôme Antoni, Ge Xin and Nacer Hamzaoui. 'Fast computation of the spectral correlation'. In: *Mechanical Systems and Signal Processing* 92 (2017), pp. 248–277. ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2017.01.011. URL: https://www.sciencedirect.com/science/article/pii/S0888327017300134.

[9] E. April. *On the Implementation of the Strip Spectral Correlation Algorithm for Cyclic Spectrum Estimation*. DREO technical note. Defence Research Establishment Ottawa, 1994. URL: https://books.google.com.au/books?id=7QD7MwEACAAJ.

[10] Viduneth Ariyarathna et al. 'Analog Approximate-FFT 8/16-Beam Algorithms, Architectures and CMOS Circuits for 5G Beamforming MIMO Transceivers'. In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8.3 (2018), pp. 466–479. DOI: 10.1109/JETCAS.2018.2832177.

[11] Mohammed Bahoura. 'Efficient FPGA-Based Architecture of the Overlap-Add Method for Short-Time Fourier Analysis/Synthesis'. In: *Electronics* 8.12 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8121533. URL: https://www.mdpi.com/2079-9292/8/12/1533.

[12] Nilangshu Bidyanta et al. '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)*. San Diego, CA: Springer, 2015, pp. 70–79.

[13] P. Borghesani. 'The envelope-based cyclic periodogram'. In: *Mechanical Systems and Signal Processing* 58-59 (2015), pp. 245–270. ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2014.11.009. URL: https://www.sciencedirect.com/science/article/pii/S0888327014004221.

[14] P. Borghesani and J. Antoni. 'A faster algorithm for the calculation of the fast spectral correlation'. In: *Mechanical Systems and Signal Processing* 111 (2018), pp. 113–118. ISSN: 0888-3270. DOI: https://doi.org/10.1016/j.ymssp.2018.03.

059. URL: https://www.sciencedirect.com/science/article/pii/S0888327018301869.

[15] R.N. Bracewell. *The Fourier Transform and Its Applications*. Circuits and systems. McGraw Hill, 2000. ISBN: 9780073039381. URL: https://books.google.com.hk/books?id=ZNQQAQAAIAAJ.

[16] Ronald N. Bracewell. *The Fourier Transform and its applications*. Boston: McGraw Hill, 1986.

[17] Anders Brandt. *Noise and vibration analysis: signal analysis and experimental procedures*. Chichester, West Sussex, U.K. ; John Wiley & Sons, 2011.

[18] William A Brown and Herschel H Loomis. 'Digital implementations of spectral correlation analyzers'. In: *IEEE Transactions on Signal Processing* 41.2 (1993), pp. 703–720.

[19] William Alexander Brown. 'On the theory of cyclostationary signals'. PhD thesis. University of California Davis, 1987, p. 413.

[20] James W. Cooley and John W. Tukey. 'An algorithm for the machine calculation of complex Fourier series'. eng. In: *Mathematics of computation* 19.90 (1965), pp. 297–301. ISSN: 0025-5718.

[21] Intel Corporation. *FFT IP Core: User Guide*. Doc. ID 683374, ver. 17.1. Intel FPGA. 2017. URL: https://www.intel.com/content/www/us/en/docs/programmable/683374/17-1/about-this-ip-core.html (visited on 22/04/2025).

[22] *cuFFT API Reference*. https://docs.nvidia.com/cuda/cufft/. Accessed: 2024-10-27.

[23] P. D'Alberto et al. 'Generating FPGA-Accelerated DFT Libraries'. eng. In: *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. IEEE, 2007, pp. 173–184. ISBN: 9780769529400.

[24] P. Duhamel and M. Vetterli. 'Fast fourier transforms: A tutorial review and a state of the art'. In: *Signal Processing* 19.4 (1990), pp. 259–299. ISSN: 0165-1684. DOI: https://doi.org/10.1016/0165-1684(90)90158-U. URL: https://www.sciencedirect.com/science/article/pii/016516849090158U.

[25] A. Fehske, J. Gaeddert and J.H. Reed. 'A new approach to signal classification using spectral correlation and neural networks'. In: *First IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks, 2005. DySPAN 2005.* 2005, pp. 144–150. DOI: 10.1109/DYSPAN.2005.1542629.

[26] M. Frigo and S.G. Johnson. 'FFTW: an adaptive software architecture for the FFT'. In: *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*. Vol. 3. 1998, 1381–1384 vol.3. DOI: 10.1109/ICASSP.1998.681704.

[27] Matteo Frigo and Steven G. Johnson. *FFTW Library version 3.3.10*. Accessed: July 4, 2023. URL: http://www.fftw.org/download.html.

[28] Matteo Frigo and Steven G. Johnson. 'The Design and Implementation of FFTW3'. In: *Proceedings of the IEEE* 93.2 (2005). Special issue on "Program Generation, Optimization, and Platform Adaptation", pp. 216–231.

[29] William A Gardner. *Cyclostationarity in communications and signal processing*. New York: IEEE Press, 1994.

[30] William A Gardner. 'The spectral correlation theory of cyclostationary time-series'. In: *Signal processing* 11.1 (1986), pp. 13–36.

[31] William A Gardner, Antonio Napolitano and Luigi Paura. 'Cyclostationarity: Half a century of research'. In: *Signal processing* 86.4 (2006), pp. 639–697.

[32] William A. Gardner. 'Exploitation of spectral redundancy in cyclostationary signals'. In: *IEEE Signal Processing Magazine* 8 (Apr. 1991), pp. 14–36. DOI: 10.1109/79.81007.

[33] Mario Garrido et al. 'Pipelined Radix-2^k Feedforward FFT Architectures'. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 21.1 (2013), pp. 23–32. DOI: 10.1109/TVLSI.2011.2178275.

[34] I. J. Good. 'The Interaction Algorithm and Practical Fourier Analysis'. In: *Journal of the Royal Statistical Society: Series B (Methodological)* 20.2 (Dec. 2018), pp. 361–372. ISSN: 0035-9246. DOI: 10.1111/j.2517-6161.1958.tb00300.x. eprint: https://academic.oup.com/jrsssb/article-pdf/20/2/361/

49096113/jrsssb_20_2_361.pdf. URL: https://doi.org/10.1111/j.2517-6161.1958.tb00300.x.

[35] Pankaj Gupta. 'Accurate performance analysis of a fixed point FFT'. eng. In: *2016 Twenty Second National Conference on Communication (NCC)*. India: IEEE, 2016, pp. 1–6. ISBN: 1509023615.

[36] Haitham Hassanieh. *The Sparse Fourier Transform: Theory and Practice*. Vol. 19. Association for Computing Machinery and Morgan & Claypool, 2018. ISBN: 9781947487079.

[37] Haitham Hassanieh et al. 'Nearly Optimal Sparse Fourier Transform'. In: *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*. STOC '12. New York, New York, USA: Association for Computing Machinery, 2012, pp. 563–578. ISBN: 9781450312455. DOI: 10.1145/2213977.2214029. URL: https://doi.org/10.1145/2213977.2214029.

[38] Haitham Hassanieh et al. 'Simple and Practical Algorithm for Sparse Fourier Transform'. In: *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '12. Kyoto, Japan: Society for Industrial and Applied Mathematics, 2012, pp. 1183–1194.

[39] DeepSig Inc. *RF Datasets For Machine Learning*. https://www.deepsig.ai/datasets. 2018.

[40] Xilinx Inc. *Fast Fourier Transform LogiCORE IP Product Guide PG109*. 2021. URL: https://docs.xilinx.com/v/u/en-US/pg109-xfft.

[41] Xilinx Inc. *Vivado Design Suite User Guide High-Level Synthesis UG902 (v2019.2)*. 2019. URL: https://www.xilinx.com/support/documentation/sw%5C_manuals/xilinx2019_2/ug902-vivado-high-level-synthesis.pdf.

[42] Dina Katabi et al. *SFFT Website*. URL: https://groups.csail.mit.edu/netmit/sFFT/.

[43] Jong Hwan Ko et al. 'Design of an Energy-Efficient Accelerator for Training of Convolutional Neural Networks using Frequency-Domain Computation'. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. DAC '17. Austin, TX, USA: Association for Computing Machinery, 2017. ISBN: 9781450349277.

DOI: 10.1145/3061639.3062228. URL: https://doi.org/10.1145/3061639.3062228.

[44] Carol Jingyi Li et al. 'Fixed-point FPGA Implementation of the FFT Accumulation Method for Real-time Cyclostationary Analysis'. In: *ACM Trans. Reconfigurable Technol. Syst.* 16.3 (June 2023). ISSN: 1936-7406. DOI: 10.1145/3567429. URL: https://doi.org/10.1145/3567429.

[45] Carol Jingyi Li et al. 'S^3CA: A Sparse Strip Spectral Correlation Analyzer'. In: *IEEE Signal Processing Letters* 31 (2024), pp. 646–650. DOI: 10.1109/LSP.2024.3364062.

[46] Xiangwei Li et al. 'A Scalable Systolic Accelerator for Estimation of the Spectral Correlation Density Function and Its FPGA Implementation'. In: *ACM Trans. Reconfigurable Technol. Syst.* (June 2022). Just Accepted. ISSN: 1936-7406. DOI: 10.1145/3546181. URL: https://doi.org/10.1145/3546181.

[47] Weiqiang Liu et al. 'Approximate Designs for Fast Fourier Transform (FFT) With Application to Speech Recognition'. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 66.12 (2019), pp. 4727–4739. DOI: 10.1109/TCSI.2019.2933321.

[48] Xueyuan Liu et al. 'Wireless Signal Representation Techniques for Automatic Modulation Classification'. In: *IEEE Access* 10 (2022), pp. 84166–84187. DOI: 10.1109/ACCESS.2022.3197224.

[49] Alexander López-Parrado and Jaime Velasco-Medina. 'SoC-FPGA implementation of the sparse fast fourier transform algorithm'. In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2017, pp. 120–123. DOI: 10.1109/MWSCAS.2017.8052875.

[50] Richard G. Lyons. *Understanding Digital Signal Processing*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN: 0201634678.

[51] Yishay Mansour. 'Randomized interpolation and approximation of sparse polynomials stPreliminary version'. In: *Automata, Languages and Programming*. Ed. by W. Kuich. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 261–272. ISBN: 978-3-540-47278-0.

[52] Scott Marshall et al. ‘GPGPU based parallel implementation of spectral correlation density function’. In: *Journal of Signal Processing Systems* 92.1 (2020), pp. 71–93.

[53] Razvan Nane et al. ‘A Survey and Evaluation of FPGA High-Level Synthesis Tools’. eng. In: *IEEE transactions on computer-aided design of integrated circuits and systems* 35.10 (2016), pp. 1591–1604. ISSN: 0278-0070.

[54] Alan V Oppenheim and Clifford J Weinstein. ‘Effects of finite register length in digital filtering and the fast Fourier transform’. In: *Proceedings of the IEEE* 60.8 (1972), pp. 957–976.

[55] Alan V. (Alan Victor) Oppenheim, John R. Buck and Ronald W. (Ronald William) Schafer. *Discrete-time signal processing*. eng. 2nd ed. / Alan V. Oppenheim, Ronald W. Schafer, with John R. Buck. Prentice-Hall signal processing series. London: Prentice-Hall International, 1999. ISBN: 0130834432.

[56] Pedro Paz and Mario Garrido. ‘A 12.8-GS/s 32-Parallel 1 Million-Point FFT’. In: *2024 39th Conference on Design of Circuits and Integrated Systems (DCIS)*. 2024, pp. 1–6. DOI: 10.1109/DCIS62603.2024.10769108.

[57] Barathram Ramkumar. ‘Automatic modulation classification for cognitive radios using cyclic feature detection’. In: *IEEE Circuits and Systems Magazine* 9.2 (2009), pp. 27–45. DOI: 10.1109/MCAS.2008.931739.

[58] R.S. Roberts. *Architectures for Digital Cyclic Spectral Analysis*. UMI Dissertation Services, 1989. URL: https://books.google.com.au/books?id=zWOSYmH2ldIC.

[59] Randy S Roberts, William A Brown and Herschel H Loomis. ‘Computationally efficient algorithms for cyclic spectral analysis’. In: *IEEE Signal Processing Magazine* 8.2 (1991), pp. 38–49.

[60] Wolfgang Schlecker, Christiane Beuschel and Hans-Jörg Pfleiderer. ‘Quantisation noise in fixed-point multiplications’. In: *Electrical Engineering* 89.4 (2007), pp. 339–342.

[61] Steven R Schnur. *Identification and classification of OFDM based signals using preamble correlation and cyclostationary feature extraction*. Tech. rep. Naval Postgraduate School, Monterey CA, 2009.

[62] R. Singleton. ‘An algorithm for computing the mixed radix fast Fourier transform’. In: *IEEE Transactions on Audio and Electroacoustics* 17.2 (1969), pp. 93–103. DOI: 10.1109/TAU.1969.1162042.

[63] Julius O. Smith. *Spectral Audio Signal Processing*. online book, 2011 edition. http://ccrma.stanford.edu/~jos/sasp/, 2024.

[64] Sundaramurthy and Reddy. ‘Some Results in Fixed-Point Fast Fourier Transform Error Analysis’. eng. In: *IEEE transactions on computers* C-26.3 (1977), pp. 305–308. ISSN: 0018-9340.

[65] Charles. Van Loan, Society for Industrial and Applied Mathematics. *Computational frameworks for the fast fourier transform*. eng. Frontiers in applied mathematics ; vol. 10. Philadelphia, Pa: Society for Industrial and Applied Mathematics SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104, 1992. ISBN: 9781611970999.

[66] *VCK5000 Versal Development Card*. https://www.xilinx.com/products/boards-and-kits/vck5000.html. Accessed: 2024-10-27.

[67] Martin Vetterli, member Eurasip and Henri J. Nussbaumer. ‘Simple FFT and DCT algorithms with reduced number of operations’. eng. In: *Signal processing* 6.4 (1984), pp. 267–278. ISSN: 0165-1684.

[68] Bernard Widrow and Istvá Kollár. *Quantization noise: roundoff error in digital computation, signal processing, control, and communications*. 2008.

[69] Shmuel Winograd. ‘On computing the Discrete Fourier Transform’. In: *Proceedings of the National Academy of Sciences* 73.4 (1976), pp. 1005–1006. DOI: 10.1073/pnas.73.4.1005.

[70] Wojciech Zabolotny. *Versatile FFT: Parametrized FFT Engine*. 2015. URL: https://opencores.org/projects/versatile_fft (visited on 22/04/2025).