High-Level Accelerator Design for Plane-Wave Ultrasound Beamforming in Fourier Domain

by

Mahdi Babajan Rahaghi

B.Sc., Shahed University, Tehran, Iran, 2013

M.Sc., Shahid Beheshti University, Tehran, Iran, 2017

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Department of Electrical and Computer Engineering

We acknowledge and respect the Lək^{w}əŋən (Songhees and X^{w}sepsəm/Esquimalt) Peoples on whose territory the university stands, and the Lək^{w}əŋən and W̱ SÁNEĆ Peoples whose historical relationships with the land continue to this day.

High-Level Accelerator Design for Plane-Wave Ultrasound Beamforming in Fourier Domain

by

Mahdi Babajan Rahaghi

B.Sc., Shahed University, Tehran, Iran, 2013

M.Sc., Shahid Beheshti University, Tehran, Iran, 2017

Supervisory Committee

Dr. Daler N. Rakhmatov, Supervisor
Department of Electrical and Computer Engineering

Dr. David Capson, Departmental Member
Department of Electrical and Computer Engineering

Abstract

This thesis presents the design and implementation of a high-throughput receive beamforming system for Fourier-domain multi-angle plane-wave ultrasound image reconstruction using high-level synthesis (HLS) on a AMD Versal™ adaptive system-on-chip device. The proposed architecture implements a customized Temme–Mueller migration algorithm entirely within the programmable logic fabric, avoiding off-chip memory and relying solely on on-chip resources. The system operates in a stream-based multi-stage pipelined fashion, with each stage — including temporal and spatial Fast Fourier Transforms (FFTs), dynamic phase delay, spectral remapping, multi-angle coherent compounding, and inverse FFTs— realized as modular, latency-aware HLS blocks.

Unlike previous methods, the system performs remapping and interpolation without relying on large, precomputed lookup tables, instead computing all migration-related parameters on the fly. The architecture achieves a sustained processing throughput of over 500 frames per second (with five-angle compounding), enabled by four-lane parallelization and efficient pipelining across all stages. The system HLS specification is statically parameterized in terms of user-controlled FFT lengths, data frame sizes. and number of compounding angles. On the other hand, the key imaging parameters, such as speed of sound, sampling frequency, and probe geometry, are configurable at runtime. Extensive module-level simulation and architecture-level integration testing have validated the system's correctness confirmed against a MATLAB-based reference model.

Table of Contents

List of Tables

List of Figures

Glossary

AXI – Advanced eXtensible Interface (Stream).

BRAM – Block Random Access Memory

CPWC – Coherent Plane-Wave Compounding

DAS – Delay-and-Sum (Beamforming)

DSP – Digital Signal Processing (Block)

FFT – Fast Fourier Transform

FPGA – Field-Programmable Gate Array

HLS – High-Level Synthesis

IFFT – Inverse Fast Fourier Transform

II – Initiation Interval

ILA – Integrated Logic Analyzer

LUT – Look-Up Table

PL – Programmable Logic

PS – Processing System

TM – Temme-Mueller (Algorithm)

URAM – Ultra Random Access Memory

PWI – Plane Wave Imaging

RTL – Register Transfer Level (Design)

SNR – Signal-to-Noise Ratio

1. Introduction

Ultrasound imaging remains one of the most widely used diagnostic tools in medical imaging due to its real-time capability, portability, and safety. Unlike CT or MRI, ultrasound does not rely on ionizing radiation or large-scale infrastructure, making it well-suited for point-of-care diagnostics, fetal imaging, cardiac assessment, and tissue stiffness characterization. Despite its clinical popularity, ultrasound systems are often constrained by trade-offs between spatial resolution, frame rate, and computational complexity. Traditional imaging systems typically acquire and process one focused beam at a time, which limits acquisition speed and impairs real-time responsiveness in dynamic applications such as cardiac imaging, interventional procedures, and elastography.

To overcome these limitations, ultrafast ultrasound techniques, such as Coherent Plane-Wave Compounding (CPWC), have been developed. The key idea is to insonify a large subsurface region at once using an unfocused transmission and combine data from multiple transmit angles to produce high-quality images. However, these methods dramatically increase data volume and place a heavy computational burden on the beamforming stage, which must now process large, multi-channel datasets.

Conventional beamforming approaches, such as time-domain Delay-and-Sum (DAS) method, become performance bottlenecks under high-frame-rate conditions. While DAS beamforming is well suited for parallelization on GPU hardware, it remains challenging to implement it on resource-constrained embedded platforms. Frequency-domain beamforming techniques, such as those based on Fourier-domain migration, reformulate the beamforming problem into a series of spectral operations (FFTs, phase shifts, and spectral remapping). Among these, the Temme–Mueller (TM) algorithm stands out for its ability to perform computationally efficient and accurate image reconstruction in a pipelined fashion. However, implementing such spectral-domain algorithms in hardware presents challenges. These include managing multi-dimensional streaming data, maintaining synchronization across pipelined modules, and ensuring efficient use of hardware resources—all while meeting stringent throughput and latency requirements.

1.1. Limitations in Real-Time Imaging

As ultrasound technology continues to expand into ultrafast imaging domains, the demand for higher frame rates, lower latency, and real-time responsiveness has grown considerably. Applications such as cardiac imaging, vascular flow monitoring, and interventional guidance now require systems to deliver full frames in under a millisecond.

However, traditional ultrasound systems — particularly those based on line-by-line focused beam transmissions and DAS beamforming — are fundamentally limited in how quickly they can acquire and process data. Each focused scan line entails a dedicated transmit-receive cycle, and modern images may contain hundreds of lines. Even with optimized hardware, this approach results in total frame delays in the order of tens of milliseconds, which is too slow for real-time imaging of fast physiological events [1].

This bottleneck has led to the adoption of ultrafast imaging methods, such as CPWC. These techniques dramatically increase the imaging frame rate by illuminating the entire imaging field with unfocused acoustic beams and beamforming echoes from multiple transmit angles [2]. However, the benefits of CPWC come at a computational cost:

- Each transmit produces channel data from every element, across hundreds or thousands of samples per channel.
- Multiples transmit angles are typically needed (e.g., 8–16), leading to a ten-fold or more increase in total data volume.
- Beamforming algorithms must run at accelerated rates to keep pace when reconstructing a full images frame at a rate exceeding 1,000 frames per second (fps).

Time-domain DAS beamformers, even when offloaded to GPUs, often struggle to meet these real-time constraints without compromising resolution or flexibility [3]. This performance pressure has motivated the exploration of Fourier-domain beamforming techniques, such as Temme-Mueller migration, which allows for streamable FFT-based implementations on FPGA-like hardware [4]. This thesis presents a pipelined design of a Fourier-domain beamformer based on the TM algorithm, synthesized using HLS and targeting a Versal™ XCVC1902 adaptive SoC.

1.2. Related Work

Prior work on coherent plane-wave compounding (CPWC) imaging has explored both time-domain and frequency-domain beamforming architectures. While time-domain delay-and-sum (DAS) methods are widely used, their high computational demand has led to interest in alternative formulations such as Fourier-domain migration. Time-domain approaches offer implementation simplicity and flexibility for per-pixel delay adaptation [5], with hardware designs supporting both focused and divergent wave imaging [6]. Some architectures have addressed 2D [7][8][9] and 3D [10] imaging; however, this thesis focuses exclusively on 2D CPWC reconstruction.

The method employed here builds upon the Temme–Mueller (TM) migration algorithm due to its computational efficiency, primarily leveraging 2D FFTs. Compared to the $O(N_t N_x N_x)$ complexity of traditional DAS beamforming, TM migration scales as $O(N_t N_x \log(N_t N_x))$, where N_x and N_t refer to the number of probe channels and the number of samples per channel (forming a single-angle data frame). Previous Fourier-domain techniques [4][11][12][13][14][15] have laid the groundwork for such methods, but few implementations have translated these into FPGA-accelerated platforms. A GPU-based software implementation of the method from [15], which is practically identical to Lu's beamforming [12], was reported in [16]; however, no corresponding hardware implementation was developed.

A central challenge in hardware-based CPWC is achieving frame rates above 1,000 fps to support continuous acquisition. For example, assuming a 70 mm depth and a 1540 m/s speed of sound, the round-trip time limits single-angle imaging to under 11 kHz, with compounded frame rates (e.g., using 7 angles) requiring at least ~1,570 fps for uncompressed throughput. Several implementations targeting high-throughput DAS beamforming have been reported. A system using Intel Arria V FPGAs achieved 543 fps for 7-angle compounding at 1280×96 resolution [17]. The fastest reported FPGA-based DAS architecture achieved a throughput of 29,466 fps using a Kintex KU19P, albeit without coherent compounding [18].

Some FPGA-based efforts have adopted Fourier-based methods with varying levels of fidelity. The most recent hardware realization [9] using a AMD Versal™ VC1902 [19] achieved high throughput for 7-angle compounding. Another work [20] evaluated the feasibility of fixed-point arithmetic for Fourier-domain beamformers inspired by Stolt migration [21], yet lacked a concrete hardware design. In earlier work, a VHDL-based implementation of a Fourier domain ultrasound beamforming system was developed for the AMD Virtex™ UltraScale+ VU13P device [22]. This design, which

implemented the TM migration algorithm, achieved a clock frequency of 385 MHz and delivered a throughput of approximately $[1380/N_a]$ frames per second, where N_a is the number of compounding angles, with each frame consisting of 2048×128 complex-valued samples (64-bit per sample). The implementation used 54.4K LUTs, 238 DSPs, 135.5 BRAMs, and 768 URAMs, with a total processing latency of 278,735 cycles (approximately 724.7 μs). However, the design required off-chip memory to store remapping and phase correction factors, which constrained scalability. This VHDL-based approach demonstrated strong performance, but its complexity and limited configurability motivated the development of a more flexible, fully on-chip HLS-based solution explored in this thesis.

Considering these earlier VHDL-based efforts, the goal of this work was to push toward higher frame rates while adopting a higher-level hardware design flow. Manual RTL implementations often achieve excellent performance but come at the cost of development time and limited flexibility, whereas HLS-based designs are typically easier to develop but their automatically synthesized implementations can have lower performance. For this project, a baseline requirement was to sustain at least 300 frames per second for a five-angle compounding cycle, which is greater than $1380/N_a$ (for $N_a = 5$) offered by the manually VHDL-coded architecture from [22]. As will be presented in the following chapters, the proposed HLS-based pipeline successfully met the baseline requirement and achieved a throughput exceeding 500 frames per second.

1.3. Challenges in Hardware Implementation

Implementing TM migration in hardware introduces several system-level challenges. Our goal is to create a high-throughput pipeline using exclusively C/C++ HLS descriptions that implements the TM algorithm while targeting a AMD Versal™ XCVC1902 device. This requires careful consideration of the following key issues:

- Input 2D data must be transformed along both horizontal and vertical axes, which demands performing sequential 1D FFTs in different dimensions and 2D data transposition between them. Achieving this on hardware necessitates internal buffering to reorder data between processing stages. The spectral remapping step further adds complexity, as it requires resampling the frequency-wavenumber spectrum onto a new grid.
- A high frame-rate beamformer must carry out multiple operations sequentially – such as Fourier transformations, phase correction, interpolation, and

accumulation of results across multiple transmit angles. The execution of these tasks needs to be carefully pipelined, so that different stages operate in parallel on different data. For example, while one data frame is undergoing phase correction, another data frame may simultaneously undergo interpolation. The challenge is to partition the algorithm into pipeline stages and ensure each stage can run in parallel with the others whenever possible, thus achieving high throughput.

- In a streaming architecture, each computational module must consume and produce data at a compatible rate, and any slight mismatches in latency or throughput must be reconciled. Buffering is especially critical when implementing required data transpositions and frame compounding. One may need to use on-chip dual-port memory blocks as ping-pong buffers to temporarily hold intermediate results between stages, effectively acting as FIFO queues. Managing these queues entails careful addressing, handshaking, and double-buffering (using one memory bank for reading while another is being written) to ensure continuous data throughput.
- We want to avoid relying on external memory or large pre-computed lookup tables. External memory accesses would introduce unacceptable latency and bandwidth limitations, and therefore our system needs to compute migration-related parameter values on the fly. The challenge here is twofold: 1) ensuring that these dynamic computations keep pace with the streaming data, and 2) doing so without excessive resource consumption. By generating phase shifts and interpolation indices in hardware rather than storing them, we reduce off-chip memory traffic at the cost of extra on-chip computation.
- Building a complex beamforming engine in HLS requires a modular approach to achieve reliable synthesis and maintainable code. The overall design should be divided into manageable components (computational modules), and each component should be developed and optimized in isolation, which simplifies verification. However, integrating these modules into a complete system raises its own challenges. We must ensure stream alignment between modules – data produced by one module must be correctly timed to be consumed by the next. This often involves inserting FIFO buffers, as mentioned earlier, or adjusting read/write timing so that, for instance, the phase correction unit receives the corresponding frequency-domain data from the FFT at the right clock cycle. Another consideration is the reliability of HLS in achieving the desired parallelism – complex loop logic or insufficient pipeline depth can cause HLS to serialize

operations. Through coding techniques and pragmas, each processing stage must be instructed to operate in a streaming fashion. Finally, handling boundary conditions (frame boundaries and switching between processing one angle to the next) must be properly handled.

Each of these challenges had to be addressed to successfully map the TM algorithm onto a programmable device via HLS. Overcoming them was central to this thesis work, resulting in a pipelined implementation that can reconstruct ultrasound images at high frame rates.

1.4. Thesis Objectives and Contributions

This thesis sets out to develop a beamforming processor tailored for high-throughput plane-wave image reconstruction using the AMD Vitis™ framework. The latter is a development environment for creating designs that includes FPGA fabric, Arm® processor subsystems, and AI Engines. Building upon the hardware principles and algorithmic structure outlined in prior work by Navaeilavasani [23], which targeted the TM migration algorithm and verified its hardware design correctness against a MATLAB-based software reference, this project aims to develop a pipelined HLS accelerator core without reliance on external memory.

1.4.1 Thesis Objectives

The primary objective of this thesis is to design and evaluate a high-throughput, resource-efficient ultrasound beamforming pipeline using HLS techniques that implement the TM migration algorithm. By structuring the design as a fully pipelined and stream-based architecture, the aim is to achieve near-continuous dataflow operation with minimal latency and efficient use of available hardware resources.

This work focuses on achieving post-synthesis and place-and-route closure using Vitis™ HLS and Vivado™. Within this scope, the goal is to demonstrate that a full beamforming chain can be developed at a high level of abstraction while still achieving high CPWC throughput of several hundred frames per second. Additionally, the design serves as a proof of concept for using modular, reusable HLS-based components in complex medical imaging pipelines, potentially accelerating development cycles in future implementations.

1.4.2 Key Contributions

This thesis presents a complete, modular implementation of an ultrasound beamforming processor developed entirely in C++ using Vitis™ HLS. A major contribution of this work lies in the integration of all core beamforming stages—FFT, phase delay correction, spatial frequency remapping, and accumulation—into a tightly pipelined and dataflow-oriented architecture. The design leverages streaming AXI interfaces, customized fixed-point arithmetic, and HLS directives such as loop unrolling, array partitioning, and URAM binding to optimize throughput and resource usage. Unlike earlier designs that relied heavily on static LUTs, the approach developed in this thesis supports dynamic computation of phase and mapping parameters. Overall, this work demonstrates that advanced signal processing for medical imaging can be effectively designed, optimized, and validated at the HLS level, offering a practical foundation for future FPGA-based deployment without a significant design burden and costs associated with RTL development.

1.5. Thesis Roadmap

The thesis is organized as follows:

- Chapter 2 provides a technical background on ultrasound imaging principles, with a focus on Fourier domain beamforming and the mathematical foundations of spectral migration.
- Chapter 3 presents a detailed system-level architecture of the proposed beamforming pipeline. It introduces the modular design, outlines core computational blocks and describes relevant memory organization, buffering strategy, and streaming interfaces.
- Chapter 4 discusses hardware implementation on the AMD Versal™ XCVC1902 device, including resource utilization, timing analysis, throughput metrics, and validation methodology illustrated by waveform traces and MATLAB comparisons.
- Chapter 5 concludes the thesis by summarizing the key contributions, highlighting design limitations, and reflecting on architectural trade-offs. Directions for future system enhancements are briefly noted as well.

2. Fundamentals of Ultrasound Imaging

Ultrasound reflection imaging involves transmitting high-frequency acoustic wave pulses into a medium and analyzing the returning echoes to visualize internal reflectors. These echoes carry critical information about the medium, such as the depth, size, and acoustic properties of various structures. Ultrasound operates at frequencies above 20 kHz [24], well beyond the audible range of human hearing. In medical applications, frequencies between 1 and 15 MHz are commonly used, balancing between image resolution and penetration depth [24]. Higher frequencies offer greater resolution but penetrate less deeply, making them ideal for imaging superficial structures, while lower frequencies are used for visualizing deeper tissues. The interaction of sound waves with tissue interfaces, governed by acoustic impedance, is fundamental to ultrasound imaging [25]. Acoustic impedance refers to the resistance a medium offers to the propagation of sound waves. When waves encounter interfaces between tissues with differing acoustic impedances, a portion of the sound is reflected back to the transducer. Acoustic impedance forms the basis of the detailed visualizations achieved with ultrasound technology [26].

Ultrasound imaging has become an indispensable tool in modern medicine due to its ability to provide real-time, non-invasive diagnostics. Unlike other imaging modalities, such as X-rays or CT scans, ultrasound does not use ionizing radiation, making it particularly suitable in healthcare for pregnant women and pediatric patients [24]. Its ability to visualize soft tissues with high resolution makes it an ideal choice for applications ranging from fetal monitoring in obstetrics to detecting abnormalities in organs like the liver, kidneys, and heart [27]. Furthermore, advancements in portable ultrasound devices have significantly increased accessibility to this technology, especially in remote or resource-limited settings [28].

Cardiology leans heavily on the use of ultrasound to visualize structure and function in the heart via echocardiography. These applications provide critical insights into heart valve performance, chamber size, and blood flow dynamics, which are essential for diagnosing and managing conditions such as heart failure, arrhythmias, and congenital heart defects [29]. Doppler ultrasound, a specialized mode, measures the velocity and direction of blood flow, aiding in the diagnosis of vascular diseases, including arterial blockages and deep vein thrombosis [30]. Additionally, ultrasound-guided procedures, such as biopsies and catheter placements, improve accuracy and reduce risks by providing real-time visual guidance [31].

Beyond diagnostics, ultrasound is increasingly being used in therapeutic applications. High-intensity focused ultrasound (HIFU) is an emerging treatment modality for conditions such as uterine fibroids and certain types of cancer, offering a non-invasive alternative to surgery [32]. Additionally, advancements in point-of-care ultrasound (POCUS) have revolutionized emergency and critical care settings, enabling rapid bedside assessments for trauma, internal bleeding, or organ dysfunction [33]. The versatility, safety, and real-time capabilities of ultrasound make it a cornerstone technology in medical imaging and therapeutic applications, contributing significantly to improved patient outcomes [34]

2.1. Importance of Plane Wave Imaging

Ultrafast ultrasound imaging is an advanced technique that captures raw data at very high frame rates, often exceeding thousands of frames per second (fps). By utilizing plane-wave imaging (PWI), a single unfocused acoustic beam insonifies a large region of the medium, enabling simultaneous acquisition of echoes from a wide area.

Unlike conventional focused imaging, which transmits sequentially focused beams to reconstruct an image line by line, PWI emits plane wave that insonifies the entire field of view in one shot [35]. The echoes received by the transducer array are then processed to reconstruct the desired image. By eliminating the need for sequential beam steering and focusing, PWI drastically reduces acquisition time, enabling the capture of high-frame-rate ultrasound data [36] .

One of the key advantages of PWI is its ability to capture fast-moving physiological phenomena with minimal motion artifacts [37]. This is particularly important in applications like cardiology, where rapid cardiac cycles demand high temporal resolution to analyze heart valve dynamics or blood flow accurately [38]. In conventional focused imaging, the slow acquisition process may result in blurring or missing critical transient events, while PWI ensures that entire frames are captured in real-time without the need for repeated scans [39]. Furthermore, by eliminating the need for multiple transmit pulses per image frame, PWI reduces patient exposure time and improves clinical workflow efficiency [36].

2.2. Understanding Coherent Plane Wave Compounding

Coherent Plane-Wave Compounding (CPWC) is an advanced imaging technique that significantly enhances the quality of ultrasound images. In traditional PWI, a single

plane wave is emitted, and the resulting echo data is used to reconstruct an image. While this method provides high frame rates, the images can suffer from lower resolution and increased artifacts due to the lack of focusing during transmission. CPWC overcomes these limitations by emitting multiple plane waves at varying angles and combining their individually beamformed datasets into a single high-quality image [2].

For instance, using as few as three to five angles provides a noticeable improvement in resolution while preserving a high frame rate suitable for real-time imaging [36]. This flexibility makes CPWC suitable for various clinical applications, including vascular imaging, musculoskeletal diagnostics, and cardiac assessments, where both high resolution and real-time processing are essential [36][40][41].

The core principle of CPWC is to leverage the angular diversity of plane waves to improve spatial resolution, contrast, and overall image clarity. Each plane wave insonifies the medium from a different direction, capturing information that is unique to that specific angle. By coherently compounding the data from all angles, CPWC enhances the signal-to-noise ratio (SNR) and reduces the presence of speckle noise and other artifacts [36]. This results in sharper and more detailed tissue visualization compared to single-angle PWI. Additionally, CPWC improves contrast resolution by averaging out undesired variations in scattered signals [36]. The compounding process involves aligning the data from different angles to ensure that the echoes from the same point in the medium are constructively summed, leading to a sharper and more detailed representation of the imaged structures [42].

Figure 1 presents a high-level overview of image formation based on CPWC. The process begins with the emission of unfocused plane waves at various steering angles θ, followed by the acquisition of corresponding raw RF data frames $D_\theta(t, x)$, where t denotes time and x is the lateral position along the transducer aperture. For each transmit angle, the received data undergoes an independent beamforming stage, which reconstructs an intermediate image frame $\widetilde{D}_\theta(z, x)$, where z represents the axial imaging depth. This step is performed separately for all steering angles, producing a set of angle-specific beamformed frames.

Once individual beamformed frames have been generated, they are coherently compounded. This compounding step involves pixel-wise summation of the complex-valued analytic representations across all transmit angles, yielding a high-contrast image $\widetilde{\mathrm{D}}_{cc}(z, x)$. Constructive interference among spatially aligned reflectors enhances image resolution and suppresses incoherent noise, a key advantage of

CPWC methods. The compounded image is subsequently passed through a signal post-processing stage. This includes envelope detection to extract the amplitude of the analytic signal, followed by logarithmic compression to adjust the signal dynamic range for visual display. The final result is a B-mode image suitable for clinical or diagnostic interpretation.

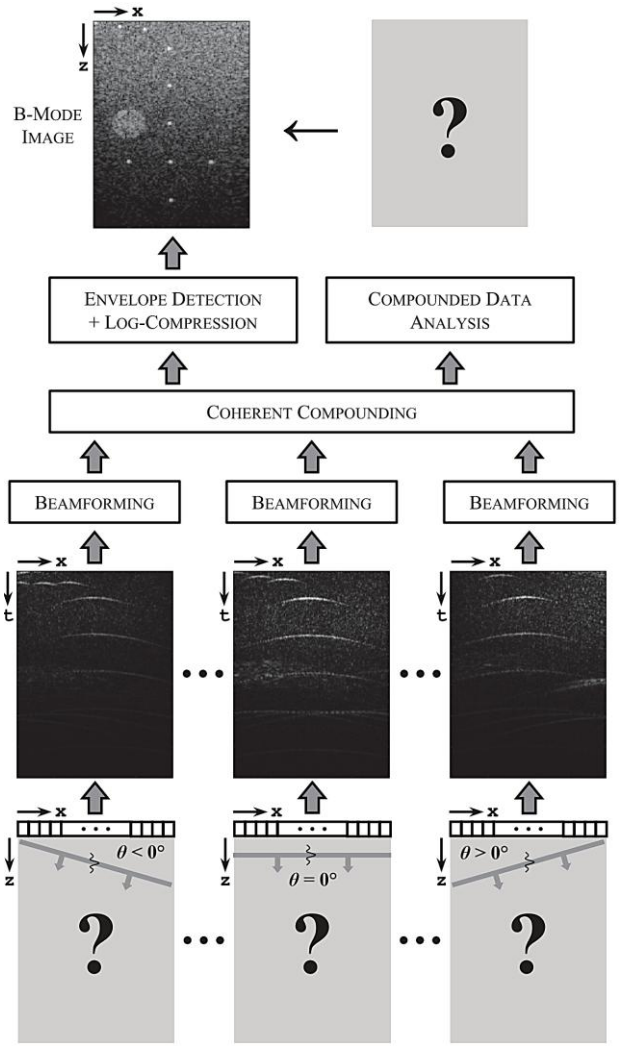

Figure 1 Illustration of CPWC Image Formation [43]

In addition to improving image quality, CPWC is highly compatible with modern computational techniques and hardware platforms, such as FPGAs and GPUs. These systems enable the real-time processing of large volumes of data, making CPWC a practical choice for clinical ultrasound systems [44]. Recent advancements in hardware acceleration, such as AI-core-assisted beamforming and FPGA-based implementations, have further optimized CPWC for high-speed imaging [45]. Overall, CPWC represents a significant advancement in ultrasound imaging, providing a

powerful tool for clinicians and researchers to achieve excellent image quality without sacrificing efficiency [46].

2.3. Fourier-Domain Migration

In seismic and ultrasound imaging, data acquisition typically involves recording wavefields over time (t) at sensor spatial locations (x). This results in data represented in the (t, x) domain. Migration is the process of repositioning recorded reflections to their correct spatial locations, thereby transforming the data into the axial depth (z) and lateral position (x) coordinates, known as the (z, x) domain.

Traditional migration methods operate in the (t, x) domain, applying travel time delays of waves to reposition reflection events. However, these time-domain methods can be computationally intensive. Fourier-domain migration addresses this issue by converting the data into the frequency-wavenumber domain, where wave propagation effects can be handled more efficiently using frequency-domain interpolation techniques [11].

Let $D_\theta(t, x)$ represent the time-domain signal recorded after a plane wave is transmitted at angle θ. The objective is to produce a beamformed image $\widetilde{D}_\theta(z, x)$ where z represents the axial depth. This transformation is carried out using the Temme–Mueller (TM) migration algorithm through the following five-step process:

1. A one-dimensional Fourier transform is first applied along the temporal axis t, converting the raw RF signal into its frequency-domain representation $\Phi_\theta(f, x)$ where f is the temporal frequency.
2. Next, a phase correction is applied to $\Phi_\theta(f, x)$. This is performed by multiplying each frequency component with a complex exponential:

$$\Phi'_\theta(f, x) = \Phi_\theta(f, x).\, exp\left(j\frac{2\pi f x \sin\theta}{c}\right) \qquad (2.1)$$

   where c is the assumed speed of sound in the medium.
3. A second Fourier transform is then taken along the lateral coordinate x, resulting in a spectral representation $\Psi_\theta(f, k_x)$ with k_x denoting the lateral spatial frequency.

4. This is the main remapping step, where one obtains a migrated spectrum $\widetilde{\Psi}_\theta(k_z, k_x)$, by applying the following equation:

$$\widetilde{\Psi}_\theta(k_z, k_x) = \mathrm{A}_\theta(k_z, k_x).\, \Psi_\theta\left(\tilde{f}_\theta(k_z, k_x), k_x\right) \qquad (2.2)$$

where the remapped frequency $\tilde{f}_\theta(k_z, k_x)$ is computed as:

$$\tilde{f}_\theta(k_z, k_x) = \frac{c k_z \left(1 + \left(\frac{k_x}{k_z}\right)^2\right)}{2\left(\cos\theta + \sin\theta \frac{k_x}{k_z}\right)} \qquad (2.3)$$

and the associated scaling factor $\mathrm{A}_\theta(k_z, k_x)$ is given by:

$$\mathrm{A}_\theta(k_z, k_x) = \frac{c\left|\cos\theta\left(1 - \left(\frac{k_x}{k_z}\right)^2\right) + 2\sin\theta \frac{k_x}{k_z}\right|}{2\left(\cos\theta + \sin\theta \frac{k_x}{k_z}\right)^2} \qquad (2.4)$$

5. Finally, the beamformed image $\widetilde{\mathrm{D}}_\theta(z, x)$ is reconstructed via a two-dimensional inverse Fourier transform of $\widetilde{\Psi}_\theta(k_z, k_x)$ with axial distance z defined as $z = \frac{ct}{2}$.

A significant advantage of this approach becomes evident in the CPWC framework. Instead of performing an inverse transform for each angle-specific image, the compounded image $\widetilde{\mathrm{D}}_{cc}(z, x) = \sum_\theta \widetilde{\mathrm{D}}_\theta(z, x)$ can equivalently be obtained by summing the migrated spectra first and applying the inverse 2D Fourier transform only once:

$$\widetilde{\Psi}_{cc}(k_z, k_x) = \sum_\theta \widetilde{\Psi}_\theta(k_z, k_x) \;\Rightarrow\; \widetilde{\mathrm{D}}_{cc}(z, x) = \mathcal{F}_{2D}^{-1}\left[\widetilde{\Psi}_{cc}(k_z, k_x)\right] \qquad (2.5)$$

The flow of the TM algorithm is illustrated in Figure 2. The first computational stage, the Temporal FFT (TFFT), transforms the input RF data from the time domain to the frequency domain. Since the ultrasound signals input sequences are real-valued, we can exploit symmetry properties of the Fourier transform to improve computational efficiency. Specifically, by pairing two real-valued signals into a single complex sequence, a single complex FFT can be used to compute the frequency-domain representations of both sequences simultaneously.

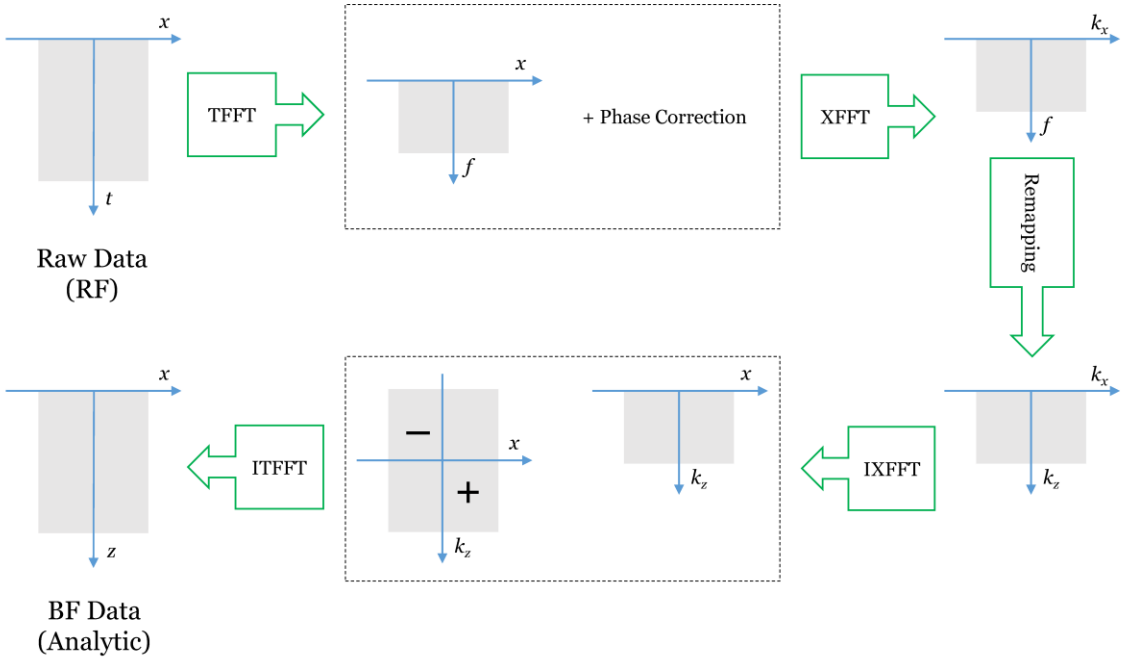

Figure 2 Key Steps in Temme-Mueller (TM) Migration Process [47]

According to the algorithm described in [48], two real-valued sequences of length N can be packed into a single complex sequence and transformed using one complex FFT of size N, effectively reducing the number of required FFT operations by half. Suppose $p[n]$ and $q[n]$ are real-valued input sequences. By combining them into a complex signal $\mathrm{s}[n] = p[n] + jq[n]$, their respective Fourier transforms $P[k]$ and $Q[k]$ can be extracted from the FFT of $s[n]$. Since the FFT is a linear operation, it yields:

$$FFT\{\mathrm{s}[n]\} = S[k] = S_r[k] + jS_i[k] \tag{2.6}$$

where $S_r[k]$ and $S_i[k]$ are the real and imaginary components of the FFT output. The transforms $P[k]$ and $Q[k]$ can be recovered as follows:

$$P_r[k] = \frac{1}{2}(S_r[k] + S_r[N-k]), k = 1,2,\dots,\frac{N}{2} \tag{2.7}$$

$$P_i[k] = \frac{1}{2}(S_i[k] + S_i[N-k]), k = 1,2,\dots,\frac{N}{2} \tag{2.8}$$

$$Q_r[k] = \frac{1}{2}(S_i[N-k] + S_i[k]), k = 1,2,\dots,\frac{N}{2} \tag{2.9}$$

$$Q_i[k] = \frac{1}{2}(S_r[N-k] + S_r[k]), k = 1,2,\dots,\frac{N}{2} \tag{2.10}$$

For $k = 0$, these formulas simplify to:

$$P[0] = S_r[0] \quad , \quad Q[0] = S_i[0] \qquad (2.11)$$

Following the TFFT, the phase correction module applies frequency-dependent phase shifts to the frequency-domain signals $P[k]$ and $Q[k]$, which correspond to neighboring even- and odd-indexed transducer elements, respectively. Here, k represents the frequency bin index. This operation aligns the spectral components of each channel according to the current steering angle θ, producing the phase-adjusted spectrum $\Phi'_\theta(f, x)$, computed across all temporal frequencies f and locations x, as defined by Equation (2.1).

Subsequently, the spatial FFT (XFFT) transforms the phase-corrected data from the (f, x) domain to the (f, k_x) domain. The next step, Remapping, translates the spectral content from the (f, k_x) domain into the (k_z, k_x) domain using a linear interpolation strategy based on Equations (2.2)-(2.4). This step is essential for correctly reconstructing the final image in the (z, x) domain, after the inverse transformations.

Once the remapping stage is completed, the compounding process takes place, where remapped data from multiple imaging angles is summed together in the (k_z, k_x) domain, forming a compounded spectrum that enhances image quality and reduces speckle noise. The compounding operation is governed by Equation (2.5).

The compounded spectrum is then passed through an inverse spatial FFT (IXFFT), effectively reversing the prior XFFT operation and returning the data to the (k_z, x) domain. Finally, the inverse temporal FFT (ITFFT) converts the reshaped spectrum into the spatial (z, x) domain, yielding the reconstructed image.

The flowchart below summarizes all the steps described above.

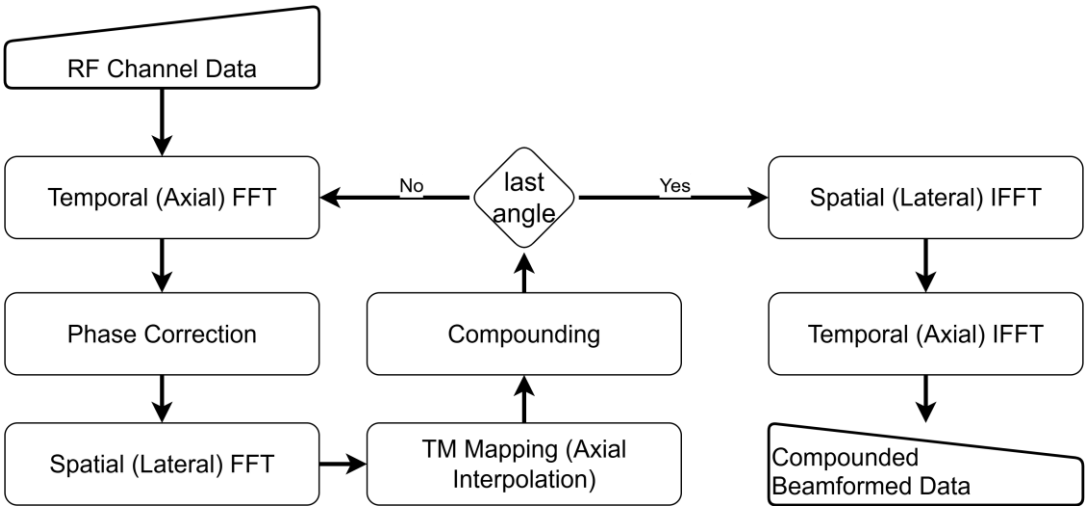

Figure 3 Flowchart of Computational Steps

3. System Architecture

3.1. Top-Level Block Diagram

The proposed beamforming architecture is designed for high-throughput ultrasound imaging based on the Temme–Mueller (TM) migration algorithm. The system adopts a modular and pipelined structure, implemented using high-level synthesis (HLS). Each processing stage operates independently and communicates via streaming interfaces (`hls::stream<T>`) to support continuous data processing.

Each functional block is implemented as a standalone HLS module (e.g., FFT, remapper, accumulator), allowing targeted optimization and easier scaling. A simplified block diagram of the architecture is shown in Figure 4 and detailed in the rest of this chapter. It illustrates the sequential processing chain from RF input to beamformed image output. These modules operate under DATAFLOW pragma to enable concurrent execution. The system is organized as a spatial pipeline where each module consumes and produces data in a streaming fashion with ping-pong buffering. All inter-module communication is handled through AXI streaming interfaces. These streams are implemented using FIFOs with handshake signals (valid, ready, etc.) to make sure that modules can operate at independent speeds without losing data. Each module is optimized for a one-cycle initiation interval (II=1), enabling sustained data movement across the pipeline without idle cycles or buffering bottlenecks.

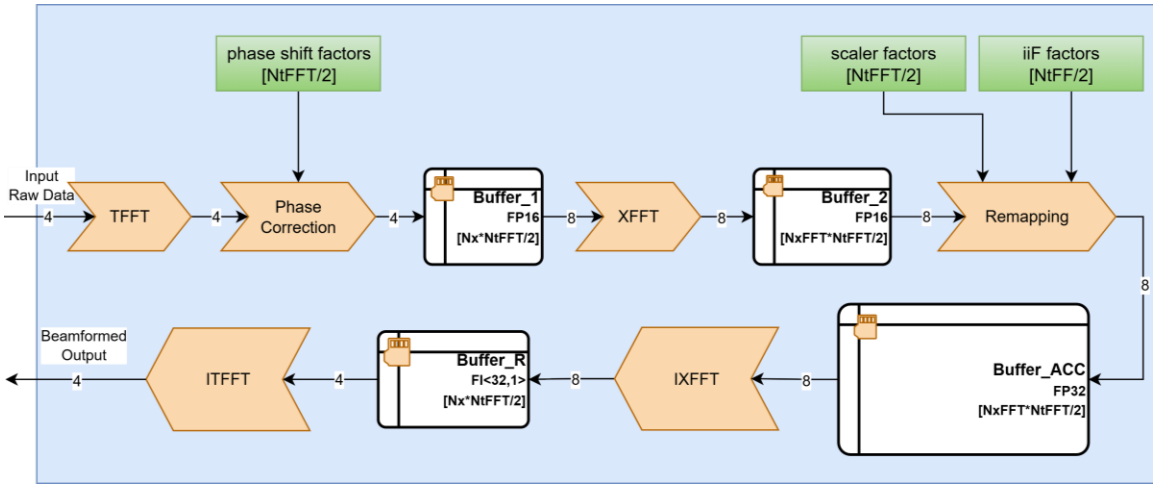

Figure 4 Top-Level Architecture View

The processing flow begins with raw RF channel data from an ultrasound probe. For each pulse emission, every transducer element records a sequence of sampled echo

signals. These are stored as time-domain 2D matrix, where rows represent sampling time instances, and columns represent channels (i.e., array elements). This input data comes in through a streaming interface as a stream of samples in the column-wise order.

The key parameters such as the PW emission angles (`in_Theta_S`), sampling frequency (`Fs`), wave propagation speed (`C`), and probe geometry (`probe_geo[Nx]`) are passed as runtime inputs to the top-level `Beamforming_MIG` function, enabling dynamic adaptation to varying probe configurations without requiring re-synthesis. Externally, the probe pitch (`probe_diff`) is also provided explicitly: it is equivalent to the difference between the first two probe element positions (`probe_geo[0] - probe_geo[1]`). These inputs are internally converted to fixed-point format `ap_fixed<32, 16>` (i.e., signed 32-bit representation with a 16-bit integer part) and used to calculate derived parameters, such as temporal frequency resolution (`dF0`) and lateral wavenumber increments (`dkx0`), which govern the behavior of the phase delay and spectral remapping modules.

```
typedef std::complex<ap_fixed<16, 1>> T_FFTZ_in;
typedef std::complex<ap_fixed<32, 1>> T_IFFTZ_out;
void Beamforming_MIG(hls::stream<float> &in_Theta_S, float probe_geo[Nx],
                     float probe_diff, float C, float Fs,
                     hls::stream<T_FFTZ_in>   HW_rawRF_S[4],
                     hls::stream<T_IFFTZ_out> HW_out_stream[4]};
```

```
float dF0_f = Fs / float(NtFFT);
float dkx0_f = 1.0 / (probe_diff * NxFFT);
ap_fixed<32, 16>  dF0 = static_cast< ap_fixed >(dF0_f);
ap_fixed<32, 16> dkx0 = static_cast< ap_fixed >(dkx0_f);
```

Once the data enters the system, the first stage applies a temporal FFT (TFFT) to convert each time-domain data column into its frequency-domain representation. Following this, a phase correction stage adjusts each signal based on the steering angle of the incoming plane wave. These corrections are implemented as complex rotations. The output of the phase adjustment is then passed to the spatial FFT (XFFT) module, which processes data across the transducer channel axis. After the spatial transformation, the remapping stage modifies the frequency components in frequency-wavenumber space. This step is essential for accurate image formation in Fourier-domain beamforming.

If multiple plane waves are used—as in CPWC—the accumulation stage sums remapped spectral from different angles. This results in a compounded dataset with improved contrast and resolution. Finally, the beamformed data is passed through two inverse FFT stages producing final beamformed data in the image domain. The

result is the analytic form of the beamformed ultrasound image, ready for envelope detection or further post-processing. The final output is streamed out column-by-column through an output stream interface.

Our pipeline uses a ping-pong buffer system and a light-weight control mechanism to ensure that row-by-row and column-by-column data accesses do not collide. At any given moment, one set of memory blocks is consuming incoming data, while the other is supplying data to the next stage. After each frame, the buffers swap their consumer-supplier roles.

3.2. Processing Modules

Each module in the processing chain — from input to final output — operates as a self-contained unit with well-defined streaming interfaces. This modular streaming architecture provides a high degree of flexibility. Individual stages can be debugged, reconfigured, or optimized independently. Performance bottlenecks are easier to identify and address, and the system can scale by adding more parallel lanes or replicating processing blocks. The system's parallel processing capability is structured around a modular architecture, where the parameter `N_Lanes` (set to 4 in the current design) defines the level of concurrency across all processing stages.

These lanes carry complex values continuously from one module to the next, forming the backbone of the streaming pipeline. At the input stage, four parallel lanes accept raw RF data samples, each lane moving one complex value per cycle. Recall that each input complex-valued sample is formed by combining two real-valued RF samples, as described in Section 2.3. As data progresses through subsequent computational stages, such as the XFFT, Remapping, and IXFFT, these four lanes are expanded to eight, as illustrated in Figure 4. Each lane is strongly typed: fixed-point or FP16 formats are used in the early stages to minimize memory footprint and resource cost, while FP32 precision is reserved for the accumulation (i.e., compounding) stage. After compounding and IXFFT, the data flows into the final ITFFT stage via the last buffer, where the number of lanes is reduced back to four for the final streaming output. In this way, the number of lanes, the data type, and the streaming width evolve across the pipeline—always tuned to the computational load of each stage—so that both input and output are handled efficiently without stalls or noticeable losses in accuracy. Further details about the lanes are provided in the sequel.

3.2.1 Temporal FFT (TFFT)

The beamforming process begins with the TFFT module, which serves as the entry point for raw radio-frequency (RF) data and marks the initial transition from time-domain to frequency-domain processing. The input consists of acquired real-valued RF data representing signals from $N_x = 128$ transducer channels. To efficiently utilize the FFT IP core, which expects complex-valued input, a pairing strategy is employed: the data from odd-numbered channels is assigned to the real component, while data from even-numbered channels is assigned to the imaginary component. This technique effectively folds two real channels into a single complex input, as described in Section 2.3. This optimization leverages the symmetry of the FFT when operating on real signals.

Prior to FFT processing, to satisfy the FFT core's input length requirement, the original time-domain sequences of length $N_t = 1536$ to are zero-padded to $N_{tFFT} = 4096$, After padding and pairing, the data is streamed into the FFT core for processing along the temporal axis. The input stream `in_rf_data[x,t]` represents a transposed form of the original acquisition matrix $D_\theta(t, x)$, where time samples and channel indices are rearranged to align with the dataflow order expected by the processing pipeline.

```
// Even-odd channel folding and zero-padding before FFT
typedef ap_fixed<16,1>                          FFTZ_in_t;
typedef std::complex<FFTZ_in_t>                 T_FFTZ_in;
typedef std::complex<ap_fixed<32, 17>>          T_FFTZ_out;

for (int x = 0; x < Nx; x += 2) {
    for (int t = 0; t < NtFFT; t++) {
        #pragma HLS PIPELINE II=1
        FFTZ_in_t real_val = (t < Nt) ? in_rf_data[x][t]     : 0;
        FFTZ_in_t imag_val = (t < Nt) ? in_rf_data[x + 1][t]: 0;
        T_FFTZ_in fft_input = T_FFTZ_in {real_val, imag_val};
        fft_in_stream.write(fft_input);
    }
}
```

The TFFT block is implemented using the HLS-provided `hls::fft` IP core configured for complex input and output with a fixed transform length of N_{tFFT} = 4096 points per channel, fully pipelined with an initiation interval (II) of 1, allowing new input samples to be fed into the transform every clock cycle after the initial pipeline latency. The FFT inputs and outputs are defined using fixed-point complex data types to balance precision and hardware efficiency. A representative invocation of the core is shown below:

```
hls::fft<config_FFTZ>(fft_input_stream, fft_output_stream, &status);
```

The `config_FFTZ` structure specifies the transform parameters, including transform length (4096), direction (`FFT_FWD`), and scaling behavior.

To sustain real-time throughput and preserve streaming compatibility, four identical FFT cores are instantiated in parallel. Each core handles one complex-valued row per cycle, corresponding to a pair of original RF channels. With 128 total channels packed into 64 complex rows, the system completes the temporal FFT stage over 16 scheduling iterations, processing four (`N_Lanes`) rows in parallel per iteration. This architectural decision significantly reduces processing latency and enables the TFFT stage to sustain a high input data rate. All FFT cores are configured with the same parameters but operate on independent input lanes. The resulting spectral data is streamed directly to the phase compensation module via AXI-compliant `hls::stream<T_FFTZ_out>` interface.

After the FFT stage, the complex frequency-domain outputs are split into their real and imaginary components and temporarily stored in separate buffers `FFT_Buffer_Re[]` and `FFT_Buffer_Im[]`. Once a complete FFT frame of N_{tFFT} points are collected, the subsequent step reconstructs positive-frequency spectra of the original even- and odd-numbered real-valued channel data that were previously folded into complex input pairs. This reconstruction follows the formulation described in Section 2.3, where each positive frequency bin t is combined with its mirrored counterpart at $N_{tFFT} - t$ to generate two complex-valued spectral lines corresponding to the analytic signals of the original even- and odd-numbered channels. Note that during this process the system relies on alternating buffer banks selected by a `ping_pong` flag, which toggles between two memory sets to support continuous frame processing without overwriting in-flight data. The resulting streams are written to `analytic_data_stream[*]` and forwarded to the phase compensation stage.

The relevant section of the HLS code is shown below:

```
#define  data_t        ap_fixed<32, 17>
bool ping_pong = false;
data_t vRe_0, vRe_1, vIm_0, vIm_1;
for (int x = 0; x < Nx ; x += 2) {
    for (int t = 0; t < NtFFT/2; t++) {
        if (t == 0) {
            vRe_0 = 2 * FFT_Buffer_Re[ping_pong][0];
            vIm_0 = 0;
            vRe_1 = 2 * FFT_Buffer_Im[ping_pong][0];
            vIm_1 = 0;
        } else {
            data_t tmp_bri_Re = FFT_Buffer_Re[ping_pong][t];
            data_t tmp_briPos_Re = FFT_Buffer_Re[ping_pong][NtFFT - t];
            data_t tmp_bri_Im = FFT_Buffer_Im[ping_pong][t];
            data_t tmp_briPos_Im = FFT_Buffer_Im[ping_pong][NtFFT - t];

            data_t vRe_0 = tmp_bri_Re + tmp_briPos_Re;
            data_t vIm_0 = tmp_bri_Im - tmp_briPos_Im;
            data_t vRe_1 = tmp_briPos_Im + tmp_bri_Im;
            data_t vIm_1 = tmp_briPos_Re - tmp_bri_Re;
        }
        analytic_data_stream[0].write( T_FFTZ_out {vRe_0 , vIm_0} );
        analytic_data_stream[1].write( T_FFTZ_out {vRe_1 , vIm_1} );
    }
    ping_pong = !ping_pong;
}
```

3.2.2 Phase Compensation

Figure 5 shows the block diagram of the `PhaseDelay()` unit, which applies frequency-dependent phase compensation to the TFFT stage output according to Equation (2.1).

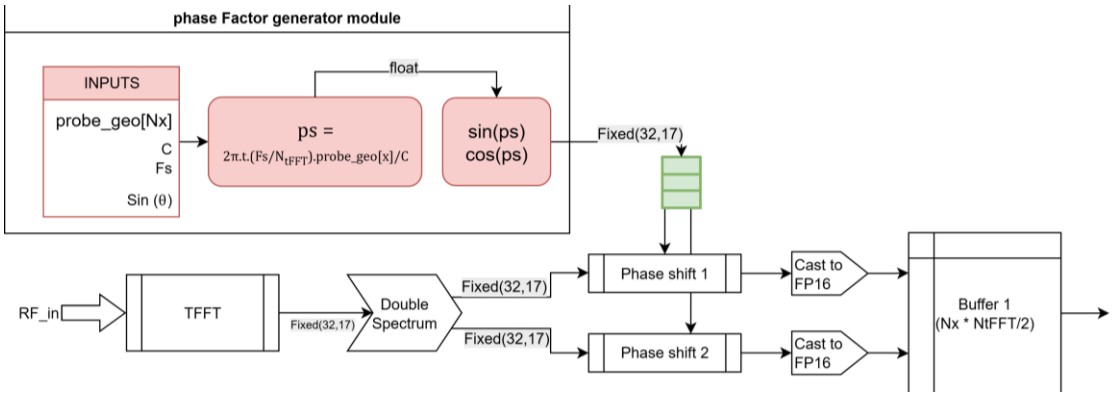

Figure 5 Phase Delay Module and Its Associated Phase Shift Factor Generator

Phase compensation starts with the on-the-fly generation of phase factor for the current steering angle. This is handled by the `phasedelay_factors()` module, which receives three inputs: the normalized temporal frequency step size (`step_size`) calculated as $\frac{\pi}{0.5*C}(\frac{F_s}{N_{tFFT}})$, a stream of sine values for the steering angle (`in_sinTheta_stream`), and the lateral coordinates of the probe elements (`probe_geo[Nx]`). Internally, the function calls `produce_ps()` to compute the phase offset values for each probe element based on these parameters. These values

are then passed to `compute_sincos()`, which calculates their corresponding sine and cosine components. The resulting values are streamed out via `out_sincos_stream[]` to the corresponding `PhaseDelay()` unit for required phase shift (`PS`) correction.

```
void phasedelay_factors(ap_fixed<32, 6> step_size, float probe_geo[Nx],
                        hls::stream<float> &in_sinTheta_stream,
                        hls::stream<sin_cos_t> out_sincos_stream[8]) {
  hls::stream<float> ps_stream[8];
  for (int f = 0; f < FRAMES; ++f) {
// FRAMES is the number of angle-specific RF frames being processed
    float sinTheta = in_sinTheta_stream.read();
    produce_ps(step_size, sinTheta, probe_geo, ps_stream);
    compute_sincos(ps_stream, out_sincos_stream);
  }
}
```

The sine and cosine computations in `compute_sincos()` are performed in floating-point arithmetic, which is fully synthesizable in Vitis™ HLS. This enables high-accuracy evaluation of trigonometric functions without relying on lookup tables or external preprocessing. Once calculated, these values are immediately cast to a fixed-point format (`ap_fixed<32,17>`) ensuring compatibility with the fixed-point arithmetic used in the upstream TFFT processing chain.

```
// Inputs:
//   sinTheta: beam steering parameter
//   probe_geo[x]: lateral position of element x
//   step_size: normalized temporal frequency step = 2π(Fs/NtFFT)/C.
//   NtFFT: number of temporal FFT bins
//   Nx: number of lateral positions per frame
// Output:
//   Stream of ps values per (x, t)
void produce_ps(ap_fixed<32,6> step_size, float sinTheta, float probe_geo[Nx],
                hls::stream<float> ps_stream) {
    for (int x = 0; x < Nx; ++x) {
#pragma HLS UNROLL factor=8
        for (int t = 0; t < NtFFT / 2; ++t) {
            float ps = t * step_size * sin_theta * probe_geo[x];
            ps_stream[x % 8].write(ps);  // shift amount for (x, t)
        }
    }
}
```

```
struct SinCos_t {
  T_FFTZ_out sinPS;
  T_FFTZ_out cosPS;
};
// Inputs:             Stream of ps values per (x, t)
// Output:             Stream of SinCos values per (x, t)
void compute_sincos(hls::stream<float> ps_stream[8],
                    hls::stream<sin_cos_t> out_sincos_stream[8]){
    for (int x = 0; x < Nx; ++x) {
#pragma HLS UNROLL factor=8
        for (int t = 0; t < NtFFT / 2; ++t) {
            float ps = ps_stream[x % 8].read();
            SinCos_t factor;
            factor.cosPS = static_cast< T_FFTZ_out >( cosf(ps) );
            factor.sinPS = static_cast< T_FFTZ_out >( sinf(ps) );
            sincos_out_stream[x % 8].write(factor); //phase correction factor output
        }
    }
}
```

The computed sine–cosine pairs are streamed directly into the `PhaseDelay()` unit along with the pre-processed analytic spectra. It processes eight complex samples per clock cycle, leveraging four parallel lanes that are fully unrolled for simultaneous execution.

For each frequency index, the unit reads the corresponding sine and cosine values—one pair for the even channel and one for the odd—and applies the complex rotation as follows:

$$v'_{Re} = v_{Re}\cos PS - v_{Im}\sin PS \quad , \quad v'_{Im} = v_{Re}\sin PS + v_{Im}\cos PS \qquad (3.1)$$

Following the complex rotation, the output values are cast to float to align with the data format expected by the subsequent buffering stage. The following code demonstrates a portion of the phase correction process.

```
struct cmpx2xD_t {
  std::complex<float> X1;
  std::complex<float> X2;
};
for (int x = 0; x < Nx; x += (2*N_Lanes)) {
  for (int t = 0; t < NtFFT/2; ++t) {
    for (int L = 0; L < N_Lanes; ++L) {
#pragma HLS UNROLL
       // Read complex input data (analytic signal from FFT)
       T_FFTZ_out in_0 = analytic_data_stream[2 * L].read();  //even: vRe_0, vIm_0
       T_FFTZ_out in_1 = analytic_data_stream[2 * L + 1].read();//odd: vRe_1, vIm_1
       // Read corresponding phase shift (cos/sin) values
       SinCos_t phase_0 = sincos_stream[2 * L].read();      // cosPS0, sinPS0
       SinCos_t phase_1 = sincos_stream[2 * L + 1].read(); // cosPS1, sinPS1
       // Apply complex rotation (frequency-dependent phase shift)
       T_FFTZ_out out_0, out_1;
       out_0.real = in_0.real * phase_0.cos - in_0.imag * phase_0.sin;
       out_0.imag = in_0.real * phase_0.sin + in_0.imag * phase_0.cos;

       out_1.real = in_1.real * phase_1.cos - in_1.imag * phase_1.sin;
       out_1.imag = in_1.real * phase_1.sin + in_1.imag * phase_1.cos;

       cmpx2xD_t out_data; // Forward corrected data
       out_data.X1 = static_cast< float >( out_0 );
       out_data.X2 = static_cast< float >( out_1 );
       PD_out_stream[L].write (out_data);
    }
  }
}
```

3.2.3 Intermediate Buffering: Buffer 1 Module

The primary role of `buffer_1` is to store the complex-valued data received from the phase correction stage in preparation for the spatial FFT (XFFT) processing. It captures four parallel streams of `cmpx2xD_t`, separates their real and imaginary components, and puts them into two URAM-backed 2D arrays:

```
half Buffer1_R[Nx][NtFFT/2];
half Buffer1_I[Nx][NtFFT/2];
```

These arrays store the real and imaginary parts of the complex samples received. The type `half` corresponds to IEEE 754 half-precision binary floating-point format (binary16). The samples are indexed by lateral position (x) and temporal frequency index (t).

The following code snippet illustrates how `Buffer1_R` and `Buffer1_I` are filled:

```
for (int x = 0; x < Nx; x+= (2*N_Lanes) ) {
    for (int t = 0; t < NtFFT_half; ++t) {
        for (int L = 0; L < N_Lanes; ++L) {
#pragma HLS UNROLL
        cmpx2xD_t temp;
        in_stream[L].read(temp);

        Buffer1_R[2 * L][t] = (half)temp.X1.real();
        Buffer1_I[2 * L][t] = (half)temp.X1.imag();

        Buffer1_R[2 * L + 1][t] = (half)temp.X2.real();
        Buffer1_I[2 * L + 1][t] = (half)temp.X2.imag();
```

The stored data is then streamed row-by-row to the XFFT module. During this stage, zero-padding is applied to extend the lateral dimension from N_x = 128 to N_{xFFT} = 256. To preserve throughput despite the increased output width, the design doubles the number of output lanes, effectively instantiating twice as many XFFT engines as were used on the input side.

The following snippet shows how zero-padding is applied during this readout phase

```
for (int t = 0; t < NtFFT/2; t += (2*N_Lanes) ) {
    for (int x = 0; x < NxFFT; x++) {
        for (int L = 0; L < (2*N_Lanes); L ++) {
#pragma HLS UNROLL factor=8
            T_FFTX_in out_temp;
            if(x < Nx)
                out_temp = T_FFTX_in (Buffer1_R[x][t + L], Buffer1_I[x][t + L])
            else
                out_temp = T_FFTX_in (0, 0);
            out_stream[L].write(out_temp);
        }
    }
}
```

To enable high-throughput memory access, both `Buffer1_R` and `Buffer1_I` arrays are cyclically partitioned along both dimensions using HLS pragmas:

```
#pragma HLS ARRAY_PARTITION cyclic variable=Buffer1_R, Buffer1_I dim=1 factor=8
#pragma HLS ARRAY_PARTITION cyclic variable=Buffer1_R, Buffer1_I dim=2 factor=4
```

This partitioning strategy allows multiple rows (dim=1) and columns (dim=2) to be accessed in parallel without causing memory access contention. The lateral (x) dimension is partitioned by a factor of 8 to match the number of output lanes, ensuring that eight spatial positions can be accessed simultaneously. The temporal (t) dimension is partitioned by a factor of 4, allowing multiple frequency bins to be served in parallel. Together, these optimizations are critical for sustaining the required throughput during both buffer read and write operations across the fully unrolled datapath. All loops are pipelined with

```
#pragma HLS PIPELINE II=1
```

enabling full throughput once the pipeline is filled. Parallelization is applied across lanes with

```
#pragma HLS UNROLL factor=8
```

which increases resource utilization of the hardware fabric but allows simultaneous writing and reading across 8 output lanes. Despite these parallelization directives, `buffer_1` slightly disrupts the streaming nature of the pipeline. The observed

processing interval is approximately 68002 clock cycles instead of the ideal 65536, introducing a small pause between consecutive frames.

3.2.4 Spatial FFT (XFFT)

The spatial FFT operates along the channel axis, i.e., it computes a one-dimensional FFT across adjacent probe elements for each temporal frequency bin. The input to this module is a matrix of complex frequency-domain samples indexed by $[x][t]$, where x is the channel number, and t is the temporal frequency bin. For each fixed t, the FFT is applied across x. To sustain high throughput, input data is fed through a stream interface and stored in local buffers until a complete row is filled with data for a given frequency. The FFT is then executed in a burst, and the output is streamed forward immediately.

```
// For each frame and temporal frequency bin, perform lateral FFT across channels
for (int t = 0; t < NtFFT / 2; ++t) {
    Complex buffer[Nx];  // temporary buffer for spatial FFT

    // Collect complex inputs across lateral dimension
    for (int x = 0; x < Nx; ++x) {
        buffer[x] = phase_corrected_stream[x].read();
    }

    // Perform FFT across lateral axis
    fft_core(buffer);

    // Stream out transformed data
    for (int kx = 0; kx < NxFFT; ++kx) {
        spatial_fft_stream[kx].write(buffer[kx]);
    }
}
```

The XFFT is implemented using the HLS `hls::fft` core, configured to operate in the forward direction, Input and output data are formatted as fixed-point complex numbers:

```
typedef ap_fixed<32, 1> FFTX_in_t;
typedef std::complex<FFTX_in_t> T_FFTX_in;
typedef ap_fixed<32, 1> FFTX_out_t;
typedef std::complex<FFTX_out_t> T_FFTX_out;
```

To maintain dataflow continuity, the architecture instantiates 8 parallel FFT cores. This parallel structure allows the entire frame to be processed in a fully pipelined manner across 256 iterations, with each iteration feeding 8 FFT rows in parallel, totalling 2048 rows. The cores operate with an initiation interval (II) of 1, meaning new rows can be introduced into each core every clock cycle. The output stream from each FFT core is forwarded via synchronized `hls::stream<T_FFTX_out>` interfaces into the remapping pipeline.

3.2.5 Intermediate Buffering: Buffer 2 Module

After the spatial FFT, the complex-valued output data is streamed in 8 parallel lanes as `T_FFTX_out` objects. The role of `buffer_2` is to receive these XFFT outputs and store them in a dual-banked ping-pong memory structure, normalize and reshape the data into a fixed-point format for interpolation and spectral remapping, and stream the resulting values into the remapping module through synchronized parallel lanes.

Internally, `buffer_2` employs two static URAM-backed 2D arrays to store the real and imaginary components of the XFFT output:

```
half Buffer2_R[NxFFT][NtFFT/2];
half Buffer2_I[NxFFT][NtFFT/2];
```

Each array stores the real and imaginary part of the FFT output, respectively. They are indexed along spatial frequency (`NxFFT`) and temporal frequency (`NtFFT/2`) and are partitioned cyclically for parallel access across the output lanes, to facilitate high-throughput operation and avoid memory contention during simultaneous read and write access.

```
#pragma HLS ARRAY_PARTITION cyclic variable=Buffer2_R, Buffer2_I dim=1 factor=8
#pragma HLS ARRAY_PARTITION cyclic variable=Buffer2_R, Buffer2_I dim=2 factor=8
```

The arrays are also bound to URAM with dual-port URAM (`ram_t2p`) to support concurrent read and write operations.

```
#pragma HLS BIND_STORAGE variable = Buffer2_R, Buffer2_I type = ram_t2p impl = uram
```

This function receives data from the XFFT output stream and performs casting to 16-bit floating-point format:

```
for (int t = 0; t < NtFFT/2; t += (2*N_Lanes) ) {
  for (int x = 0; x < NxFFT; x++) {
    for (int L = 0; L < (2*N_Lanes); L++) {
#pragma HLS UNROLL
      T_FFTX_out in_data;
      in_stream[L].read(in_data);
      float tmp_r = static_cast< float >(in_data.real());
      float tmp_i = static_cast< float >(in_data.imag());
      Buffer2_R[x][t + L] = static_cast< half >(tmp_r);
      Buffer2_I[x][t + L] = static_cast< half >(tmp_i);
    }
  }
}
```

Once data is fully ingested, it is streamed out over 8 output lanes to the remapping stage:

```
for (int x = 0; x < NxFFT; x += (2*N_Lanes) ) {
  for (int t = 0; t < NtFFT/2; t++) {
    for (int L = 0; L < (2*N_Lanes); L++) {
#pragma HLS UNROLL
      ap_fixed<32, 16> real_part = static_cast< ap_fixed >(Buffer2_R[x + L][t]);
      ap_fixed<32, 16> imag_part = static_cast< ap_fixed >(Buffer2_I[x + L][t]);
      std::complex<ap_fixed<32, 16>> temp(real_part, imag_part);
      out_stream[L].write(temp);
    }
  }
}
```

The `buffer_2` module is designed for high-throughput pipelined operation and achieves an exact processing interval of 65,536 cycles, which aligns perfectly with the theoretical ideal for continuous streaming (`NtFFT × NxFFT / 8`). Consequently, this module does not introduce any delay or pause between frames, enabling seamless data propagation through the pipeline with II = 1.

3.2.6 Spectral Remapping Engine

The input to the remapping block consists of complex-valued frequency-domain signals that have already undergone temporal FFT, phase compensation, and spatial FFT. There are two key units: the map generator (`map_factors_gen`) and the interpolation unit (`mapping_Kz`). The map generator calculates the target temporal frequency indices `iFnew` and scaling coefficients `scaler`, based on equations (2.3) and (2.4), taking into account the runtime inputs, such as the PW angle θ, the sampling frequency (`Fs`), the speed of sound (`C`), and the spatial frequency step size (`dKx0`), and the probe geometry (`probe_geo[Nx]`). The relevant code is shown below.

```
typedef ap_fixed<24, 8>     sin_t;
typedef ap_fixed<24, 14>    Kx_t;
typedef ap_fixed<36, 16>    Kz_t;
typedef ap_fixed<22, 12>    KxKzR_t;
typedef ap_fixed<30, 17>    KxKzR2_t;
typedef ap_fixed<24, 3>     DenomInv_t;
typedef ap_fixed<32, 17>    KxKzR_ar_t;
typedef ap_fixed<36, 24>    ScalerBase_t;
#define D_t                 ap_fixed<40, 24>
#define CD_t                std::complex<D_t>
#define Data_t              ap_fixed<32, 16>
#define CData_t             std::complex<Data_t>
void map_factors_gen(float Thetha,                          // steering angle input
                     ap_fixed<16, 12> C, ap_fixed<24, 20> Fs, ap_fixed<28, 14> dKx0
                     hls::stream<Data_t> out_scaler_stream[2*N_Lanes],
                     hls::stream<Data_t> out_iiF_stream[2*N_Lanes]) {
sin_t sinTheta = hls::sinf(Thetha);
sin_t cosTheta = hls::cosf(Thetha);

for (int x = 0; x < NxFFT; x ++) {
  for (ap_uint<12> ind_t = 0; ind_t < NtFFT_half; ind_t++) {
    int L = x % (2*N_Lanes);
    Kz_t Kz = (ind_t != 0)  (ind_t * NtFFT / Fs) : (C *(Fs/NtFFT)/2);

    Kx_t Kx = (x-NxFFT/2) * dKx0;
    KxKzR_t KxKzR = Kx / Kz;

    KxKzR2_t KxKzR2 = KxKzR * KxKzR;
    D_t numer = Kz_val * (1 + KxKzR2);
    sin_t denom = cosTheta * 2 + KxKzR * (sinTheta * 2);

    D_t iFnew = C * numer / denom / (NtFFT / Fs);
    D_t LUT_iiF = (iFnew < 0 ) ? 0 :
                  (iFnew > (NtFFT/2-1) ? (NtFFT/2-1) : iFnew);
    out_iiF_stream[L].write(LUT_iiF);

    KxKzR2_t KxKzR_ar = cosTheta * (1 - KxKzR2) + (sinTheta * 2) * KxKzR;
    KxKzR2_t scaler = C * 2 * hls::abs(KxKzR_ar) / (denom)^2;

    Kx_t threshold1 = (Kx * cosTheta) / (sinTheta - 1.0);
    Kx_t threshold2 = (Kx * cosTheta) / (sinTheta + 1.0);
    bool is_left = x < NxFFT_half;
    bool skip = (x < NxFFT_half && Kz_val <= threshold1) ||
                (x >= NxFFT_half && Kz_val <= threshold2);
    ScalerBase_t LUT_scaler = skip ? 0 : scaler;

    out_scaler_stream[L].write(LUT_scaler);

  }
}
```

The map generator outputs are streamed directly into the interpolation unit, where the target frequency index is split into its integer and fractional parts to allow linear interpolation between adjacent bins. For each remapping step, the interpolation unit uses the integer part of the remapped index to read adjacent bins from the buffer and computes the interpolated result using the fractional part, as shown in the code snippet below. It then multiplies the result by the scale factor and forwards the output to the next stage.

```
for (int t = 0; t < NtFFT/2; ++t) {
    auto input_data = input_stream.read();
    auto scale_factors = scale_factor_stream.read();
    auto iiF_factors = iiF_factor_stream.read();
    int idx = floor(iiF_factors);            // remapped FFT index
    ap_fixed frac = iiF_factors - (idx);     // fractional offset
    ap_fixed scale = scale_factors;
    Complex v0 = buffer[idx];
    Complex v1 = buffer[idx + 1];
    Complex interp;
    interp.real = scale * (v0.real + frac * (v1.real - v0.real));
    interp.imag = scale * (v0.imag + frac * (v1.imag - v0.imag));
    output_stream.write(interp);
}
```

To support the system's 8-lane parallel architecture, eight instances of this interpolation module are instantiated — one per lane — allowing all remapping operations to proceed concurrently and in sync with the incoming data streams. Both the mapping and interpolation units operate at initiation interval 1, and data flows continuously from input to output, with remapping operations dynamically adapting to each steering angle. This remapping strategy avoids large LUTs.

3.2.7 Accumulation and Multi-Angle Compounding

The next step involves summing frequency-domain data from multiple plane wave transmissions to implement CPWC. This functionality is implemented in the `buffer_acc()` module, which is accepts input from 8 parallel lanes, accumulates data over `FRAMES` transmission angles, and then outputs the compounded result to the IXFFT module.

To manage accumulation efficiently and maintain pipeline continuity, the module is backed by a pair of URAM-based 2D arrays `BufferAcc_R` (real part) and `BufferAcc_I` (imaginary part). They are large enough to handle the entire 2D spectrum and are implemented to sustain parallel lane access with II=1.

```
float BufferAcc_R[NxFFT][NtFFT/2];
float BufferAcc_I[NxFFT][NtFFT/2];
```

The accumulation is performed by reading the current spectral contribution from each lane's input stream, retrieving the existing partial sum from the corresponding URAM buffer, adding the two values together, and writing the result back to the same memory location. This operation is fully pipelined and supports concurrent access across all lanes. The relevant segment of the HLS implementation is given below:

```
for (int f = 0; f < FRAMES; f++) {
  for (int x = 0; x < NxFFT; x ++) {
#pragma HLS UNROLL factor = 8
    for (int t = 0; t < NtFFT/2; ++t) {
      int L = x % (2 * N_Lanes);
      in_stream[L].read(in_data);
      BufferAcc_R[x][t] += static_cast<float>(in_data.real());
      BufferAcc_I[x][t] += static_cast<float>(in_data.imag());
    }
  }
```

To support angle-wise accumulation control, the module maintains an internal frame counter. The buffer is reinitialized at the start of a new frame, and once all `FRAMES` have been accumulated, a final flush cycle is triggered, which streams out the compounded result to the inverse spatial FFT stage. Normalization and output formatting are performed prior to transmission. The accumulated real and imaginary components are scaled and converted to `ap_fixed<32,1>` and then streamed out row-wise across 8 parallel lanes. The relevant segment of the HLS implementation is given below:

```
typedef ap_fixed<32, 1>              IXFFT_in_t;
typedef std::complex<IXFFT_in_t>     T_IXFFT_in;
for (int t = 0; t < NtFFT/2; t += (2*N_Lanes) ) {
  for (int x = 0; x < NxFFT; x++) {
#pragma HLS PIPELINE II = 1
    for (int L = 0; L < (2*N_Lanes); ++L) {
#pragma HLS UNROLL
      float val_r = BufferAcc_R[x][t + L];
      float val_i = BufferAcc_I[x][t + L];
      IXFFT_in_t tmp_r = static_cast<IXFFT_in_t>(val_r);
      IXFFT_in_t tmp_i = static_cast<IXFFT_in_t>(val_i);
      T_IFFTX_in out_data_temp = T_IFFTX_in(tmp_r, tmp_i);
      out_stream[L].write(out_data_temp);
    }
  }
}
```

To manage inter-frame synchronization, the accumulation module includes a controlled pause phase. After emitting the compounded result, the system waits for a single frame period before accepting new input. This pause allows `buffer_acc()` to finalize output operations and reset internal counters.

3.2.8 Image Domain Recovery

The final stage of the beamforming pipeline reconstructs the ultrasound image in the spatial domain (x, z) by transforming the compounded (k_x, k_z) domain data. This transformation is accomplished through two sequential inverse Fast Fourier Transforms: The Inverse Spatial FFT (IXFFT), converting k_x to lateral position x, followed by the Inverse Temporal FFT (ITFFT), which converts k_z to axial depth z.

To comply with the FFT IP core constraints in Vivado™, both inverse FFT stages operate on fixed-point inputs formatted as `ap_fixed<32,1>`. The preceding accumulation module uses 32-bit floating-point (FP32) arithmetic. Therefore, prior to entering the IXFFT module, the data is converted from FP32 to `ap_fixed<32,1>`.

The IXFFT module consists of eight parallel cores, each processing a single k_z bin across all lateral positions. These cores are fully pipelined and operate on a fixed length of N_{xFFT} complex points per stream. The result of this stage is a reconstructed image representation in the (x, k_z) domain. To prepare this data for the ITFFT module, the system employs `buffer_R` that captures the full (x, k_z) spectral frame into dual-port URAM arrays defined below:

```
ap_fixed<32, 1> BufferR_Re[NxFFT][NtFFT/2];
ap_fixed<32, 1> BufferR_Im[NxFFT][NtFFT/2];
```

The buffer module performs two key procedures:

- `bufferR_write_data()` reorganizes the buffered data and transmits it to the ITFFT stage via `T_ITFFT_in` streams. Initially, it traverses each spatial lane and outputs values from the corresponding k_z bins that are expanded to the full expected length (`NtFFT`) via zero-padding beyond the `NtFFT/2` boundary.

```
typedef ap_fixed<32, 1>              ITFFT_in_t;
typedef std::complex<ITFFT_in_t>     T_ITFFT_in;
typedef ap_fixed<32, 1>              ITFFT_out_t;
typedef std::complex<ITFFT_out_t>    T_ITFFT_out;
for (int x = 0; x < Nx; x += 4) {
  for (int t = 0; t < NtFFT; t++) {
    for (int L = 0; L < 4; L++) {
#pragma HLS UNROLL
      out_temp = (t < NtFFT_half)
                  ? T_ITFFT_in(BufferR_R[x + L][t], BufferR_I[x + L][t])
                  : T_ITFFT_in(0, 0);
      out_stream[L].write(out_temp);
    }
  }
}
```

- `bufferR_read_data()` reads incoming streams (`T_IXFFT_out`) from 8 parallel lanes. Each stream provides a complex output from the IXFFT stage, which is decomposed into real and imaginary parts and stored in `BufferR_R` and `BufferR_I`, respectively. At $(t = 0)$, the IXFFT output is divided by 2 to correctly scale the DC component during the reconstruction of the analytic signal from a one-sided spectrum.

```
for (int x = 0; x < NxFFT; x++) {
T_IFFTX_out in_data;
    in_stream[0].read(in_data);
    ap_fixed<32, 1> tmp_r = (in_data.real())>>1;
    ap_fixed<32, 1> tmp_i = (in_data.imag())>>1;
    if (x < Nx) {
      Buffer_R[x][0] = (tmp_r);
      Buffer_I[x][0] = (tmp_i);
    }
  for (int l = 1; l < (2*N_Lanes); l++) {
#pragma HLS UNROLL
    in_stream[l].read(in_data);
    ap_fixed<32, 1> tmp_r = (in_data.real());
    ap_fixed<32, 1> tmp_i = (in_data.imag());
    if (x < Nx) {
      Buffer_R[x][l] = (tmp_r);
      Buffer_I[x][l] = (tmp_i);
    }
  }
}
for (int t = (2*N_Lanes); t < NtFFT_half; t += (2*N_Lanes)) {
    for (int x = 0; x < NxFFT; x++) {
      for (int l = 0; l < (2*N_Lanes); l++) {
#pragma HLS UNROLL
        T_IFFTX_out in_data;
        in_stream[l].read(in_data);
         ap_fixed<32, 1> tmp_r = (in_data.real());
         ap_fixed<32, 1> tmp_i = (in_data.imag());
        if (x < Nx) {
          Buffer_R[x][t + l] = (tmp_r);
          Buffer_I[x][t + l] = (tmp_i);
        }
      }
    }
}
```

The ITFFT stage performs the final frequency-to-depth conversion. It uses four pipelined cores operating on N_{tFFT} -element vectors. Each core transforms a given data vector from the spectral (k_z) domain into the depth (z) domain.

Each output sample corresponds to a pixel in the (x, z) image plane and contains a complex (analytic) value capturing both magnitude and phase information. At this point, the data is converted back from `ap_fixed<32,1>` to floating-point format and written to an output file for visualization, envelope detection, and further MATLAB-based post-processing.

3.3. Frame Scheduling

The beamforming pipeline implements a five-angle CPWC, in which each transmit angle contributes one frame of RF data to the final compounded image. During each compounding cycle, the system processes a predefined number of frames, one per transmit angle, and aggregates their spectral contributions into a final image representation.

At the start of each frame's execution, the system reads the transmit steering angle from `in_Theta_Stream` and computes its sine and cosine values. These values are streamed through `in_sinTheta_stream` and `in_SinCos_stream`, respectively, and are subsequently consumed by multiple downstream modules, including `phasedelay_factors()` and `mapping_Kz()`, to dynamically compute angle-dependent transformations in the frequency domain. This allows the system to adaptively adjust delay and remapping calculations for each transmit angle.

The `buffer_acc` module performs accumulation of migrated Fourier-domain data across all angles. For each incoming frame, it retrieves the existing partial sum from URAM, adds the new contribution from the remapping module output, and writes the updated result back into memory. This process is repeated for every angle until all frames in the compounding set have been processed. Once accumulation across all five frames is complete, the pipeline transitions to the inverse FFT modules — IXFFT and ITFFT — are launched in sequence to convert the data from spectral (k_x, k_z) representation into the spatial domain (x, z). The overall frame-level processing flow is illustrated in Figure 6.

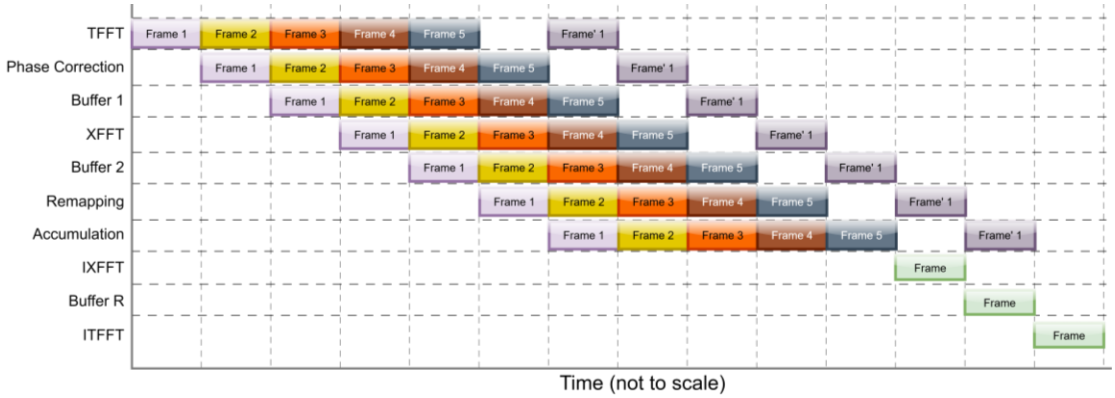

Figure 6 Frame Scheduling Diagram

3.4. Performance and Flexibility

3.4.1 Performance: High Frame Rate Processing

Throughput was prioritized to achieve high frame rate processing with minimal pipeline stalls. The aim was to reduce the initiation interval (II) and latency of each pipeline stage, as well as to enable inter-stage synchronization within the dataflow structure. The system is designed to process frames composed of 2048 × 256 samples,

distributed across 8 parallel lanes, targeting a theoretical minimum interval of 65,536 cycles per frame to maintain uninterrupted streaming.

Synthesis results, presented in the next chapter, show that most modules operate close to this target, with small deviations caused by architectural constraints. For instance, the `PhaseDelay()` module operates with an interval of 67,603 cycles, the remapping module requires 67,600 cycles. The buffering stages `buffer_1()` and `buffer_2()` take 68,002 and 65,536 cycles, respectively. The `buffer_acc()` stage, which performs accumulation across five angles, introduces a longer interval of 131,079 cycles due to additional memory and arithmetic overhead.

During accumulation, the pipeline intentionally stalls for one frame interval after processing five consecutive frames. This pause is required to transfer the fully compounded frequency-domain data to the downstream IXFFT and ITFFT modules. Limited by this delay, the pipeline achieves an effective utilization rate of approximately 83.3%, while supporting sustained high-throughput operation across compounding sequences. Note that ping-pong memory structures between critical stages allow concurrent read and write transactions without contention.

3.4.2 Flexibility: Supporting Diverse Imaging Configurations

An important feature of the proposed beamforming pipeline is its architectural flexibility, which is rooted in the system's modular composition, parameterized HLS implementation, and streamed dataflow structure. Each stage in the pipeline — from FFT computation to spectral remapping and accumulation — is implemented as an independent module with well-defined interfaces and configurable parameters.

The design supports compile-time parameterization of key attributes such as FFT lengths, frame dimensions, and number of PW angles via a user-defined configuration header. In addition, several runtime parameters — including sampling frequency (`Fs`), speed of sound (`C`), and probe geometry — are streamed into the system and directly influence phase delay and spectral remapping computations without requiring re-synthesis.

For example, altering the plane wave steering direction requires only the regeneration of sine/cosine angle streams, with no modification to the structural design. Likewise, variations in sampling frequency or propagation velocity are handled by recalculating phase delay and mapping coefficients on-the-fly within the corresponding compute units. Furthermore, the streaming architecture facilitates

straightforward horizontal scaling. The number of parallel processing lanes can be increased to match growing performance demands by duplicating existing modules.

4. FPGA Implementation and Performance

The architecture described in previous chapters was implemented, synthesized, and simulated on a high-end AMD Versal™ XCVC1902 device. This chapter presents the hardware-level aspects of the implementation, including design configuration, synthesis results, resource usage, timing performance, and system throughput.

4.1. Target Device and Implementation Parameters

The XCVC1902 device provides the architectural freedom necessary to implement fully streaming, and floating-point-based beamforming logic with ample room for future scaling in resolution, channel count, and frame rate. The key technical details are summarized in Table 1.

Table 1 Device Selection

Chip	XCVC1902-3MSEVSVA2197
Architecture	7nm FinFET
DSP Slices	1,968 DSP58, optimized for high-speed float operations
Memory	130 Mb URAM (UltraRAM) 34 Mb BRAM 900K distributed LUTRAM for control and tags
Toolchain	Synthesis: Vitis™ HLS 2024.2 Verification: Vivado™ Simulator

Key top-level design parameters used in high-level synthesis are shown in Table 2.

Table 2 Top-Level Pre-Synthesis Design Parameters

Parameter	**Value**
Number of channels	128 (`Nx`)
Temporal samples	1536 (`Nt`)
Number of PW angles	5 (`FRAMES`)
Parallel input/output lanes	4 (`N_Lanes`)

Parameter	Value
Temporal FFT size	4096 (`NtFFT`)
Spatial FFT size	256 (`NxFFT`)
Input data type	`hls::stream<std::complex<ap_fixed<16, 1>>>`
Output data type	`hls::stream<std::complex<ap_fixed<32, 1>>>`
Phase correction and remapping parameters	Computed dynamically

In addition to the design parameters, the architecture accepts several runtime inputs that allow it to adapt to different scanning configurations and probe characteristics without re-synthesis. These inputs are listed in Table 3.

Table 3 Top-Level Run-Time Design Parameters

Parameter	Variable Type	Variable Name
Probe element positions	`1D array float`	`probe_geo[Nx]`
Element spacing	`float`	`probe_diff`
Speed of sound	`float`	`C`
Sampling frequency	`float`	`Fs`
Steering angles	`hls::stream<float>`	`in_Theta_S`

4.2. Resource Utilization

The beamforming system was synthesized and implemented on the AMD Versal™ XCVC1902 device, targeting the programmable logic (PL) region. The design leverages dataflow pipelining, optional parallel lanes, and mixed-point arithmetic to achieve high-throughput ultrasound beamforming. While this implementation focuses on the PL fabric, the Versal™ architecture offers additional advanced resources — such as the AI Engine and Network-on-Chip (NoC) — which can significantly boost processing capabilities. The integration of these features is left for future work and presents a promising avenue for further acceleration and intelligent signal processing.

At the top level, the design consumes approximately 1121 DSPs (64%), 384 URAMs (83%), 1520 BRAMs (78%), 198,903 flip-flops (12%), and 243,609 LUTs (29%), operating at a clock frequency of approximately 213 MHz with corresponding to an estimated clock period of 4.7 ns. The system supports a sustained pipeline interval of approximately 1,120,000 cycles per frame, while several submodules achieve ideal or near-ideal intervals to maintain stream continuity.

4.3. Buffering and Timing Intervals

In the current beamforming pipeline, four key buffering stages — `buffer_1()`, `buffer_2()`, `buffer_acc()`, and `buffer_R()` — are used to manage timing alignment and decouple initiation intervals across processing stages. Among them, `buffer_1()`, positioned between the phase correction unit and the FFTX block, introduces the highest frame latency of 68,002 cycles compared to the ideal 65,536, making `buffer_1()` the primary throughput-limiting stage. Consequently, the overall pipeline performance is effectively bounded by the rate at which `buffer_1()` can accept and forward data.

In contrast, `buffer_2()`, placed after FFTX, exhibits ideal streaming characteristics with an observed interval of exactly 65,536 cycles. This means it seamlessly receives and delivers data at the frame rate dictated by the temporal FFT output, thus maintaining uninterrupted throughput between FFTX and the remapping stage.

The `buffer_acc()` module supports much more coarse-grained buffering domain. Since it accumulates multi-angle spectral data over five input frames, it inherently requires more cycles to complete its accumulation task. Once the final compounded data for a frame is written, it clears its memory and accepts new data in the next accumulation window.

The last major timing boundary is governed by `buffer_R()`, which captures fully accumulated (k_x, k_z) domain data and reshapes it into a format consumable by the ITFFT. It is not implemented with DATAFLOW or ping-pong logic, as there is no benefit from frame-overlap at this stage. The latency of writing and reading an entire frame is smaller than the compounding duration, and hence the system remains frame-synchronous without any unnecessary duplication of resources.

4.4. Throughput, Latency, and Frame Rate

The best possible frame throughput for this architecture is based on processing primarily data frames of size $\frac{N_{\text{tFFT}}}{2} \times N_{\text{xFFT}}$, where N_{tFFT} = 4096 (zero-padded axial dimension) and $N_{xFFT} = 256$ (zero-padded lateral dimension). Given a total of 8 parallel complex input lanes, the ideal processing interval per compounded frame becomes:

2048 × 256 / 8 = 65,536 cycles.

However, the actual frame interval for each module includes a combination of its internal latency related buffering boundaries. The module-specific processing intervals are summarized in Table 4.

Table 4 Modules Measured Intervals

Module name	Latency (us) Single Frame (clock: 4.7ns)	Interval Single Frame	Interval Compounding
TFFT	3.08E+02	65,536	327,680
PhaseDelay	3.18E+02	67,603	338,015
buffer_1	3.20E+02	68,002	340,010
XFFT	3.08E+02	65,536	327,680
buffer_2	3.08E+02	65,536	327,680
remapping	3.18E+02	67,600	338,001
buffer_acc	-	-	393,240
IXFFT	-	-	65,536
buffer_R	-	-	196,748
ITFFT	-	-	131,072

While the slight variation above 65,536 in some modules introduces minor delays, the pipeline remains nearly saturated with negligible stalls between stages. The most significant timing boundary is introduced by the accumulation block. Since five angles must be compounded for each frame, `buffer_acc()` maintains partial sums across multiple iterations, introducing a natural inter-frame delay.

After five RF frames are consumed and accumulated, a pause of approximately one frame cycle is introduced to flush the output from `buffer_acc()` through IFFTX. This single-frame delay is required because `buffer_R()` uses full-frame buffering without ping-pong memory, thus serialization is required before streaming to the final inverse FFT stage.

Figure 7 shows a representative portion of co-simulation waveforms, where one can monitor the input streaming behavior by observing handshake signals such as `HW_rawRF_S_*_read`, `HW_rawRF_S_*_empty_n`, and `HW_out_stream_*_write`. These signals implement the basic ready/valid protocol, which reflects the default behavior of AXI-Stream interfaces in Vitis HLS. This minimal interface supports straightforward module-to-module data exchange and is sufficient for baseline functional verification.

As illustrated in Figure 7, five plane wave frames (F1 to F5) are processed sequentially, with each input stream lane (`HW_rawRF_S_0` to `HW_rawRF_S_3`) contributing data toward a compounded output. The processing of individual frames and their accumulation in Box A, spanning from approximately 0 μs to 1581 μs, during which all raw RF frames are continuously ingested. Following this, Box B represents the computation phase after the accumulation buffer (`buffer_acc()`). Finally, in Box C, the output streams (`HW_out_stream_0` to `HW_out_stream_3`) begin transmitting the fully beamformed image data at approximately 2649 μs, resulting in a total latency of about 1070 μs from the end of input ingestion to the onset of output.

Given a system clock frequency of 213 MHz, and worst-case cumulative latency of 340,010 cycles for the buffering stage (`buffer_1()`) plus an additional 65,536 cycles corresponding to one full frame delay required for sequential compounding, the effective processing interval per compounded frame becomes 405,546 cycles, so the effective frame rate for 5-angle CWPC is:

$$Frame\ Rate\ = \frac{213E6}{340{,}010 + 65536} \approx 526\ fps$$

This estimate reflects a conservative configuration in which all input frames are fully buffered and accumulated sequentially, one angle at a time. The total frame interval includes both the latency of the `buffer_1()` stage and a full-frame delay required for angle-wise accumulation.

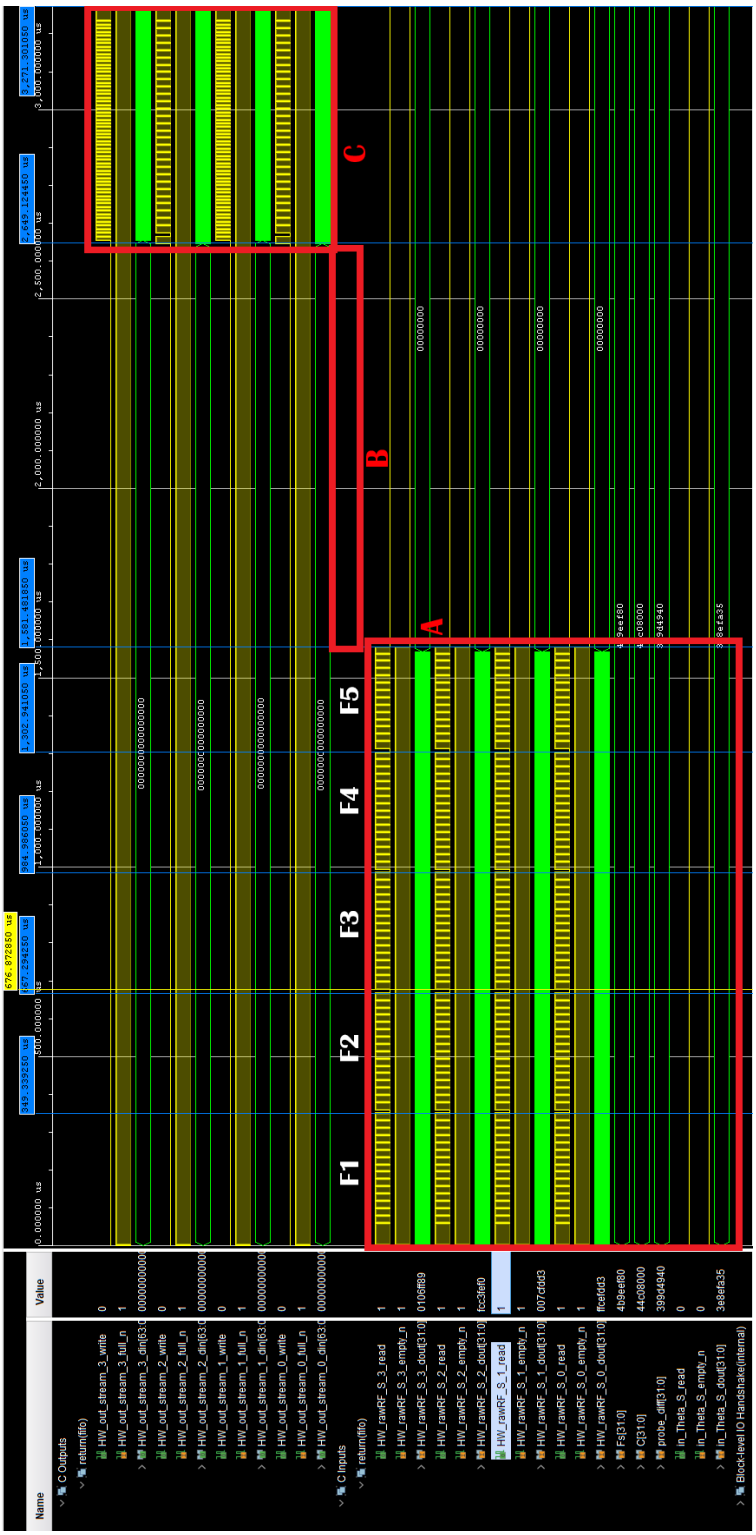

Figure 7 Co-Simulation Waveform

4.5. Clocking, Frequency, and Timing Closure

In our case, the best achievable clock period constraint of 4.7 ns was applied globally across all synthesized modules using the Vitis™ HLS and Vivado™ integration flow. The synthesis and implementation tools generated timing reports confirming successful timing closure without violations. The key timing metrics are summarized in Table 5 based on the routed design timing report.

These results indicate that the design comfortably meets timing requirements across all hierarchical levels and across all clock regions of the Versal™ device. Throughout the implementation, clock domain crossings were avoided by ensuring that all modules are synchronized under the same clock. This simplifies both the timing analysis and the physical implementation, avoiding the need for complex asynchronous interfaces or handshake bridges. All FIFOs, memory buffers, and DSP pipelines are clocked using this single 213 MHz signal, making timing integration straightforward.

Table 5 Routed Design Timing Report

Achieved Frequency	212.766 MHz (4.70 ns clock period)
Achieved Clock Period	4.70 ns
Worst Negative Slack (WNS)	+0.006 ns
Total Negative Slack (TNS)	0.000 ns
Setup Violation Paths	0
Hold Violation Paths	0

4.6. Estimated Power Consumption

Power analysis of the beamforming design was included in the post-route report generated by Vivado™. As shown in Table 6, the overall estimated power consumption is 27.106 W. This includes contributions from logic, memory blocks, DSPs, and clocking infrastructure. Among these components, DSP slices and signal wires are the largest contributor to dynamic power. This is consistent with the design's heavy reliance on Fourier transforms, remapping-related calculations, and wide-lane streaming operations. Static power, which represents baseline leakage and always-on power regardless of switching activity, accounts for nearly 59% of the total consumption. This high contribution is not usual for large Versal™ devices due to their dense logic and memory fabric, considering that our design utilizes only the PL

portion of the device. The reported static power was estimated using a default worst-case junction temperature of 100 °C, as no specific thermal constraints (such as ambient temperature or Theta JA) were applied during analysis. Despite the relatively high static power, the thermal margin remains acceptable, with the reported margin exceeding 25 °C under worst-case assumptions, supporting stable long-duration operation.

Table 6 On-Chip Components Power Consumption

On-Chip	Power (W)
Clocks	1.687
Logic	1.844
Signals	2.915
Block RAM	1.032
URAM	0.840
DSPs	2.784
Static Power	16.003
Total	**27.106**

4.7.Scalability and Architectural Trade-offs

At the architectural level, the modular HLS-based design enables performance scaling through replication of key units, such as FFT lanes and accumulation buffers. For instance, the current implementation instantiates eight FFTX modules for parallel spatial transforms. This number can be increased, subject to resource constraints, to accommodate higher performance demands.

Another form of scaling is achieved through parameterization of core variables — including the number of I/O lanes (`N_Lanes`), probe channels (`Nx`), temporal samples (`Nt`), FFT sizes (`NtFFT, NxFFT`), and the number of compounding frames (`FRAMES`). By modifying these constants and re-synthesizing the design, the system can be adapted to different probe configurations or imaging requirements. Several runtime parameters — such as sampling frequency (`Fs`), speed of sound (`C`), and probe geometry (`probe_geo[]`) — are provided dynamically, which eliminates the need for re-synthesis when adapting to new imaging conditions or probes, enabling real-time adjustment to the acoustic environment.

Scaling certain parameters, however, introduces non-trivial impacts on resource utilization. Table 7 summarizes how changes to key configuration variables affect memory, logic, and compute resources. For example, increasing the number of processing lanes significantly raises URAM consumption due to the need for dedicated memory slices per lane. Likewise, larger FFT sizes increase BRAM and DSP usage, while the number of compounding angles has a minimal effect due to buffer reuse across iterations.

Another architectural trade-off involves dataflow depth versus modular isolation. While deeper pipelines improve throughput, they make timing closure and debugging more difficult. To address this, the design encapsulates major stages — such as FFT, remapping, and accumulation — as self-contained modules with clear handshake protocols, allowing independent optimization and integration.

Clock frequency also constrains scalability. While the current implementation runs at 213 MHz, further throughput improvements would require additional lanes or higher clock rates — both of which increase power consumption and routing complexity in resource-dense layouts.

Table 7 Parameters Impact on Resources

Parameter	Resource Impact
More lanes	URAM usage increases linearly. Each new lane requires separate URAM slices for independent access. With cyclic partitioning, doubling the lane count approximately doubles the total URAM usage.
Larger FFTs	Linearly increases BRAM and DSP usage per FFT instance
More PW angles	Minimal logic increase; handled via reuse in accumulation buffer

4.8. Verification Methodology

For this project, simulation-based verification was embedded into every stage of development, from early functional testing to post-synthesis timing validation. All computational blocks were accompanied by a standalone testbench to isolate and confirm its behavior before integration into the full pipeline.

Functional simulation formed the foundation of this methodology. All major modules were verified using hand-written C++ testbenches created within the Vitis™ HLS

environment. These testbenches supplied known input data streams and examined the generated outputs against software-based reference baselines or expected structural patterns. For example, the `phaseDelay_main.cpp` testbench fed known complex-valued samples into the `PhaseDelay()` module and validated the correctness of the output phase adjustments based on the angle-specific sine and cosine values generated dynamically. Similarly, `phasedelay_factors_main.cpp` confirms that the sine and cosine values themselves are consistent with the computed geometry and transmit angles.

Buffer-related modules such as `buffer1_main.cpp`, `buffer2_main.cpp`, `bufferAcc_main.cpp`, and `bufferR_main.cpp` were validated to ensure correct address indexing, write-read synchronization, and cyclic memory behavior. In these testbenches, corner cases were simulated, including wrap-around conditions, multiple frames of continuous data, and staggered stream inputs. The goal was to guarantee that memory access patterns preserved data integrity while maintaining pipeline-friendly access patterns.

The remapping module was evaluated using the `mapping_factors_main.cpp` and `mapping_main.cpp` testbenches. The correctness of index generation, interpolation factor computation, and clamping logic were all verified. These tests ensured that the correct behavior of spectral remapping was not compromised under varying imaging angles and channel geometries.

Once individual modules were verified in isolation, the full beamforming pipeline was integrated and tested using the top-level `BF_TB_main.cpp` testbench. This comprehensive simulation environment loaded realistic RF input data from prerecorded files, applied runtime configuration parameters (e.g., probe geometry, speed of sound, sampling frequency), and streamed the data through the complete pipeline. The output — after all processing stages — was captured for verification and analysis.

Our testbench supports evaluation across multiple compounding configurations, including single-, three-, and five-angle plane wave scenarios. It features configurable debug flags that control which intermediate pipeline outputs are dumped for inspection. These checkpoints include the phase delay, buffer_1, XFFT, buffer_2, remapping, accumulation, and final inverse FFT stages. For each activated dump point, the corresponding internal output stream is redirected to `HW_out_stream[]` using preprocessor directives, as shown in the example below:

```
#define DUMP_AFTER_PHASE_DELAY 0
#define DUMP_AFTER_BUFFER1     0
#define DUMP_AFTER_FFTX_STREAM 0
#define DUMP_AFTER_BUFFER2     0
#define DUMP_AFTER_MAPPING     0
#define DUMP_AFTER_ACC         0
#define DUMP_AFTER_FinalStage  1
void Beamforming_MIG(hls::stream<float> &in_Theta_S,
                     float probe_geo[Nx], float probe_diff, float C, float Fs,
                     hls::stream<T_FFTZ_in> HW_rawRF_S[N_Lanes],
#if DUMP_AFTER_PHASE_DELAY
                     hls::stream<cmpx2xD_t> HW_out_stream[N_Lanes]
#elif DUMP_AFTER_BUFFER1
                     hls::stream<T_FFTX_in> HW_out_stream[2 * N_Lanes]
#elif DUMP_AFTER_FFTX_STREAM
                    hls::stream<T_FFTX_out> HW_out_stream[2 * N_Lanes]
#elif DUMP_AFTER_BUFFER2
                   hls::stream<CData_t> HW_out_stream[2 * N_Lanes]
#elif DUMP_AFTER_MAPPING
                     hls::stream<CData_t> HW_out_stream[2 * N_Lanes]
#elif DUMP_AFTER_ACC
                     hls::stream<T_IFFTX_in> HW_out_stream[2 * N_Lanes]
#elif DUMP_AFTER_FinalStage
                   hls::stream<T_IFFTZ_out> HW_out_stream[N_Lanes]
#endif
```

To evaluate the quality of the reconstructed data, the final 2D analytic image — composed of real and imaginary components — was exported and visualized in MATLAB. These outputs were compared to ground-truth results from a software-based beamformer to assess CPWC reconstruction accuracy (see Figure 8 and Figure 9). After passing functional simulation, the design was implemented and routed at the RTL level. Post-implementation validation included detailed inspection of Vivado™ timing reports, confirming that the critical timing constraint of 4.7 ns was met without violations.

4.9. Comparison of HLS Results against Software-Based Reference

To verify the correctness of the entire beamforming pipeline, the final output of the HLS implementation was directly compared with a MATLAB-based reference code of the same imaging process. Figure 8 shows the B-mode image generated by the MATLAB code, which serves as the numerical reference or "golden output" for comparison. Figure 9 depicts the B-mode image obtained from the reconstruction results generated by the synthesized pipeline, using the same input dataset and parameters.

Visual inspection of the two images reveals a near-identical reconstruction across depth and lateral dimensions. No significant artifacts, distortions, or edge inconsistencies were introduced by the hardware pipeline, confirming the bit-accurate correctness throughout all processing stages. This close match between HLS and MATLAB implementation outputs confirms that our design does not introduce

any reconstruction errors. It demonstrates that the proposed mixed-point architecture can serve as a hardware-based accelerator for floating-point software methods in high-throughput imaging applications.

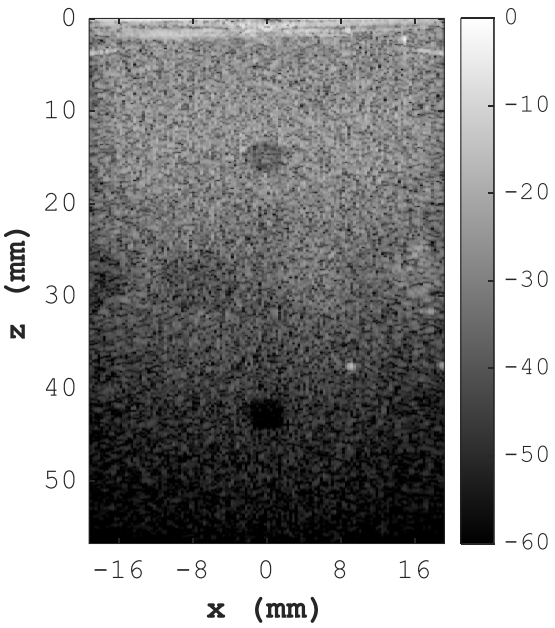

Figure 8 B-Mode Image from MATLAB-Based Reconstruction

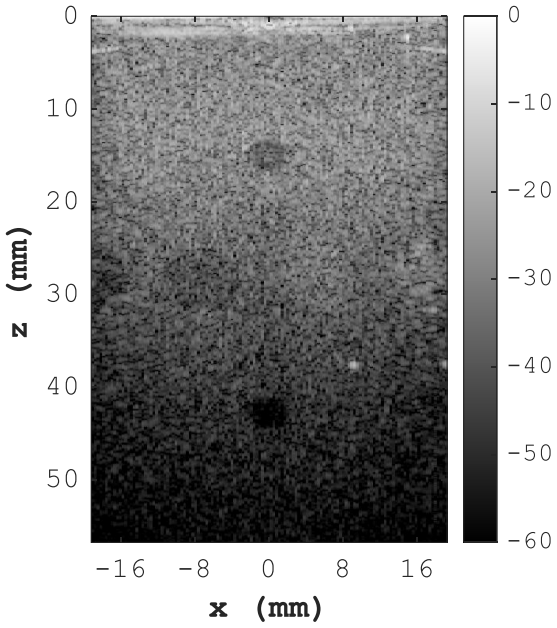

Figure 9 B-Mode Image from HLS-Based Reconstruction

4.10. Current Design Limitations

The design adopts full-frame buffering for all major transform and staging operations using cyclically partitioned URAM arrays, enabling high parallelism and fine-grained stream control across the pipeline. However, with 4 parallel processing lanes, the system already utilizes approximately 82% of the available URAM resources on the XVCK1902 device. Expanding the design to support 16 lanes or increasing frame dimensions (e.g., longer FFTs or more accumulation depth) would exceed the available on-chip memory unless the buffering strategy is restructured or replaced with time-multiplexed access patterns.

Another limitation of our design approach is as follows. The system HLS specification is configurable through the `constants.h` header, where key parameters — including FFT dimensions, number of transmit angles, number of channels, and fixed-point precision — are statically defined and resolved at compile time. Hence, any changes to these parameters would require re-synthesis of the entire HLS project.

While this compile-time parameterization is beneficial for timing closure, resource usage, and streamlined pipeline behavior, it inherently limits the system's adaptability at runtime. As a result, true runtime flexibility — such as dynamically modifying imaging geometry or adjusting frame resolution on the fly — is not supported in the current implementation. This limitation reflects a broader trade-off in fixed HLS-based pipelines, where architectural determinism and performance are prioritized over dynamic reconfigurability.

Finally, we should note that the current beamforming pipeline is implemented entirely within the programmable logic (PL) fabric of the Versal™ XVCK1902 and does not yet leverage the device's AI Engine (AIE) cores. While the architecture is modular and structurally compatible with future integration of AI-augmented functionality, this version does not include any adaptive logic or learning-based components. Features such as dynamic transmit angle selection, coherence-based apodization, or learned remapping techniques are not yet implemented. The system remains strictly deterministic and feed-forward in its execution model, which simplifies control, timing, and validation, but also limits its ability to adapt in real time to dynamic changes of imaging scenarios.

5. Conclusion

This thesis has presented the design, implementation, and verification of a Fourier-domain ultrasound beamforming pipeline mapped onto the AMD Versal™ XVCK1902 device. The system integrates all core processing stages — including temporal FFT, phase correction, spatial FFT, spectral remapping, coherent compounding, and inverse FFTs— using a modular architecture developed entirely as a high-level synthesis (HLS) design.

Through structured dataflow pipelining, mixed-point arithmetic, and URAM-backed buffering, the system supports continuous streaming across multiple input frames with CPWC data accumulation. Each module has been tested independently and integrated into a complete architecture, which achieves timing closure at 213 MHz and maintains frame-level synchronization and dataflow consistency throughout the processing chain.

5.1. Summary of Contributions

The proposed system implements the Temme–Mueller (TM) migration algorithm as a pipelined hardware architecture. By restructuring the TM migration steps into discrete pipelined modules, the implementation supports continuous dataflow while preserving the theoretical underpinnings of the algorithm. My main contribution is the development of all processing stages — from the temporal FFT to the final inverse FFT — as standalone, pipelined modules that communicate via `hls::stream` interfaces with handshake signaling.

My second contribution is that one of the most computationally complex stages — spectral remapping — is implemented without the use of precomputed lookup tables. All required quantities are computed dynamically on-chip using mixed-point arithmetic, based on the input parameters (imaging angle, probe geometry, and speed of sound). Remapping and interpolation are performed inline for each frequency bin using dedicated logic performing linear interpolation between FFT bins.

My third contribution is as follows. The beamforming architecture employs URAM-based dual-port memory to support full-frame on-chip buffering across the pipeline. There are four buffering modules, each cyclically partitioned to enable concurrent access by parallel lanes, ensuring low-latency data movement without reliance on external DRAM. To improve processing continuity, the system adopts an interleaved

execution model in which data from new angles can propagate through early stages while previous angles are still being accumulated.

My fourth contribution is enabling imaging settings, such as probe geometry (`probe_geo[]`), speed of sound (`C`), and sampling frequency (`Fs`), to be treated as runtime-configurable parameters. These values are passed through the top-level interface and used dynamically within key modules, including remapping and phase correction stages. This allows the beamforming system to adjust to different probes, acoustic media, or PW angles without requiring hardware recompilation. Additionally, design-time parameters — such as FFT sizes (`NtFFT`, `NxFFT`), frame count (`FRAMES`), and lane parallelism (`N_Lanes`) — are abstracted into a global header file (`constants.h`), enabling easy adaptation of the design to various spatial resolutions and frame rates through a single synthesis pass.

Finally, the system was validated through a combination of module-level and full-pipeline testbenches developed in C++/HLS, followed by hardware synthesis, placement, routing, and Vivado™ co-simulation. The design achieved timing closure at 213 MHz on the Versal™ XVCK1902 device, with consistent data throughput across all stages and no timing violations reported. Output data was compared against MATLAB-generated reference results to validate functional accuracy across all pipeline stages.

5.2. Future Work

Although the current implementation is optimized for the Versal™ XVCK1902 device, the design remains portable across other AMD FPGAs such as Ultrascale+ or Virtex™ series. Porting the pipeline to new targets involves adapting the synthesis constraints to the resource availability of the target device, particularly in terms of URAM, BRAM, and DSP slices. Thanks to modular encapsulation and standardized stream interfaces (`hls::stream`), the design's integration boundaries are well defined, and each major stage can be isolated or replaced as needed. This approach enables compatibility with broader SoC platforms or heterogeneous computing environments, such as systems where FPGAs interface with CPUs or GPUs over AXI.

Perhaps the most compelling direction for future work lies in leveraging the AI Engine (AIE) fabric embedded within the Versal™ architecture. These SIMD vector processors operate at up to 1 GHz and are optimized for high-throughput signal processing. Integrating AIEs into the current beamforming flow opens several pathways for significant enhancement.

The first use case involves streaming spectral-domain outputs (from the FFT and remapping modules) into AIEs for additional processing, such as adaptive apodization, clutter suppression, spatial filtering, or phase coherence enhancement. These operations are highly parallel and benefit from the AIE's wide vector execution pipeline, enabling frame-level analysis and correction with lower latency than PL-only logic. Beamformer modules implemented in HLS can stream data directly into AIE tiles via AXI-Stream interfaces.

Another application is intelligent transmit angle weighting, where the AIE computes real-time weights based on signal coherence, SNR, or even shallow neural network models trained on beamformed datasets. This enables dynamic angle selection for frame compounding that adapts to the scene, improving image contrast and frame rate without static weighting heuristics.

The third class of extensions involves deploying lightweight deep learning models for tissue classification, segmentation, or artifact suppression. With support for quantized inference via Vitis™ AI, the AIE can perform real-time interpretation of spectral or spatial-domain data, enabling advanced features like speed-of-sound estimation or target detection—all on-chip, without requiring external accelerators.

6. Reference

[1] H. Hasegawa and H. Kanai, "High-Frame-Rate Echocardiography Using Diverging Transmit Beams and Parallel Receive Beamforming," *Journal of Medical Ultrasonics*, vol. 38, no. 3, pp. 129–140, Jul. 2011, doi: 10.1007/s10396-011-0304-0.

[2] G. Montaldo, J. Bercoff, N. Benech, and M. Fink, "Coherent Plane-Wave Compounding for Very High Frame Rate Ultrasonography and Transient Elastography," *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 56, pp. 489–506, Apr. 2009, doi: 10.1109/TUFFC.2009.1067.

[3] R. Prager, "High-Speed Beamforming Using Modern GPUs," *IEEE Trans Med Imaging*, vol. 29, no. 2, pp. 347–355, 2010.

[4] M. Albulayli and D. Rakhmatov, "Fourier Domain Depth Migration for Plane-Wave Ultrasound Imaging," *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 65, no. 8, pp. 1321–1333, Aug. 2018, doi: 10.1109/TUFFC.2018.2837000.

[5] V. Perrot, M. Polichetti, F. Varray, and D. Garcia, "So You Think You Can DAS? A Viewpoint on Delay-and-Sum Beamforming," *Ultrasonics*, vol. 111, p. 106309, Mar. 2021, doi: 10.1016/j.ultras.2020.106309.

[6] N. A. Campbell and J. A. Brown, "A Real-Time Dual-Mode High-Frequency Beamformer for Ultrafast and Focused Imaging," *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 69, no. 4, pp. 1268–1276, Apr. 2022, doi: 10.1109/TUFFC.2022.3151218.

[7] E. Boni *et al.*, "Architecture of an Ultrasound System for Continuous Real-Time High Frame Rate Imaging," *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 64, no. 9, pp. 1276–1284, Sep. 2017, doi: 10.1109/TUFFC.2017.2727980.

[8] Z. Kou *et al.*, "High-Level Synthesis Design of Scalable Ultrafast Ultrasound Beamformer With Single FPGA," *IEEE Trans Biomed Circuits Syst*, vol. 17, no. 3, pp. 446–457, Jun. 2023, doi: 10.1109/TBCAS.2023.3267614.

[9] G. Corradi and J. A. Jensen, "Real Time Synthetic Aperture and Plane Wave Ultrasound Imaging with the Xilinx VERSAL™ SIMD-VLIW Architecture," in *2020 IEEE International Ultrasonics Symposium (IUS)*, IEEE, Sep. 2020, pp. 1–4. doi: 10.1109/IUS46767.2020.9251749.

[10] B. L. West *et al.*, "Tetris: Using Software/Hardware Co-Design to Enable Handheld, Physics-Limited 3D Plane-Wave Ultrasound Imaging," *IEEE*

Transactions on Computers, vol. 69, no. 8, pp. 1209–1220, Aug. 2020, doi: 10.1109/TC.2020.2990061.

[11] P. Temme and G. Mueller, "Fast plane-wave and single-shot migration by Fourier transform," *Journal of Geophysics*, vol. 60, pp. 19–27, Jan. 1986.

[12] Jiqi Cheng and Jian-yu Lu, "Extended high-frame rate imaging method with limited-diffraction beams," *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 53, no. 5, pp. 880–899, May 2006, doi: 10.1109/TUFFC.2006.1632680.

[13] P. Kruizinga, F. Mastik, N. de Jong, A. F. W. van der Steen, and G. van Soest, "Plane-Wave Ultrasound Beamforming Using a Nonuniform Fast Fourier Transform," *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 59, no. 12, Dec. 2012, doi: 10.1109/TUFFC.2012.2509.

[14] D. Garcia, L. L. Tarnec, S. Muth, E. Montagnon, J. Poree, and G. Cloutier, "Stolt's f-k Migration for Plane Wave Ultrasound Imaging," *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 60, no. 9, pp. 1853–1867, Sep. 2013, doi: 10.1109/TUFFC.2013.2771.

[15] D.-L. D. Liu and Ting-Lan Ji, "Plane Wave Image Formation in Spatial-Temporal Frequency Domain," in *2016 IEEE International Ultrasonics Symposium (IUS)*, IEEE, Sep. 2016, pp. 1–5. doi: 10.1109/ULTSYM.2016.7728905.

[16] T. Sumanaweera and D. Liu, "Medical image reconstruction with the FFT," in *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, M. Pharr, Ed., Reading, MA, USA: Addison-Wesley, 2005, ch. 48.

[17] AMD, "Arria V GX FPGA." [Online]. Available: https://www.intel.com/content/www/us/en/products/details/fpga/arria/v/gx/products.html

[18] AMD, "Kintex UltraScale+FPGAs." [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/kintex-ultrascale-plus.html

[19] AMD, "Versal AI Core Series." [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/versal/aicore-series.html

[20] J. Shi and D. Rakhmatov, "Fixed-Point CPWC Ultrasound Image Reconstruction," in *2019 IEEE International Ultrasonics Symposium (IUS)*, IEEE, Oct. 2019, pp. 1282–1285. doi: 10.1109/ULTSYM.2019.8926043.

[21] R. H. Stolt, “Migration by Fourier Transform,” *Geophysics*, vol. 43, no. 1, pp. 23–48, Feb. 1978, doi: 10.1190/1.1440826.

[22] AMD, “Virtex UltraScale+FPGAs.” [Online]. Available: https://www.amd.com/en/products/adaptive-socs-and-fpgas/fpga/virtexultrascale-plus.html

[23] P. Navaeilavasani, “Hardware Architecture for Accelerating Frequency-Domain Ultrasound Image Reconstruction,” MASc Thesis, University of Victoria, Canada, 2024.

[24] T. L. Szabo, *Diagnostic Ultrasound Imaging*, 2nd ed. New York, USA: Academic, 2004.

[25] F. W. Kremkau and F. Forsberg, *Sonography: Principles and Instruments*, 9th. St. Louis, Missouri: Elsevier, 2016.

[26] J. T. Bushberg, J. A. Seibert, E. M. Leidholdt, and J. M. Boone, *The Essential Physics of Medical Imaging*. Wolters Kluwer Health, 2011. [Online]. Available: https://books.google.ca/books?id=RKcTgTqeniwC

[27] K. W. Pulsipher, D. A. Hammer, D. Lee, and C. M. Sehgal, “Engineering Theranostic Microbubbles Using Microfluidics for Ultrasound Imaging and Therapy: A Review,” *Ultrasound Med Biol*, vol. 44, no. 12, pp. 2441–2460, 2018, doi: https://doi.org/10.1016/j.ultrasmedbio.2018.07.026.

[28] J. Shen-Wagner and M. Deutchman, “Point-of-Care Ultrasound: A Practical Guide for Primary Care.,” *Fam Pract Manag*, vol. 27, no. 6, pp. 33–40, 2020.

[29] R. M. Lang *et al.*, “Recommendations for Cardiac Chamber Quantification by Echocardiography in Adults: An Update from the American Society of Echocardiography and the European Association of Cardiovascular Imaging,” *Eur Heart J Cardiovasc Imaging*, vol. 16, no. 3, pp. 233–271, Mar. 2015, doi: 10.1093/ehjci/jev014.

[30] A. Benchimol and K. B. Desser, “Doppler Ultrasound in Cardiology: Physical Principles and Clinical Applications,” *JAMA*, vol. 249, no. 11, p. 1502, Mar. 1983, doi: 10.1001/jama.1983.03330350070037.

[31] T. Lorentzen *et al.*, “EFSUMB Guidelines on Interventional Ultrasound (INVUS), Part I,” *Ultraschall Med*, vol. 36, pp. E3–E16, Oct. 2015, doi: 10.1055/s-0035-1553593.

[32] J. E. Kennedy, “High-Intensity Focused Ultrasound in the Treatment of Solid Tumours,” *Nat Rev Cancer*, vol. 5, no. 4, pp. 321–327, Apr. 2005, doi: 10.1038/nrc1591.

[33] C. L. Moore and J. Copel, “Point-of-Care Ultrasonography.,” *N Engl J Med*, vol. 364 8, pp. 749–57, 2011, [Online]. Available: https://api.semanticscholar.org/CorpusID:1621892

[34] J. H. Miao, “The Art and Science of Ultrasound Imaging: Medical Applications of Ultrasound in Diagnosis and Therapy and its Impact on Patient Care,” *Cardiovasc Diagn Ther*, vol. 13, no. 1, pp. 109–111, Feb. 2023, doi: 10.21037/cdt-22-537.

[35] J. A. Jensen, *Estimation of Blood Velocities Using Ultrasound: A Signal Processing Approach*. Cambridge University Press, 1996. [Online]. Available: https://books.google.ca/books?id=2pZOAAAAIAAJ

[36] G. Montaldo, M. Tanter, J. Bercoff, N. Benech, and M. Fink, “Coherent Plane-Wave Compounding for Very High Frame Rate Ultrasonography and Transient Elastography,” *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 56, no. 3, pp. 489–506, Mar. 2009, doi: 10.1109/TUFFC.2009.1067.

[37] M. Tanter and M. Fink, “Ultrafast Imaging in Biomedical Ultrasound,” *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 61, no. 1, pp. 102–119, 2014, doi: 10.1109/TUFFC.2014.2882.

[38] J. Jensen *et al.*, “Accuracy and Precision of a Plane Wave Vector Flow Imaging Method in the Healthy Carotid Artery,” *Ultrasound Med Biol*, vol. 44, no. 8, pp. 1727–1741, 2018, doi: https://doi.org/10.1016/j.ultrasmedbio.2018.03.017.

[39] B.-F. Osmanski, M. Pernot, G. Montaldo, A. Bel, E. Messas, and M. Tanter, “Ultrafast Doppler Imaging of Blood Flow Dynamics in the Myocardium,” *IEEE Trans Med Imaging*, vol. 31, pp. 1661–1668, 2012, [Online]. Available: https://api.semanticscholar.org/CorpusID:206747848

[40] J. A. Jensen and P. Munk, “A New Method for Estimation of Velocity Vectors,” *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 45, no. 3, pp. 837–851, 1998, doi: 10.1109/58.677749.

[41] J. Bercoff, G. Montaldo, T. Loupas, D. Savéry, F. Mézière, and M. Fink, “Ultrafast Compound Doppler Imaging: Providing Full Blood Flow Characterization,” *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 58, pp. 134–147, Jan. 2011, doi: 10.1109/TUFFC.2011.1780.

[42] P. Song *et al.*, “Improved Super-Resolution ultrasound Microvessel Imaging with Spatiotemporal Nonlocal Means Filtering and Bipartite Graph-Based Microbubble Tracking,” *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 65, no. 2, pp. 149–167, 2018, doi: 10.1109/tuffc.2017.2778941.

[43] P. Navaeilavasani and D. Rakhmatov, “Accelerator Architecture for Plane-Wave Ultrasound Image Reconstruction in Fourier Domain,” *IEEE Open Journal of Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 4, pp. 231–246, 2024, doi: 10.1109/OJUFFC.2025.3530395.

[44] B. Y. S. Yiu, I. K. H. Tsang, and A. C. H. Yu, “GPU-Based Beamformer: Fast Realization of Plane Wave Compounding and Synthetic Aperture Imaging,” *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 58, no. 8, pp. 1698–1705, 2011, doi: 10.1109/TUFFC.2011.1999.

[45] X. Zhou, L. Wu, and W. Wang, “FPGA Implementation of Real-Time Plane-Wave Ultrasound Imaging with Coherence Compounding,” *IEEE Trans Biomed Circuits Syst*, vol. 14, no. 5, pp. 1023–1033, 2020.

[46] C. Hu, Z. Ma, Z. Shen, Z. Wang, and H. Liu, “AI-Assisted Beamforming for Ultrafast Ultrasound Imaging Using Deep Learning and FPGA Acceleration,” *IEEE Trans Ultrason Ferroelectr Freq Control*, vol. 69, no. 6, pp. 1292–1305, 2022.

[47] S. V Musti, “Plane-Wave Fourier-Domain Beamforming with CNN-Assisted Resolution Enhancement,” MEng Report, University of Victoria, Canada, 2022.

[48] H. Sorensen, D. Jones, M. Heideman, and C. Burrus, “Real-Valued Fast Fourier Transform Algorithms,” *IEEE Trans Acoust*, vol. 35, no. 6, pp. 849–863, 1987, doi: 10.1109/TASSP.1987.1165220.