Fixed-Point Hardware Design for CPWC Image Reconstruction

by

Ji Shi
B.Sc., Beijing Jiaotong University, 2013
M.Sc., University College London, 2014

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

Fixed-Point Hardware Design for CPWC Image Reconstruction

by

Ji Shi
B.Sc., Beijing Jiaotong University, 2013
M.Sc., University College London, 2014

Supervisory Committee

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

Dr. Kin Fun Li, Departmental Member
(Department of Electrical and Computer Engineering)

ABSTRACT

Coherent plane-wave compounding (CPWC) ultrasonography is an important imaging modality that allows for very high frame rates. During CPWC image reconstruction, computationally expensive delay-and-sum beamforming can be replaced by faster Fourier-domain remapping. The thesis deals with the MATLAB and hardware implementation of one of the recently proposed Fourier-domain CPWC reconstruction methods, namely, plane-wave (PW) Stolt's migration algorithm.

We first present the floating- and fixed-point implementations of the said migration algorithm in MATLAB, and then perform quantitative evaluation of the reconstruction results, showing that it is feasible to obtain high-quality compounded images using hardware-oriented scaled fixed-point calculations, as opposed to more expensive software-oriented floating-point arithmetic.

We also generate Xilinx FPGA-based implementations of both floating- and fixed-point MATLAB-based algorithms, using a high-level synthesis (HLS) design flow that collaboratively employs MATLAB Coder and Vivado HLS tool. MATLAB Coder can automatically convert a MATLAB code into a C program, while Vivado HLS can convert the resulting C program into a synthesizable Verilog/VHDL description. Results show that our fixed-point FPGA implementation is more resource and power efficient and can also operate at a higher clock frequency compared to its floating-point counterpart.

Contents

List of Tables

List of Figures

Listings

ACKNOWLEDGEMENTS

First and foremost, I would like to express my most sincere gratitude to my supervisor, **Dr. Daler N. Rakhmatov**, for his mentorship, enthusiasm, and encouragement throughout all stages of my study and research. Without his invaluable guidance and inspiration, this thesis would not have been possible.

I would also like to thank **Dr. Kin Fun Li** (my supervisory committee member) for his support and insightful feedback, which have contributed to the improvement of this thesis.

Lastly, I would like to thank my family and friends for their love, understanding, and support along the way.

DEDICATION

To my grandparents, Shoujie and Jiuzhang.
To my parents, Yuqiang and Xiang.
To my wife, Ying.

List of Acronyms

ADC	Analog-to-Digital Converter
ALU	Arithmetic Logic Unit
BRAM	Block Random Access Memory
CORDIC	COordinate Rotation DIgital Computer
CPWC	Coherent Plane-Wave Compounding
CT	Computed Tomography
DAC	Digital-to-Analog Converter
DAS	Delay-And-Sum
DSP	Digital Signal Processing
ERM	Exploding Reflector Mode
FF	Flip-Flop
FFT	Fast Fourier Transform
FIL	FPGA-In-the-Loop
FPGA	Field-Programmable Gate Array
FWHM	Full Width at Half Maximum
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
IFFT	Inverse Fast Fourier Transform
IP	Intellectual Property
LUT	Look-Up Table
MRI	Magnetic Resonance Imaging

MSE	Mean-Squared Error
PSNR	Peak Signal-to-Noise Ratio
ROM	Read-Only Memory
RTL	Register-Transfer Level
SSIM	Structural Similarity Index
WHS	Worst Hold Slack
WNS	Worst Negative Slack

Chapter 1

Introduction

Ultrasound imaging is a medical imaging technique that utilizes high-frequency sound waves and their echoes to generate images of the internal structure of the body for both diagnosis and therapeutic purposes. Unlike other imaging modalities, such as CT and MRI, ultrasound imaging is relatively safe, noninvasive and cost-effective, hence this technique has been widely used in the medical field.

1.1 Ultrasound Imaging Basics

1.1.1 Ultrasound Imaging Modes

The images generated by an ultrasound device can be displayed in various ways, which are called ultrasound imaging modes.

A-mode, or amplitude mode, which is the oldest ultrasound technique, shows the amplitude distribution at different depths. It measures the arrival time of the echoes and displays the envelope of pulse-echoes versus depth. A-mode has a one-dimensional format, which provides little spatial information on the imaged structure. Fig. 1.1 shows an A-mode image example.

B-mode, or brightness mode, is the most common form of ultrasound imaging. Unlike A-mode, B-mode generates two or three-dimensional images where the amplitude of the echoes is converted into pixels of different brightness. The horizontal and vertical distance of a pixel indicates an actual position in the imaged body, and the magnitude of the gray level corresponds to the echo strength. Fig. 1.2 shows an example of a B-mode image.

M-mode, or motion mode, is defined as time motion display of the acoustic wave

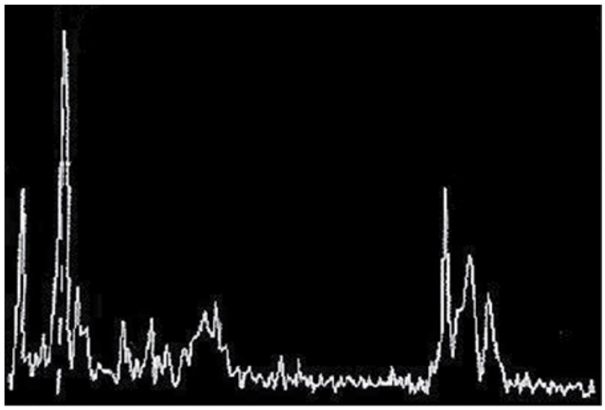


Figure 1.1: A-mode image example [1]

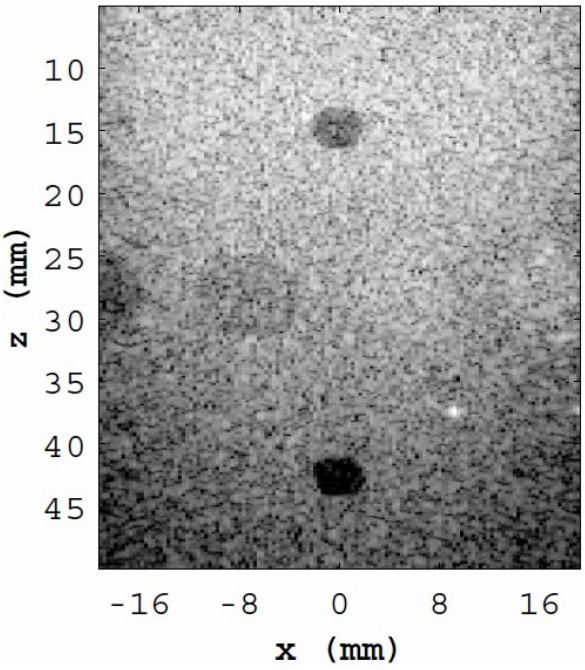


Figure 1.2: B-mode image example [2]

along a chosen image line. Similar to B-mode, the amplitude of the echoes in an M-mode image is also represented by the grayscale of pixels. The difference from a B-mode image is that only a single B-mode line is required and the information is repeatedly obtained from that line in order to analyze the motion of the medium. Fig. 1.3 is an example showing an M-mode image.

Since our work mainly focuses on the two-dimension B-mode imaging, all the discussions in the remaining thesis fall into this category.

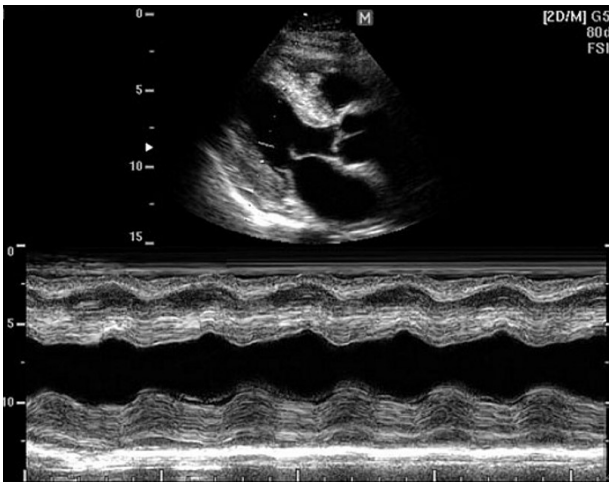


Figure 1.3: M-mode image example [1]

1.1.2 Ultrasound Imaging System

Typically, ultrasound systems operate in the frequency range between 2 MHz and 20 MHz [8]. Fig. 1.4 is the block diagram showing the implementation of a conventional multi-focus ultrasound system.

The transducer consists of an array of piezoelectric elements, which are capable of converting electric energy to acoustic energy and vice versa. As a result, each element can be regarded as both a transmitter (Tx) emitting ultrasound waves during the transmission phase and a receiver (Rx) collecting echoes during the reception phase. The Tx beamformer is responsible for adjusting the signal amplitude and phase for each transducer element to form a beam toward desired directions for each image frame [9]. During focused ultrasound imaging, the Tx beamformer first sends the delayed and weighted version of the digital pulses through the digital-to-analog converter (DAC), providing required voltages on transducer elements to generate sound waves focused at the desired focal point along the scan line. Alternatively, the TX beamformer can generate unfocused beams, such as plane waves, which allows the system to insonify large sections of an imaging medium at a faster rate than in the case of multiple focused transmissions. The sound wave then propagates in the medium. Reflections occur when the sound wave encounters structures with different acoustic impedances. The backscattered signals are collected by the same transducer elements and converted into analog voltages which are subsequently sampled by the analog-to-digital converter (ADC) before being fed to the Rx beamformer. The latter applies appropriate delay to generate beamformed signals. Next, the envelope detection

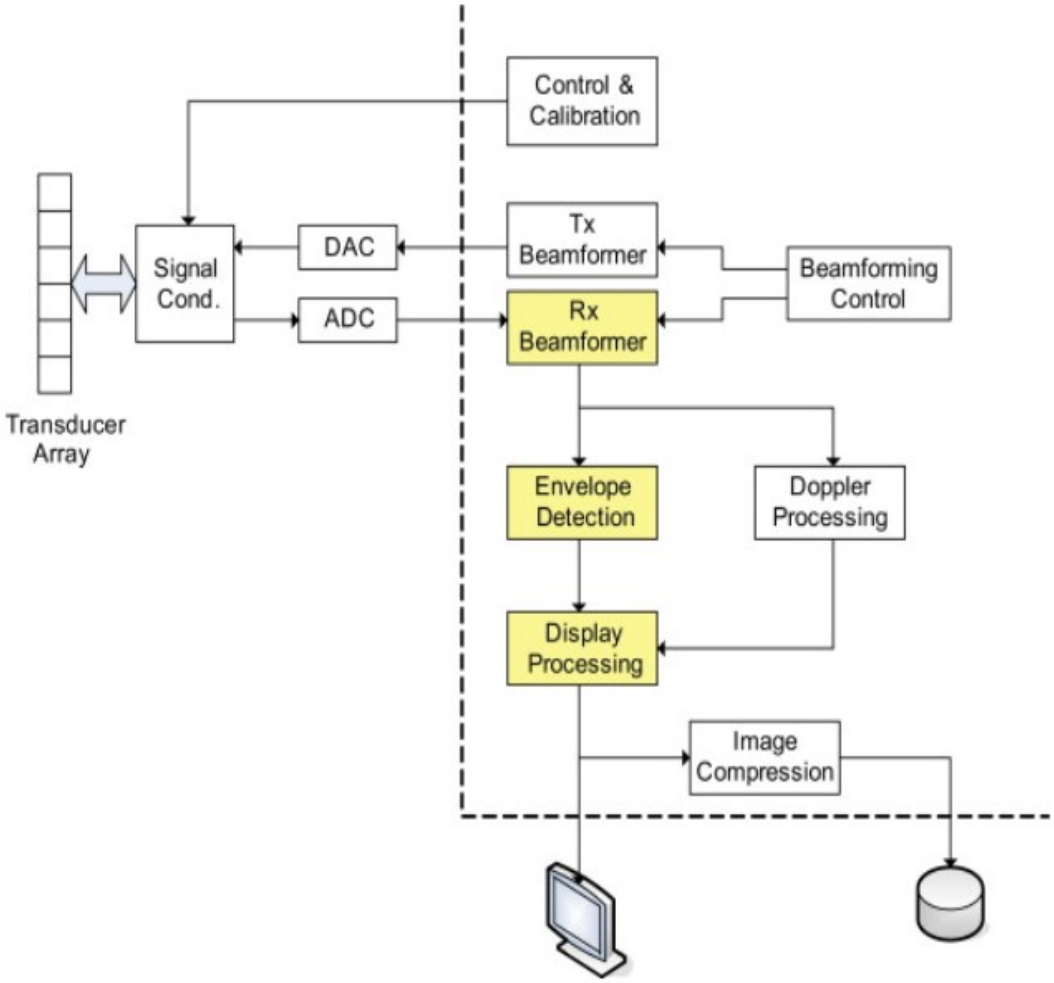


Figure 1.4: An ultrasound imaging system [3]

block takes the absolute values of the Hilbert transform of the beamformed signals for the purpose of display. Doppler processing block is used for the detection and measurement associated with velocities, for example, of wall and valve motion and blood flow. The image compression block is then employed to reduce the dynamic range of the received signals by performing the logarithmic compression [10].

1.1.3 Image Quality Metrics

To evaluate the image quality obtained in an ultrasound imaging system, the following quantitative performance metrics are widely utilized.

Spatial Resolution

Spatial resolution refers to the ability to distinguish two points from one another. In other words, a higher spatial resolution represents a smaller distinguishable distance between two points. One quantitative measurement of spatial resolution is known as the full-width at half maximum (FWHM).

One category of spatial resolution is called axial resolution, which also known as depth, longitudinal, or linear resolution. It refers to the minimum separation that an imaging system can distinguish along the axis of wave propagation. It is equal to half of the spatial pulse length. For example, emitting a pulse consisting of M sinusoidal periods gives the following axial resolution [11] :

$$R_{axial} = \lambda M/2 \tag{1.1}$$

where λ is the wavelength of the transmitted pulse. Equation 1.1 shows that the shorter the wavelength of the pulse, the smaller the distinguishable distance, which means a greater axial resolution. Also, from equation 1.1, it can be found that the axial resolution does not vary with the image depth.

The other category is lateral resolution, which indicates the ability to distinguish two points in the direction perpendicular to the wave propagation axis. Lateral resolution is affected by the width of the beam and the depth of imaging. The lateral FWHM resolution can be expressed as [11]:

$$R_{lateral} = \lambda z/D_t \tag{1.2}$$

where λ, z, and D_t represent the wavelength, the imaged depth, and the width of the active transducer, respectively. Equation 1.2 shows that the lateral resolution gets worse at a greater depth or with a smaller transducer.

Contrast-to-Noise Ratio

Contrast-to-noise ratio (CNR) measures a system's ability to distinguish a certain region of interest (ROI) from its surrounding background. It can be expressed as [12]

$$CNR = 20\log_{10}(\frac{|\mu_{in} - \mu_{out}|}{\sqrt{(\sigma_{in}^2 + \sigma_{out}^2)/2}}) \tag{1.3}$$

where μ_{in} and μ_{out} are the mean signal levels inside and outside the ROI, and σ_{in} and σ_{out} are the corresponding standard deviations.

Mean-Squared Error

Mean-squared error (MSE) is a common way of measuring the degree of similarity between two images. It is represented as the cumulative squared error between the processed and the original image. The MSE between two 2-dimension images $I_1(m,n)$ and $I_2(m,n)$ is defined as

$$MSE = \frac{\sum_{m,n}[I_1(m,n) - I_2(m,n)]^2}{MN} \tag{1.4}$$

where M and N are the number of rows and columns in the input images, respectively.

From equation 1.4, MSE is always non-negative, and a value closer to zero indicates a better degree of similarity. MSE has been the dominant quantitative performance metric in the field of signal processing due to its simple and clear physical meaning and its efficiency in describing similarity. However, one issue is that MSE depends strongly on the image intensity scaling.

Peak Signal-to-Noise Ratio

Peak Signal-to-Noise Ratio (PSNR) can resolve the above-mentioned problem by scaling the MSE based on the intensity range of images. Since some signals have a wide dynamic range, the PSNR is usually represented in terms of the logarithmic decibel scale:

$$PSNR = 10\log_{10}(\frac{R^2}{MSE}) \tag{1.5}$$

where MSE is the mean-squared error which can be calculated using equation 1.4, and R is the maximum fluctuation in pixel values. For example, if the input image has an 8-bit unsigned integer data type, R is 255.

Structural Similarity Index

Structural Similarity Index (SSIM) is another metric for measuring the similarity between two images. It is based on visible structures in the image and compares local patterns of pixel intensities that have been normalized for luminance and contrast [13].

Mathematically, SSIM can be expressed as

$$SSIM(x,y) = [l(x,y)]^{\alpha} \cdot [c(x,y)]^{\beta} \cdot [s(x,y)]^{\gamma} \tag{1.6}$$

where $l(x,y)$, $c(x,y)$ and $s(x,y)$ represent the luminance term, contrast term, and structural term, respectively, based on the following formulas:

$$l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1} \tag{1.7}$$

$$c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2} \tag{1.8}$$

$$s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3} \tag{1.9}$$

where μ_x, μ_y, σ_x, σ_y and σ_{xy} are respectively the local means, standard deviations, and cross-covariance for images x, y.

SSIM is a fractional value ranging from -1 to 1 and indicates a better structural similarity as it approaches 1. With the default settings $\alpha = \beta = \gamma = 1$ and $C_3 = C_2/2$, SSIM can be simplified into:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_x\sigma_y + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \tag{1.10}$$

SSIM can give better image quality assessment than other methods. For the images with different visual quality but close MSE/PSNR values, SSIM can better quantify their image quality variations.

1.2 Coherent Plane-Wave Compounding

As discussed in Chapter 1.1.2, in a conventional multi-focus ultrasound system, each image frame is formed by sequentially scanning the region of interest line per line using focused beams on the transmission end and processing the returning echoes using delay-and-sum (DAS) beamforming per scanline on the receiving end. This improves the image resolution at the cost of a reduction of frame rate, which is only approximately 30 to 60 frames per second (fps) [14]. The applications of such imaging system are limited because they are not capable of capturing or tracking the movement at a relatively high velocity, such as the heart movement during the cardiac cycle [15].

Another option is to insonify the region of interest at once using plane wave (PW) imaging, which enables the system to produce an entire frame simultaneously from one single PW emission. Fig. 1.5 shows the axis convention of a plane-wave imaging system. Several transducer elements forming the ultrasound array are placed at the top of the imaging medium. The x axis represents the transducer locations and z axis, perpendicular to the x axis, is the depth of the medium.

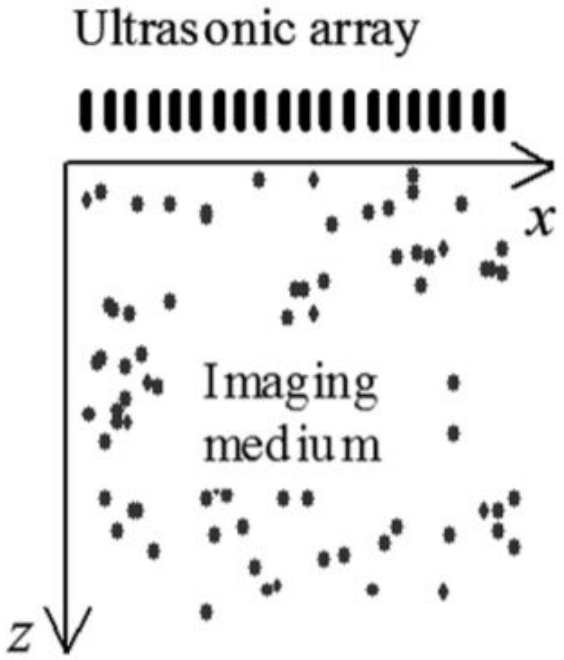


Figure 1.5: Axis convention [4]

In the course of the plane wave imaging, a plane-wave pulse is first emitted by the array into the medium and backscattered when it encounters structure with different acoustic impedances as shown in Fig. 1.6 and 1.7. The returning echoes $\mathrm{RF}(x_1, t)$ are then recorded over time t by the transducers and processed to form an PW image in two spatial dimensions (x, z).

As shown in Fig. 1.8, the traveling duration for a plane wave from the transducer array to point (x, z) and back to a transducer placed in x_1 can be expressed as

$$\tau(x_i, x, z) = (z + \sqrt{z^2 + (x - x_1)^2})/c \tag{1.11}$$

where c is the speed of the sound, which is assumed to be constant in the medium.

By coherently adding the contribution of each scatter, point (x, z) can be represented as

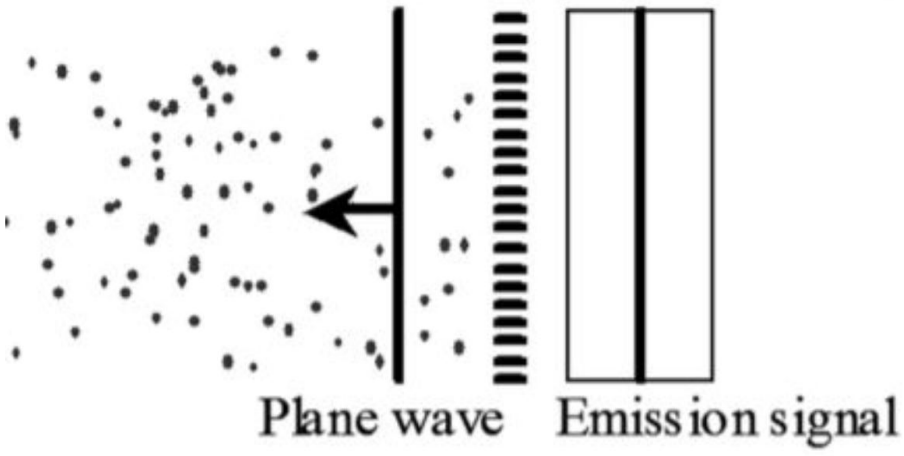


Figure 1.6: PW transmission [4]

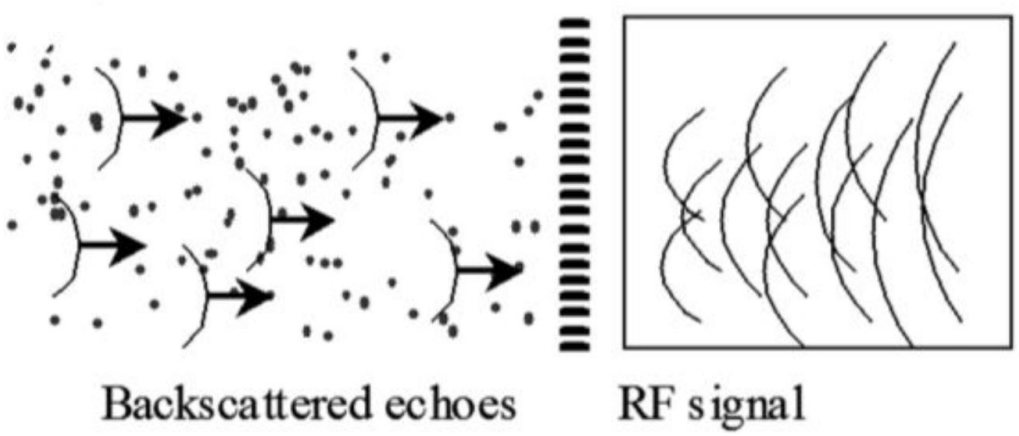


Figure 1.7: Backscattered echoes [4]

$$s(x,z) = \sum_{i=n_{start}}^{n_{end}} RF(x_i, \tau(x_i, x, z)) \tag{1.12}$$

where the transducers between n_{start} and n_{end} represent the elements that contribute to the signal.

In dynamic beamforming, a complete line of image is obtained by computing the delays τ at each depth z. The whole image is then formed after performing dynamic beamformings using the same RF data, as illustrated in Fig. 1.9.

The frame rate for a single PW image reconstruction is much faster than the conventional ultrasound imaging system, but the image quality is compromised due to the lack of focusing during transmission. To tackle the issue of image quality

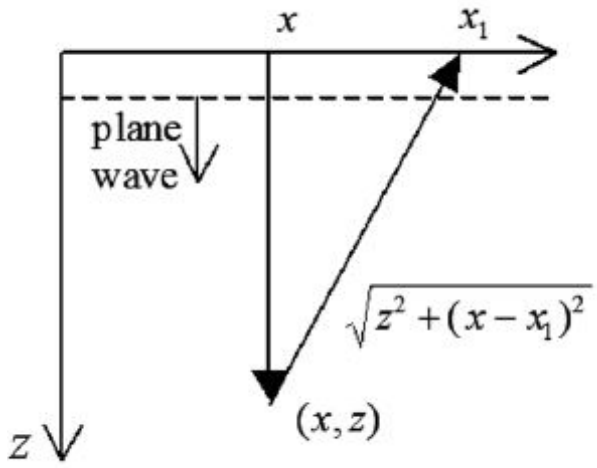


Figure 1.8: Time delay for a plane wave [4]

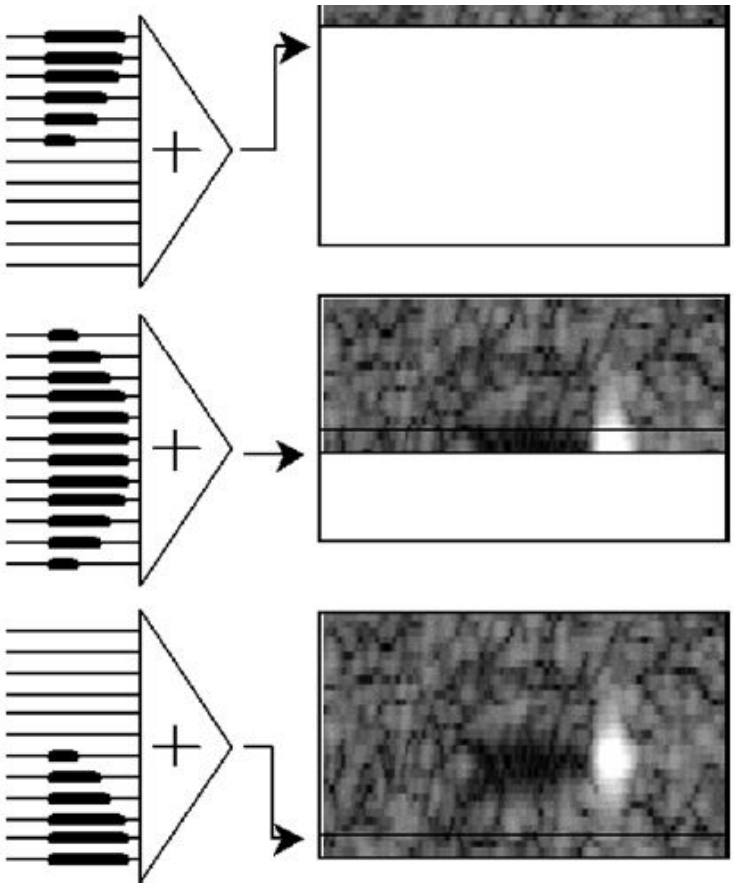


Figure 1.9: Dynamic beamforming [4]

degradation caused by the PW imaging, coherent plane-wave compounding (CPWC) was proposed in [4]. Instead of using only one PW, the transducer emits multiple PWs at various angles in order to get multiple image data sets. For a plane wave with a tilted angle α as shown in Fig. 1.10, the traveling duration τ from the transducer array to point (x, z) and back to a transducer placed in x_1 can be expressed as

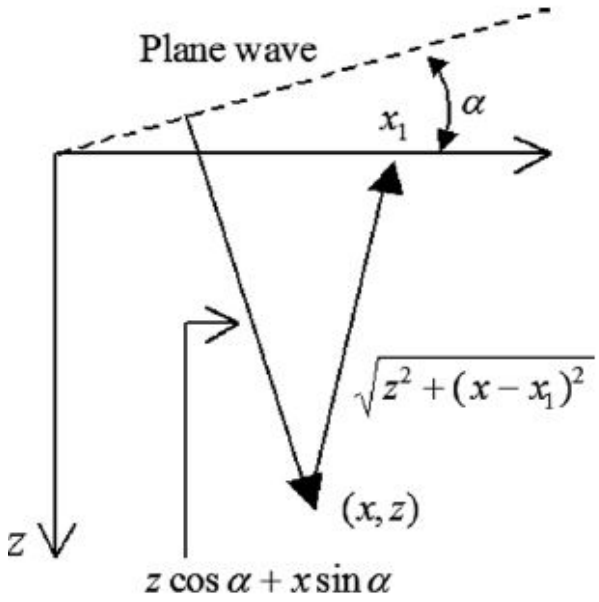


Figure 1.10: Time delay for a plane wave of angle α [4]

$$\tau(\alpha, x_1, x, z) = [(zcos\alpha + xsin\alpha) + \sqrt{z^2 + (x - x_1)^2}]/c \tag{1.13}$$

Each plane wave of a specific tilted emission angle produces an image using equation 1.12 but with the delay derived from equation 1.13. Prior to envelope detection and log compression of the beamformed signals, all images are coherently added to form a compounded one. The quality of the final image can be significantly enhanced due to the use of CPWC. Although emitting multiple plane waves takes more time and thus reduces the frame rate, CPWC can still achieve a relatively high frame rate of over 1000 fps [4], compared to the standard sequential imaging using focused transmission beams.

1.3 Thesis Contributions and Organization

Our work deals with the MATLAB and hardware implementation of one of the Fourier-domain CPWC reconstruction methods proposed in [2], namely, PW Stolt's migration algorithm. We show that it is feasible to obtain high-quality compounded images using hardware-oriented scaled fixed-point calculations in MATLAB. In addition to that, we map both floating- and fixed-point algorithm versions onto a Xilinx FPGA using a combination of two design automation software tools: MATLAB Coder and Vivado HLS. The hardware implementation is achieved without any manual intervention, which offers significant productivity boost during mapping of the MATLAB-based algorithm to the actual FPGA-based hardware.

Chapter 2 provides a background of the fixed-point representation, high-level synthesis (HLS), PW Stolt's migration algorithm and some related work. In Chapter 3, we first summarize the computational procedure of the PW Stolt's migration algorithm. Based on that, we present its MATLAB implementation using both floating-point and fixed-point arithmetic and then perform quantitative evaluation of the results, showing that the fixed-point and floating-point versions of CPWC image reconstruction are practically indistinguishable.

Chapter 4 presents our workflow of using MATLAB Coder and Vivado HLS for the hardware implementation of an algorithmic specification written in MATLAB. Based on that, Xilinx FPGA implementations of both fixed- and floating-point plane-wave Stolt's migration algorithm are detailed. Results show that the fixed-point FPGA implementation is more resource and power efficient and can also operate at a higher clock frequency compared to its floating-point counterpart. Apart from that, we verify the correctness of our Verilog implementation by performing the C/RTL cosimulation of a fixed-point toy case after creating a test bench in C. We also provide an example of exploring different FPGA implementation options by changing the original MATLAB code. In our example, we take advantage of parallel execution of the spatial FFT blocks and implement a reduced-latency RTL design. Finally, our conclusions and suggested future work directions are given in Chapter 5.

Chapter 2

Background

2.1 Fixed-Point Representation

Generally speaking, there are two phases in the course of a DSP application design: algorithm development and hardware/software implementation. In the algorithm development phase, MATLAB is a major tool for system-level modeling and often used to explore ideas, verify assumptions and perform data analysis and optimization given the fact that it is easier to use for numerical calculation and visualization of the results than other languages, such as C/C++.

When developing a DSP algorithm in MATLAB, one usually starts with the floating-point data representation due to its high precision and computation speed in MATLAB. Depending on the requirements of the application, we might also need to convert the code to fixed-point arithmetic after floating-point verification of the algorithm and then resolve issues associated with the fixed-point precision and range limitations. While standard 32-bit or 64-bit floating-point arithmetic can yield more accurate numerical results than using fixed-point arithmetic with word lengths of less than 32 bits, from the hardware point of view, fixed-point calculations are more cost-effective in terms of the resources utilization and power consumption. The use of fixed–point computation is even more appealing in field programmable gate arrays (FPGAs), which can fully take advantage of their fine-grain reconfigurability and work with the fixed-point data of any word length needed. Therefore, the requirements associated with the target application will dictate the choice of using fixed-point or floating-point arithmetic.

Typically, to define a fixed-point data type, three parameters are required: integer

bitwidth, fractional bitwidth, and signedness. Fig 2.1 shows a binary representation of a fixed-point number with a integer length of n, a fractional length of m and an unknown signedness.

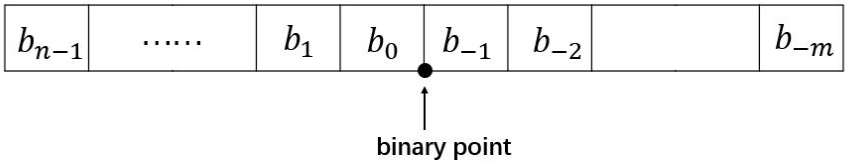


Figure 2.1: Binary representation of a fixed-point number

A fixed-point number can be either signed or unsigned. An unsigned fixed-point number can be converted into a decimal number using formula 2.1:

$$x = \sum_{i=-m}^{n-1} b_i \cdot 2^i \tag{2.1}$$

where b_i is either 0 or 1, and x is the decimal number after conversion.

For a signed binary fixed-point number, two's complement is the most common representation used in hardware due to its consistency of handling both negative and positive numbers without involving any extra logic. An signed fixed-point number represented by two's complement notion can be converted into a decimal number using formula 2.2:

$$x = -b_{n-1} \cdot 2^{n-1} + \sum_{i=-m}^{n-2} b_i \cdot 2^i \tag{2.2}$$

From equation 2.2, we can see that using two's complement notation to represent a signed binary fixed-point number is similar to the unsigned binary representation except that the most significant bit (MSB) has a weight of -2^{n-1} instead of 2^{n-1}.

Generally speaking, a DSP design using fixed-point arithmetic follows these following steps [16]:

1. Develop and verify the algorithm using floating-point arithmetic;

2. Covert to fixed-point arithmetic and resolve issues associated with precision and range limitations;

3. Implement fixed-point algorithm in actual hardware and compare the results obtained from hardware with simulation tools to verify correctness.

2.2 High-Level Synthesis

High-level synthesis (HLS) is an automated design process that interprets an algorithmic description of a desired behavior and creates digital hardware that implements that behavior. Typically, it involves the transformation from an algorithm written in a programming language like C, C++ or MATLAB to a register transfer level (RTL) implementation written in HDL such as Verilog and VHDL.

HLS bridges the gap between algorithm design and hardware design by raising the abstraction level for designing digital circuits. Therefore, it opens up hardware design to system engineers who have little hardware design experience and offers them an opportunity to use hardware for acceleration by moving the computationally intensive parts of the algorithm to an FPGA. Hardware engineers can also benefit from HLS, because they can directly start the hardware design from the high-level code (such as MATLAB/C/C++) written by the algorithm designer rather than re-implementing the design in a different language. As a result, the amount of HDL code to be written by hardware engineers is reduced dramatically, which saves time and minimizes the risk of implementation errors.

Additionally, HLS can be used to quickly explore different design options by changing the original high-level code or the optimization directives from the software tool, which increases the likelihood of finding an optimal implementation. Verification can also become easier if the testbench of the high-level code (not HDL) is in place in the HLS project, because that testbench can be reused to verify the RTL design.

Gerenally speaking, several software tools can be used to achieve an HLS design, such as Vivado HLS [17], Intel HLS Compiler [18], and MATLAB HDL Coder [19].

Vivado HLS is a software tool produced by Xilinx Corporation to perform high-level synthesis. It supports the transformation from C, C++ and SystemC programs into RTL implementations that can be synthesized into a Xilinx FPGA device. Scheduling and binding are two phases that are used for the mapping from C to HDL. Scheduling determines which operations should occur during each clock cycle based on the target FPGA device, clock frequency and optimization directives specified by the users. Binding determines which hardware resource implements each scheduled operation. Vivado HLS synthesizes the C code following the basic rules [17]:

1. Top-level function arguments synthesize into RTL I/O ports;

2. C functions synthesize into blocks in the RTL hierarchy;

3. Arrays in the C code synthesize into block RAMs.

By default, Vivado HLS creates the HDL implementation based on pre-determined settings. However, users can use optimization directives to modify and control the default behavior of the internal logic and I/O ports, which opens up the opportunities for the variations of the hardware implementation from the same C code.

Intel HLS Compiler allows users to convert C++ into RTL code that is optimized for an Intel FPGA, which is a competitor to Xilinx FPGA. Intel HLS Compiler performs important tasks such as generic compiler optimizations, FPGA specific optimizations like data path pipelining and technology specific optimizations. The system of tasks feature allows expression of thread level parallelism within a HLS component. It can be applied to the cases such as executing multiple loops in parallel or sharing an expensive compute block. Compared to traditional RTL development, Intel HLS Compiler provides the the following advantages [18]:

1. Fast and easy verification;

2. Algorithmic development in C++;

3. Automatic integration of RTL verification with a C++ testbench;

4. Powerful microarchitecture optimizations.

MATLAB HDL Coder generates portable, synthesizable Verilog and VHDL code from MATLAB functions, Simulink models, and Stateflow charts. The generated HDL code can be used for FPGA programming or ASIC prototyping and design. HDL coder automatically converts MATLAB Code from the floating-point to fixed-point data representation and generates synthesizable HDL code. Users can also explore the design area and speed tradeoffs by enabling the high-level optimization features. With Simulink, one can also generate HDL code from a library of more than 200 blocks including state flow charts and functions for signal processing and communications. Moreover, HDL coder provides a work flow advisor that is integrated with Xilinx Vivado and Altera Quartus to help users to program FPGA devices from different vendors. Users are also allowed to control the HDL architecture and implementation, highlight critical paths and generate hardware resource utilization estimates, etc [19].

2.3 PW Stolt's Migration Algorithm

There are numerous methods we can use to obtain compounded PW images in two spatial dimensions (z, x), where z and x refer to the axial and lateral coordinates, respectively. For example, one common technique is the standard delay-and-sum (DAS) beamformer that operates in the (t, x) domain, where t represents the temporal axis (sampling time instances). Montaldo *et al.* [4] have shown that the PW image plane can be subdivided into synthetic scanlines, and each scanline can be reconstructed (in parallel with the others) via conventional DAS beamforming applied to all received echoes, subject to appropriate scanline-dependent delays (see section 1.2).

Alternatively, image reconstruction can be done in the *spatio-temporal frequency domain*: the (f, k_x)-domain dataset is remapped into the (k_z, k_x)-domain dataset, where f denotes the temporal frequencies, while k_x and k_z denote the spatial frequencies.

One of the Fourier-domain remapping methods, proposed in [2] offers substantially lower computational latency compared to conventional DAS beamforming. The following discussion in this section briefly summarizes this Fourier-domain migration method for reconstructing coherently compounded PW images, borrowing from [2] with a slight change in notation to streamline the presentation.

Let θ represent a PW emission angle, and let $P(t, z, x, \theta)$ denote the resulting acoustic wavefield. Given the wavefield $P(t, 0, x, \theta)$ recorded over time at the surface (i.e., at depth $z = 0$), we want to reconstruct the subsurface image dataset $P(0, z, x, \theta)$ at time $t = 0$. This goal can be accomplished using Fourier-domain interpolation as follows.

Let $\Psi(f, 0, k_x, \theta)$ and $\widetilde{\Psi}(0, k_z, k_x, \theta)$ denote the Fourier transforms of known $P(t, 0, x, \theta)$ and unknown $P(0, z, x, \theta)$:

$$\Psi(f, 0, k_x, \theta) = \iint P(t, 0, x, \theta) e^{-j2\pi(k_x x + ft)} dx dt, \tag{2.3}$$

$$P(0, z, x, \theta) = \iint \widetilde{\Psi}(0, k_z, k_x, \theta) e^{j2\pi(k_x x + k_z z)} dk_x dk_z. \tag{2.4}$$

We have $\Psi(f, 0, k_x, \theta)$ as an input, generated by the Fourier transform of $P(t, 0, x, \theta)$. We need to obtain $\widetilde{\Psi}(0, k_z, k_x, \theta)$ from $\Psi(f, 0, k_x, \theta)$, so that sought $P(0, z, x, \theta)$ can be computed via the inverse Fourier transform of $\widetilde{\Psi}$. In [2], (extending classic Stolt's migration method [20]), the *intermediate spectrum* is produced using equation 2.5

$$\widetilde{\Psi}^*(0, k_z, k_x, \theta) = A(k_z, k_x, \theta) \cdot \Psi\left(f_{\text{mig}}(k_z, k_x, \theta), 0, k_x, \theta\right), \tag{2.5}$$

with the values of f_{mig} and A determined by

$$f_{\text{mig}}(k_z, k_x, \theta) = \frac{ck_z}{1 + \cos(\theta)} \left[1 + (k_x/k_z)^2\right] \ , \tag{2.6}$$

$$A(k_z, k_x, \theta) = \frac{c}{1 + \cos(\theta)} \left[1 - (k_x/k_z)^2\right] \ , \tag{2.7}$$

where c is the speed of sound.

Applying the inverse Fourier transform to $\widetilde{\Psi}^*(0, k_z, k_x, \theta)$ will yield a preliminary image dataset $P(0, z^*, x, \theta)$, from which we can obtain $P(0, z, x, \theta)$ by repositioning data points at locations (z^*, x) to their new coordinates $(z = z^* + x\tan(\theta)/2, x)$ [2].

To compound multiple angle-specific $P(0, z, x, \theta_n)$ over a given angular set $\{\theta_n \mid n = 1, 2, ..., N_a\}$, we perform the following summation [2]:

$$C(k_z, x) = \sum_{n=1}^{N_a} e^{j\pi k_z x \tan(\theta_n)} \cdot \overline{\Psi}(k_z, x, \theta_n), \tag{2.8}$$

$$\overline{\Psi}(k_z, x, \theta_n) = \int \widetilde{\Psi}^*(0, k_z, k_x, \theta_n) e^{j2\pi k_x x} dk_x, \tag{2.9}$$

where $\overline{\Psi}(k_z, x, \theta_n)$ represents the 1D inverse Fourier transform of $\widetilde{\Psi}^*(0, k_z, k_x, \theta_n)$ along the k_x-axis, and $C(k_z, x)$ is the compounded (k_z, x)-domain dataset. We get the final image dataset, denoted by $D(z, x)$, via the 1D inverse Fourier transform of $C(k_z, x)$ along the k_z-axis:

$$D(z, x) = \int C(k_z, x) e^{j2\pi k_z z} dk_z. \tag{2.10}$$

2.4 Related Work

Besides PW Stolt's migration algorithm discussed in section 2.3, Garcia *et al.* [21] have proposed an alternative modification of classic Stolt's method by fitting the exploding reflector model (ERM) velocity model. They account for PW emissions by changing $c/2$ to $v = c/\sqrt{1 + \cos(\theta) + \sin^2(\theta)}$ and modifying Stolt's original frequency remapping formula. Other modifications introduced by Garcia *et al.* [21] include multiplying the (x, f)-domain data by $\exp(j2\pi f x \sin(\theta)/c)$ before migration and multiplying the (k_x, z)-domain data by $\exp(j2\pi k_x z \sin(\theta)/(2 - \cos(\theta)))$ after mi-

gration. For our method discussed in section 2.3, the (x, k_z)-domain data is multiplied by $\exp(j\pi k_z x \tan(\theta))$ after migration. The key difference between Garcia's approach and the one in section 2.3 is that the former relies on approximating PW travel-time hyperbolas to those arising from synthetic-aperture ERM assumptions, whereas the latter adjusts "explosion" time of reflectors.

Based on the theory of limited-diffraction beams, Lu [22] reported another frequency-domain method, which models the propagation of two-way scalar wave in a weakly attenuating medium. In the case of steered PW imaging, the (k_x, f)-domain data is interpolated along the f-axis using the formula $c(k_x^2+k_z^2)/(2k_x \sin(\theta)+2k_z \cos(\theta))$. Prior to such frequency remapping, the (x, f)-domain data is multiplied by $\exp(j2\pi f x \sin(\theta)/c)$ to account for PW steering delays (as in Garcia's method). Note that for $\theta = 0$, Lu's f-to-k_z remapping formula becomes $c(k_x^2 + k_z^2)/(2k_z)$, which is equivalent to equation 2.6 used by our Stolt's method discussed in section 2.3, except for the presence of the scaling factor $(c/2)(1 - (k_x/k_z)^2)$. However, for $\theta \neq 0$, the remapping formula used by Lu's method and the one in section 2.3 are different. Another important difference is that Lu's pointwise multiplications of the (x, f)-domain data by $\exp(j2\pi f x \sin(\theta)/c)$ before remapping are discarded; instead, our Stolt's method discussed in section 2.3 multiplies the (x, k_z)-domain data by $\exp(j\pi k_z x \tan(\theta))$ after remapping. Other literature related to Fourier domain methods for PW ultrasound image reconstruction includes [23, 24, 25, 26]. These works, however, are closed related to Lu's method described above.

Although PW ultrasound imaging has shown potential in reaching a high frame rate, its real-time implementation is technically challenging due to the massive amount of data to be processed. One particular approach is to take advantage of graphics processing units (GPUs) [27], [28]. Yiu et al. realized plane wave compounding and synthetic aperture imaging at more than 3000 frames per second (fps) using a three-GPU system [29]. Choe et al. developed a GPU-based real-time imaging software suite for medical ultrasound which is capable of reconstructing real-time ultrasound images using various imaging schemes including conventional beamforming, synthetic beamforming, and plane wave compounding [30]. Hewener et al. integrated the plane wave beamforming and imaging into the mobile ultrasound processing application on the iOS platform using parallel GPU hardware [31].

Another promising alternative in ultrasound imaging is to use field-programmable gate array (FPGA) devices, which can achieve high-performance computing with low power consumption [32]. The high amount of hardware resources available in

modern FPGAs also enables the development of more complex hardware designs [33]. Research related to FPGA implementation of ultrasound imaging systems has been conducted in [33, 34, 35, 36]. However, the FPGA implementation for CPWC imaging is still an unexplored area, which has motivated the work presented here.

Chapter 3

MATLAB Implementation of PW Stolt's Migration Algorithm

In this Chapter, we will present both floating-point and fixed-point MATLAB implementations of PW Stolt's migration algorithm. Based on the algorithm introduction discussed in section 2.1, we first summarize the computational procedure and introduce relevant variables. Each step of the algorithm is then implemented using floating- and fixed-point arithmetic in MATLAB. Results in section 3.4 show that the reconstructed images obtained using fixed-point arithmetic are very close to their respective floating-point counterparts.

3.1 Introduction

Before we describe the PW Stolt's migration algorithm implementation in MATLAB, the pseudocode in Fig. 3.1 outlines its computational procedure discussed in section 2.1.

The 3D input $\texttt{P}[\,\cdot\,]$ is a raw RF channel dataset recorded over N_t time instances $\{t_l = 0, \Delta_t, ..., (N_t - 1)\Delta_t\}$, N_x sensor locations $\{x_m = -\frac{N_x}{2}\Delta_x, ..., 0, (\frac{N_x}{2} - 1)\Delta_x\}$, and N_a PW emission angles $\{\theta_n \mid n = 1, 2, ..., N_a\}$, i.e., $\texttt{P}[\,\cdot\,]$ represents $P(t, 0, x, \theta)$. The other 3D inputs $\texttt{M}[\,\cdot\,]$, $\texttt{A}[\,\cdot\,]$, and $\texttt{E}[\,\cdot\,]$ represent $f_{\text{mig}}(k_z, k_x, \theta)$, $A(k_z, k_x, \theta)$, and $\exp(j\pi k_z x \tan(\theta))$, respectively. The 2D "analytic signal" output $\texttt{H}[\,\cdot\,]$ is obtained via the Hilbert transform (along the z-axis) of the final dataset $D(z, x)$ represented by $\texttt{D}[\,\cdot\,]$ (lines 12 and 13 in Fig. 3.1). This output is useful for subsequent data processing (e.g., detecting the envelope), and it is computed by the HILBERT function for each

x_m value.

Input: Raw dataset $\mathtt{P}[\,\cdot\,]$, map $\mathtt{M}[\,\cdot\,]$, scaler $\mathtt{A}[\,\cdot\,]$, phase shift $\mathtt{E}[\,\cdot\,]$
Output: Hilbert-transformed compounded image dataset $\mathtt{H}[\,\cdot\,]$

1. $\mathtt{C} \leftarrow \mathbf{0}$;
2. **for** $n = 1 : N_a$ **do** {
3. **for** $m = 1 : N_x$ **do** { $\mathtt{F}[:, m] \leftarrow \mathrm{FFT}(\mathtt{P}[:, m, n], N_t^{\mathrm{FFT}})$; }
4. **for** $l = 1 : N_t^{\mathrm{FFT}}$ **do** { $\mathtt{F}[l, :] \leftarrow \mathrm{FFT}(\mathtt{F}[l, :], N_x^{\mathrm{FFT}})$; }
5. **for** $m = 1 : N_x^{\mathrm{FFT}}$ **do** {
6. $\mathtt{K}[:, m] \leftarrow \textsc{Remap}(\mathtt{F}[:, m], \mathtt{M}[:, m, n])$;
7. $\mathtt{K}[:, m] \leftarrow \mathtt{A}[:, m, n] \times \mathtt{K}[:, m]$; }
8. **for** $l = 1 : N_t^{\mathrm{FFT}}$ **do** { $\mathtt{K}[l, :] \leftarrow \mathrm{IFFT}(\mathtt{K}[l, :], N_x^{\mathrm{FFT}})$; }
9. **for** $m = 1 : N_x$ **do** { $\mathtt{K}[:, m] \leftarrow \mathtt{E}[:, m, n] \times \mathtt{K}[:, m]$; }
10. $\mathtt{C} \leftarrow \mathtt{C} + \mathtt{K}$; }
11. **for** $m = 1 : N_x$ **do** {
12. $\mathtt{D}[:, m] \leftarrow \mathrm{IFFT}(\mathtt{C}[:, m], N_t^{\mathrm{FFT}})$;
13. $\mathtt{H}[:, m] \leftarrow \textsc{Hilbert}(\mathtt{D}[:, m])$; }

Figure 3.1: Spectral migration algorithm [5]

The 1D temporal and spatial Fourier transforms and their inverses are computed by the FFT and IFFT functions, using the power-of-2 transform lengths denoted by N_t^{FFT} and N_x^{FFT}. Upon execution of lines 3 and 4 in Fig. 3.1, we obtain the (f, k_x)-domain spectrum $\mathtt{F}[\,\cdot\,]$ of size $N_t^{\mathrm{FFT}} \times N_x^{\mathrm{FFT}}$. Then, for each k_x bin indexed by $m = 1, 2, ..., N_x^{\mathrm{FFT}}$, we remap the f-axis points to the k_z-axis points according to $\mathtt{M}[:, m, n]$ and multiply the resulting data pointwise by $\mathtt{A}[:, m, n]$ (lines 6 and 7 in Fig. 3.1). Next, for each k_z bin indexed by $l = 1, 2, ..., N_t^{\mathrm{FFT}}$, we transform $\mathtt{K}[\,\cdot\,]$ back to the (k_z, x) domain (line 8 in Fig. 3.1) and apply appropriate phase shifts $\exp(j\pi k_z x_m \tan(\theta_n))$ specified by $\mathtt{E}[:, m, n]$. Finally, each angle-specific $\mathtt{K}[\,\cdot\,]$ is added to $\mathtt{C}[\,\cdot\,]$ that represents the compounded 2D dataset $C(k_z, x)$ of size $N_t^{\mathrm{FFT}} \times N_x$. After processing all angles, the inverse Fourier transform of $\mathtt{C}[\,\cdot\,]$ along the k_z-axis yields the (z, x)-domain image dataset $\mathtt{D}[\,\cdot\,]$. Then, the complex-valued output $\mathtt{H}[\,\cdot\,]$ of size $N_z \times N_x$ is obtained by putting $\mathtt{D}[\,\cdot\,]$ as the real part and the Hilbert transform of $\mathtt{D}[\,\cdot\,]$ as the imaginary part. Taking the absolute value of $\mathtt{H}[\,\cdot\,]$ produces the envelope (needed for B-mode image generation).

In the following MATLAB implementation, all the complex-number computations are divided into real and imaginary parts, thus variable names such as `F`, `K`, `D`, `C` and `H` in Fig. 3.1 will be changed to `ReF/ImF`, `ReK/ImK`, `ReD/ImD`, `ReC/ImC` and `ReH/ImH` accordingly.

3.2 Floating-Point MATLAB Implementation

Based on the procedure discussed in section 2.1, we first implement the Stolt's method in MATLAB using float-point arithmetic. All integer variables (such as array indices and FFT sizes) are set to 32-bit long and non-integer numerical variables (such as `P` and `A`) are in 32-bit (single) floating-point format in the following code snippets of this section.

Listing 3.1 shows the MATLAB implementation of the temporal FFT which transforms the data from t-x to f-x domain, corresponding to line 3 in Fig. 3.1.

```
% -- transform from t-x to F-x domain
% P, n -- see Fig 3.1
% NtFFT_half -- half of temporal FFT size
% ReF/ImF -- real/imaginary part of dataset F in Fig. 3.1
% Re/Im -- intermediate 1-dimensional real/imaginary data
% BR_NtFFT -- reversed order of temporal FFT
% bitsra -- bit shift right
% split_radix_FFT_temporal -- temporal split-radix FFT function block
for index = 1:2:Nx-1        % Nx/2 iterations
    Re = P(:,index,n);
    Im = P(:,index+1,n);
    % -- perform temporal FFT from t-x to F-x
    [Re,Im] = split_radix_FFT_temporal(Re,Im,NtFFT);
    % -- bit-reverse output and (f, x)-domain spectrum construction
    ReF(1,index) = Re(1); ImF(1,index) = 0;  %BR_NtFFT(1) = 1
    ReF(1,index+1) = Im(1); ImF(1,index+1) = 0;  %BR_NtFFT(1) = 1
    for k = 2:NtFFT_half
        BRk = BR_NtFFT(k);
        BRk_pos = BR_NtFFT(NtFFT-k+2);
        ReF(k,index) = bitsra(Re(BRk) + Re(BRk_pos), 1);
        ImF(k,index) = bitsra(Im(BRk) - Im(BRk_pos), 1);
        ReF(k,index+1) = bitsra(Im(BRk_pos) + Im(BRk), 1);
        ImF(k,index+1) = bitsra(Re(BRk_pos) - Re(BRk), 1);
    end
end
```

Listing 3.1: Temporal FFT MATLAB snippet

The temporal Fourier transform calculations are performed by the 1D split-radix FFT blocks. Note that the number of iterations over both indices $m = 1, 2, ..., N_x$

and $l = 1, 2, ..., N_t^{\mathrm{FFT}}$ is cut in half. Given that the input $\mathtt{P}[\,\cdot\,]$ is real-valued, its (f, x) spectrum is symmetric. Thus, the negative frequency half of the (f, x) spectrum contains redundant information with respect to the positive frequency half, i.e., we only need to keep its positive-f portion of the (f, x) spectrum. Moreover, we can transform two real-valued sequence simultaneously by computing only one complex-valued FFT. To be more specific, if $x[n]$ and $y[n]$ are real-valued vectors, their FFT results, $X[k]$ and $Y[k]$, have an even real part and an odd imaginary part. Since an FFT is a linear transform, the FFT of $z[n] = x[n] + jy[n]$ can be expressed as

$$FFT\left\{z[n]\right\} = Z[k] = Z_r[k] + jZ_i[k], \tag{3.1}$$

where r and i represent real and imaginary part, respectively. $X[k]$ and $Y[k]$ can then be derived as [37]

$$X_r[k] = \frac{1}{2}(Z_r[k] + Z_r[N-k]), \quad k = 0, 1, ...\frac{N}{2}; \tag{3.2}$$

$$X_i[k] = \frac{1}{2}(Z_i[k] - Z_i[N-k]), \quad k = 0, 1, ...\frac{N}{2}; \tag{3.3}$$

$$Y_r[k] = \frac{1}{2}(Z_i[N-k] + Z_i[k]), \quad k = 0, 1, ...\frac{N}{2}; \tag{3.4}$$

$$Y_i[k] = \frac{1}{2}(Z_r[N-k] - Z_r[k]), \quad k = 0, 1, ...\frac{N}{2}. \tag{3.5}$$

For k =0, the above equations can be further simplified to

$$X[0] = Z_r[0] \quad \text{and} \quad Y[0] = Z_i[0]. \tag{3.6}$$

Using the computational trick mentioned above, we can process two columns of t-axis data in pairs: the former ($P(:, index, n)$) being the real part of the FFT input, while the latter ($P(:, index{+}1, n)$) being the corresponding imaginary part. Therefore, the number of iterations is reduced by half to $N_x/2$ iterations (see line 9 in listing 3.1). After bit reversal and spectrum construction using equations 3.2-3.6 (see line 15-24 in listing 3.1), the positive-f portion of (f, x)-domain spectrum is obtained. Note that the scaling factor of $1/2$ in equations 3.2-3.6 is performed by the MATLAB function *bitsra* that implements an equivalent arithmetic bit shift to the right.

Next are the spatial FFTs (line 4 in Fig. 3.1) transforming the data from (f, x) to

(f, k_x) domain, which is shown in listing 3.2 where the (f, k_x) half-spectrum is computed along x-axis for each f bin for the subsequent remapping and scaling process.

```
% -- transform from F-x to F-Kx domain
% NtFFT_half -- half of temporal FFT size
% ReF/ImF -- real/imaginary part of dataset F in Fig. 3.1
% Re/Im -- intermediate 1-dimensional real/imaginary data
% BR_NxFFT -- reversed order of spatial FFT
% split_radix_FFT_spatial -- spatial split-radix FFT function block
for index = 1:NtFFT_half     % NtFFT/2 iterations
    Re = ReF(index,:);
    Im = ImF(index,:);
    % -- perform spatial FFT from F-x to F-Kx
    [Re,Im] = split_radix_FFT_spatial(Re,Im,NxFFT);
    % -- bit reverse output
    ReF(index,1:NxFFT) = Re(BR_NxFFT(1:NxFFT));
    ImF(index,1:NxFFT) = Im(BR_NxFFT(1:NxFFT));
end
```

Listing 3.2: Spatial FFT MATLAB snippet

Then, given a particular k_x bin, we let the k_z-axis data values equal those found (via linear interpolation) at $f_{\text{mig}}(k_z, k_x, \theta)$ (`M`$[\cdot]$ in Fig. 3.1) and scaled by $A(k_z, k_x, \theta)$ (`A`$[\cdot]$ in Fig, 3.1). Listing 3.3 shows the MATLAB snippet implementing the remapping and scaling process corresponding to lines 6 and 7 in Fig. 3.1. We first call the function *get_index* (line 12 in listing 3.3) for each k_x bin and find the corresponding bin in `M` and `A`, both of which have been calculated beforehand based on equation 2.6 and 2.7, respectively. Then, we interpolate `F` (i.e.,`ReF` and `ImF`) using `M` and then scale it by `A` to get the (k_z, k_x)-domain spectrum `K` (i.e.,`ReK` and `ImK`) of size $N_t^{\text{FFT}} \times N_x^{\text{FFT}}$ (line 13-21 in listing 3.3).

```
% -- remap from F to Kz and scale in Kz-Kx domain
% NtFFT_half -- half of temporal FFT size
% M, A, n -- see Fig. 3.1
% Re_v/Im_v -- intermediate real/imaginary value
% ReF/ImF/ReK/ImK -- real/imaginary part of F and K in Fig. 3.1
% Re/Im -- intermediate 1-dimensional real/imaginary data
% get_index -- function for index_mod-index conversion
% remap -- interpolation function block
```

```
for index_mod = 1:NxFFT
    Re = ReF(:,index_mod);
    Im = ImF(:,index_mod);
    index = get_index (index_mod);
    for j = 1: NtFFT_half
        % -- perform interpolation from F to Kz
        [Re_v,Im_v] = remap(Re, Im, M(j,index,n));
        % -- scale Re_v/Im_v by A(j,index,n)
        Re(j) = Re_v * A(j,index,n);
        Im(j) = Im_v * A(j,index,n);
    end
    ReK(1:NtFFT_half,index_mod) = Re(1:NtFFT_half);
    ImK(1:NtFFT_half,index_mod) = Im(1:NtFFT_half);
end
```

Listing 3.3: Remapping MATLAB snippet

After that, for each k_z bin with index ranging from 1 to $N_t^{\mathrm{FFT}}/2$, we reuse the same spatial FFT function block as the one in Listing 3.2 to perform IFFT along k_x-axis by conjugating both the input and output of the FFT function. This corresponds to line 8 in Fig. 3.1, which transforms `K` back to the (k_z, x) domain. The MATLAB code implementing the spatial IFFTs is shown in Listing 3.4.

```
% -- transform from Kz-Kx back to Kz-x domain
% NtFFT_half -- half of temporal FFT size
% ReK/ImK -- real/imaginary part of dataset K in Fig. 3.1
% Re/Im -- intermediate 1-dimension real/imaginary data
% BR_NxFFT -- reversed order of spatial FFT
% split_radix_FFT_spatial -- spatial split-radix FFT function block
for index = 1:NtFFT_half
    % -- conjugate input for IFFT
    Re = ReK(index,:);
    Im = -ImK(index,:);
    [Re,Im] = split_radix_FFT_spatial(Re,Im,NxFFT);
    % -- bit-reverse and conjugate output
    ReK(index,1:Nx) = Re(BR_NxFFT(1:Nx));
    ImK(index,1:Nx) = -Im(BR_NxFFT(1:Nx));
end
```

Listing 3.4: Spatial IFFT MATLAB snippet

The next step is to apply appropriate phase shifts $\exp(j\pi k_z x_m \tan(\theta_n))$ to the resulting (k_z, x) half-spectrum. To this end, CORDIC-based [38] phase rotations are performed as seen in Listing 3.5 by using the embedded MATLAB function (in line 16). Note that the phase values are provided by $shiftZ$ in Listing 3.5 (as opposed to the input $\texttt{E}[\cdot]$ in Fig. 3.1).

```
1  % -- apply phase rotation
2  % NtFFT_half -- half of temporal FFT size
3  % ReK/ImK -- real/imaginary part of K in Fig. 3.1
4  % shiftZ -- phase shiftZ value from outside
5  % twoPI -- equal to 2
6  % cordicrotate -- embedded MATLAB function for cordic rotation
7  for nx = 1:Nx
8      ps = single(0);
9      % -- determine phase shift
10     ps = ps + shiftZ(nx);
11     if ps >= twoPI, ps = ps - twoPI;
12     elseif ps < -twoPI, ps = ps + twoPI;
13     end
14     for nKz = 2:NtFFT_half
15         % -- perform CORDIC-based phase rotation
16         vC = cordicrotate(ps, complex(ReK(nKz,nx), ImK(nKz,nx)));
17         ReK(nKz,nx) = real(vC);
18         ImK(nKz,nx) = imag(vC);
19     end
20 end
```

Listing 3.5: Phase rotation MATLAB snippet

After the phase-rotation step, each angle-specific half-spectrum in the (k_z, x) domain is compounded with the others (line 10 in Fig. 3.1) to produce half-sized `ReC` and `ImC`, representing the real and imaginary parts of half-sized $\texttt{C}[\cdot]$ over positive-k_z bins. Instead of expanding such $\texttt{C}[\cdot]$ into the full-sized symmetric (k_z, x) spectrum to undergo the N_t^{FFT}-point IFFTs, followed by the Hilbert transforms (lines 12 and 13 in Fig. 3.1), we obtain the desired "analytic signal" output $\texttt{H}[\cdot]$ (`ReH` and `ImH`) directly from half-sized $\texttt{C}[\cdot]$ (`ReC` and `ImC`). For each $m = 1, 2, ..., N_x$, we first expand

$\mathtt{C}[\,\cdot\,]$ into its full-sized "analytic" version $\mathtt{C}^*[\,\cdot\,]$, based on [39]:

$$\mathtt{C}^*[l,m] \leftarrow \begin{cases} \mathtt{C}[l,m], & l=1; \\ \mathtt{C}[l-1,m], & l=\frac{N_t^{\mathrm{FFT}}}{2}+1; \\ 2\mathtt{C}[l,m], & l=2,3,...,\frac{N_t^{\mathrm{FFT}}}{2}; \\ 0, & l=\frac{N_t^{\mathrm{FFT}}}{2}+2,\frac{N_t^{\mathrm{FFT}}}{2}+3,...,N_t^{\mathrm{FFT}}. \end{cases} \tag{3.7}$$

Then, we compute the N_t^{FFT}-point IFFTs of the individual column vectors of $\mathtt{C}^*[\,\cdot\,]$ to get the compounded image dataset:

$$\mathtt{H}[:,m] \leftarrow \mathrm{IFFT}(\mathtt{C}^*[:,m], N_t^{\mathrm{FFT}}). \tag{3.8}$$

The MATLAB snippet for the "analytic signal" output construction and Fourier transform computation is shown in listing 3.6. Note that we have effectively eliminated line 13 in Fig. 3.1 by replacing symmetric $\mathtt{C}[\,\cdot\,]$ with $\mathtt{C}^*[\,\cdot\,]$ according to [39] (line 10-16 in listing 3.6). After computing the IFFTs along the k_z-axis for each x bin using the same temporal split-radix FFT block with input and output conjugation, we get the (z,x)-domain real and imaginary outputs `ReH` and `ImH`, followed by the computation of the envelope, which is the absolute values of $\mathtt{H}[\,\cdot\,]$ (line 23 in listing 3.6).

```
1  % -- construct and computer "analytic" signal
2  % NtFFT_half -- half of temporal FFT size
3  % Re/Im -- intermediate 1-dimension real/imaginary data
4  % ReC/ImC/ReH/ImH -- real/imaginary part of C and H in Fig. 3.1
5  % AbsH -- absolute values of H
6  % bitsra -- bit shift right
7  % split_radix_FFT_temporal -- temporal split-radix FFT function block
8  for index = 1:Nx
9      % -- expand into "analytic" spectrum
10     Re(1) = bitsra(ReC(1,index),1);
11     Re(2:NtFFT_half) = ReC(2:NtFFT_half,index);
12     Re(NtFFT_half+1) = bitsra(ReC(NtFFT_half,index),1);
13     % -- conjugate input
14     Im(1) = -bitsra(ImC(1,index),1);
15     Im(2:NtFFT_half) = -ImC(2:NtFFT_half,index);
16     Im(NtFFT_half+1) = bitsra(ImC(NtFFT_half,index),1);
```

```
17      % -- perform IFFT
18      [Re,Im] = split_radix_FFT_temporal(Re, Im, NtFFT);
19      % -- bit-reverse and conjugate output
20      ReH(1:Nz,index) = Re(BR_NtFFT(1:Nz));
21      ImH(1:Nz,index) = -Im(BR_NtFFT(1:Nz));
22      % -- compute absolute values for envelope
23      AbsH(1:Nz,index) = sqrt(ReH(1:Nz,index).^2 + ImH(1:Nz,index).^2);
24  end
```

Listing 3.6: "Analytic" spectrum construction and temporal IFFT MATLAB snippet

3.3 Fixed-point MATLAB Implementation

This section discusses the fixed-point MATLAB implementation of the PW Stolt's migration algorithm. The fixed-point implementation follows the same basic computation procedure as the floating-point implementation examined in section 3.2, except for the introduction of scaling factors and equalization blocks. Fig. 3.2 depicts the scaled fixed-point computational flow implementing the Fourier-domain reconstruction algorithm from the section 3.1 (see Fig. 3.1).

As in the floating-point implementation, the fixed-point Fourier transform calculations are also performed by the 1D split-radix FFT blocks [37]. The difference is that during fixed-point computations, the real/imaginary data values are restricted to the interval $[-1, +1]$. Whenever these values fall outside the allowed limits, they undergo binary scaling (i.e., division by some fitting power of 2) to enforce the range restriction.

Listing 3.7 is the MATLAB example showing how the binary scaling works during the split-radix FFT computation. The basic idea behind the split-radix FFT is the application of a radix-2 index map to the even-indexed terms and a radix-4 map to the odd-indexed terms, which results in a L-shaped "butterfly". The MATLAB code implementing the split-radix FFT is based on the FORTRAN program given in [37], and the snippet that has been selected as the example in listing 3.7 is part of the FFT computation associated with X, Y and S, all of which are arrays of size 4. Array X and Y consist of four elements from real and imaginary part of the data, respectively. Variable S, which is the scaling factor array, keeps track of the scaling factor for each element pair of X and Y. Before being fed into the FFT function, each element pair $X(k)$ and $Y(k)$ get equalized based on the difference between the

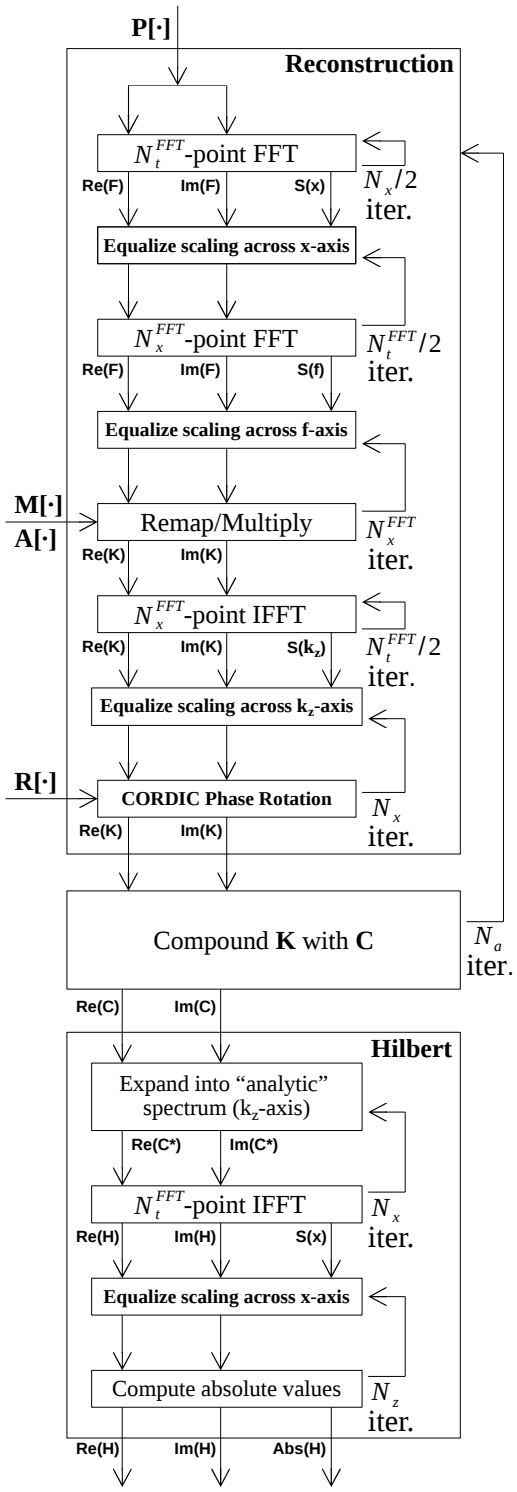


Figure 3.2: Fixed-point scaled reconstruction [5]

corresponding scaling factor $S(k)$ and maximum scaling factor sI to ensure that all elements from X and Y have the same level of scaling (line 11-15 in listing 3.7). After calling the function, the outputs of that function undergo scaling again to be restricted to interval $[-1, +1]$ by imposing different scaling factors to each individual element pair (line 18-29 in listing 3.7). Finally, the maximum scaling factor element from S is recorded as $maxS$ for future use (line 30 in listing 3.7).

```
1  % -- computation under binary scaling
2  % X, Y -- input arguments
3  % S -- scaling factor array
4  % main_computation -- function for main FFT computation
5
6  X = [X1,X2,X3,X4];
7  Y = [Y1,Y2,Y3,Y4];
8  S = [S1,S2,S3,S4];
9  sI = max(S);
10 % -- equalize input scaling
11 for k = 1:4
12     dS = sI - S(k);
13     XI(k) = bitsra(X(k),dS);
14     YI(k) = bitsra(Y(k),dS);
15 end
16     [XO, YO] = main_computation(XI,YI);
17 % -- perform output scaling
18 for k = 1:4
19     maxAbs = max(abs(XO(k)),abs(YO(k)));
20     if maxAbs <= 1, dS = 0;
21     elseif (maxAbs > 1)&&(maxAbs <= 2) dS = 1;
22     elseif (maxAbs > 2)&&(maxAbs <= 4) dS = 2;
23     elseif (maxAbs > 4)&&(maxAbs <= 8) dS = 3;
24     else dS = 4;
25     end
26     Xnew(k) = bitsra(XO(k),dS);
27     Ynew(k) = bitsra(YO(k),dS);
28     Snew(k) = sI + dS;
29 end
30     maxS = max(Snew);
```

Listing 3.7: Scaling MATLAB snippet

After completing the temporal FFTs over all x bins, we have the scaling factors

$S(x)$ associated with the f-axis data vectors. Before starting the spatial FFTs, the scaling equalization is performed as shown in listing 3.8. The x-axis data vectors is equalized using the maximum of $S(x)$, denoted by $maxS_x$ (line 13 in listing 3.8), which has been calculated from the previous step.

```
1  % -- transform from F-x to F-Kx domain
2  % NtFFT_half -- half of temporal FFT size
3  % ReF/ImF -- real/imaginary part of dataset F in Fig. 3.1
4  % Re/Im -- intermediate 1-dimensional real/imaginary data
5  % maxS -- maximum scaling factor from previous step
6  % split_radix_FFT_spatial -- spatial split-radix FFT function block
7  % bitsra -- bit shift right
8  for index = 1:NtFFT_half
9      Re = ReF(index,:);
10     Im = ImF(index,:);
11     % -- scaling equalization
12     for x = 1:Nx
13         dS = maxS_x - S(x);
14         Re(x) = bitsra(Re(x),dS);
15         Im(x) = bitsra(Im(x),dS);
16     end
17     [Re,Im] = split_radix_FFT_spatial(Re,Im,NxFFT);
18 ......
19
20 end
```

Listing 3.8: Equalization MATLAB snippet

Similar to the temporal FFTs, the spatial FFTs are also performed with the real/imaginary data values restricted to the interval $[-1, +1]$ by using binary scaling as shown in Fig. 3.7. After completing the spatial FFTs over all positive-f bins, we obtain the scaling factors $S(f)$ associated with k_x-axis to equalize the f-axis data vectors prior to frequency remapping. In addition to that, we find the maximum scaling factor element from $S(f)$, which is recorded as $maxS_f$ for use in subsequent calculations.

The Remap/Multiply block in Fig. 3.2 implements lines 6 and 7 in Fig. 3.1. As in the floating-point implementation, for each k_x bin, we map k_z-axis data values to those found at $f_{\text{mig}}(k_z, k_x, \theta)$ via linear interpolation and scaled by $A(k_z, k_x, \theta)$. In Fig. 3.2, the values of $f_{\text{mig}}(k_z, k_x, \theta)$ and $A(k_z, k_x, \theta)$ are provided by the inputs

$\mathtt{M}[\,\cdot\,]$ and $\mathtt{A}[\,\cdot\,]$, respectively. Next, the resulting (k_z, k_x) half-spectrum is fed into the spatial IFFTs yielding the (k_z, x) half-spectrum, scaling factors $S(k_z)$ and maximum scaling factor $maxS_{k_z}$. To apply the phase shifts $\exp(j\phi(k_z, x, \theta))$, where $\phi(k_z, x, \theta) = \pi k_z x \tan(\theta)$, we first equalize the x-axis data vectors using the maximum scaling factor $maxS_{k_z}$, and then perform CORDIC-based phase rotations [38] specified by $\phi(k_z, x, \theta)$. In Fig. 3.2, the values of $\phi(k_z, x, \theta)$ are provided by the input $\mathtt{R}[\,\cdot\,]$ (as opposed to the input $\mathtt{E}[\,\cdot\,]$ in Fig. 3.1).

Before the compounding is performed, all the maximum scaling factors recorded before ($maxS_x$, $maxS_f$ and $maxS_{k_z}$) need to be added up to form the final scaling factor (S_K) for this particular data frame ($\mathtt{K}$) at emission angle θ. Apart from that, the previously-compounded data frame ($\mathtt{C}$) and its scaling factor (S_C) also needs to be accounted for during the summation. Then based on the difference between S_K and S_C, either $\mathtt{K}$ or $\mathtt{C}$ gets equalized to ensure the summation is performed at the same scaling level.

Finally, HILBERT block in Fig. 3.2 is implemented the same way as the floating-point implementation except for the consideration of the scaling and equalization. After computing the IFFTs along the k_z-axis for each x bin, we get the (z, x)-domain output $\mathtt{H}[\,\cdot\,]$ and the scaling factors $S(x)$. The last computational block in Fig. 3.2 equalizes the z-axis data vectors using the maximum of $S(x)$, and it also outputs the absolute values of $\mathtt{H}[\,\cdot\,]$ giving the envelope.

3.4 PICMUS Reconstruction Results

For floating-point and fixed-point arithmetic testing, we have used several experimental datasets from PICMUS-2016 [12] that utilize $N_a = 11$ plane waves emitted at angles $\pm16°$, $\pm13°$, $\pm9.5°$, $\pm6.5°$, $\pm3.0°$, and $0°$. Specifically, we evaluate the following imaging cases:

A) Two anechoic cylinder targets (cyst phantoms), Fig. 3.3 and Fig. 3.4;

B) Seven wire targets (point phantoms), Fig. 3.5 and Fig. 3.6;

C) Carotid artery – longitudinal section, Fig. 3.7 and Fig. 3.8;

D) Carotid artery – cross section, Fig. 3.9 and Fig. 3.10.

where the first figure in each case shows the floating-point image and the second figure shows the fixed-point image.

For any given angle θ, the N_t-by-N_x size of raw RF channel data frames was 3328×128 in cases A and B, and 1536×128 in cases C and D. We generated the compounded B-mode images by log-compressing their respective normalized envelope sections of size 1216×128, covering the imaging depth from 5 to 50 mm as shown in Fig. 3.3-3.10 using the 60-dB dynamic range. In all four cases, we let $N_t^{\mathrm{FFT}} = 4096$ and $N_x^{\mathrm{FFT}} = 256$, giving $M_t^{\mathrm{FFT}} = \log_2 N_t^{\mathrm{FFT}} = 12$ and $M_x^{\mathrm{FFT}} = \log_2 N_x^{\mathrm{FFT}} = 8$.

The MATLAB version used is R2019a with Fixed-Point Designer installed. For the floating-point implementation, all integer-value variables are set to int32 and fraction-value variables to single in MATLAB. The fixed-point parameter settings are summarized in Table 3.1, with the wordlengths limited to 16 or 24 bits only. Since the values of $\mathtt{P}[\cdot]$, $\mathtt{F}[\cdot]$, and $\mathtt{K}[\cdot]$ are restricted to the $[-1, +1]$ range, the integer part of their signed fixed-point representation is only 1 bit long. The number of fractional-part bits has been set to 14, which is equal to $\max\{M_t^{\mathrm{FFT}}, M_x^{\mathrm{FFT}}\}$ plus 2 extra bits, in order to match the fractional-part length of the FFT twiddle factors having sufficient resolution. When computing $\mathtt{C}[\cdot]$ and $\mathtt{H}[\cdot]$, we have increased their fixed-point wordlength by additional 8 bits for the benefit of CPWC.

Table 3.1: Fixed-Point Parameter Settings.

Parameter	P	M	A	R	F / K	C / H
Signed	Yes	No	Yes	Yes	Yes	Yes
Int. Part	1	12	1	3	1	1
Frac. Part	14	12	14	12	14	22
Wordlength	16	24	16	16	16	24

To accommodate the permissible range of the f_{mig} values and to allow for the adequate interpolation accuracy, we let the integer and fractional parts of unsigned $\mathtt{M}[\cdot]$ have 12 bits each, thus keeping its wordlength at 24-bit limit. As for $\mathtt{A}[\cdot]$, its fixed-point representation has been made compatible with the data format of $\mathtt{K}[\cdot]$ (via binary prescaling and redundant signedness of the A values). Since the phase rotation block in Fig. 3.2 may take any ϕ value between -2π and 2π, signed $\mathtt{R}[\cdot]$ has the integer part of 3 bits. We have allocated 12 bits to its fractional part to maintain the target 16-bit wordlength.

The resulting B-mode images are displayed in Fig. 3.3-3.10, where the odd-numbered figures show the floating-point images in 4 cases and the even-numbered figures show

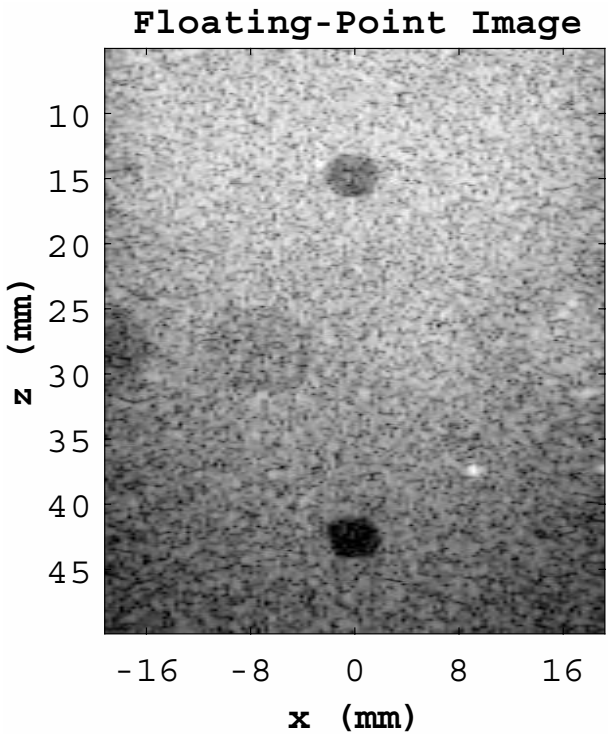


Figure 3.3: Case A: Compounded floating-point cyst phantoms image (11 PWs).

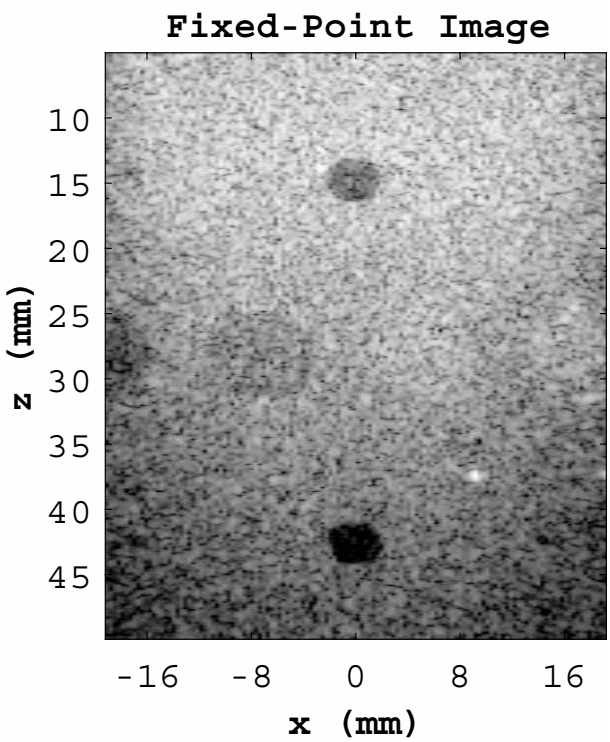


Figure 3.4: Case A: Compounded fixed-point cyst phantoms image (11 PWs).

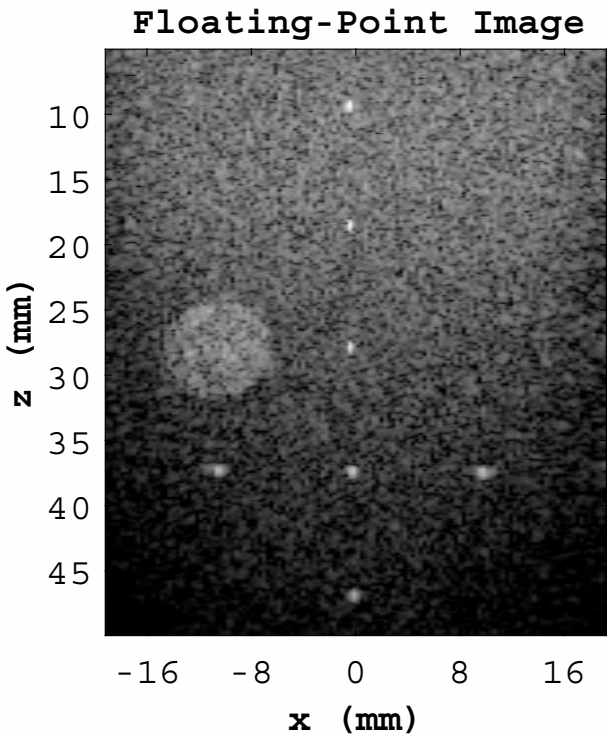


Figure 3.5: Case B: Compounded floating-point point phantoms image (11 PWs).

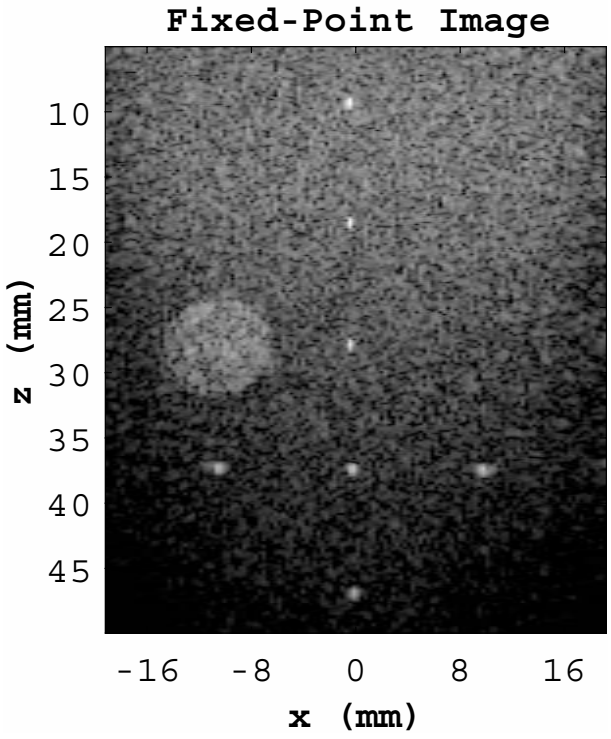


Figure 3.6: Case B: Compounded fixed-point point phantoms image (11 PWs).

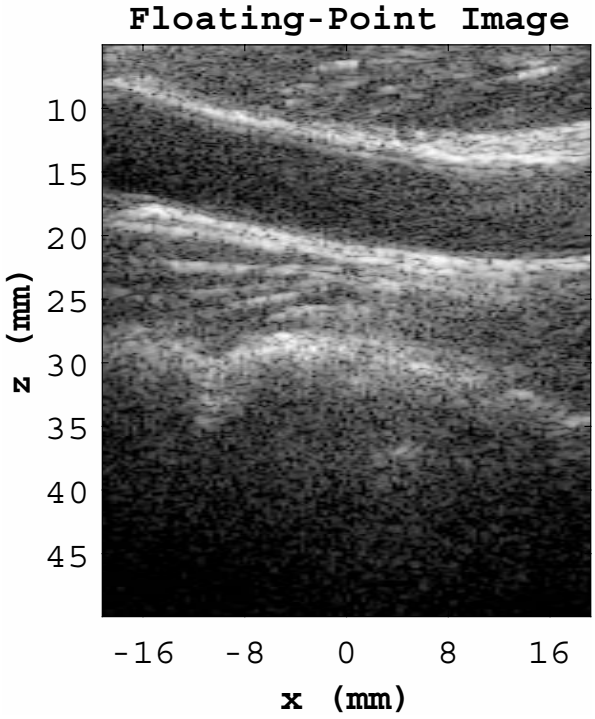


Figure 3.7: Case C: Compounded floating-point carotid artery longitudinal section image (11 PWs).

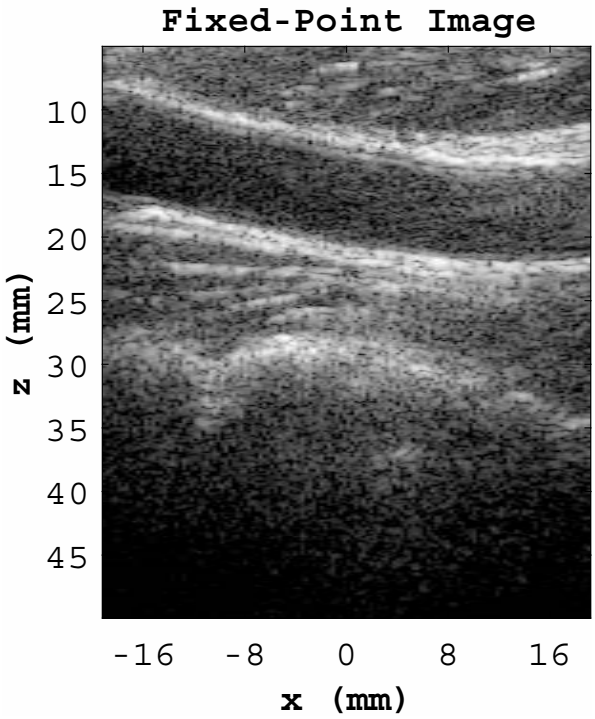


Figure 3.8: Case C: Compounded fixed-point carotid artery longitudinal section image (11 PWs).

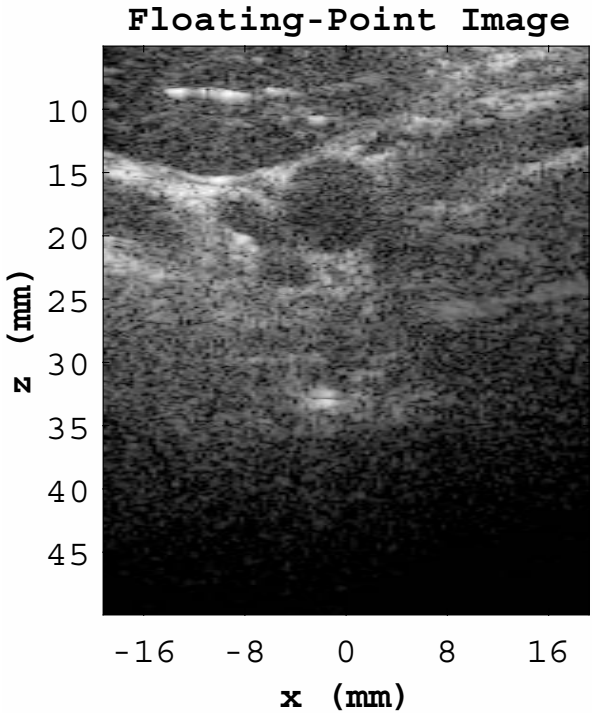


Figure 3.9: Case D: Compounded floating-point carotid artery cross section image (11 PWs).

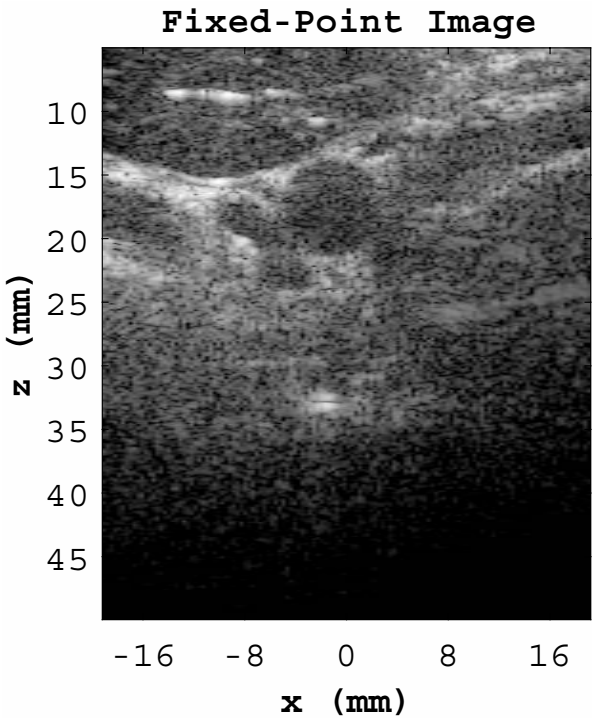


Figure 3.10: Case D: Compounded fixed-point carotid artery cross section image (11 PWs).

the fixed-point images. As one can see, although floating-point calculations are more precise and expensive than fixed-point calculations, those two versions in each case are almost indistinguishable visually.

To evaluate reconstruction differences quantitatively, we have compared the fixed-point versions of a 2D normalized envelope prior to log-compression (i.e., 1216×128 datasets of values ranging from 0 to 1) to their respective floating-point references. Our comparisons rely on three measures of similarity listed in Table 3.2: mean-square error (MSE), peak-signal-to-noise ratio (PSNR), and structural similarity index (SSIM), all of which have been discussed in section 1.1.3. Table 3.2 also shows the average power computed for each fixed-point dataset, which provides a baseline for MSE interpretation. Since the error values are three orders of magnitude smaller than the corresponding average power values, one can view the numerical distance between the fixed-point and floating-point datasets as negligible. The PSNR values ranging from 48 to 65 dB and the SSIM values exceeding 0.99 confirm that the fixed-point reconstruction results are indeed very close to their respective floating-point references.

Table 3.2: Normalized Envelope Similarity Between Fixed-Point and Floating-Point Compounded Data (11 Plane Waves)

Case	Ave. Power	MSE	PSNR	SSIM
A	2.615×10^{-2}	1.657×10^{-5}	47.81 dB	0.9965
B	5.028×10^{-4}	3.906×10^{-7}	65.09 dB	0.9993
C	5.831×10^{-3}	3.430×10^{-6}	54.65 dB	0.9981
D	1.929×10^{-3}	3.796×10^{-6}	54.21 dB	0.9987

We have also evaluated the image quality based on the following indicators [12]:

1. Contrast-to-noise ratio (CNR), obtained from two cyst phantoms (case A) as shown in Fig. 3.3 and Fig. 3.4.

2. Full-width at half-maximum (FWHM), obtained from seven point phantoms (case B) as shown in Fig. 3.5 and Fig. 3.6.

Table 3.3 and 3.4 list the respective numerical values of CNR and FWHM corresponding to the floating- and fixed-point cases. The two cyst phantoms in Fig. 3.3 and Fig. 3.4 are referred to as 'Top' and 'Bottom' in table 3.3, while table 3.4 shows the average lateral and axial FWHW values for the point phantoms in Fig. 3.5 and Fig. 3.6. From these tables, we can see that the difference of CNR and FWHM values

between floating- and fixed-point version is negligible, except for the bottom cyst CNR, which is 8.7% worse in the fixed-point case (due to precision loss affecting μ_{in} and σ_{in} of the anechoic cyst in question).

Contrast	CNR (dB)	
	Top	Bottom
Floating-Point	10.5	11.5
Fixed-Point	10.4	10.5
Difference (%)	1.0	8.7

Table 3.3: CNR of cyst phantoms

Resolution	Mean FWHM (mm)	
	Lateral	Axial
Floating-Point	0.29	0.55
Fixed-Point	0.29	0.54
Difference (%)	0	1.8

Table 3.4: FWHM of seven point phantoms

Chapter 4

Xilinx Implementation of PW Stolt's Migration Algorithm

In the previous chapter, we discussed the MATLAB implementation of PW Stolt's migration algorithm, which was the first phase of our DSP application development. Next, the algorithm written in MATLAB is implemented as a software- or hardware-oriented solution. A software solution involves a general purpose microprocessor or a specialized DSP processor. Due to the sequential execution nature of such processors, they lack adequate support for parallelism and are limited in their capabilities for high-speed and low-power processing. As for a hardware solution, FPGAs provide a reconfigurable implementation fabric allowing for higher computational throughput than software processors [40]. FPGAs are well-suited for highly parallel processing (where multiple computational tasks must be performed simultaneously), as they are electronically wired in the form of discrete programmable logic blocks that can be configured to match the user's needs.

Traditionally, for the FPGA implementation of a DSP algorithm, the RTL model needs to be first created by rewriting the MATLAB code using HDL (such as Verilog or VHDL). After functional simulation, the RTL model written in HDL is then synthesized, placed and routed by the FPGA software tool, such as Vivado or Quartus before generating the configuration bitstream to be programmed into the target FPGA. However, the process of creating an RTL model and a simulation testbench is normally time-consuming and tedious to verify. Much effort also has to be put into the design optimization.

Alternatively, such design process can be automated by using high-level synthe-

sis (HLS) in the design flow, which automatically transforms an DSP algorithmic description written in C, C++, SystemC, or Matlab into an RTL implementation. HLS can better handle the increasing design complexity by removing the need to hand-code the RTL model and testbench. Therefore, HLS along with FPGAs makes a perfect combination for rapid prototyping and a fast time to market [41].

This chapter demonstrates the FPGA implementation of PW Stolt's migration algorithm based on the HLS design flow. The methodology and workflow of collectively using MATLAB Coder and Vivado HLS to perform HLS implementation is first explained. Based on that, automatically generated Xilinx FPGA implementations of both floating-point and fixed-point versions of PW Stolt's migration algorithm are detailed.

4.1 Methodology and Workflow

For a DSP application originally written in MATLAB, HLS can be used to automate the hardware design process. Using HLS can not only avoid hand-coding the original algorithm in HDL, but also reduce the risk of making mistakes. Additionally, HLS can be used to explore different DSP design options by changing the original MATLAB code or the optimization directives from the HLS tool, which increases the likelihood of finding an optimal implementation. Verification also becomes a lot easier because the original non-HDL test code can be reused to verify the RTL design without manually creating an HDL testbench.

To obtain the Xilinx implementation of the PW Stolt's migration algorithm using HLS design flow, two design automation software tools, MATLAB Coder and Vivado HLS, are collaboratively employed. By using MATLAB Coder, the original MATLAB code can be automatically converted into a C program, then the generated C code can be directly passed to Vivado HLS to yield the synthesizable Verilog/VHDL description.

Such design flow offers a significant design productivity boost during mapping of a MATLAB algorithm onto a target FPGA. Note that to make the entire translation from MATLAB to HDL code performed automatically (without designer's manual intervention), the original MATLAB code not only has to comply with the MATLAB Coder guidelines, but also follow additional rules which can ensure the generated C program is compatible with the downstream Vivado HLS. In other words, the beneficial use of our workflow is conditioned on the MATLAB-level specification being

coded properly.

4.1.1 MATLAB Coder for C code generation

MATLAB Coder allows us to generate readable and portable C (or C++) code based on the existing MATLAB code for various software platforms, from desktop to embedded systems. The generated code can be integrated into the project as source code, static libraries, or dynamic libraries depending on the requirement [42].

The following steps need to be taken to translate a given MATLAB code into a C code:

1. Identify the entry-point function, which is to be translated into a C code. A test script calling that function could also be used for the purpose of automatic input data types detection in step 4;

2. Open the MATLAB Coder app and add the entry-point function. Fig. 4.1 is a screenshot after opening the MATLAB Coder app:

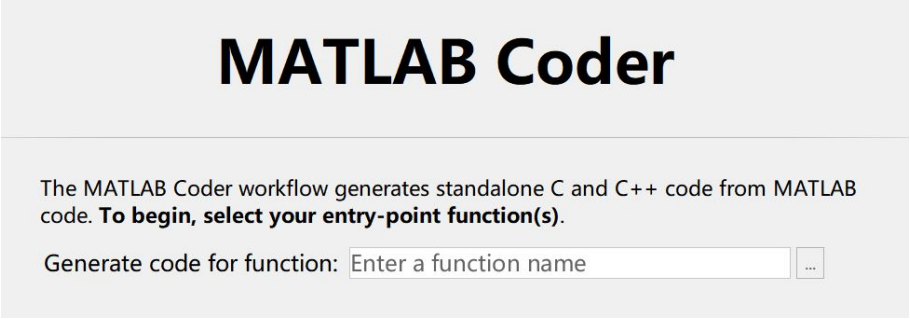


Figure 4.1: Entry-point function selection

3. Fix any issues reported by Code Generation Readiness Tool, which analyzes the entire entry-point function, including all the sub-functions, and pinpoints some unsupported MATLAB functions, syntax and language features;

4. Declare the types and sizes of all the inputs of the entry-point function by either manually defining them, or automatically detecting them using a script that calls the entry-point function. Fig. 4.2 is an example of using automatic input data types detection;

5. Generate the MEX file for verification purposes. A MEX file is essentially a function, created by MATLAB Coder, that provides an interface between

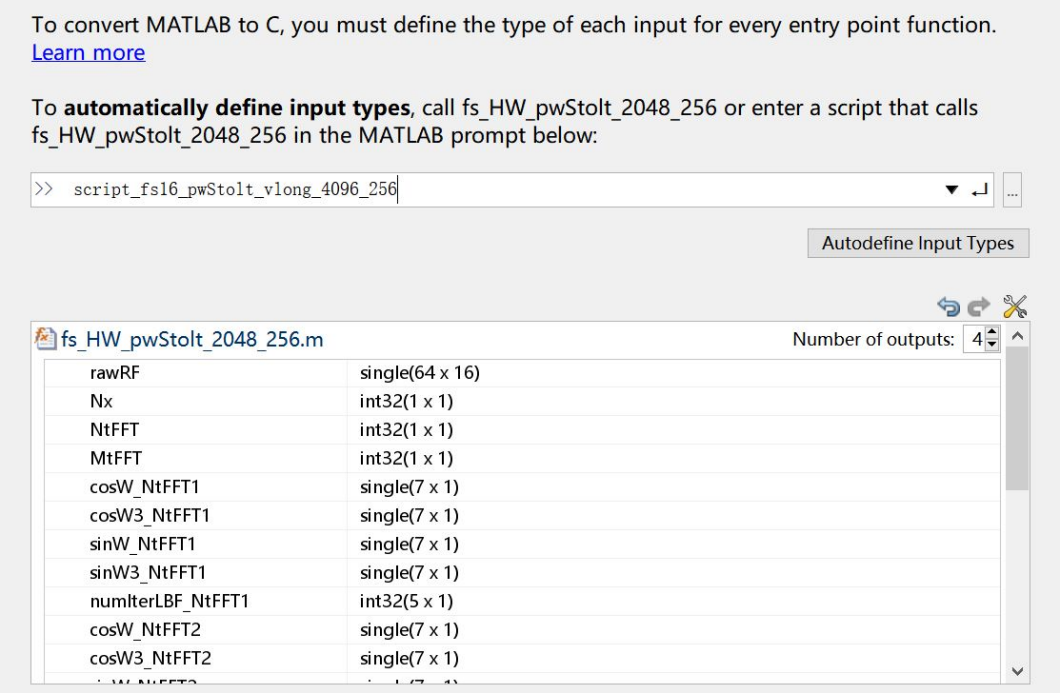


Figure 4.2: Automatic input data types detection

MATLAB and the entry-point function in C. Before generating the C code, MATLAB Coder creates a MEX file from MATLAB function(s), invokes the MEX function and reports issues that may be hard to diagnose in the generated C code. Using MEX files enable us to check for run-time errors and verify the behavior. Fig. 4.3 shows the screenshot of this step. The MEX file can also be packaged for use in the MATLAB environment. It can replace the existing MATLAB code and thus accelerate the MATLAB algorithm;

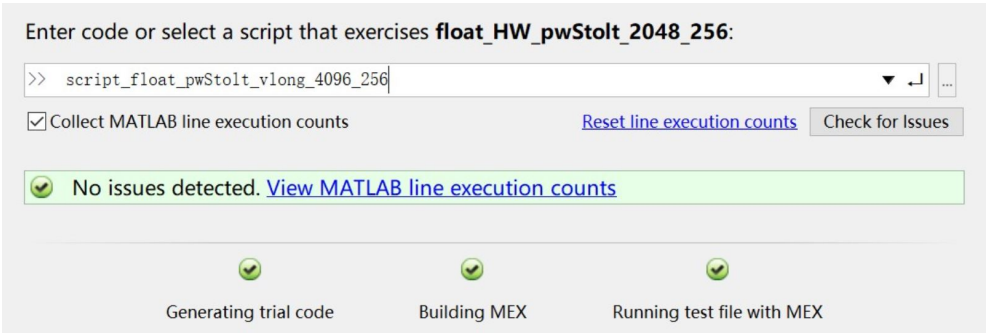


Figure 4.3: MEX function generation

6. Customize the design by changing other settings, such as array layout, threshold

for function inlining, etc., which may change the performance of the generated C code. Fig. 4.4 is the screenshot showing part of the settings available in MATLAB Coder;

⊟ Speed	
Saturate on integer overflow	Yes
Support only purely-integer numbers	No
Support nonfinite numbers	Yes
Loop unrolling threshold	5
⊟ Memory	
⊟ Variable Sizing Support	
Enable variable-sizing	Yes
Dynamic memory allocation	For arrays with max size at or above threshold
Dynamic memory allocation threshold	65536
Array layout	Column-major
Highlight potential row-major issues	Yes
Preserve array dimensions	No
Stack usage max	200000
Generate re-entrant code	No
⊟ Code Appearance	
Generated file partitioning method	Generate one file for each MATLAB file
⊟ Comments	
Include comments	Yes
Comment Style	Auto (Use standard comment style of the target language)
MATLAB source code as comments	No
MATLAB function help text	Yes
⊟ Code Style	
Convert if-elseif-else patterns to switch-case statements	No
Preserve extern keyword in function declarations	Yes
Use signed shift left for fixed-point operations and mul...	Yes
Allow right shifts on signed integers	Yes
Parentheses	Nominal (Optimize for readability)
Maximum identifier length	31
Data type replacement	Use built-in C data types in the generated code

Figure 4.4: Customized code generation settings

7. Generate C code by clicking on the red box in Fig 4.5.

4.1.2 Vivado HLS for HDL generation

Xilinx Vivado High-Level Synthesis (HLS) tool has the ability of transforming a C specification, including C, C++, or SystemC, into a register transfer level (RTL) implementation that can then be synthesized into a Xilinx field programmable gate array (FPGA). Also, the Vivado HLS tool offers capabilities for design evaluation and various optimization techniques using fine-grain parallel architecture of FPGAs [17]. This not only greatly reduces the effort of creating an RTL implementation from a

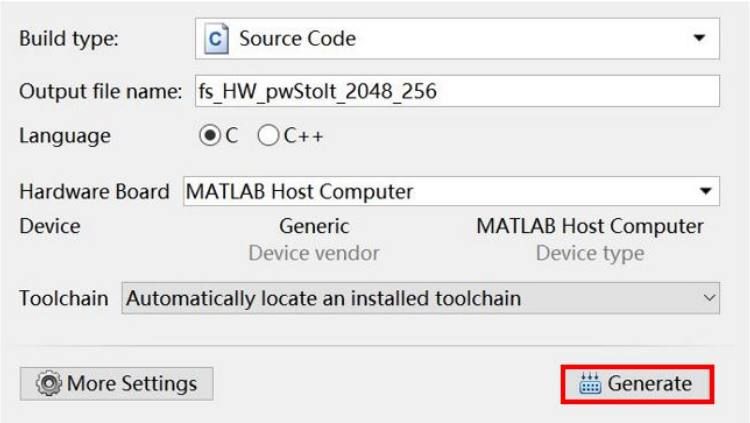


Figure 4.5: C code generation

high-level design, but also provides flexibility in the final hardware implementation to meet design constraints set by the developer.

From a user's point of view, the general design flow of using the Vivado HLS tool for an RTL implementation is as follows:

1. Create a new Vivado HLS project, import all the related C source files into the project and identify the top-level function. Fig. 4.6 is a screenshot of adding or removing C-based source files;

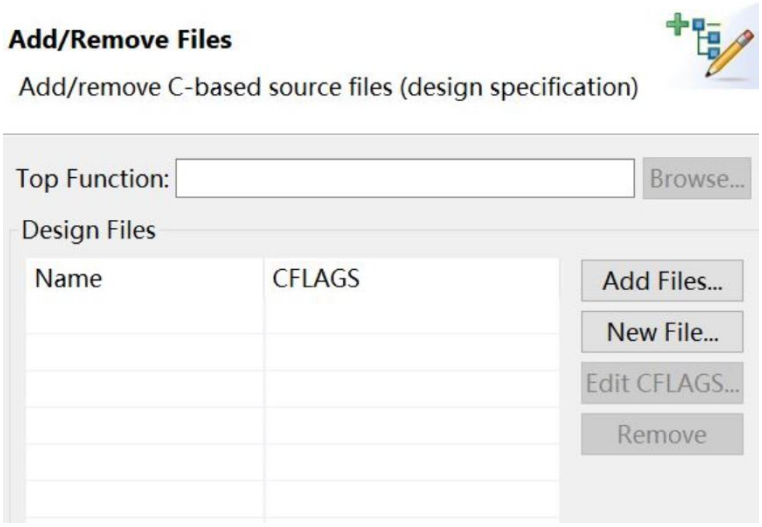


Figure 4.6: Project source files adding/removing

2. Import the C testbench files, which are then used to simulate the C function to be synthesized and verify the RTL output in C/RTL co-simulation (step 6);

3. Specify the solution configuration including clock period, uncertainty and the type of target Xilinx FPGA as shown in Fig. 4.7;

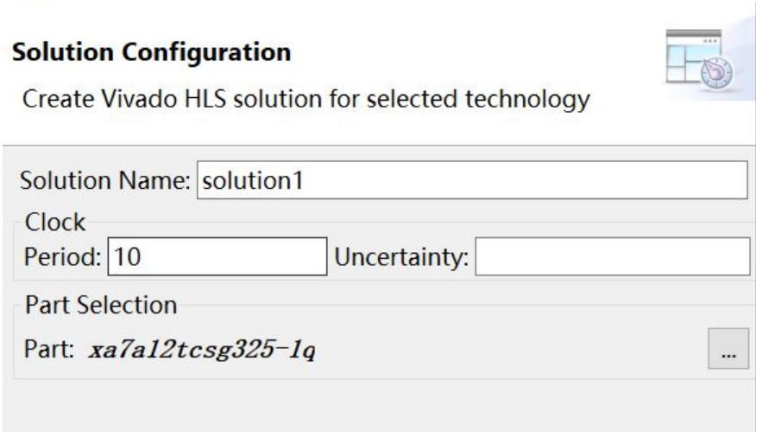


Figure 4.7: Initial solution configuration

4. Run C simulation and check the output. Note that in high-level synthesis, running the compiled C program is referred to as C simulation [17]. Vivado HLS uses the test bench to compile and execute the C simulation to validate that the algorithm is functionally correct. If C simulation does not get the expected results, debug the code in the debugger mode, which is shown in Fig. 4.8.

   The red box in Fig. 4.8 allows user to step through code. Breakpoints can also be set and the value of the variables can be directly viewed;

5. Synthesize the design and analyze the synthesis results by checking the synthesis report and analysis perspective generated by Vivado HLS. Note that synthesis can be controlled by applying different optimization directives, which provides possibilities for different design options. Fig. 4.9 shows a synthesis report example. Each item in the report (e.g., Timing) can be expanded to see the details;

6. Run C/RTL co-simulation and view the simulation waveforms in Vivado to verify the design. Fig. 4.10 shows the C/RTL co-simulation window allowing us to select which type of HDL to use for verification (Verilog or VHDL) and which HDL simulator to use for simulation;

7. Export the RTL design as an Intellectual Property (IP) block that can be used by other tools in the Xilinx design flow.

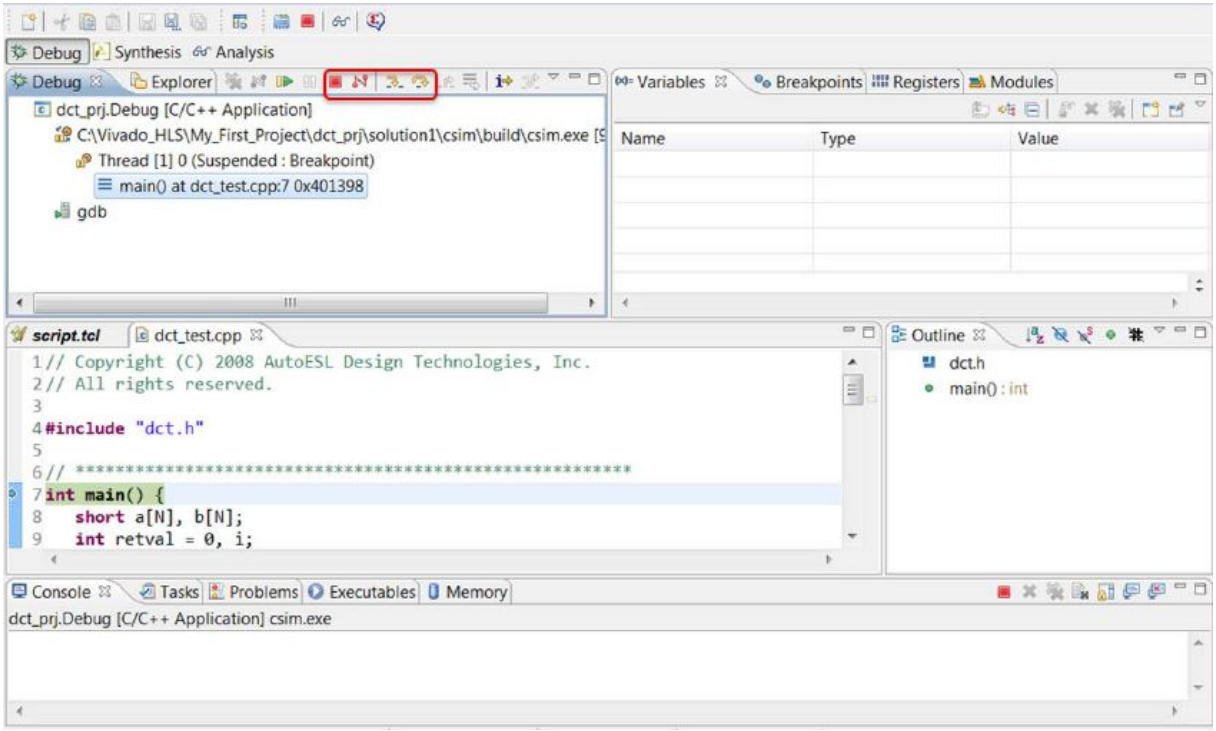


Figure 4.8: C Debug Environment

In the rest of this chapter, we will implement the PW Stolt's migration algorithm in a Xilinx FPGA by collaboratively using MATLAB Coder and Vivado HLS following the methodology and workflow discussed above.

4.2 Floating-Point Xilinx Implementation

Before feeding the floating-point MATLAB code version to the MATLAB Coder, we need to specify the entry-point function, which acts as the top-level function of the final RTL. There were originally three main function blocks before: *reconstruction.m*, *compounding.m* and *hilbert.m*, each of which corresponds to one block in the computational flow chart shown in Fig. 3.2, excluding the blocks referring to scaling equalization (used only in the fixed-point version). Considering the fact that only one top function is allowed to be specified in Vivado HLS, we encapsulate these three function blocks into one entry-point function named *stolt_hardware.m*. Also, a top-level test script named *matlab_script.m* is needed to invoke *stolt_hardware.m* in order to automatically detect the data types and sizes of the input arguments of *stolt_hardware.m*. For each emission angle, *stolt_hardware.m* gets called once by

Figure 4.9: Synthesis Report

matlab_script.m.

In Vivado HLS, arrays in the C code synthesize into block RAMs in the final FPGA design and top-level function arguments synthesize into RTL I/O ports. If a matrix or array belongs to the top-level function interface, high-level synthesis implements that array as ports to access a block RAM outside the design [17]. In other words, block RAMs resulted from matrices or arrays on the top-level function interface do not occupy the internal FPGA memory resource. Therefore, a large amount of memory resource can be saved by placing large matrix or array chunks on the top-level function interface. To this end, at the MATLAB level, instead of doing all matrix and array computations inside *stolt_hardware.m*, which would be synthesized into internal memory of the FPGA by Vivado HLS later, some matrices and arrays can actually be calculated in advance in *matlab_script.m* and accessed by *stolt_hardware.m* through its function interface.

Thus, in *matlab_script.m*, we pre-compute `M`, `A`, and `R` (see Fig. 3.2) for a given PW emission angle. In addition to that, some constant parameters related to the

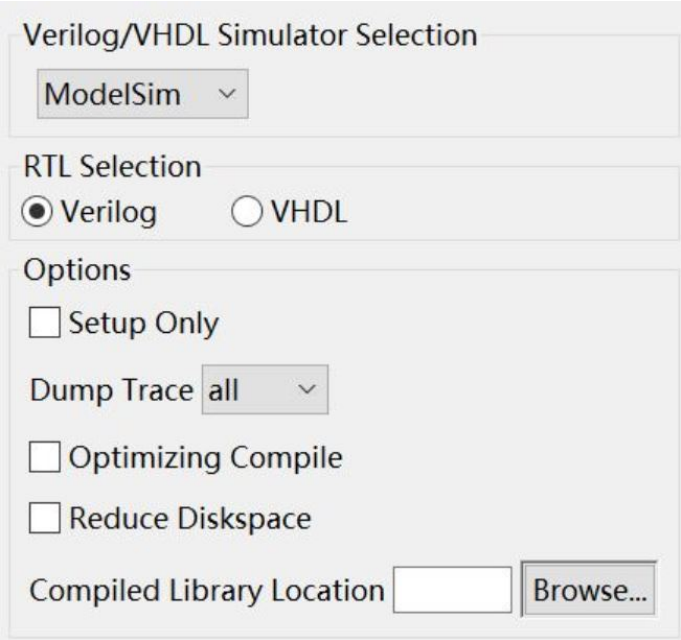


Figure 4.10: C/RTL co-simulation window

temporal and spatial split-radix FFTs are also pre-computed, such as the sine and cosine coefficients, number of internal iterations, and the bit-reversal order of the output results. Other input arguments include `P` (raw data frame corresponding to a given PW emission angle), N_x^{FFT}, N_t^{FFT}, M_x^{FFT}, M_t^{FFT}, N_x, N_z, and `C` (see Fig. 3.2). All of them serve as the input arguments on the interface of *stolt_hardware.m* in order to save memory resources of the FPGA to be synthesized later.

As discussed in section 3.3, we have multiple PW emission angles, so the entry-point function (*stolt_hardware.m*) is invoked multiple times accordingly. At each iteration, we need to update the compounded data frame `C`, which is implemented by *Compounding* block. We place matrix `C` in the input argument list, as it needs to be updated at every iteration. Once all angle-specific migrated (beamformed) frames have been compounded, we need to calculate `H` (the "analytic signal" version of final `C`), which is done by *Hilbert* block. The latter needs to be called only once for the last angle. To this end, we use a flag signal named *last_flag* as another input argument to indicate if the current emission angle is the last one. Similar to `C`, we place `H` in the input argument list, so that it can be initialized and stored outside of an FPGA. Both `C` and `H` are also the output arguments of hardware-bound *stolt_hardware.m*, since their values are affected by FPGA-mapped computations.

Based on the above description, we have the diagram in Fig. 4.11 showing the hi-

erarchy with the input and output arguments listed. During computation, all integer variables (such as array indices and FFT sizes) are set to *int32* and floating-point variables (such as `P` and `A`) to *single*. Before feeding the design to MATLAB Coder, one local data matrix is created in the *Reconstruction* block to store the intermediate results obtained from each sub-block inside *Reconstruction*, such as *FFT*, *Remap/-Multiply*, *Phase Rotation*, and so on. Instead of using different data matrices, we use the same matrix to store the output data from different sub-blocks and the values in that matrix get updated throughout the computation process in order to further reduce the memory requirements.

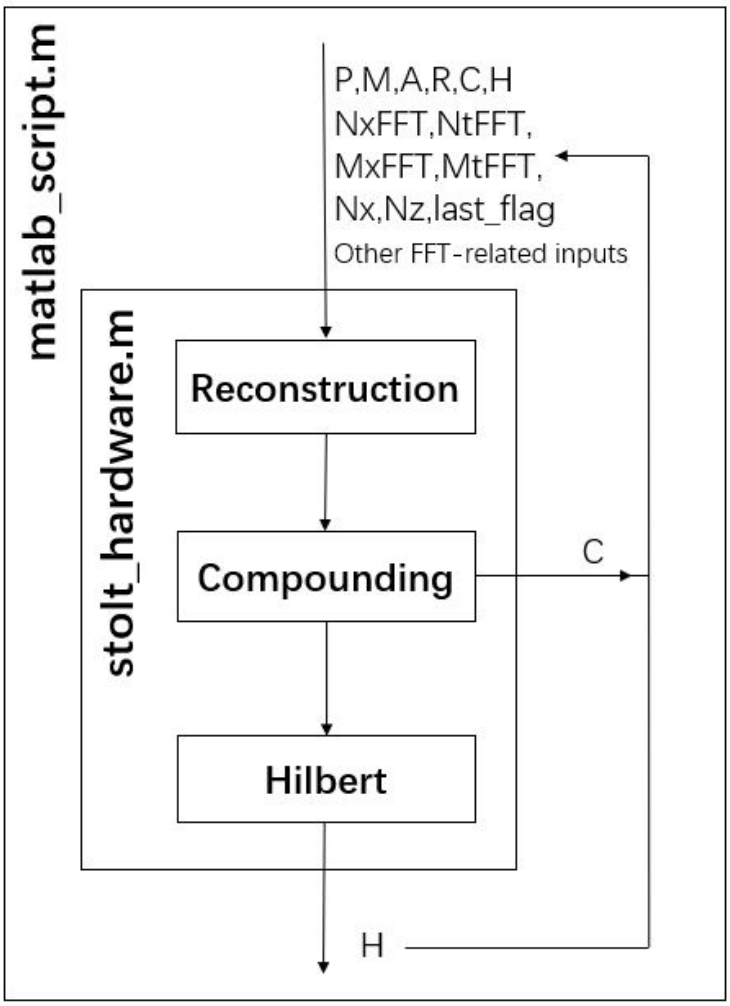


Figure 4.11: Floating-point MATLAB code hierarchy

After all the above-mentioned adjustments, the data type and size for each input argument are automatically obtained from MATLAB Coder by *matlab_script.m*

calling *stolt_hardware.m*. Table 4.1 summarizes each input argument, its data type, size, and the corresponding symbol seen in Fig. 3.2. Note that *stretchZ* shown in the input argument list is an angle dependent value used to compute `M` and `A`. That is, $stretchZ = 2/(1 + \cos(\theta))$, which can be viewed as a scaling factor in equations 2.6 and 2.7.

Input Argument	Data Type & Size	Notation Mapping
HW_rawRF	single(4096 x 256)	P
Nx	single(1 x 1)	Nx
NtFFT	single(1 x 1)	N_t^{FFT}
MtFFT	single(1 x 1)	M_t^{FFT}
NxFFT	single(1 x 1)	N_x^{FFT}
MxFFT	single(1 x 1)	M_x^{FFT}
LUT_iiF	single(2048 x 129)	M
LUT_scaler	single(2048 x 129)	A
stretchZ	single(1 x 1)	
shiftZ	single(256 x 1)	R
cosW_NtFFT	single(511 x 1)	
cosW3_NtFFT	single(511 x 1)	
sinW_NtFFT	single(511 x 1)	
sinW3_NtFFT	single(511 x 1)	
BR_NtFFT	int32(4096 x 1)	
numIterLBF_NtFFT	int32(11 x 1)	
numIter2BF_NtFFT	int32(1 x 1)	FFT-related inputs
cosW_NxFFT	single(1 x 31)	
cosW3_NxFFT	single(1 x 31)	
sinW_NxFFT	single(1 x 31)	
sinW3_NxFFT	single(1 x 31)	
BR_NxFFT	int32(256 x 1)	
numIterLBF_NxFFT	int32(7 x 1)	
numIter2BF_NxFFT	int32(1 x 1)	
Nz	single(1 x 1)	Nz
PW_last_flag	logical(1 x 1)	last_flag
HW_Comp	single(4096 x 256)	C
HW_Hilb	single(12288 x 256)	H

Table 4.1: Floating-point input argument lists

Before generating an actual C code, several settings need to be adjusted for the purpose of the compatibility with the downstream Vivado HLS. For example, in Vivado HLS, using dynamic memory allocation in C is not supported, so it must be disabled in the MATLAB Coder settings before generating a C program.

Moving to Vivado HLS, we first create a Vivado HLS project and import all the generated C source files into that project. The C source file: *stolt_hardware.c*, which is generated from *stolt_hardware.m* by MATLAB Coder is specified to be the top function of the Vivado HLS project. Then we select the FPGA to be targeted. In our case, we use *xq7vx690trf1930-1i*, which belongs to Xilinx Virtex-7 FPGA family. This FPGA device has 433200 6-input look-up tables (LUTs), 866400 flip-flops (FFs), 3600 DSP blocks and 2940 block RAMs (BRAMs).

The LUT is the building block of an FPGA. Essentially, a LUT is a truth table in which different combinations of the inputs implement different functions to yield output values [6]. An N-input LUT is capable of implementing any logic function of N Boolean variables. The value for N in our FPGA device is 6.

The flip-flop is the basic storage unit in an FPGA. As shown in Fig. 4.12, the structure of a flip-flop includes a data input, clock input, clock enable, reset, and data output. Any value at the data input port is latched and passed to the output on every pulse of the clock. The flip-flop is always paired with a LUT to perform logic pipelining and data storage [6].

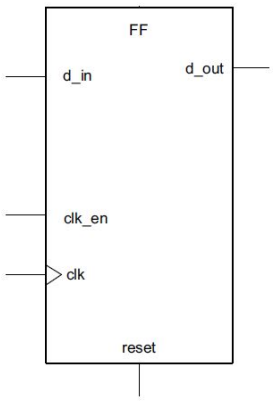


Figure 4.12: Structure of flip-flop

The DSP block is an arithmetic logic unit (ALU) embedded into the FPGA fabric. As shown in Fig. 4.13, the computational chain in a DSP block consists of an add/sub-

tract unit, a multiplier and a final add/subtract/accumulate engine. Such structure allows a single DSP unit to implement computation of the form: $p = a \times (b + d) + c$ or $p+ = a \times (b + d)$.

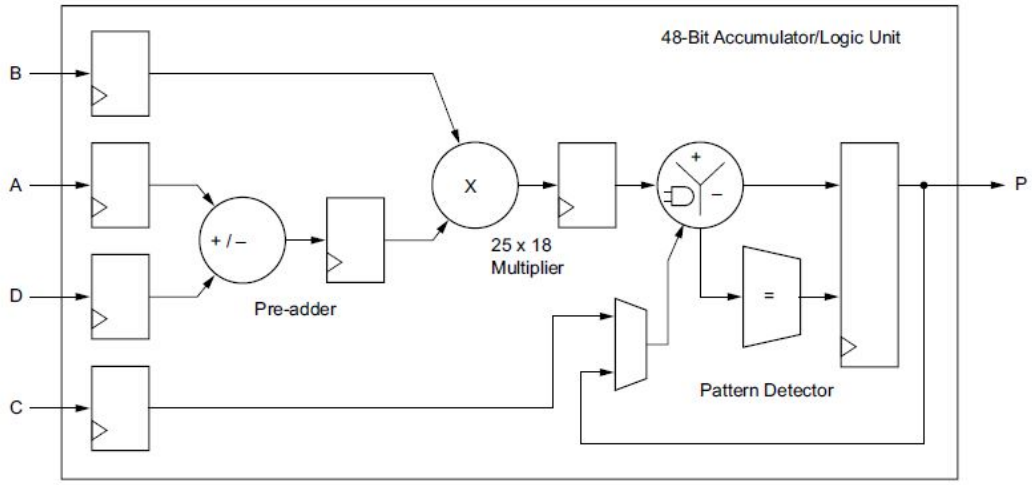


Figure 4.13: Structure of DSP block [6]

The BRAM is a dual-port RAM module which offers on-chip storage for a large dataset. The BRAM memory capacity available in an FPGA device can be either 18K or 36K bits. Depending on how arrays are represented in C code, BRAMs can implement either a RAM or a ROM in Vivado HLS. In a RAM configuration, the data can be read and written at any time during the runtime of the circuit. In contrast, in a ROM configuration, data can only be read during the runtime. The writing stage of a ROM occurs during the FPGA configuration and cannot be modified later [6].

Since C simulation has been performed in MATLAB Coder using the MEX file, we can skip this step in Vivado HLS and start to synthesize the design directly. When the C synthesis completes, Verilog and VHDL code generate automatically. Fig. 4.14 shows the resource utilization report after C synthesis, in which the percentage of BRAM, DSP, FF, and LUT utilization is 75%, 6%, 2%, and 8%, respectively. Note that the utilization of any resource is below 100%, indicating that the target FPGA can accommodate our floating-point design.

A more detailed analysis and comparison will be performed in section 4.4, where the generated HDL along with the time constraint will be synthesized, placed and routed in the target FPGA using Vivado.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	3253
FIFO	-	-	-	-
Instance	135	234	19024	33562
Memory	2076	-	0	0
Multiplexer	-	-	-	2128
Register	-	-	2036	-
Total	2211	234	21060	38943
Available	2940	3600	866400	433200
Utilization (%)	75	6	2	8

Figure 4.14: Floating-point utilization estimates

4.3 Fixed-Point Xilinx Implementation

The fixed-point Xilinx implementation of our design follows the same procedure as the floating-point version discussed in the previous section. All the input and output arguments from the floating-point implementation are preserved (with different data types). Since the data of the fixed-point algorithm is scaled throughout the computation, the scaling factors after compounding and Hilbert transform are added to the interface of the entry-point function to keep track of the data scaling process for re-scaling adjustment after the hardware computation. We also use an overflow flag to indicate saturation after compounding, which allows us to compensate for it at the next computational step.

The fixed-point version of the code hierarchy before feeding into MATLAB Coder is shown in Fig. 4.15, where the scaling factors after compounding and Hilbert transform are labeled C_s and H_s, respectively, and the overflow flag is labeled $OFflag$. Also, a suffix "$\_fi$" is added to each Matlab file name to differentiate from the previous floating-point version.

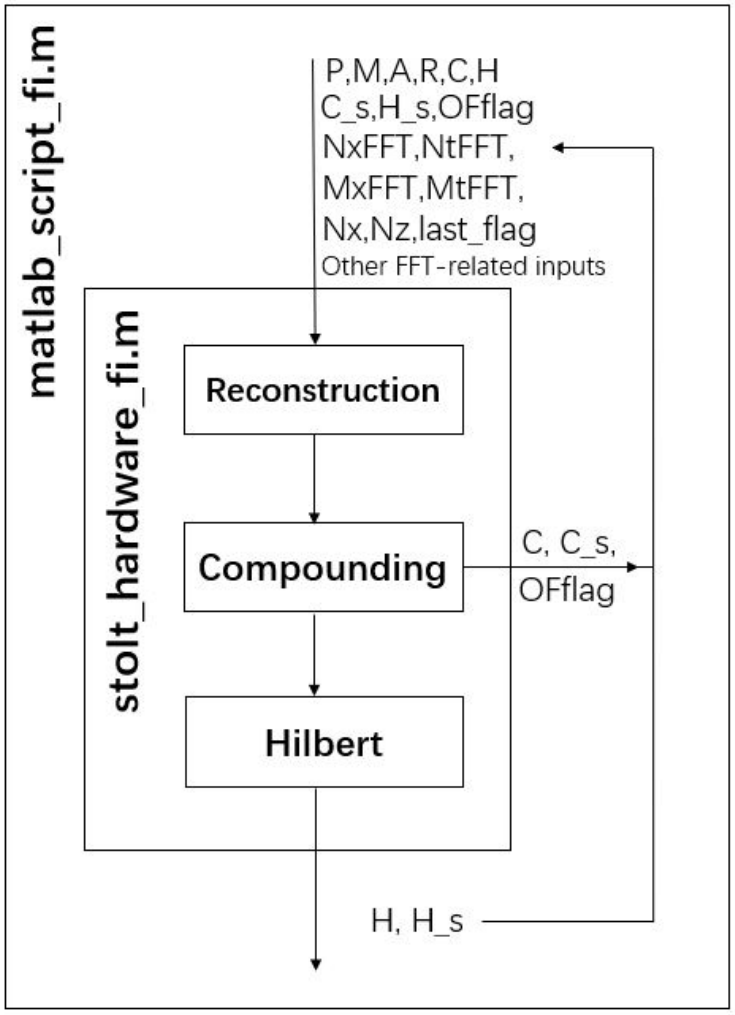


Figure 4.15: Fixed-point MATLAB code hierarchy

Table 4.2 summarizes each input argument, including the input name, data type, size, and the notation mapping (see Fig. 3.2 and table 3.1) after automatic input detection through MATLAB Coder.

Input Argument	Data Type & Size	Fixed-Point Precision	Notation Mapping
HW_rawRF	fi(4096 x 256)	*numerictype(1, 16, 14)*	P
Nx	int16(1 x 1)		Nx
NtFFT	int16(1 x 1)		N_t^{FFT}
MtFFT	int8(1 x 1)		M_t^{FFT}
NxFFT	int16(1 x 1)		N_x^{FFT}
MxFFT	int8(1 x 1)		M_x^{FFT}
LUT_iiF	fi(2048 x 129)	*numerictype(0, 24, 12)*	M
LUT_scaler	fi(2048 x 129)	*numerictype(1, 16, 14)*	A
HW_stretchZ	fi(1 x 1)	*numerictype(1, 16, 14)*	
HW_shiftZ	fi(256 x 1)	*numerictype(1, 16, 12)*	R
cosW_NtFFT	fi(511 x 1)	*numerictype(1, 16, 14)*	
cosW3_NtFFT	fi(511 x 1)	*numerictype(1, 16, 14)*	
sinW_NtFFT	fi(511 x 1)	*numerictype(1, 16, 14)*	
sinW3_NtFFT	fi(511 x 1)	*numerictype(1, 16, 14)*	
BR_NtFFT	int16(4096 x 1)		
numIterLBF_NtFFT	int8(11 x 1)		
numIter2BF_NtFFT	int8(1 x 1)		
cosW_NxFFT	fi(1 x 31)	*numerictype(1, 16, 14)*	FFT-related inputs
cosW3_NxFFT	fi(1 x 31)	*numerictype(1, 16, 14)*	
sinW_NxFFT	fi(1 x 31)	*numerictype(1, 16, 14)*	
sinW3_NxFFT	fi(1 x 31)	*numerictype(1, 16, 14)*	
BR_NxFFT	int16(256 x 1)		
numIterLBF_NxFFT	int8(7 x 1)		
numIter2BF_NxFFT	int8(1 x 1)		
Nz	int16(1 x 1)		Nz
PW_last_flag	logical(1 x 1)		last_flag
HW_Comp	fi(4096 x 256)	*numerictype(1, 24, 22)*	C
SmaxComp	int8(1 x 1)		C_s
OFComp	int8(1 x 1)		OFflag
HW_Hilb	fi(12288 x 256)	*numerictype(1, 24, 22)*	H
SmaxHilb	int8(1 x 1)		H_s

Table 4.2: Fixed-point input arguments list

After verifying the algorithm using the MEX file, the fixed-point version of C code is generated. We then pass the C code to Vivado HLS, and perform the C synthesis using the same target FPGA (xq7vx690trf1930-1i) as we used in the floating-point version. Fig. 4.16 shows the resulting resource utilization report, in which the percentage of BRAM, DSP, FF, and LUT utilization is 36%, 5%, 1%, and 19%, respectively, indicating our fixed-point design can also fit into the target FPGA. More details will be discussed in section 4.4.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	3576
FIFO	-	-	-	-
Instance	94	201	22584	76920
Memory	1052	-	0	0
Multiplexer	-	-	-	2554
Register	-	-	2260	-
Total	1146	201	24844	83050
Available	2940	3600	866400	433200
Utilization (%)	38	5	2	19

Figure 4.16: Fixed-point utilization estimates

4.4 Results and Comparisons

After having the HDL code generated for the floating- and fixed-point version of our design by using MATLAB Coder and Vivado HLS, we pass the Verilog code to Vivado to perform the actual Xilinx implementation.

We first create a RTL project in Vivado and add all the Verilog files generated from Vivado HLS as the source files. The target FPGA is set to xq7vx690trf1930-1i, which is the same as what we used in Vivado HLS. We have also specified a clock with a period of 12.5ns (80MHz) as the timing constraint. For the floating-point version, some computations are performed using Xilinx's IP cores, all of which can be added to Vivado by running the .tcl scripts automatically generated by Vivado HLS. After all the preparation mentioned above, we run synthesis, placement and routing for the floating- and fixed-point versions of HDL.

Fig. 4.17 shows an utilization summary generated after placement and routing. We can see the I/O pins that the target FPGA can provide is 1000, whereas the actual use of I/O after implementation is 1032. Our floating-point HDL implementation thus fails the placement due to the excessive I/O utilization, which was not caught by Vivado HLS.

To tackle this issue, some changes can be made in the original floating-point MATLAB code. All non-array input arguments, such as Nx, Nz, $NtFFT$, $strechZ$, etc., can be replaced by a single setting array, which is introduced in *matlab_script.m* to reduce the pin count. The setting array holds the said non-array input arguments in a certain order and acts as a single input argument of *stolt_hardware.m*. All the

Resource	Utilization	Available	Utilization %
LUT	26390	433200	6.09
LUTRAM	46	174200	0.03
FF	20168	866400	2.33
BRAM	1063.50	1470	72.35
DSP	168	3600	4.67
IO	1032	1000	103.20

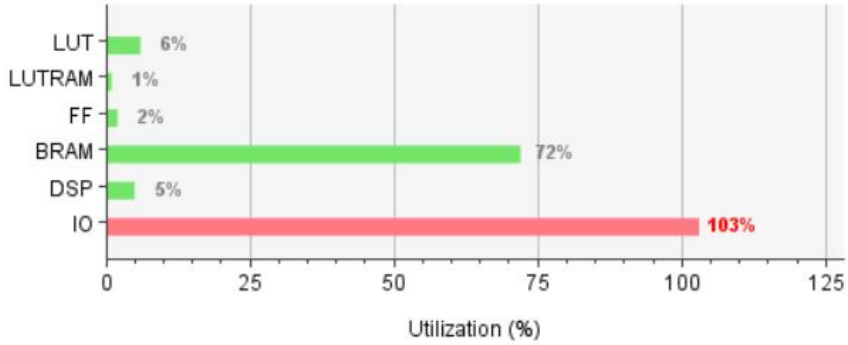


Figure 4.17: Floating-point utilization report

variables are typecast to *single* before assigning to that setting array. After the setting array is passed to the *stolt_hardware.m*, each element in that array is restored to its original data type, which is then used by the following computation. After the adjustment mentioned above, all I/Os can be fitted into the target FPGA.

Fig. 4.3 summarizes the resource utilization from the floating- and fixed-point synthesis reports, and Fig. 4.18 shows a comparison between these two versions in terms of the percentage of utilization for each resource category. The fixed-point design is more resource efficient in terms of the use of flip-flops, block RAMs, DSPs and I/O. Especially, the utilization of BRAM in the fixed-point design is almost half of that in the floating-point counterpart. This is mainly because the bit width used for fixed-point design (mostly 16 bits) is far less than that used in the floating-point design (32 bits), leading to half of the utilization of BRAMs for the storage of the intermediate data matrix throughout the fixed-point computation.

Fig. 4.19 and 4.20 show the timing reports for floating- and fixed-point designs. The timing constraints are met for both versions. The value of worst negative slack (WNS) and worst hold slack (WHS), which correspond to the worst slack of all the timing paths for maximum and minimum delay analysis [43], respectively, show that

Resource	Floating-point	Fixed-point	Available
LUT	29005	38001	433200
LUTRAM	32	128	174200
FF	23787	18928	866400
BRAM	1063.5	571	1470
DSP	168	152	3600
IO	981	926	1000

Table 4.3: Resource utilization summary

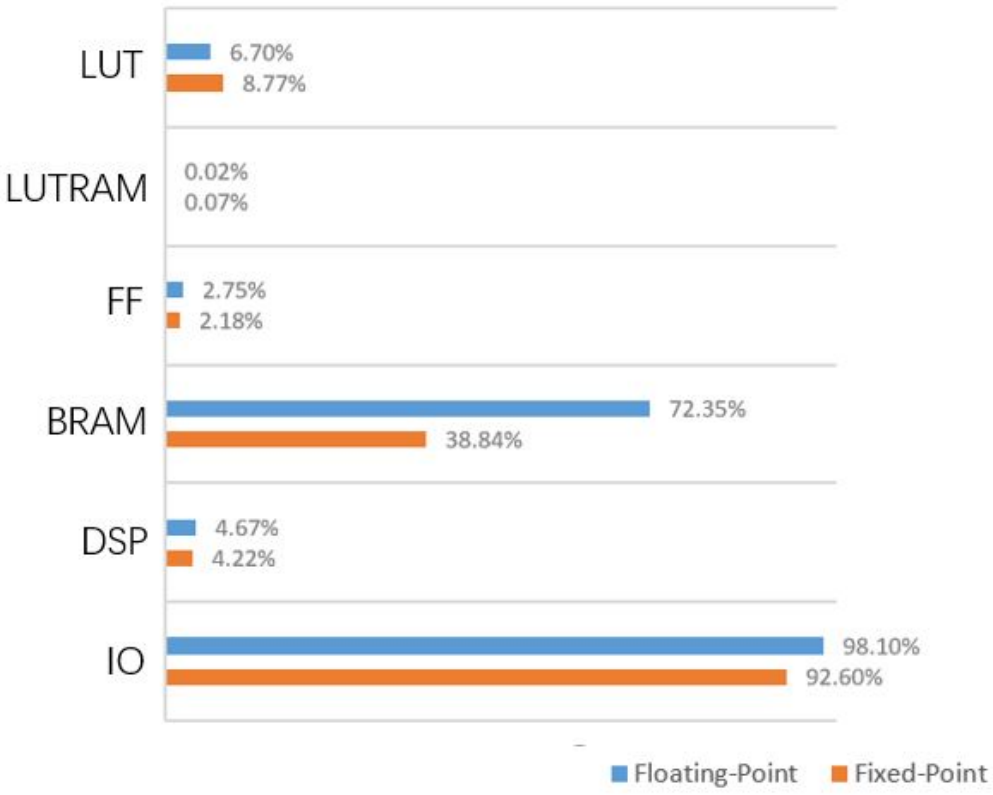


Figure 4.18: Resource utilization comparison

the fixed-point version has a better timing in terms of both setup and hold times. In other words, our fixed-point design can operate at a higher clock frequency than the floating-point counterpart.

Fig. 4.21 and 4.22 show rough estimations in terms of the power consumption for those two implementations. It can be seen that our fixed-point design is more power

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	0.476 ns	Worst Hold Slack (WHS):	0.033 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	81466	Total Number of Endpoints:	81466

All user specified timing constraints are met.

Figure 4.19: Floating-point timing summary

Design Timing Summary

Setup		Hold	
Worst Negative Slack (WNS):	1.454 ns	Worst Hold Slack (WHS):	0.054 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	68417	Total Number of Endpoints:	68417

All user specified timing constraints are met.

Figure 4.20: Fixed-point timing summary

efficient then the floating-point one.

From the above discussion, it can be seen that although the introduction of the scaling factors and equalization blocks add complexity to the fixed-point Stolt's method design, our fixed-point FPGA implementation is more resource and power efficient and can also operate at a higher clock frequency compared to its floating-point counterpart.

4.5 Post-HLS Verification

To verify the correctness of the RTL output generated from Vivado HLS, we need to perform post-HLS verification after C synthesis. In Vivado HLS, such process can be automated through the C/RTL co-simulation feature which can take advantage of the

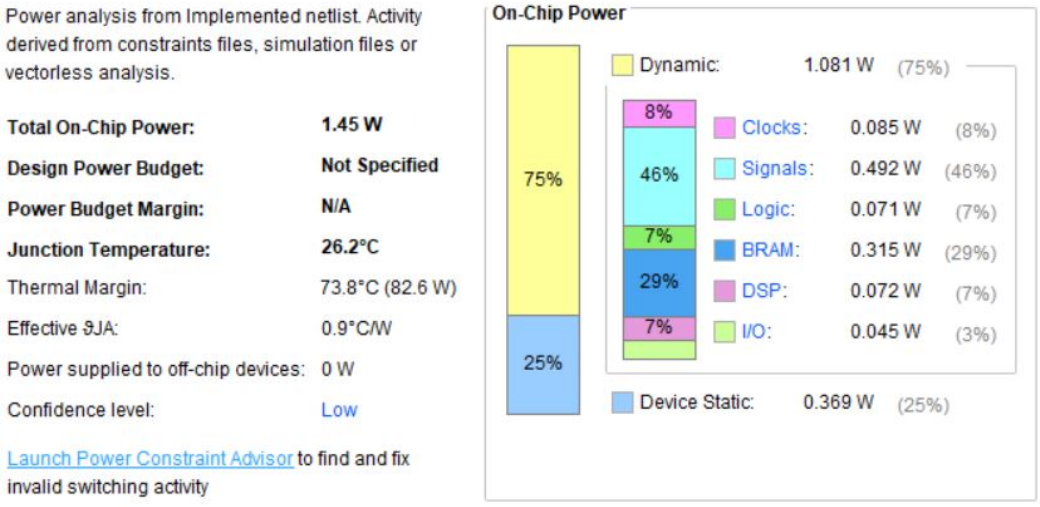


Figure 4.21: Floating-point power summary

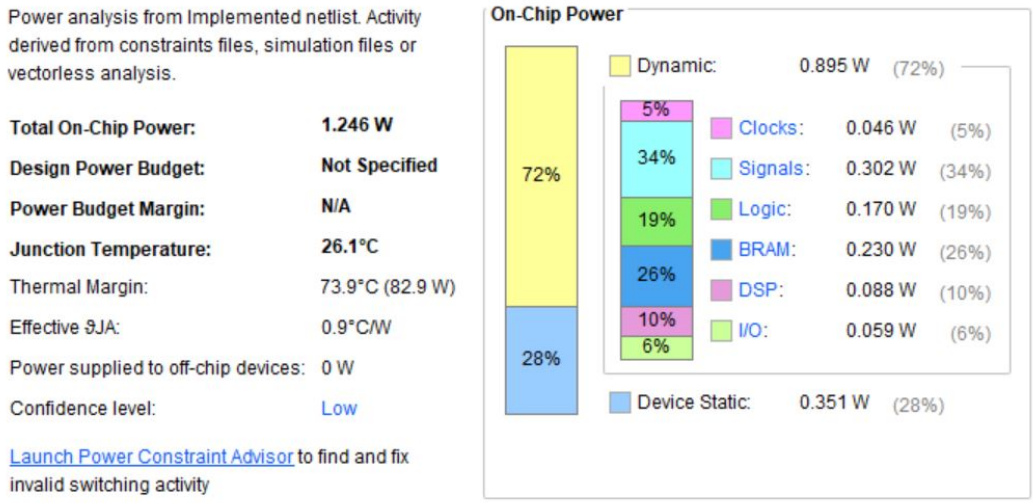


Figure 4.22: Fixed-point power summary

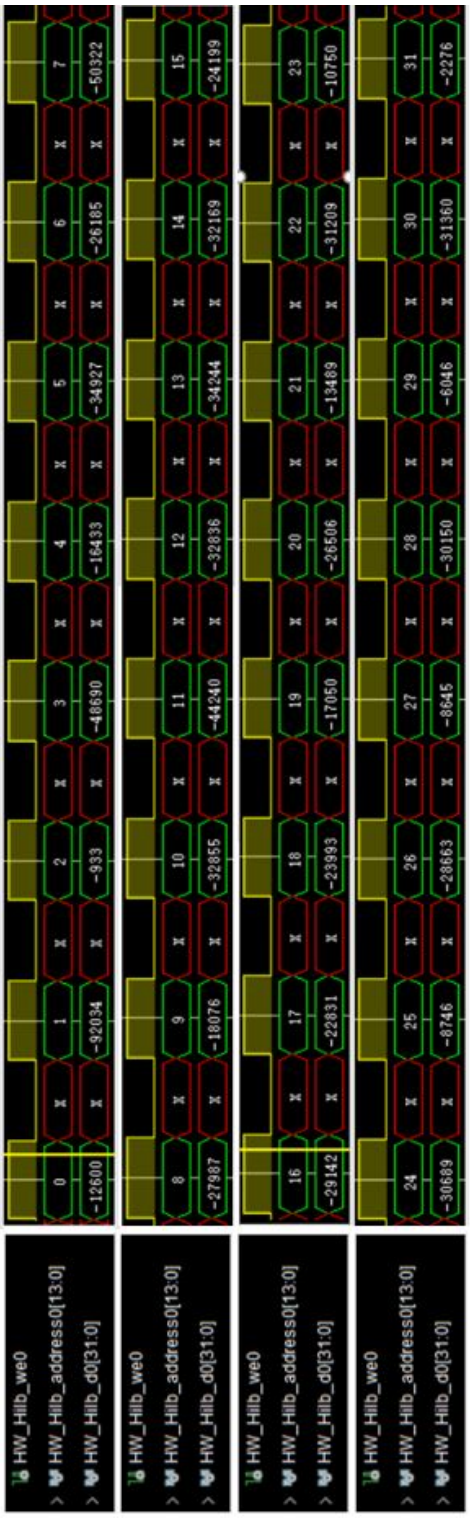
pre-synthesis C test bench and reuse it for verification on the output RTL. However, our current C code, which is automatically generated from MATLAB Coder, does not necessarily require a C test bench as the MEX file in MATLAB Coder has played the role of verifying the C code. For the purpose of automatically performing the post-synthesis verification in Vivado HLS, a C test bench has to be created in the first place.

Moreover, as demonstrated in Chapter 3, our PW Stolt's migration algorithm involves data-intensive computations, such as 4096/256-point FFTs and a large num-

ber of multiplication operations in the RAMAP/MULTIPLY block (see Fig. 3.2), which make the simulation for the entire data frame impractical to perform within a short period of time. Besides that, the complexity of floating-point operations themselves also dramatically prolong the simulation process. For these reasons, we simulate a toy case using fixed-point arithmetic to ensure that our simulation is not excessively time-consuming.

In the MATLAB code of our fixed-point toy case, the size of dataset for each frame is shrank by a factor of 16 to 256-by-16. The sizes of other pre-computed matrices or arrays, such as `M`, `A`, and `R` (see Fig. 3.2) also get reduced accordingly. Note that the precision of all fixed-point data used in the toy case remains unchanged. Then we feed the modified MATLAB code to MATLAB Coder to get the C code and create the C testbench which is used to prepare the data for all input arguments of the entry-point function before calling it. To obtain the values of the input arguments in the C testbench, the MATLAB data fed into each input argument of the entry-point function (*stolt_hardware_fi.m*) are exported and saved as an individual file. For our fixed-point MATLAB code, each fixed-point number can be regarded as an integer with a binary point at a particular position depending on the lengths of integer and fraction parts. As a result, before we export the fixed-point data, they need to be firstly converted to their integer interpretation by shifting the binary point to the right-hand side of the least significant bit (LSB). The files storing the equivalent integer representation of the fixed-point numbers are then fed into the C testbench. The C testbench is also responsible for recording the values of outputs from the entry-point function to verify the correctness of the C testbench itself. (Our PW Stolt's algorithm in C has already been verified using the MEX file in MATLAB Coder.)

Next, we pass our toy case in C along with the testbench to Vivado HLS and run C simulation. The results obtained from C simulation match with those from MATLAB, indicating that our testbench is functionally correct. We then run C synthesis, followed by C/RTL cosimulation. C/RTL cosimulation provides the actual values of latency for the simulation, which is 384783 cycles for our toy case. We can also review the waveform after C/RTL cosimulation. Fig. 4.23a shows a portion of values and addresses of the output data to be written to the external memory after the *Hilbert* block. Comparing with the data obtained from MATLAB shown in Fig 4.23b (after conversion to interger representation), which act as our reference, we can find that these two sets of data are identical indicating the correctness of the RTL output generated from Vivado HLS.



(a) Waveform results

16x768 int32

1	2	3	4	5	6	7	8
-12600	-92034	-933	-48690	-16433	-34927	-26185	-50322
9	10	11	12	13	14	15	16
-27987	-18076	-32855	-44240	-32836	-34244	-32169	-24199
17	18	19	20	21	22	23	24
-29142	-22831	-23993	-17050	-26506	-13489	-31209	-10750
25	26	27	28	29	30	31	32
-30689	-8746	-28663	-8645	-30150	-6046	-31360	-2276

(b) MATLAB results

Figure 4.23: Comparison between waveform and MATLAB results

We then identify the enable signal in the Verilog code which controls the execution of each sub computational block and add them to the waveform window. As shown in Fig. 4.24, the actual FPGA computational flow perfectly matches the algorithmic flow shown in Fig. 3.2. Note that the N_x^{FFT}-point FFT module is instantiated only once and shared before and after Remap/Multiply block, because those two N_x^{FFT}-point FFT blocks are identical in terms of the bit width of data (16 bits) for computation and the way of block execution. However, two different N_t^{FFT}-point FFT modules are instantiated and each one is responsible for one N_t^{FFT}-point FFT block, respectively. This is because the first N_t^{FFT}-point FFT block requires 16-bit data for computation and a trick is employed to reduce the number of iterations by combining two real-valued sequence into one complex-valued sequence (see section 3.2), whereas the second N_t^{FFT}-point FFT block requires 24-bit data and no such trick is used throughout the computation.

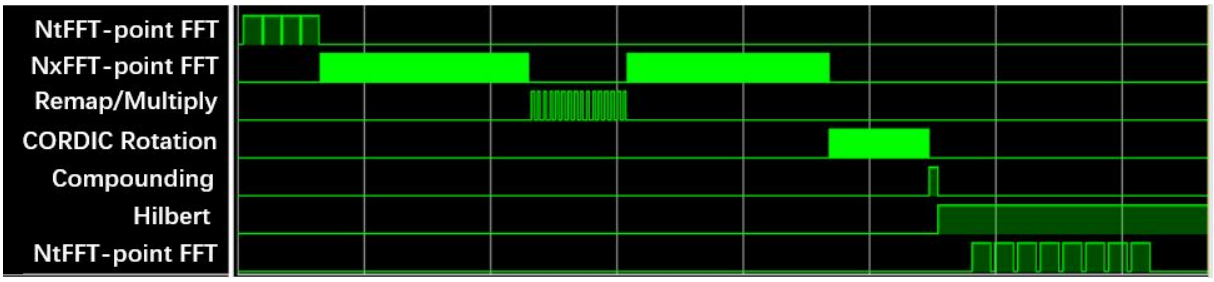


Figure 4.24: FPGA computational flow

4.6 Example of Design Exploration

Vivado HLS has the ability to quickly explore different FPGA implementation options with various performance and resource utilization by modifying the C code used for synthesis. As discussed in section 4.1, in our FPGA implementation workflow, the C code is automatically obtained from MATLAB code using MATLAB Coder. Therefore, changing C code for a different RTL design is equivalent to changing the original MATLAB code. This section shows an example of getting a different RTL design by changing the MATLAB code based on the toy case in section 4.5.

For the RTL implementation discussed before, N_x^{FFT}-point FFT block is executed multiple times sequentially before and after Remap/Multiply block, accounting for a large portion of the final latency. Alternatively, a RTL design which takes advantage

of parallel execution of this block can result in a shorter latency at the cost of a higher resource utilization due to multiple copies of that block generated in the HDL code.

Listing 4.1 shows a MATLAB snippet which enables 4 parallel execution of the N_x^{FFT}-point FFT block before Remap/Multiply block. Compared to the original MATLAB code shown in listing 3.8, the increment value of the outer for loop in the modified version is increased to 4 (line 9 in listing 4.1), resulting in a reduction of the total number of iterations by a factor of 4. Accordingly, for each iteration, 4 sets of the real and imaginary data array pairs are created from the local data matrix (ReF and ImF in listing 4.1) row by row. The assignments mentioned above are performed pointwise in order to ensure that those 4 array pairs are not optimized by MATLAB Coder during the C code generation process (line 11-18 in listing 4.1). After scaling equalization, 4 FFT functions are invoked (line 28-31 in listing 4.1), with each one responsible for the computation of one array pair. Other input arguments related to FFT computation (which are not shown in listing 4.1) also need to have an individual copy for each function call if they are arrays rather than numbers to ensure parallel execution. For the N_x^{FFT}-point FFT block after Remap/Multiply block, similar techniques of changing the MATLAB code are applied as well.

```
1  % -- transform from F-x to F-Kx domain
2  % NtFFT_half -- half of temporal FFT size
3  % ReF/ImF -- real/imaginary part of dataset F in Fig. 3.1
4  % Re/Im(1,2,3,4) -- intermediate 1-dimensional real/imaginary data
5  % maxS -- maximum scaling factor from previous step
6  % split_radix_FFT_spatial -- spatial split-radix FFT function block
7  % bitsra -- bit shift right arithmetic
8
9  for index = 1:4:NtFFT_half-3
10     for index_2 = 1:NxFFT
11         Re1(index_2) = ReF(index,index_2);
12         Im1(index_2) = ImF(index,index_2);
13         Re2(index_2) = ReF(index+1,index_2);
14         Im2(index_2) = ImF(index+1,index_2);
15         Re3(index_2) = ReF(index+2,index_2);
16         Im3(index_2) = ImF(index+2,index_2);
17         Re4(index_2) = ReF(index+3,index_2);
18         Im4(index_2) = ImF(index+3,index_2);
19     end
20 % -- scaling equalization
```

```
21      for x = 1:Nx
22          dS = maxS - S(x);
23          Re1(x) = bitsra(Re1(x),dS);  Im1(x) = bitsra(Im1(x),dS);
24          Re2(x) = bitsra(Re2(x),dS);  Im2(x) = bitsra(Im2(x),dS);
25          Re3(x) = bitsra(Re3(x),dS);  Im3(x) = bitsra(Im3(x),dS);
26          Re4(x) = bitsra(Re4(x),dS);  Im4(x) = bitsra(Im4(x),dS);
27      end
28      [Re1,Im1,Smax1] = split_radix_FFT_spatial(Re1,Im1,NxFFT,..);
29      [Re2,Im2,Smax2] = split_radix_FFT_spatial(Re2,Im2,NxFFT,..);
30      [Re3,Im3,Smax3] = split_radix_FFT_spatial(Re3,Im3,NxFFT,..);
31      [Re4,Im4,Smax4] = split_radix_FFT_spatial(Re4,Im4,NxFFT,..);
32
33  ......
34
35  end
```

Listing 4.1: MATLAB snippet for parallel execution

The utilization report shown in Fig. 4.25 is generated after the C code generated from MATLAB Coder runs C synthesis in Vivado HLS. For comparison purposes, the report for the toy case employing sequential execution in section 4.5 is also shown in Fig. 4.26. The parallel execution scenario consume more resources in each category with respect to the sequential one.

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	4680
FIFO	-	-	-	-
Instance	38	381	41002	153686
Memory	15	-	624	78
Multiplexer	-	-	-	3759
Register	-	-	3087	-
Total	53	381	44713	162203
Available	2940	3600	866400	433200
Utilization (%)	1	10	5	37

Figure 4.25: Utilization report for parallel execution

Then we run the C/RTL cosimulation using the same C testbench as in section 4.5 and obtain a simulation latency of 270610 cycles. Compared to the simulation latency for the sequential execution, which is 384783 cycles (see section 4.5), the latency is

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	3279
FIFO	-	-	-	-
Instance	23	201	22657	82511
Memory	15	-	208	26
Multiplexer	-	-	-	2617
Register	-	-	2024	-
Total	38	201	24889	88433
Available	2940	3600	866400	433200
Utilization (%)	1	5	2	20

Figure 4.26: Utilization report for sequential execution

reduced by 29.7%.

After checking the correctness of waveforms, we extract the signals controlling the execution of each sub computational block and add them to the waveform window, which is shown in Fig. 4.27. Compared to the waveforms for sequential execution shown in Fig 4.24, the red box in Fig. 4.27 shows that 4 N_x^{FFT}-point FFT modules are instantiated and executed in parallel, which leads to the reduction of the final latency.

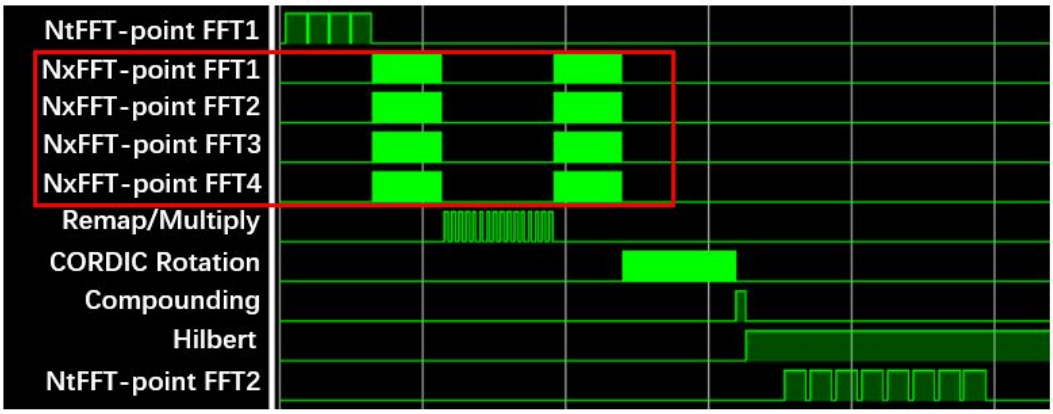


Figure 4.27: Parallel computation flow

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Our work deals with the MATLAB and hardware implementation of one of the recently proposed Fourier-domain CPWC reconstruction methods from [2], namely, PW Stolt's migration algorithm. In Chapter 3, we have presented its MATLAB implementation using both floating-point and fixed-point arithmetic. The floating-point version uses a data type of 32-bit single for all non-integer data representation. For the fixed-point reconstruction, the wordlength of 24 bits is used for the results after compounding and Hilbert transform and 16 bits for the rest of the computations; the scaling factors and equalization blocks are also introduced to enforce the range restriction. We then performed quantitative evaluation of the results, showing that the fixed-point and floating-point versions of CPWC image reconstruction are practically indistinguishable.

In Chapter 4, we have described our workflow of using MATLAB Coder and Vivado HLS for the hardware implementation of an algorithmic specification written in MATLAB. Based on that, Xilinx FPGA implementations of both fixed- and floating-point versions of PW Stolt's migration algorithm have been generated and verified. Results show that the fixed-point FPGA implementation is more resource and power efficient and can also operate at a higher clock frequency compared to its floating-point counterpart. Apart from that, another possible design option has been provided by taking advantage of parallel execution, which illustrates one of the performance-enhancing possibilities and Vivado HLS's ability to quickly implement it.

5.2 Future Work

There are several promising directions that one can pursue based on the work presented in this thesis.

In section 4.5, we have verified our design by using a toy case with one emission angle and a data size of 256-by-16 to avoid a prolonged simulation process. Our first suggestion for the future work is to perform the C/RTL co-simulation for the entire frame with all angles compounded in Vivado HLS, followed by the FPGA-in-the-loop (FIL) verification, which allows us to use Simulink or MATLAB software to thoroughly test our RTL design of PW Stolt's method in the target FPGA.

In section 4.6, we have given an example to show how to explore a different design option. In the future, one can also investigate other design options by either changing the original MATLAB code or adding optimization directives from Vivado HLS. Possible design options include using pipelining execution, loop unrolling, array partitioning, etc. Then, one can compare different design alternatives and find an optimal solution.

Our third suggestion for the future work is to investigate different ways of getting HDL code. For example, instead of using MATLAB, we can start from SystemC, which also has fixed-point support, and then convert SystemC into HDL using Vivado HLS. Another alternative is to use MATLAB HDL Coder, which can generate portable, synthesizable Verilog and VHDL code directly from MATLAB. One can even perform manual optimizations of an HDL description by hand-coding the RTL design.

The last suggestion is to explore the possibility of implementing PW Stolt's migration algorithm using the posit arithmetic [44]. It provides compelling advantages over floating-point arithmetic, including larger dynamic range, higher accuracy, simpler hardware, etc. Therefore, it will be worthwhile to investigate how posit arithmetic affects the FPGA resources, performance, and power consumption in the context of our target application.

Appendix A

CORDIC Phase Rotation

CORDIC, which stands for COordinate Rotation DIgital Computer, is an iterative algorithm suited to hardware implementations because it requires only iterative shift-add operations [45] and eliminates the need for explicit multipliers. Using CORDIC, one can calculate the trigonometric functions of sine, cosine, magnitude and phase (arctangent) to any desired precision.

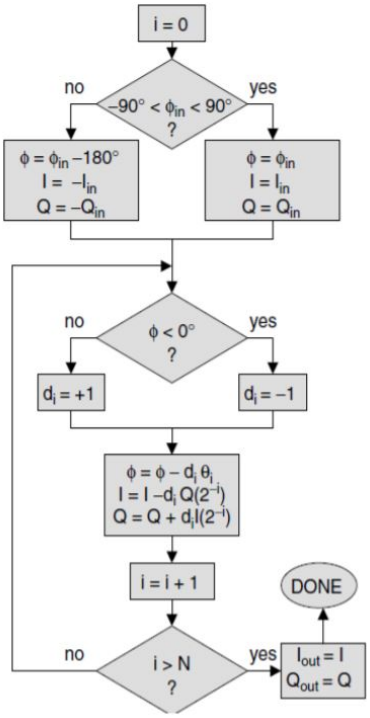


Figure A.1: CORDIC algorithm flow chart [7]

Fig. A.1 is a flow chart showing CORDIC algorithm to implement phase rotations

described in Section 3.3. The inputs to the algorithm include I_{in} (the real part), Q_{in} (the imaginary part), and Φ_{in} (the desired phase shift). The variable i, which is initialized to zero, keeps track of the processing stage being performed. The algorithm loops through N iterations for the purpose of driving the residual phase error, Φ, to zero. In each iteration, a new Φ value is obtained by subtracting or adding a tabulated θ_i value to the previous value of Φ. In each stage, the Q (or I) input is divided by a factor of 2^i, using an arithmetic shift to the right by i bits. The result is then added to or subtracted from the I (or Q) input, depending on the sign of Φ. The variable i is incremented as the process repeats, and the phase-shifted results are available after this algorithm completes N interations.

Bibliography

[1] A. Hadzic, *Hadzic's Peripheral Nerve Blocks and Anatomy for Ultrasound-guided Regional Anesthesia*. McGraw Hill Professional, 2011.

[2] M. Albulayli and D. Rakhmatov, "Fourier domain depth migration for plane-wave ultrasound imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 65, no. 8, pp. 1321–1333, 2018.

[3] P. Barry and P. Crowley, *Modern Embedded Computing: Designing Connected, Pervasive, Media-rich Systems*. Elsevier, 2012.

[4] 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 Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 56, no. 3, pp. 489–506, 2009.

[5] J. Shi and D. Rakhmatov, "Fixed-point CPWC ultrasound image reconstruction," in *2019 IEEE International Ultrasonics Symposium (IUS)*. IEEE, 2019, pp. 1282–1285.

[6] Xilinx, *UG-998 Introduction to FPGA Design with Vivado High-Level Synthesis*, January 2019.

[7] S. R. Triveni.C, "Implementation of phase shifter using cordic on FPGA for RADAR application," *International Journal of Advanced Research in Electronics and Communication Engineering (IJARECE)*, vol. 5, 2016.

[8] M. Ali, D. Magee, and U. Dasgupta, "Signal processing overview of ultrasound systems for medical imaging," *SPRAB12, Texas Instruments, Texas*, 2008.

[9] S. ShahbazPanahi and Y. Jing, "Recent advances in network beamforming," in *Academic Press Library in Signal Processing, Volume 7*. Elsevier, 2018, pp. 403–477.

[10] R. S. Cobbold, *Foundations of Biomedical Ultrasound*. Oxford University Press, 2006.

[11] J. A. Jensen, "Medical ultrasound imaging," *Progress in Biophysics and Molecular Biology*, vol. 93, no. 1-3, pp. 153–165, 2007.

[12] H. Liebgott, A. Rodriguez-Molares, F. Cervenansky, J. A. Jensen, and O. Bernard, "Plane-wave imaging challenge in medical ultrasound," in *2016 IEEE International Ultrasonics Symposium (IUS)*. IEEE, 2016, pp. 1–4.

[13] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli *et al.*, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[14] T. Szabo, *Diagnostic Ultrasound Imaging: Inside Out*. MA: Elsevier, 2014.

[15] M. Tanter and M. Fink, "Ultrafast imaging in biomedical ultrasound," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 61, no. 1, pp. 102–119, 2014.

[16] W. T. Padgett and D. V. Anderson, "Fixed-point signal processing," *Synthesis Lectures on Signal Processing*, vol. 4, no. 1, pp. 1–133, 2009.

[17] Xilinx, *UG902 Vivado Design Suite User Guide: High-Level Synthesis*, July 2018.

[18] Intel, *UG-20266 Intel High Level Synthesis Compiler Standard Edition User Guide*, December 2019.

[19] MathWorks, *HDL Coder User's Guide*, 2018.

[20] Ö. Yilmaz, *Seismic Data Analysis: Processing, Inversion, and Interpretation of Seismic Data*. Society of exploration geophysicists, 2001.

[21] D. Garcia, L. Le Tarnec, S. Muth, E. Montagnon, J. Porée, and G. Cloutier, "Stolt's fk migration for plane wave ultrasound imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 60, no. 9, pp. 1853–1867, 2013.

[22] J. Cheng and J.-y. Lu, "Extended high-frame rate imaging method with limited-diffraction beams," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 53, no. 5, pp. 880–899, 2006.

[23] D. D. Liu and T.-L. Ji, "Plane wave image formation in spatial-temporal frequency domain," in *2016 IEEE International Ultrasonics Symposium (IUS)*. IEEE, 2016, pp. 1–5.

[24] A. Besson, M. Zhang, F. Varray, H. Liebgott, D. Friboulet, Y. Wiaux, J.-P. Thiran, R. E. Carrillo, and O. Bernard, "A sparse reconstruction framework for Fourier-based plane-wave imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 63, no. 12, pp. 2092–2106, 2016.

[25] J.-y. Lu, "2D and 3D high frame rate imaging with limited diffraction beams," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 44, no. 4, pp. 839–856, 1997.

[26] P. Kruizinga, F. Mastik, N. de Jong, A. F. van der Steen, and G. van Soest, "Plane-wave ultrasound beamforming using a nonuniform fast Fourier transform," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 59, no. 12, pp. 2684–2691, 2012.

[27] A. Eklund, P. Dufort, D. Forsberg, and S. M. LaConte, "Medical image processing on the gpu–past, present and future," *Medical Image Analysis*, vol. 17, no. 8, pp. 1073–1094, 2013.

[28] T. Y. Phuong and J.-G. Lee, "Design space exploration of sw beamformer on gpu," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 7, pp. 1718–1733, 2015.

[29] B. Y. Yiu, I. K. Tsang, and C. Alfred, "Real-time gpu-based software beamformer designed for advanced imaging methods research," in *2010 IEEE International Ultrasonics Symposium (IUS)*. IEEE, 2010, pp. 1920–1923.

[30] J. W. Choe, A. Nikoozadeh, Ö. Oralkan, and B. T. Khuri-Yakub, "Gpu-based real-time imaging software suite for medical ultrasound," in *2013 IEEE International Ultrasonics Symposium (IUS)*. IEEE, 2013, pp. 2057–2060.

[31] H. Hewener and S. Tretbar, "Mobile ultrafast ultrasound imaging system based on smartphone and tablet devices," in *2015 IEEE International Ultrasonics Symposium (IUS)*. IEEE, 2015, pp. 1–4.

[32] I. S. Uzun, A. Amira, and A. Bouridane, "FPGA implementations of fast Fourier transforms for real-time signal and image processing," *IEEE Proceedings-Vision, Image and Signal Processing*, vol. 152, no. 3, pp. 283–296, 2005.

[33] J. Amaro, B. Y. Yiu, G. Falcao, M. A. Gomes, and C. Alfred, "Software-based high-level synthesis design of FPGA beamformers for synthetic aperture imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 62, no. 5, pp. 862–870, 2015.

[34] J. A. Jensen, H. Holten-Lund, R. T. Nilsson, M. Hansen, U. D. Larsen, R. P. Domsten, B. G. Tomov, M. B. Stuart, S. I. Nikolov, M. J. Pihl *et al.*, "Sarus: A synthetic aperture real-time ultrasound system," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 60, no. 9, pp. 1838–1852, 2013.

[35] J.-y. Lu, J. Cheng, and J. Wang, "High frame rate imaging system for limited diffraction array beam imaging with square-wave aperture weightings high frame rate imaging system for limited diffraction array beam imaging with square-wave aperture weightings," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 53, no. 10, pp. 1796–1812, 2006.

[36] P. Tortoli, L. Bassi, E. Boni, A. Dallai, F. Guidi, and S. Ricci, "ULA-OP: An advanced open platform for ultrasound research," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 56, no. 10, pp. 2207–2216, 2009.

[37] H. Sorensen and C. Burrus, "Fast DFT and convolution algorithms," in *Handbook for Digital Signal Processing*, S. Mitra and J. Kaiser, Eds. NY: Wiley, 1993, ch. 8, pp. 491–610.

[38] J. Muller, *Elementary Functions*, 3rd ed. NY: Springer, 2016.

[39] L. Marple, "Computing the discrete-time "analytic" signal via FFT," *IEEE Transactions on Signal Processing*, vol. 47, no. 9, pp. 2600–2603, 1999.

[40] Altera, *DSP Design Flow User Guide*, June 2009.

[41] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt, "An overview of today's high-level synthesis tools," *Design Automation for Embedded Systems*, vol. 16, no. 3, pp. 31–51, 2012.

[42] MathWorks, *MATLAB Coder User's Guide*, 2019.

[43] Xilinx, *UG-906 Design Analysis and Closure Techniques*, June 2012.

[44] J. L. Gustafson and I. T. Yonemoto, "Beating floating point at its own game: Posit arithmetic," *Supercomputing Frontiers and Innovations*, vol. 4, no. 2, pp. 71–86, 2017.

[45] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the May 18-20, 1971, spring joint computer conference*, 1971, pp. 379–385.