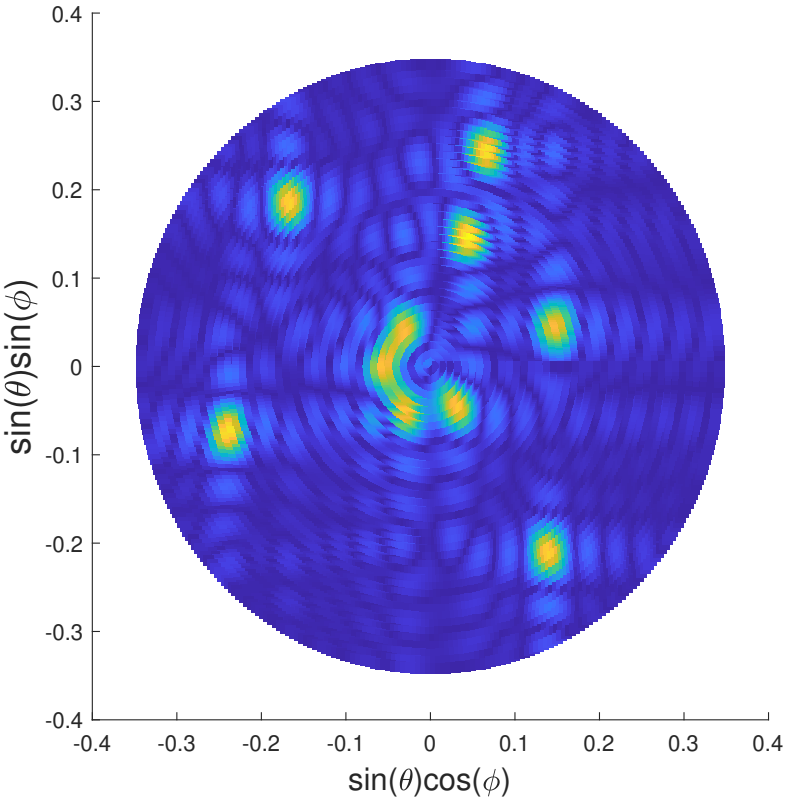
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Beamforming of Multi-Channel Digital Radar System on System-on-Chip

Master's thesis in Embedded Electronic System Design

FREDRIK JOHANSSON
LUKAS ÖGNELOD

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Beamforming of Multi-Channel Digital Radar System on System-on-Chip

FREDRIK JOHANSSON
LUKAS ÖGNELOD

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Beamforming of Multi-Channel Digital Radar System on System-on-Chip

FREDRIK JOHANSSON
LUKAS ÖGNELOD

Supervisor: Arne Linde, Department of Computer Science and Engineering
Advisor: Christian Takvam, Saab AB
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis, Spring 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Circular sweep of beams from implemented system on hardware. Ten signal sources can be observed.

Typeset in LaTeX
Gothenburg, Sweden 2022

Beamforming of Multi-Channel Digital Radar System on System-on-Chip
FREDRIK JOHANSSON
LUKAS ÖGNELOD
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

When speaking about radar systems, most people might straight away think about them being used for military operations but radar systems today are a part of everyday life. To deal with increasing amounts of data from larger antenna arrays while still having an agile system, FPGAs are a natural choice when dealing with high throughput applications as beamforming.
This project explores the capabilities of the Xilinx Versal VCK190 ACAP, combining FPGA with a SIMD processor architecture called AI Engine to perform digital beamforming on radar signals. We realised an implementation of a digital beamforming system on a Versal VCK190 and were able to run continuously achieving a throughput of up to 7.8 Msamples/s.

Sammanfattning

Radar är ett brett område med en stor variation av problem som kräver specifika lösningar. Samtidigt finns det ett ökat intresse för informationsinsamling där radar visat sig användbart i t.ex. antikollisionssystem i bilar. Det ökade behovet av tillgång till information betyder att mer data behöver bearbetas. Idag används FPGAs till de flesta högpresterande radarsystem tack vare deras flexibilitet och kapacitet att transportera och bearbeta stor mängd data.
Detta projekt utforskar kapaciteten av Xilinx Versal VCK190 ACAP, som kombinerar FPGA med en SIMD processorarkitektur, kallad AI Engine för att utföra digital lobformning på radarsignaler. Vi tog fram en implemetation för ett digitalt lobformningssytem på en Versal VCK190 och vi lyckades uppnå en kontinuerlig genomströmning uppåt 7.8 Msamples/s.

Keywords: Computer science, Radar, beamforming, programmable hardware, FPGA, Xilinx Versal, AI Engine, AESA.

Acknowledgements

We would like to thank Saab AB for giving us the opportunity to work with this interesting project. A special thanks to Christian Takvam for his help supervising this project at the company and to Arne Linde for his role as supervisor from Chalmers. We would also like to thank last years master's thesis students Christoffer Krull and Gustav Holst for helping, answering our daily questions regarding programming and implementing on Xilinx Versal VCK190. We would also like to thank Daniel Wallström at Saab AB for allowing us to work on this project. Lastly we would like to thank Per Larsson-Edefors for his role as examiner during the project.

Fredrik Johansson & Lukas Ögnelod, Gothenburg, July 2022

Contents

List of Figures xi

List of Tables xiii

List of Abbreviations xv

1 Introduction 1
- 1.1 Problem description . . . 2
- 1.2 Limitations . . . 2
- 1.3 Thesis outline . . . 2

2 Technical Background 5
- 2.1 Pulse-Doppler radar . . . 5
- 2.2 Beamforming . . . 6
- 2.3 Active Electronically Scanned Array . . . 7
  - 2.3.1 Array elements . . . 8
  - 2.3.2 Transmitter Receiver Modules . . . 8
  - 2.3.3 Beamformer . . . 9
- 2.4 Digital beamforming . . . 9
- 2.5 Versal architecture . . . 11
  - 2.5.1 Scalar units . . . 11
  - 2.5.2 Programmable Logic . . . 11
  - 2.5.3 AI Engine . . . 12
  - 2.5.4 Network-on-chip . . . 13
  - 2.5.5 Kahn process network . . . 14
  - 2.5.6 PetaLinux . . . 14

3 Methods and Implementation 15
- 3.1 Engine programming . . . 15
  - 3.1.1 Intrinsics . . . 15
  - 3.1.2 Kernels/AI Tiles . . . 16
- 3.2 Design flow . . . 16
- 3.3 System architecture . . . 17
  - 3.3.1 Splitters . . . 18
  - 3.3.2 Partial beamformer . . . 18
  - 3.3.3 Weight calculations . . . 19
  - 3.3.4 Adders . . . 20

3.3.5 Program controller 20
3.4 Generation of test data 21

4 Results **23**
4.1 AI Engine simulation evaluation 23
4.1.1 Figures of Merits 23
4.1.2 Simulation results 23
4.2 Hardware evaluation 24
4.2.1 Figures of Merits 24
4.2.2 Hardware results 25
4.2.3 Resource usage 26
4.2.4 Accuracy 26

5 Discussion **31**
5.1 Result comparison 31
5.2 Hardware accuracy 32
5.3 Design choices 32
5.3.1 PL adders 32
5.3.2 Placement of kernels 33
5.3.3 Hardware conversion 33
5.4 Ethical considerations 33

6 Conclusion **35**
6.1 Future research 35
6.1.1 Actual radar target testing 35
6.1.2 Scaling up the system 35
6.1.3 Combining usage 36

List of Figures

2.1 A: One antenna element radiating in all directions with no beamforming. B: Four antenna elements beamforming at 0°. C: Four antenna elements beamforming at -30°. 6
2.2 1D array of antenna elements listening to incoming signal from θ. Adapted from [14] 7
2.3 Overview of a basic AESA system, antennas going to each TRM. Each TRM is then added together in the beamformer and sent to the output. Adapted from [14]. 8
2.4 Basic layout of a TRM. Adapted from [17]. 9
2.5 The layout of Xilinx Versal VCK190. Each processing area is connected via the AXI-4 based NoC. Adapted from [23]. 11
2.6 The AI Engine hierarchy, from array to tile to instruction word . . . 12
2.7 Interconnections and communications for an AI Tile. 13
2.8 A small Kahn process network with four tasks. 14

3.1 Design flow for the system. 17
3.2 The basic architecture of the beamformer. Samples get split, and each PBF performs its calculations separately before being added together and sent out as M beams. 18
3.3 One Tile and Cascade versions of partial beamformer. Each square is one AI Tile used and the arrows show the dataflow. 18
3.4 16x16 weights divided into 4x4 matrix and then stored in memory as blocks. 19

4.1 Average speed of implemented system running 16 million sample points. 25
4.2 Average speed of implemented system running 256 million sample points. 25
4.3 Circular sweep without noise for Matlab and hardware results. 27
4.4 The difference between Matlab and hardware using logarithmic scale. 27
4.5 Square sweep with noise for Matlab and Hardware results. 28
4.6 The difference between Matlab and hardware using logarithmic scale. 29

List of Tables

4.1 FoM layout for simulations. 23
4.2 Kernel runtime for weight calculation. 24
4.3 Simulated maximum throughput for Cascade implementation of PB. . 24
4.4 Simulated maximum throughput for One Tile implementation of PB. 24
4.5 Throughput for different implementations. Values are calculated from those shown in Fig 4.2. 26
4.6 Resource usage: Total resource usage of PB and addition of partial beams. 26
4.7 Power consumption for One Tile and Cascade implementation. 26

List of Abbreviations

ACAP	Adaptive compute acceleration platform
AESA	Active electronically scanned array
ARM	Advanced RISC machine
AXI	Advanced extensible interface
DMA	Direct memory access
DSP	Digital signal processor
FDB	Full digital beamfroming
FIFO	First In First Out
FoM	Figure of merit
FPGA	Field programmable gate array
HLS	High-level synthesis
HPA	High power amplifier
IDE	Integrated development environment
IP	Intellectual property
IQ	In-phase and quadrature components
KPN	Kahn process network
LNA	Low noise amplifier
LUT	Look up table
MAC	Multiply–accumulate
NoC	Network on chip
PB	Partial beamformer
PL	Programmable logic
RAM	Random accesses memory
RF	Radio frequency
SIMD	Similar instruction multiple data
SoC	System on chip
TRM	Transmit receive module
VLIW	Very long instruction word

1

Introduction

Radar (radio detection and ranging) was first coined in 1943 by an American officer and has since been the general term for when discussing radio frequency (RF) waves [1]. Radar systems were initially developed for military applications, but today radar systems are an integrated part of various areas such as navigation [2], automotive anti-collision [3], astronomy [4], and weather [5][6].

Military radar systems have for a long time foregone gimbal structured antennas and nowadays consist of mechanically fixed antenna arrays called Active Electronically Scanned Array (AESA). Focusing the radar signal is done by either delaying or phasing each antenna element's incoming/outgoing signal; this is a technique known as beamforming. Beamforming refers to spatially filtering out and amplifying signals with the use of wave interference. Two or more antenna elements interfere with each other and create a beam-like attribute that can be steered by phasing signals in the wanted direction. Beamforming is a vital component in radar communication devices for filtering and amplifying their signals. The boundary for beamforming between digital and analogue signals moves closer toward each singular antenna element.

Specifically for large AESA radar implementations with many array elements, keeping power consumption low in each transmitter/receiver module (TRM) is essential. Moving some or all of the analogue phase shifting and mixing to the digital domain can reduce power consumption in each element. However, going to the digital domain creates higher data flow and require high speed and accuracy computing capacity to perform the phase shift. Programmable hardware such as FPGAs is often used to deal with the high throughput, making the Versal platform attractive.

Xilinx, one of the world-leading manufacturers of FPGAs and System-on-Chip(SoC), provides with their latest Versal VCK190 circuit a single instruction multiple data (SIMD) processor matrix, AI Engine [7]. The AI Engine is designed for high throughput vector operations used in signal processing and machine learning applications. The AI Engine consists of 400 AI Engine Tiles arranged in a 2D grid where each tile has a Very Long Instruction Word (VLIW) SIMD architecture with its data and instruction cache. Similar architectures have previously been proposed and used for beamforming radar [8]. With claims from the manufacture of a reduction in power consumption by 40% compared to standard programmable logic, the AI Engine was chosen to be tested on a AESA system [7].

This project aims to study and evaluate the gains of using a SoC implementation and compare results to a pure FPGA implementation for beamforming.

1.1 Problem description

The Xilinx Versal VCK190 presents a new type of programmable hardware in its AI Engine, with a GPU-like structure and customisable processors is an interesting case to study. This project will integrate a previous FPGA AESA implementation on this circuit to observe the performance and utilisation of the new AI Engine architecture.

The project will also evaluate the usability of the AI Engine in regards to beamforming and examine how many beams it can calculate and how it could scale with the number of array elements. The suitability of different algorithms for the platform will be evaluated, and the theoretical throughput will be analysed.

1.2 Limitations

One main limitation will be that since we aim to evaluate the performance of the AI Engine within a Versal VCK190 platform, we are bound to this specific hardware and will make design choices for that hardware.

A beamforming unit is a small part of an entire radar system, and this thesis will not look at any integration with other features, such as last year's thesis [9] where Holst and Krull evaluated the AI Engine for object identification.

The radar system will be assumed stationary and monostatic, meaning only observable objects are moving. The system's whose transmitter and receiver are placed at the same location. This work will not explore changes to the antenna array's physical layout.

The thesis will not explore the usage of machine learning or similar methods and will not consider possible post-processing of the beams.

Due to time constraints, the system will not be tested on an actual radar antenna, instead only running using generated test data.

1.3 Thesis outline

The thesis follows a modified IMRAD structure [10], with an introduction, theory, methods, implementation, results and discussion. Chapter 2 begins the theory where the first section presenting pulse-Doppler radar, its principles, and how data is created for a radar system. The following section describes AESAs with each part of the system, array elements, transmit-receive module, and beamformer. The chapter continues with a section regarding digital beamforming, with an introduction to the math behind digital beamforming and its advantages. Lastly, in the technical background chapter, the Versal architecture is presented.

The theory chapter is followed by methods and implementation in Chapter 3 where our methods and implementations are presented. The chapter starts with what methods were used for implementing our system, and how the design flow was structured.

The results of the measurements are presented in Chapter 4 in the order of simulation followed by hardware. Hardware results are divided into speed, resource usage and accuracy. The results and design choices are discussed in Chapter 5.

Finally, in Chapter 6 our conclusion is presented with examples of future research.

2

Technical Background

This chapter will give a deeper introduction to radar and specify what type of radar data is available from the system. Our implementation will have no interaction with a live radar system and its hardware components during this project. Despite that, it is essential to mention some of their basic functions to understand an AESA system.

This chapter will then introduce AESA systems, outlining each component, how they function, and how they are connected. After that, digital beamforming is introduced, where newer ideas about beamforming and how it is transferred to the digital plane. The last section of this chapter presents the hardware circuit with all its parts.

2.1 Pulse-Doppler radar

The Doppler effect, first mentioned by Christian Doppler in 1842, is a measurable shift in frequency. The frequency shift occurs when the source and observer do not have the same relative velocity [11].

Pulse-Doppler radars work by transmitting RF pulses and listening to their echoes. The range R between the transmission position and the object the pulse echoed off can then be calculated. Equation 2.1 shows the range equation where c is the speed of light in which the RF pulse travels and t is the time between transmission and echo [12].

$$R = \frac{ct}{2} \tag{2.1}$$

Sometimes it is not enough to know the distance to an object. There might also be a need to know if the object is moving towards or away from the transmitter. The Doppler effect claims that the frequency of a wave as measured by the observer depends on the velocity of the source relative to the observer [13].

The time interval between two crests of the wave emitted at t_1 and t_2 is determined by

$$\Delta t_s = t_2 - t_1 = \frac{\gamma}{\lambda} \tag{2.2}$$

where $\gamma = 1/(1 - v_s^2/c^2)^{1/2}$ is a factor that represents the relativistic time dilation and c is the propagation of RF waves (speed of light) [13]. But since the velocities

achieved by objects observed by radar systems are $v_s << c \rightarrow \gamma = 1$. Then the time interval between the arrivals of two crests at the observer is:

$$\Delta t_o = \left(t_2 + \frac{r_2}{c}\right) - \left(t_1 + \frac{r_1}{c}\right) = \frac{1}{f}\left(1 - \frac{v_s \cdot \cos\theta_s}{c}\right) \tag{2.3}$$

where r_1 and r_2 are the respective ranges for a radiating object. Thus the corresponding observed frequency by the observer is:

$$f' = \frac{1}{\Delta t_o} = \frac{f}{1 - \frac{v_s \cdot \cos\theta_s}{c}} \tag{2.4}$$

From that we can calculate the Doppler shift frequency f_d as:

$$f_d = -f\frac{v_s \cos\theta_s}{c} \tag{2.5}$$

For stationary pulse radar systems with transmitter and receiver at the same point, the receiver observes double the frequency shift according to equation 2.6. The double shift comes from the base that the radar bounces off the target.

$$f_d = -f\frac{2v_s \cos\theta_s}{c} \tag{2.6}$$

2.2 Beamforming

Beamforming refers to a way of spatially filtering out and amplifying signals with the use of wave interference. Two or more antenna elements interfering with each other create a beam-like attribute that can be steered by phasing signals towards the desired direction. Beamforming has been a vital component in military radar communication and identification devices in amplifying and disguising their signals and the information they contain. Figure 2.1 displays the general difference between single element transmission and multi-element transmission.

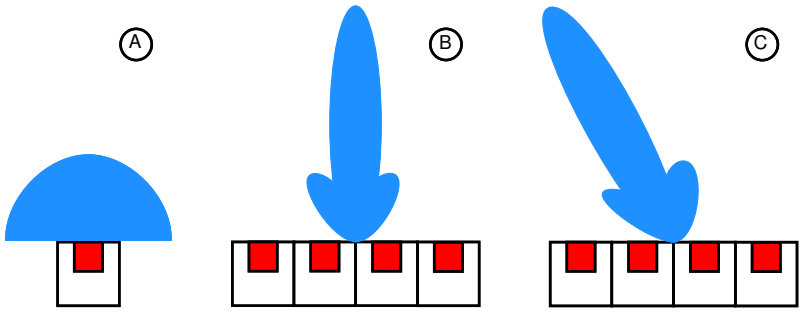


Figure 2.1: A: One antenna element radiating in all directions with no beamforming. B: Four antenna elements beamforming at 0°. C: Four antenna elements beamforming at -30°.

When spatially filtering signals, there will always be some high amplitude wave interference offshoots that differ from the main beam, called sidelobes. The magnitude of the side lobes depends on the array layout and weights used. In Figure

2.1, side lobes can be observed using a four-element 1D-array layout. As long as the side lobes are small enough, they do not impact the system much except for some unwanted noise in the sidelobe direction. With proper system implementation, their effect can be minimised.

2.3 Active Electronically Scanned Array

AESA systems can produce capable, agile, high gain beams used in radar, weather surveillance, and imaging [14]. Instead of conventional reflector antennas, which require a gimbal for beam steering, AESAs electronically scan the space with no mechanical movement needed.

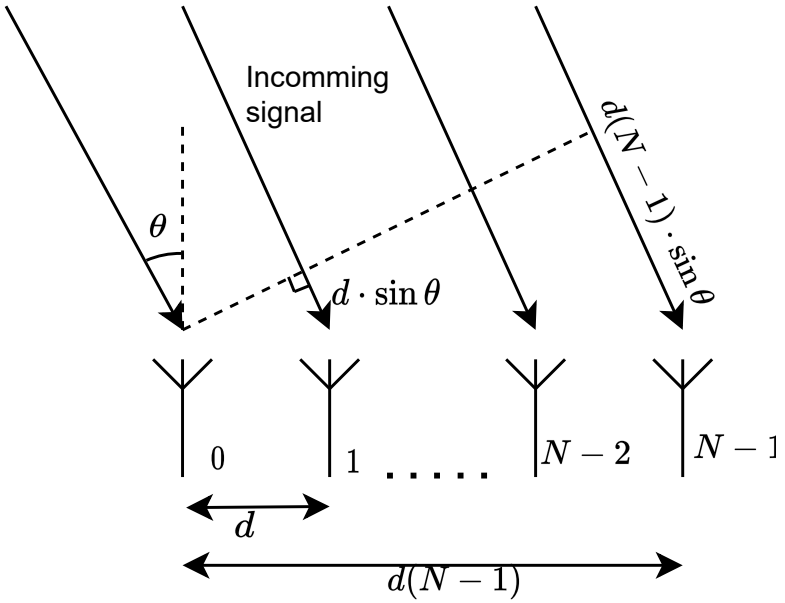


Figure 2.2: 1D array of antenna elements listening to incoming signal from θ. Adapted from [14]

One of the most critical parts of AESAs is the array configuration. The configuration decides on three factors; the number of elements, size, and layout, which in turn can significantly affect performance factors such as maximum gain, accuracy, power consumption, latency, and throughput. With 1D arrays, it is possible to steer the beam in a given direction, see Figure 2.1. For a beam to be steerable in all three dimensions, a 2D array is needed. Depending on implementation and available area, 2D arrays are mainly ordered, either circular or rectangular.

The beamforming part of an AESA system is built up of three parts. A beamformer, transmit/receive modules (TRMs), and antenna elements. When transmitting, the system input consists of an analogue signal. The signal is split in the beamformer into equal parts power-wise and transferred to each TRM. In the TRMs, the signal is phased or delayed, amplified, filtered, and then sent to their respective antenna element.

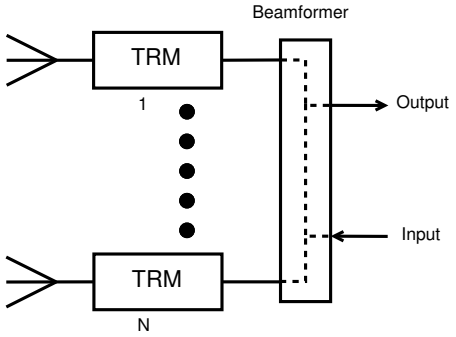


Figure 2.3: Overview of a basic AESA system, antennas going to each TRM. Each TRM is then added together in the beamformer and sent to the output. Adapted from [14].

When receiving, the input consists of an RF signal received by the antenna elements. They are then amplified, filtered, and shifted in their respective TRM before being added together in the beamformer. The beamformer's output is the added signal from each of the TRMs.

2.3.1 Array elements

The array elements are the nodes from which the system transmits and receives information. For large passive arrays, in the area of thousands of array elements, there is a maximal theoretical gain when only increasing the number of elements. This is due to losses caused by line attenuation [15]. For active arrays, since the amplifiers are at element level, they do not suffer from line attenuation and the maximal theoretical gain when increasing the number of elements is infinite.

The array gain is related to each element's gain but also to how the elements are situated to one another in the array [16].

2.3.2 Transmitter Receiver Modules

Transmitter receive modules (TRMs) are the interface between array elements and beamformers. On transmit, they apply a phase on the voltage given from the beamformer that corresponds to the scan direction [17]. The same goes for receiving, but voltage now comes from the array elements [17]. The phase given shown in Equation 2.7 has the required complex weight needed to steer the beam electronically. Here the beam is formed at angle θ_0.

$$A_m = 1 \cdot e^{-j\frac{2\pi}{\lambda_0} md \sin\theta_0} \tag{2.7}$$

A basic TRM consists of one transmit part (8, 9, and 10 in Figure 2.4) one receive part (12, 14, 15, 16 in Figure 2.4), and one phase shift/delay part (3, 4, 5, and 6 in Figure 2.4). The transmit part consists of a pre-amplifier (8), a high power amplifier (9), and a filter (10) to amplify and post ADC filter the analogue wave. The receiving part consists of a receiver protector (12), a low noise amplifier (LNA)(14), a filter (15), and the load (16). Switching between transmit and receive is done by five

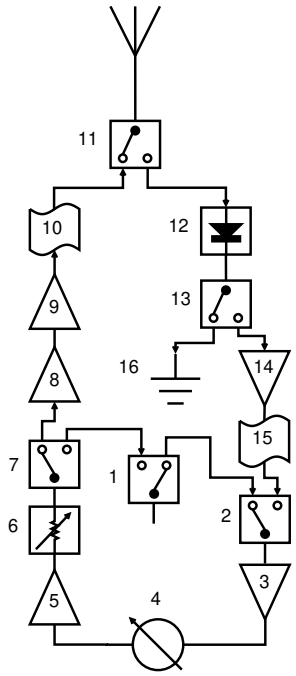


Figure 2.4: Basic layout of a TRM. Adapted from [17].

switches placed as first and last component, see 1 and 11 in Figure 2.4, before and after the phase shift/delay part, see 2 and 7 in Figure 2.4 and one last switch placed within the receiver part shown as 13 in Figure 2.4.

2.3.3 Beamformer

Beamformers in AESA systems distribute RF energy on transmit and combine RF energy on receive. On transmission, they are provided power via a modulated voltage waveform generated at the system exciter. Voltage is then distributed by the beamformer to each TRM [18]. In receive mode, beamformers instead combine receiving phased voltages from each TRM shown in Equation 2.8.

$$V_{out} = \sum_{m=1}^{M} \alpha_m V_{in_m} e^{-j\frac{2\pi}{\lambda_0} x_m \sin\theta_0} \tag{2.8}$$

2.4 Digital beamforming

Given an input vector $\mathbf{x_{Nx1}}$ with the sampled value at time t_0, it is possible to create a beam in the direction of angle θ, as in Figure 2.2, by delaying the signal from the closest antenna or using phase rotation to steer it. To create a digital beam, a weight vector $\mathbf{w}$ is used and with Equation 2.10 gets the resulting power, y, in the steered direction. For a simple 1D array as in Figure 2.2 we can use Equation 2.7 to calculate w as m goes from 0 to $N-1$:

$$\mathbf{w} = \begin{bmatrix} 1 & e^{j\frac{2\pi}{\lambda_0} d \sin\theta} & \dots & e^{j\frac{2\pi}{\lambda_0}(N-1) d \sin\theta} \end{bmatrix} \tag{2.9}$$

$$y_{1\times 1} = \mathbf{w}_{1\times N}^{H}\mathbf{x}_{N\times 1} \tag{2.10}$$

This is the basic version of digital beamforming. However, it can be modified with windowing to reduce the power of sidelobes. Hamming [19], Taylor, and Bayliss windows have all been shown to lower the gain in the sidelobes [20] and can be easily added to the digital domain when required. The downside of full digital beamforming (FDB) is the increase in required computation power as previous analogue operations are performed in the digital domain requiring high throughput. For large systems, using a combination of digital and analogue beamforming, analogue sub-arrays connect multiple antenna elements to a single digital TRM, which is connected to the digital beamformer, can mitigate this problem.

The advantage of digital beamforming vs analogue is that it becomes possible to construct more beams using the same data by changing the angle θ_0 giving us a vector-Matrix multiplication shown in Equation 2.11 where the weights for a single beam are placed in rows.

$$\mathbf{y}_{M\times 1} = \mathbf{W}_{M\times N}\mathbf{x}_{N\times 1} \tag{2.11}$$

Since the signal is in the digital domain, the weights are easy to change, allowing for changing the direction it is looking at and can actively calculate weights to suppress weights.

When suppressing jamming signals, we want to find the optimal weight matrix, $\mathbf{w}_{opt}$ that often uses the MSE of the signal to maximise the signal power and minimise the jamming signals. The vector in Equation 2.9 is referred to as the steering vector, as changing the angle θ will change the angle of the beam in Figure 2.2. If we then call then vector $\mathbf{w}(\theta)$, the co-variance matrix R will be according to Equation 2.12 where θ_0 is the direction of our desired signal, K is the number of angles where jamming is present, θ_k is the angle from a jamming signal, and a_k is its amplitude, σ_n^2 is the noise power and $\mathbf{I}$ is the identity matrix.

$$\mathbf{R} = \mathbf{w}(\theta_0)^H\mathbf{w}(\theta_0) + \sum_{k=1}^{K}|a_k|^2\mathbf{w}(\theta_k)^H\mathbf{w}(\theta_k) + \sigma_n^2\mathbf{I} \tag{2.12}$$

The expected signal power of the output for any given weight vector $\mathbf{w}$ results in:

$$E\{|y|^2\} = \mathbf{w}\mathbf{R}\mathbf{w}^H \tag{2.13}$$

When maximising the signal power, compared to the noise and jamming on the output, the optimal weight vector, $\mathbf{w}_{opt}$ can be calculated with [21][20]:

$$\mathbf{w}_{opt} = \frac{\mathbf{R}^{-1}\mathbf{w}(\theta_0)}{\mathbf{w}(\theta_0)^H\mathbf{R}^{-1}\mathbf{w}(\theta_0)} = k\mathbf{R}^{-1}\mathbf{w}(\theta_0) \tag{2.14}$$

The denominator will evaluate down to a constant k that can be used for scaling, as it affects all weights equally and adds a constant phase shift. With correct weights, digital beamforming operates the same as analogue beamforming but more flexibly as multiple beams can be calculated in parallel, and weights can be changed to suit current needs.

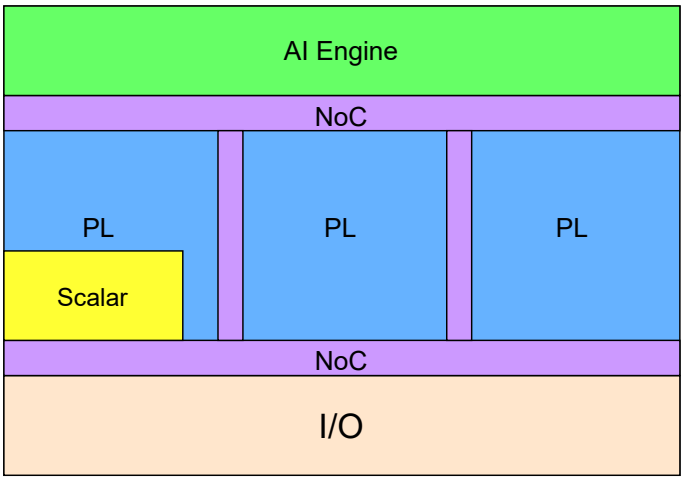


Figure 2.5: The layout of Xilinx Versal VCK190. Each processing area is connected via the AXI-4 based NoC. Adapted from [23].

2.5 Versal architecture

The hardware used in this project is a Xilinx Versal VCK190 evaluation kit with a Versal AI core. Versal AI core series is what Xilinx calls an Adaptive Compute Acceleration Platform (ACAP) design for hardware acceleration. The Versal chip contains two ARM scalar processors, programmable logic (PL), AI, and Digital signal processing (DSP) blocks, and various high-speed I/O options [22]. All the different components are connected with an AXI-4-based programmable network on chip (NoC), allowing for a low latency connection between all blocks. In Figure 2.5 the chip layout can be observed with the main processing areas marked. Xilinx has moved to 7 nm architecture for the AI core series giving a boost in energy efficiency compared to previous models [23]. This boost in energy efficiency can allow for greater acceleration in calculations as more calculations can be performed.

2.5.1 Scalar units

The circuit has two scalar processing units, one Arm Cortex-A72, and one Arm Cortex-R5F. The purpose of the scalar processors is to allow for more complex controlled applications and operating systems for real-time control systems. With simpler operation systems, the developer can debug and optimise other parts of the system while it is running and reprogram the PL and AI Engine while the system is running.

2.5.2 Programmable Logic

The PL is the adaptable engine of the core consisting of a grid with lookup tables (LUT) and memory modules. The VCK190 has a total of 899,840 LUTs, 1,968,400 Logic cells, and 27.5 Mb of RAM [22]. The PL uses an FPGA architecture to connect the components allowing for highly flexible design with a high level of parallelism.

The PL-block also contains a DSP-block with efficient MAC blocks for 27x24 bit

multiplication with dedicated units for single and half-precision floating-point multiplication [22].

2.5.3 AI Engine

The AI Engine consists of 400 AI Tiles in a 2D grid where each tile has a VLIW SIMD architecture [7]. In Figure 2.6 the architecture can be seen with the VLIW instructions. For each instruction, it can run two scalar operations, two vector read operations, one vector write operation and one vector operation. The scalar operations run on a small RISC core for more advanced operations and flow control. The address control units used for vector load and store work on 256-bit wide words, allowing up to 512 bits to be loaded at once. For the vector operation, each tile has two separate units for floating point and fixed point but can only use one unit at a time.

The vector operation can run in 8-bit, 16-bit, or 32-bit mode with a total input width of 512 bits allowing for a maximum 128 MAC/clock cycle for real 8x8-bit operations and 8 MAC/clock cycle 16x16 complex multiplications.

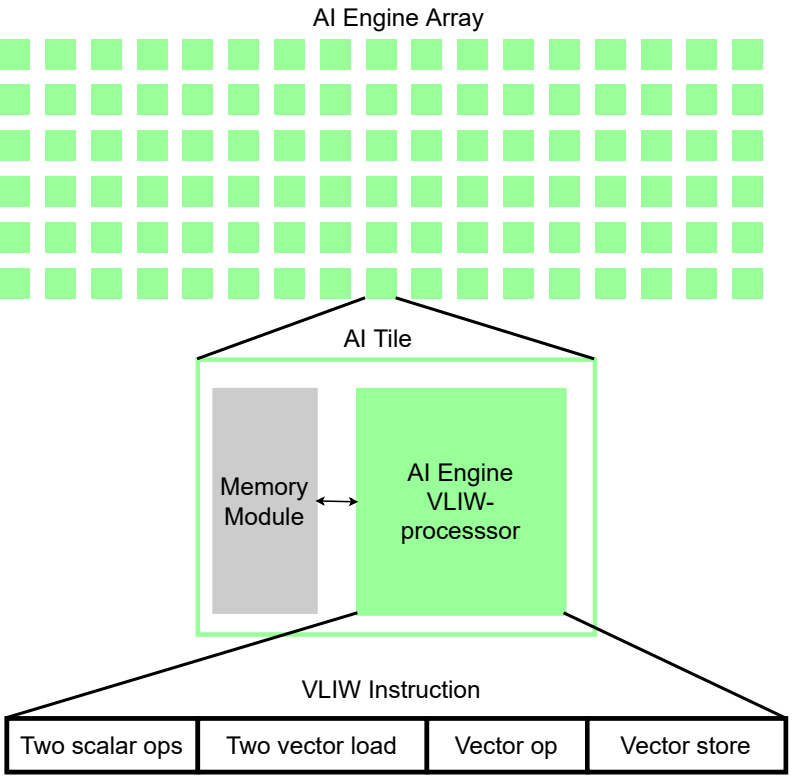


Figure 2.6: The AI Engine hierarchy, from array to tile to instruction word

In Figure 2.7 the possible communication options are shown. Each tile has 16 KB of instruction memory and 32 KB of data RAM with direct memory access (DMA) to three of its neighbours, north, south, and east or west, depending on cascade direction, allowing for access to a total of 128 KB of data RAM [24]. Additionally, it has a cascade stream access which allows it to forward a 384-bit wide accumulator

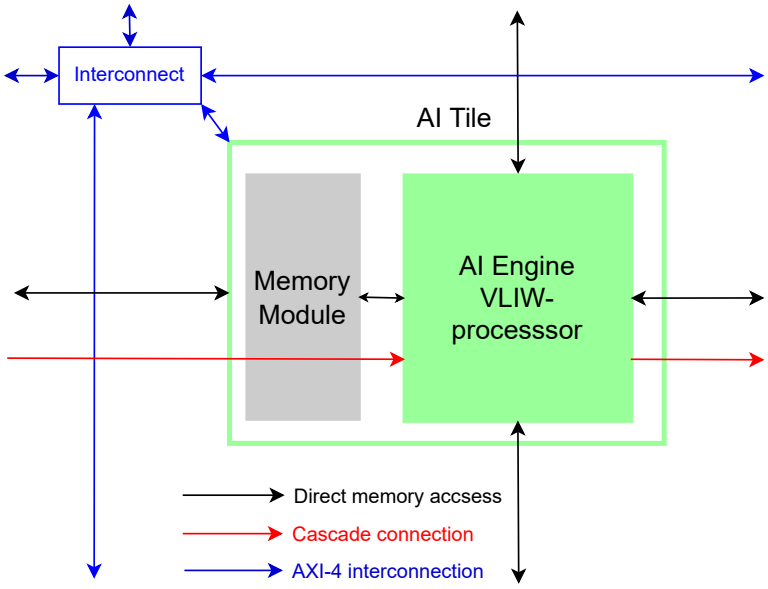


Figure 2.7: Interconnections and communications for an AI Tile.

(from vector operations) to the next tile on the same row if it goes east or west, depending on which row it is placed on. In Figure 2.7 this connection is shown with a red arrow.

For communication with tiles further away, each tile uses an AXI-4 interconnection to access data from any tiles with two input and two output streams with a width of 32 bits. This AXI-4 interconnection is separate from the one used by the NoC. However, the AI Engine interconnection connects to the NoC for passing data between the AI Engine and the rest of the chip with dedicated channels for communication with the PL.

Xilinx claims that for DSP and ML applications, the AI Engine can have up to eight times the silicon area compute density compared to a PL implementation and a 40% reduction in power consumption [7].

2.5.4 Network-on-chip

The Network-on-Chip (NoC) is an AXI-4-based network that interconnects all the different engines on the ACAP. The system runs on a master and slave-based protocol with a maximum bandwidth of 2.2 Tb/s on the VCK190. The network is programmable for high interconnection between PL, the AI Engine, IO, and scalar processors and reduces the use of the PL for data transfer. In Figure 2.5 the NoC is shown in purple and has two horizontal and two vertical connections. The I/O also contains DDR4 RAM, and the NoC allows the programmer to decide how connections are made.

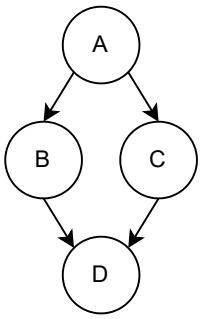


Figure 2.8: A small Kahn process network with four tasks.

2.5.5 Kahn process network

When programming the AI Engine, Kahn process networks (KPN) are used to dictate data flow and order of operations. A KPN uses graphs to represent a program or algorithm with smaller tasks. Each task or process is represented as vertices and communications channels as edges in the graph, as shown in Figure 2.8. In Figure 2.8 the circles represent the task (A,B,C,D) and the arrows the channels. All communication channels are non-blocking for the sender and blocking for the receiver, requiring the receiver to wait until data is available before proceeding. From the graph, D requires both B and C to be done before it can execute, as it needs the data from both of the tasks. KPN allows for deterministic programming regardless of communication delays but limits what programs can be represented in the graph. An example of a task that can not be modelled with a KPN is if D is a switch that waits for data from B, but if no data arrives, it reads from C instead. Since delays in transfer speed affect the outcome, it can not be modelled with a KPN.

2.5.6 PetaLinux

PetaLinux is a tool within the Xilinx family, which offers setups for using Embedded Linux distributions on Xilinx SoCs [25]. With Linux running on SoCs, it eases the running and future observation of programs by adding a file management system to the board and gives programmers an extra layer in which they can control the workflow.

3

Methods and Implementation

This chapter discusses methods and approaches for designing and implementing the digital beamforming system. First, a general view of the system is presented with an expected flow and two versions for calculating the beams. Then the implementation of each version is shown, how they differ in their workings, and how test data is generated.

3.1 Engine programming

Programming the AI Engine is done in Vitis IDE [26], where C++ is combined with intrinsic functions for greater control of the specialised hardware. Each C++ source file can be seen as a small kernel placed within one or many AI Tiles in the AI Engine. Any header files included in a source file are read to the AI Tile memory on AI Engine initialisation.

3.1.1 Intrinsics

Each AI Tile consists of a VLIW SIMD processor, as shown in Figure 2.6. To simplify coding and code readability, Xilinx has provided a long list of intrinsic, and in this project, some were used, primarily `mul4`, `mac4`, and `sincos`. The intrinsic can be seen as a call to a specific assembly instruction inside the AI Engine to reduce compiler dependencies. With 16-bit complex fixed-point input, each AI Tile can perform 8 MAC operations per clock cycle. Each AI Tile can also perform post addition, allowing for the system shown in Equation 3.1, to be calculated in a single instruction with the function `mul4` as:

```
acc = mul4(xbuff,xstart,xoffsets,xstep,zbuff,zstart,zoffsets,zstep);
```

$$\begin{cases} acc_0 = x_{0,0}z_{0,0} + x_{0,1}z_{0,1} \\ acc_1 = x_{1,0}z_{1,0} + x_{1,1}z_{1,1} \\ acc_2 = x_{2,0}z_{2,0} + x_{2,1}z_{2,1} \\ acc_3 = x_{3,0}z_{3,0} + x_{3,1}z_{3,1} \end{cases} \tag{3.1}$$

The values used in `mul4` uses one 512-bit register (`xbuff`) and one 256-bit register (`zbuff`). For 16-bit complex numbers, this allow for 16 and 8 numbers in each buffer respectively. The variables `start`, `offsets` and `step` are used to select index in the buffer used for the calculation. The formula can be seen in Equation 3.2 where

`start` is the starting point for all calculations and `offsets` is 16-bit, where 4 bits are used as offset for each row. Lastly `step` is the step size for the column.

$$\texttt{index[row,col]} = \texttt{start} + \texttt{offset[row]} + \texttt{step} \cdot \texttt{col} \bmod 16 \tag{3.2}$$

The `sincos` intrinsic takes one 32-bit integer and returns it as one complex value containing in-phase (bits[31:16]) and quadrature (bits[15:0]) components. It is worth noting that `sincos` uses 20-bits of accuracy for its calculations, reducing the accuracy as the input from previous calculations is 32 bits.

3.1.2 Kernels/AI Tiles

The AI Engine compiler uses a kernel system where more complex programs are constructed using KPNs, where multiple networks can be run in parallel. Each kernel is a C++ program or function with void as output arguments and can only be passed pointers to windows buffers or data streams as arguments. Each AI Tile will be assigned kernels to execute depending on the expected runtime of the kernel. To have multiple kernels running on the same AI Tile, the real-time requirements must be considered as the program flow on the AI Tile is scalar, and each kernel must run in a specific order. Therefore, a data delay can deadlock or stall the system.

3.2 Design flow

To compile and run the system on hardware the design flow from Figure 3.1 is followed. The system starts with a base platform design in Vivado containing the essential component for the system. This includes AI Engine blocks, PL clock, NoC, interrupts and the Arm A72-processor as a control unit.

The base platform is then exported and used as a target during the compilation of the AI Engine and PL. The PL and AI Engine are compiled separately, and the outputs are linked with the v++ linker. The linker generates a single binary for the AI Engine and PL combined and an updated platform with the necessary connections and HLS blocks added.

Separately PetaLinux is used to generate a Linux image and file system from the base platform. A control program used for testing and debugging was compiled with g++ for the system. Finally, v++ packages all the files required to generate a bootable image. The image was flashed to an SD card and booted on the system.

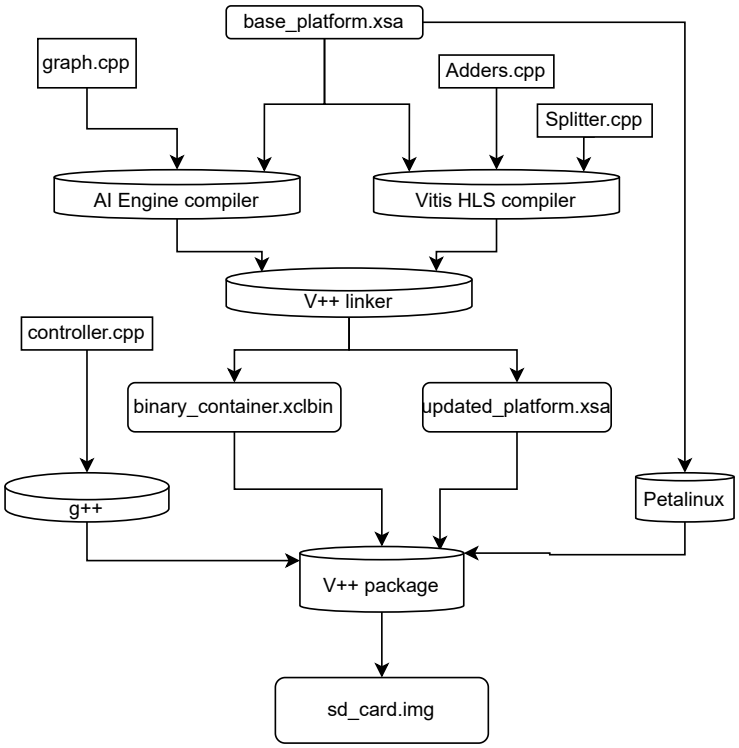


Figure 3.1: Design flow for the system.

3.3 System architecture

To calculate M number of beams from N number of antennas, Equation 3.3 is used where s are samples from antennas, w the weights used, and y is one beam. This vector-matrix multiplication must be calculated every time the radar samples its antennas. To solve Equation 3.3 on multiple cores, an architecture can be seen in Figure 3.2 where the samples are diverted dependent on which antenna it is from by the splitter to one of the partial beamformer (PB). The PB to perform the vector-matrix multiplication for N/K samples on the AI Engine. The workload can be spread by assigning specific PB to dedicated AI Tiles. This allows us to utilise the interconnections to transfer the samples and local storage on the AI Tiles for weights. The M partial beams are then passed on for summation on the PL.

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{M-1} \end{bmatrix} = \begin{bmatrix} w_{0,0} & w_{0,1} \dots & w_{0,N-1} \\ w_{1,0} & w_{1,1} \dots & w_{1,N-1} \\ \vdots & \vdots \ddots & \vdots \\ w_{M-1,0} & w_{M-1,2} \dots & w_{M-1,N-1} \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_{N-1} \end{bmatrix} \tag{3.3}$$

Between the PB and adders, FIFO buffers are inserted to increase the allowed delay between the arrival of samples to different PBs. Without the buffers, one PB could get stalled by the adders due to adders running at a lower clock frequency and requiring multiple partial beams, from different PB inputs, that can arrive at varying times.

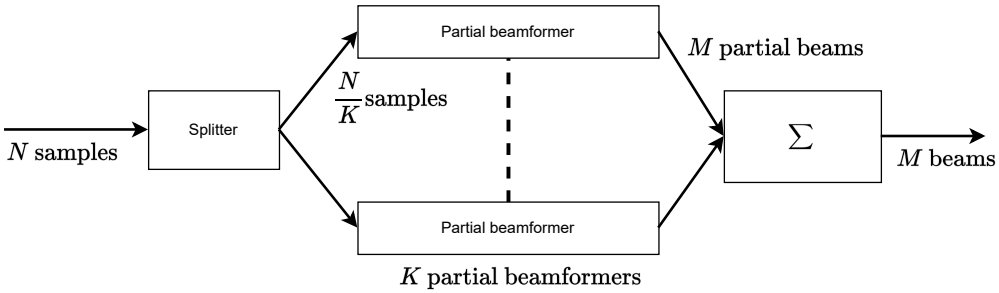


Figure 3.2: The basic architecture of the beamformer. Samples get split, and each PBF performs its calculations separately before being added together and sent out as M beams.

3.3.1 Splitters

The splitters read the samples from the DDR memory in burst and buffer them on the PL. It then uses the AXI-4 stream to send samples to the PB. The splitters for the system are divided into smaller modules, where each module reads and distributes the samples for four PBs.

3.3.2 Partial beamformer

For the PB, two different versions were designed, Cascade and One Tile, as shown in Figure 3.3. Both were implemented to receive 16 samples per PB.

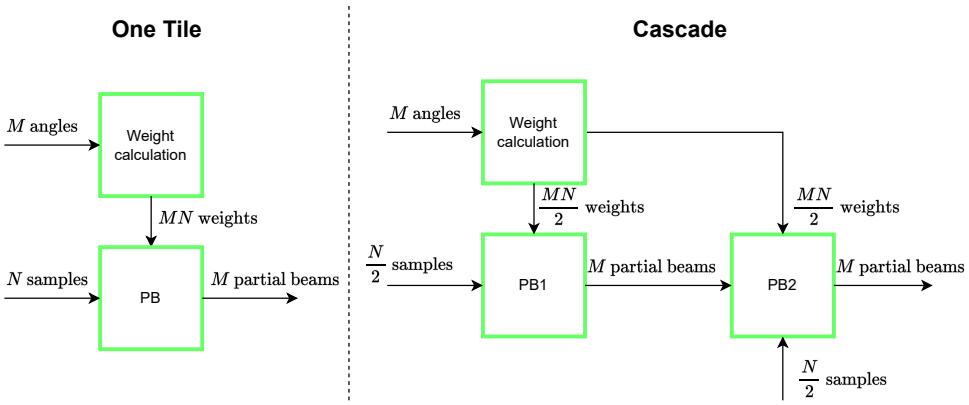


Figure 3.3: One Tile and Cascade versions of partial beamformer. Each square is one AI Tile used and the arrows show the dataflow.

One Tile performs the complete calculation on the same AI Tile. It uses the `mul4` and `mac4` intrinsics to calculate four partial beams in parallel. One Tile stores the samples in the x-buffer (16 samples) and the weights in the z-buffer (8 weights). The samples are read from the AXI-4 stream, and the weights are stored in local memory on the memory module. In Equation 3.4 and 3.5 the first two iterations for a beam calculation is shown where y_n represents the nth partial beam. This matches the selection when calling `mul4` and `mac4` according to the code below.

```
y = mul4(samples, 0, 0, 1, weights, 0, 0xC840, 1);
```

```
y = mac4(y, samples, 2, 0, 1, weights, 2, 0xC840, 1);
```

$$\begin{cases} y_0 = s_0 w_{0,0} + s_1 w_{0,1} \\ y_1 = s_0 w_{1,0} + s_1 w_{1,1} \\ y_2 = s_0 w_{2,0} + s_1 w_{2,1} \\ y_3 = s_0 w_{3,0} + s_1 w_{3,1} \end{cases} \tag{3.4}$$

$$\begin{cases} y_0 += s_2 w_{0,2} + s_3 w_{0,3} \\ y_1 += s_2 w_{1,2} + s_3 w_{1,3} \\ y_2 += s_2 w_{2,2} + s_3 w_{2,3} \\ y_3 += s_2 w_{3,2} + s_3 w_{3,3} \end{cases} \tag{3.5}$$

The second version, Cascade, sends the accumulator register to another AI Tile utilising the cascade connection. This allows two AI Tiles to work in series where each AI Tile only works on half of the samples. Only working with half of the samples allows the Cascade version to change buffers for samples and weights, storing samples in z-buffer and weights in x-buffer. This reduces the number of registers used without affecting performance. The second AI Tile waits for the first AI Tile to be done with y_0 to y_3 before performing its first calculation. The Cascade setup allows for a theoretically larger throughput as the first AI Tile can start on the next set of samples while the second AI Tile is finalising the previous set.

Both One Tile and Cascade are flexible in the number of beams they can calculate. However, the runtime increases in steps with multiple of four as it always runs four beams in parallel. They both also read weights in parallel keeping the pipeline full after the initial read of samples. To read weights linearly from memory, they are stored in a specific manner. Figure 3.4 shows how for One Tile with 16 beams, the weights are divided into 16 smaller 4x4 matrices. The 4x4 matrices are stored in row-major order, and the blocks are stored according to Figure 3.4. The weight calculation does this ordering.

$$W_0 = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} & w_{0,3} \\ w_{1,0} & w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,0} & w_{2,1} & w_{2,2} & w_{2,3} \\ w_{3,0} & w_{3,1} & w_{3,2} & w_{3,3} \end{bmatrix} \rightarrow \begin{bmatrix} W_0 & W_1 & W_2 & W_3 \\ W_4 & W_5 & W_6 & W_7 \\ W_8 & W_9 & W_{10} & W_{11} \\ W_{12} & W_{13} & W_{14} & W_{15} \end{bmatrix}$$

Weights → | W_0 | W_1 | W_2 | W_3 | W_4 | ..... | W_{15} | W_{16} |

Figure 3.4: 16x16 weights divided into 4x4 matrix and then stored in memory as blocks.

3.3.3 Weight calculations

Each weight is calculated based on beam angle, placement of antenna element, and requested frequency. Out of these three inputs, only the beam angle is produced outside the AI Tile because it is required to achieve an agile system. Both antenna

element placement and frequencies are placed in the AI Tile memory and read at system upstart.

The incoming beams arrive in a 16-bit, M-element vector containing $M/2$ angle pairs, θ and ϕ. Calculating weights is done according to Equation 3.6. The incoming angles are multiplied with a 32-element vector containing hard-coded x and y position values for each antenna element. The resulting vector is then multiplied by the frequency. After that, each element in the vector has its weights extracted using the `sincos` intrinsic, which concatenates a 32-bit value into a sine part, bit[31:16], and cosine part, bit[15:0], in signed Q.15 fixed-point format.

$$w(\theta, \phi, f_0) = e^{-j(x \sin\theta \cos\phi + y \sin\theta \sin\phi)\frac{2\pi c}{f_0})} = e^{-j(xu+yv)\frac{2\pi c}{f_0})} \tag{3.6}$$

The output is then fed into a buffer, waiting to be used by the PB kernel. The buffer between weight calculation and the PB uses dual ping-pong buffers and a selector to avoid read and write stalls. This ensures that one buffer is always readable. Each buffer is locked while being read or written to satisfy mutual exclusivity.

The weights are recalculated whenever a new beam angle vector is fed into the system. Weights calculation is done on a separate AI Tile to allow the beamformer to run while new weights are calculated. Hence, the weight calculations generate a latency issue when the weights are calculated at startup but will not affect the beamformer throughput when the same weights are reused. When weights are recalculated, the PBs continue their calculations with the old weights until the new ones are available in the ping-pong buffer. The control signal has been updated for the following samples received, and the new weights are used in the PB.

3.3.4 Adders

To perform the final summation of the beams, the adders are placed on the PL using complex 32-bit wide fixed point numbers. The outputs from the AI Engine to PL are shifted down by $\lceil \log_2(\text{PBF}) \rceil$ to account for possible overflows that can accumulate from the repeating addition. The system's last adder is responsible for truncating the value down to a complex 16-bit fixed-point representation. The adders are written in C++ and compiled into an FPGA design with HLS. This choice was made for fast prototyping and automated communication protocols with stream interfaces to the AI Engine.

3.3.5 Program controller

To control the system described in Figure 3.2, a control program running on the ARM A72 processor in a Linux environment was created. The control program's purpose is to dedicate memory areas for samples and the output beams, start each part of the system and control the number of iterations it will run. It is also responsible for when weights are to be recalculated. The controller uses a runtime control library to start the correct kernels, collecting the buffers' results and timing the system during tests.

3.4 Generation of test data

A function to generate test data was created that takes in the antenna elements' position, the distance d to N sources, the incoming angles θ and ϕ for each source, signal amplitude on the receiver, expected SNR, frequency of the signal and sample frequency. The function then uses the wave function in Equation 3.7.

$$x[n] = \sum_{s=0}^{s=N} a_s e^{-i2\pi(\frac{df_q}{c} - \frac{nf_q}{f_s})} + N(0, \sigma) \tag{3.7}$$

The samples are written into a file and read in during simulations or sent to the board as 16-bit complex values representing the in-phase and quadrature components of the signal. The samples are then used to calculate the expected output result according to the model made earlier.

4

Results

In this results chapter, both AI Engine simulation and hardware systems are presented with FoMs regarding both types of results.

Each input beam results in an output of equal size, so each beam linearly increases the data needing to be processed by the system.

4.1 AI Engine simulation evaluation

For simulation, values were extracted using the Vitis Analyzer tool for a hardware emulation on the system. Simulations are measured from traces in a graph for both the PB and weight calculation. The simulation results assume no stalls from lack of input data.

4.1.1 Figures of Merits

The FoMs considered for the simulation part of the project are throughput, latency and resource usage. Latency is the time it takes for a sample to process in the partial beamformer, added together in the system's PL adders and then fed to an output file. Throughput is defined as the number of samples the system can process. It is measured as the time it takes between the starts of two partial beamformer processes. The resource usage includes the number of AI Tiles used in a specific configuration.

The AI Engine is clocked at 1GHz, which results in a clock cycle of 1ns. The PL adders are for simulation clocked at 300 MHz.

Table 4.1: FoM layout for simulations.

	Definition
Throughput	How many samples the system can process per second
Latency	From start kernel to first output in clock cycles

4.1.2 Simulation results

Simulated results from one weight calculator feeding values to one PB can be seen in Table 4.2. Note that this is only for the weight calculation and not the entire system. As the weights are calculated separately and stored in a buffer which is

being read by the PB, it can be placed outside of the actual throughput calculations and only act like a startup and latency delay when either starting up the system or calculating new weights.

Table 4.2: Kernel runtime for weight calculation.

Beams	Clock cycles	Clock cycles/beam
8	269	34
16	520	33
32	960	30

Table 4.3: Simulated maximum throughput for Cascade implementation of PB.

Beams	Clock cycles	Clock cycles/beam	Msamples/s	Latency
8	80	10	200	43
16	92	6	177	43
32	118	4	135	43

Table 4.4: Simulated maximum throughput for One Tile implementation of PB.

Beams	Clock cycles	Clock cycles/beam	Msample/s	Latency
8	113	15	141	35
16	113	8	141	35
32	262	8	61	35

4.2 Hardware evaluation

Results from hardware evaluation were extracted using Vivado reports as well as timestamps before and after AI Engine execution in the C main program controlling the system.

4.2.1 Figures of Merits

When evaluating the result from the hardware, throughput, power consumption, area usage and accuracy were observed. The complete system consists of reading samples and weights from memory, calculating these using our AI Engine implementation, adding the partial beams together in the PL-block and writing the full beams to memory.

One sample point consists of one sample from every antenna element. Time measurements were only made during the running of AI Engine and PL, including feeding data to AI Engine and writing output to memory, removing overhead caused by memory allocations done by the main program.

4.2.2 Hardware results

Figure 4.1 and Figure 4.2 shows the different implementations average speed running for 16 and 256 million sample points respectively. Note the linear increase in execution time depending on beams for One Tile. The linearity is also apparent in Cascade from 16 to 32 beams. 8 and 16 beams achieving the same execution time is due to the implementation counting four beams at once. This is also why Cascade and One Tile achieves almost the same execution time for 8 beams; the gain running AI Tiles in parallel is more apparent for a higher number of beams. Table 4.5 displays throughput for the two implementations with different beam numbers.

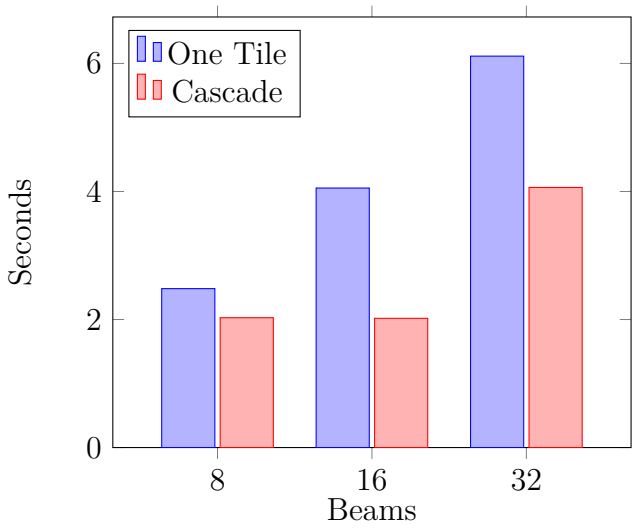


Figure 4.1: Average speed of implemented system running 16 million sample points.

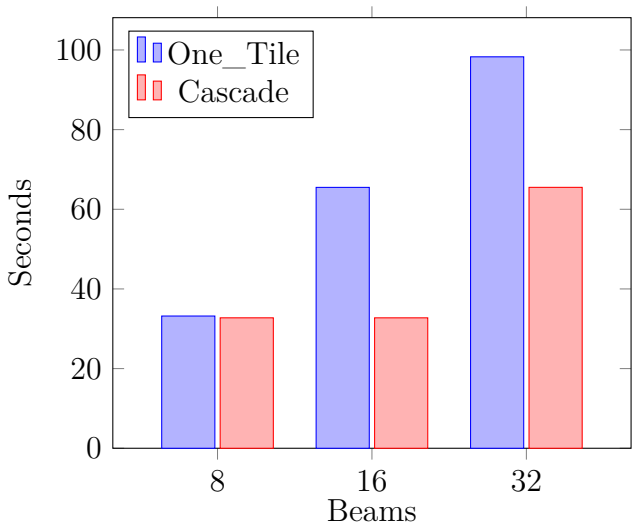


Figure 4.2: Average speed of implemented system running 256 million sample points.

Table 4.5: Throughput for different implementations. Values are calculated from those shown in Fig 4.2.

Version	Beams	Throughput(Msample point/s)	ns/Beam
One Tile	8	7.7	16.2
One Tile	16	3.9	16.0
One Tile	32	2.6	12
Cascade	8	7.8	16.0
Cascade	16	7.8	7.9
Cascade	32	3.9	7.9

4.2.3 Resource usage

For area usage on the platform, the total amount of LUTs for the partial beamformer and adders are counted towards the total usage of the data given for comparison against the existing FPGA design.

Table 4.6: Resource usage: Total resource usage of PB and addition of partial beams.

Version	AI Tiles	FF	LUT	BRAM	DSP
One Tile	96	643	6028	0	0
Cascade	144	643	6028	0	0
FPGA imp	0	179000	145000	89	1200

Table 4.7: Power consumption for One Tile and Cascade implementation.

Implementation	Total power	Dynamic	Device static
One Tile	42.9W	28.2W	14.7W
Cascade	49.2W	34.5W	14.7W

Since both implementations use the same hardware platform with the same layout of PL parts, the only difference in power consumption comes from the AI Engine.

4.2.4 Accuracy

Calculations in our two hardware implementations are the same and therefore do not affect the accuracy or output of the system. In our observations, for accuracy, the Cascade implementation using 32 beams was used and then final truncation from 32-bit down to 16-bit was removed.

In Figure 4.3 the result from 14400 different beams calculated can be seen. The area sweeper was done by sweeping ϕ from 0 to 360 degrees and θ from 0 to 30 degrees. The Matlab floating point results are shifted up to match the fixed-point representation values. The signal contains ten equal sources, without any noise added and normalized to [-1,1]. The difference between the hardware results and

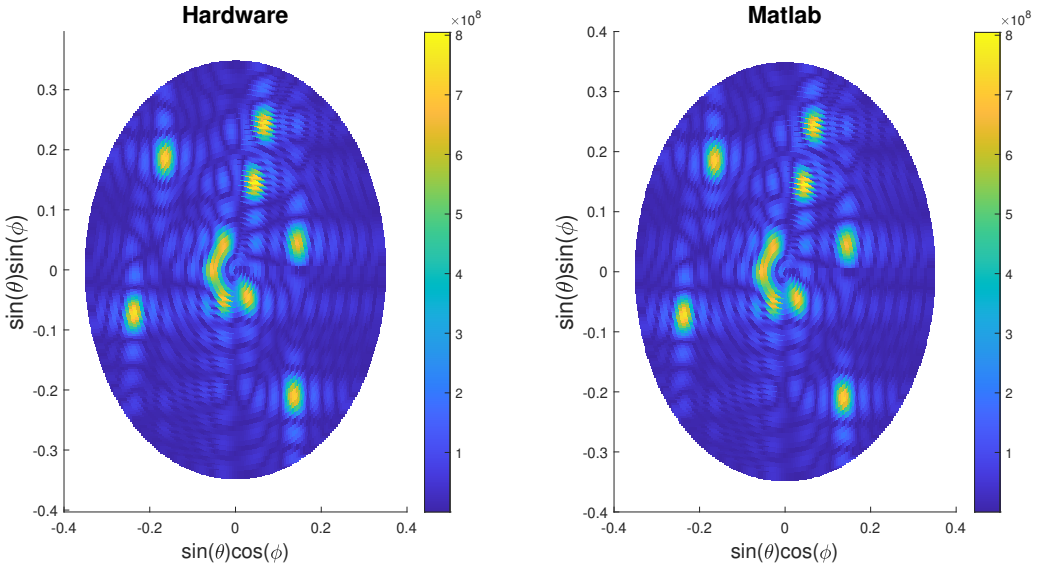


Figure 4.3: Circular sweep without noise for Matlab and hardware results.

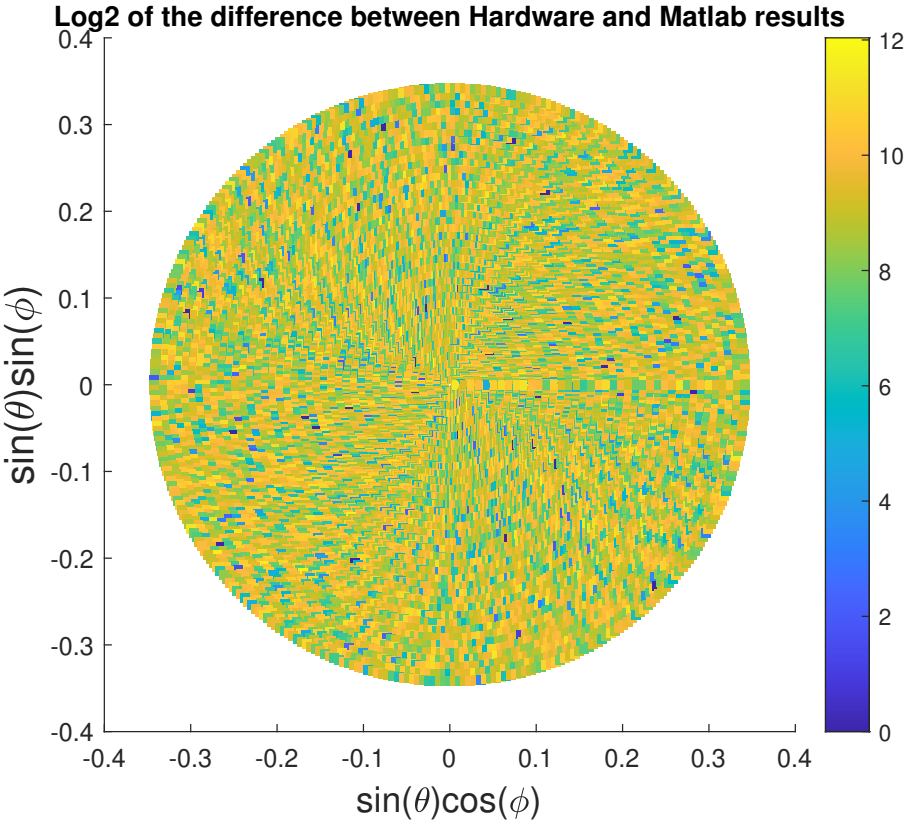


Figure 4.4: The difference between Matlab and hardware using logarithmic scale.

Matlab from Figure 4.3 can be observed in Figure 4.4. The maximum difference was 4282, equating to an error at the 13th bit.

In Figure 4.5 the results from a second larger sweep can be observed where the signal contains ten new sources but with added white Gaussian noise with an SNR of two and normalized to [-1,1]. Here the space was swept in a 200x200 grid to generate different beams compared to 4.3. Figure 4.6 shows the difference between the hardware results and Matlab. In this test, the maximum difference was 3957, equating to an error at the 12th bit.

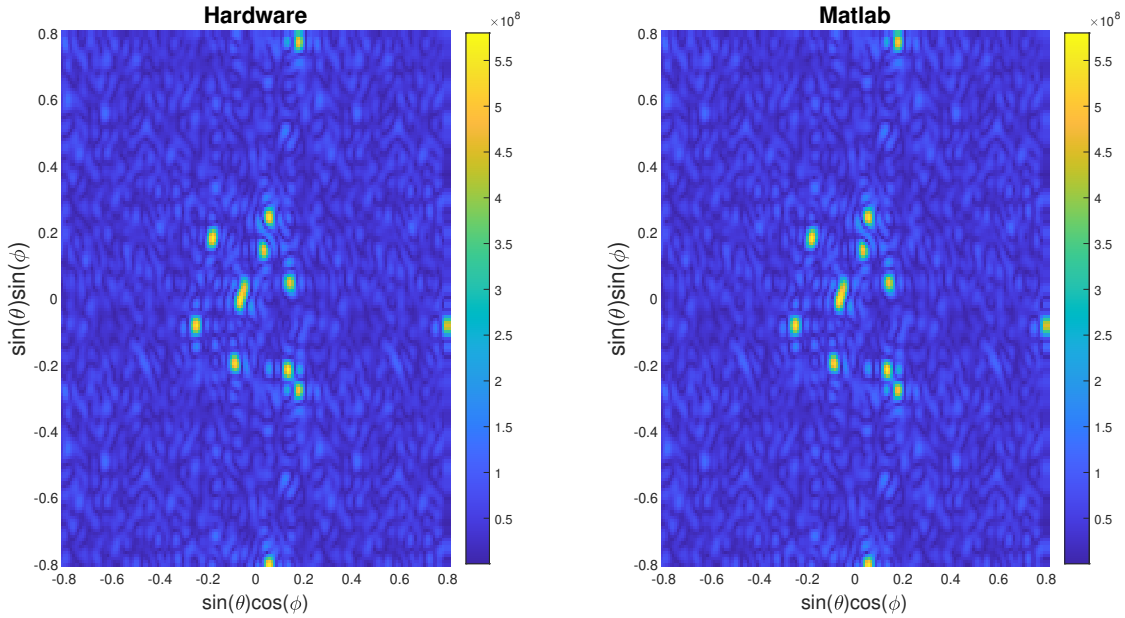


Figure 4.5: Square sweep with noise for Matlab and Hardware results.

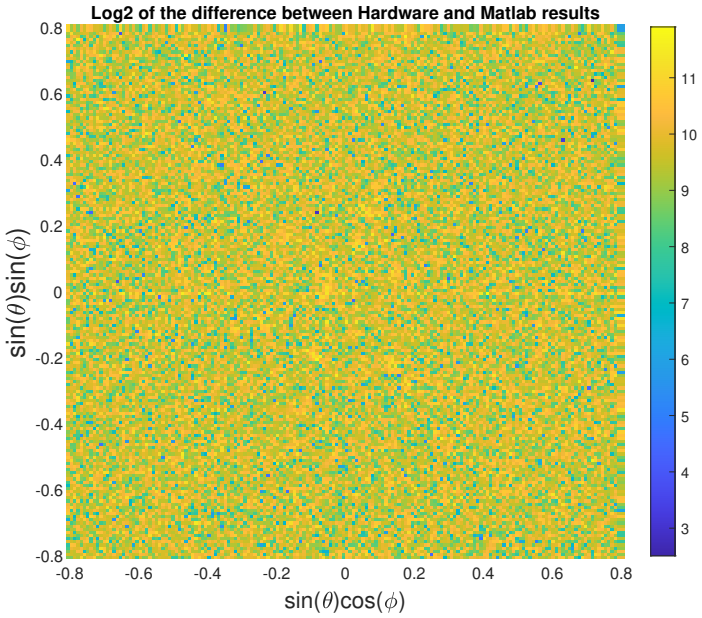


Figure 4.6: The difference between Matlab and hardware using logarithmic scale.

5

Discussion

This chapter discusses the results presented in Chapter 4, the cause and effect of design choices. Finally, some development aspects and evaluations of the Xilinx Versal VCK190 SoC and the tools used in programming it is discussed.

5.1 Result comparison

Compared to the existing solution for the FPGA, the AI Engine solutions significantly reduced the usage of LUT and DSP for the partial beamformers (PBs) in exchange for the AI Engines used. The move of utilisation can allow for other systems on the FPGA as more space is available where the AI Engine is used.

From our results, the main limiting factor in the system is the dataflow to and from the AI Engine. Comparing the simulated result of the PB maximum throughput, the hardware implementation runs at 66% of the maximum simulated throughput of the AI Engine. As the number of beams increases, the throughput decreases as more calculations are done on each sample.

If PL adders run at 300 MHz, with perfect timing and no stalls, they can process one beam per clock cycle, giving a theoretical maximum speed at the output at 3.3 ns/beam. Currently, the Cascade version can run at 7.8 ns/beam for both 8 and 16 beams, showing some delays or stalls for data to travel within the system. Since the AI Engine and adders are running on the same clock domain, the more beams get added, the more data is required to be transferred away from the bottleneck. If more speed was required, splitting the design into two clock domains, one for addition and one for providing the AI Engine with samples, could increase efficiency and throughput. One possible way to increase the total throughput is by adding more and larger buffers in the adder chain, increasing the system's stability for potential stalls. With small buffers, one stall could propagate back into the chain and affect other adders while waiting to be allowed to write data.

Both AI Engine solutions use the same base for the PL, so the difference in computation time is only based on the change of data flow inside the AI Engine. The difference in throughput for more beams is because the Cascade version can start reading samples simultaneously as it can write out data. This is due to the kernels being on separate AI Tiles and results in better throughput, as one of the AI Tiles often performs a task while the other is waiting to read/write data. The main disadvantage of the Cascade version over the One Tile is the 50% increase in usage of AI Tiles and more challenging constraints on placement. The Cascade AI Tiles

must be placed in series on the same row of the AI Engine and be able to share a memory with the weight calculator. The One Tile has the advantage of being more flexible and can be combined with other applications that want to take advantage of the AI Engine as its footprint is smaller than Cascade. One Tile also has lower power consumption and can use a smaller chip variant, if not all 400 AI Tiles are required.

5.2 Hardware accuracy

In Figure 4.5 and 4.3, we can see that the same pattern emerge from both Matlab and hardware implementation but get some accuracy loss. The loss in accuracy on the 13th LSB comes from the weight calculation and how the e^{-jx} operation differs in Matlab compared to hardware implementation. In the weight calculation, we observed a difference in the LSB for its output, which then propagates through the system resulting in our accuracy error of 13 bits.

The truncation of the 7 LSB, to compensate for overflow when moving from 48-bit accumulator register to 32-bit adders, gives a worst-case accuracy of 26 bits when 48 partial beams are added together. The truncation error is less than that observed from weight calculations, as one flip gives an error up to the ninth bit of the output. Our tests observed a maximum error at the 13th bit with peak usage of 30 bits. This results in 17 bits of accuracy, which can be seen in Figure 4.4.

5.3 Design choices

Our design choices were made focusing on throughput in the AI Engine. The project removed packet overhead and used HLS to speed up the implementation process.

5.3.1 PL adders

During early drafts of the system, the adders were placed in the AI Engine to add outputs from PBs together. However, tests showed long compile time and timeouts during "place and routing" as the system was scaled up. The timeout was because optimising the interconnections to give the system the best possible timings required manual placement of each AI Tile, which was deemed too long regarding what it would achieve in performance. The risk for deadlock and stalls increased as more AI Tiles would access the same memory areas.

The PL is also good at performing addition and can be pipelined to keep up with the output of the AI Engine. Clocking the Adders at 300 MHz allows us a potential 3.3 ns/beam time that is faster than the average maximum simulated output of the AI Engine. However, the AI Engine receives and outputs beams in bursts; therefore, crossing clock domains from AI Engine to PL requires a FIFO buffer on the AI Engine side. The buffer is required to avoid stalls in the AI Engine as the PL only reads 64-bit at 300 MHz compared to the write speed of 32-bit at 1 GHz.

5.3.2 Placement of kernels

The limitation of one kernel (weight calculation or PB) per AI Tile was a design choice under the assumption that the partial beamformer would run with bursts when it gets data. If multiple PBs were to be able to place on the same AI Tile, some buffer system would be required inside or before the AI Engine to store the samples. If multiple PBs share a single AI Tile, greater care must be taken to avoid deadlocks as they compete for the same resources, and deadlock can quickly arise if the system gets data for the wrong PB.

5.3.3 Hardware conversion

Programming hardware in an HLS fashion with Vitis is made to shorten development time while retaining control over hardware definitions. The shortened development time was felt, as the HLS solved communications between the AI Engine and PL that would have taken significant work to write in VHDL. The method has its downside with higher PL usage, making it harder to control how the design was implemented precisely.

Another part which could further improve the implementation would be to focus more on the hardware platform. Now, some IPs, like unused clocks and overly large interrupt controllers, are not used and, with proper diagnostics and programming, could be removed, reducing overall power consumption.

5.4 Ethical considerations

Any radar system, focused on the military or not, is gathering information. The information might be harmful to others, but the method used to gather it is not harmful. A radar system could be used to obtain vital information for the user against a target without their knowledge. Beamforming is not harmful but enhances systems' capabilities and thus becomes of interest ethically speaking. Now, considering that this project was done in collaboration with Saab AB, a defence company, the focus is gathering information from external threats that would help save lives rather than detecting targets before an attack, as stated in the United Nations charter [27].

6

Conclusion

With its AI Engine, Xilinx has started something new and unique within the programmable hardware field. Our implementation was successful, and the results confirm that creating a simple but fast AESA beamforming system was not only possible but achievable within a short period.

The main concerns we found were being able to supply and process the AI Engine with data and not getting precise timing results.

The AI Engine kernel programming with intrinsic functions made it possible to maximise the effectiveness of the AI Engine cores and have a high theoretical throughput. As previously mentioned, communication delays from and to the PL reduce the efficiency of the final implementation.

6.1 Future research

Future research would look into how the design can be fitted into a more extensive system. Currently, a large area of the PL is used to supply the AI Engine with samples from the DDR memory. This can be changed to include reading samples from an Ethernet connection, which would be more in line with a digital beamformer. As our work only looked at a small section of a fully realised radar system, future work is required for it to fit into a system. More possibilities for debugging and error checks might be required.

6.1.1 Actual radar target testing

Due to limited time and availability, testing was only made on generated data in a closed environment. One future part would be to extract samples from a running radar system and observe metrics such as agility and precision. More caution is required for a fully realised system to guarantee that the system does not stall and add timeouts for packets.

6.1.2 Scaling up the system

The implemented system was tested with up to 48 PBs and used 144 out of the 400 potential AI Tiles available. The scaling is linear, and each additional PB requires two AI Tiles for One Tile and three for Cascade. The system can be optimised to reduce the weight calculation AI Tiles used. As weight calculations are performed

less frequently and do not have the same demand on throughput as weights are calculated less regularly, they can have longer runtimes.

So, with our Cascade implementation and manual placement, the maximum number of PB which could theoretically fit in the Xilinx Versal VCK190 AI Engine would be 133 for Cascade and 200 for One Tile.

6.1.3 Combining usage

Last year, Holst and Krull [9] explored the possibility of using an AI Engine for radar classification and concluded that it was possible and required only a tiny part of the AI Engine to run. Combining both projects to achieve better resource usage and power consumption and working towards realising more intricate radar system parts on AI Engines.

Bibliography

[1] C. Alabaster, "Part I. Basic Concepts", in *Pulse Doppler Radar - Principles, Technology, Applications*. SciTech Publishing, 2012, ISBN: 978-1-891121-98-2. [Online]. Available: https://app.knovel.com/hotlink/khtml/id:kt00BID314/pulse-doppler-radar-principles/part-i-basic-concepts.

[2] A. G. Bole, A. Wall, A. Norris, and W. O. Dineley, "Navigation tecniques using radar and ARPA", in *Radar and ARPA Manual*. 2005, pp. 356–392, ISBN: 9780080480527.

[3] R. Dixit, "Automotive radars-development status", in *1996 IEEE MTT-S International Microwave Symposium Digest*, vol. 1, 1996, 317–320 vol.1. DOI: 10.1109/MWSYM.1996.508520.

[4] P. Rodriguez, E. Kennedy, and P. Kossey, "High frequency radar astronomy with HAARP", in *Proceedings of the 2003 IEEE Radar Conference (Cat. No. 03CH37474)*, 2003, pp. 154–159. DOI: 10.1109/NRC.2003.1203395.

[5] F. Gekat, P. Meischner, K. Friedrich, *et al.*, "The state of weather radar operations, networks and products", in *Weather Radar: Principles and Advanced Applications*. 2004, pp. 1–51, ISBN: 978-3-662-05202-0. DOI: 10.1007/978-3-662-05202-0_1. [Online]. Available: https://doi.org/10.1007/978-3-662-05202-0_1.

[6] M. Skolnik, "Radar's environmental role", *IEEE Potentials*, vol. 10, no. 2, pp. 13–16, 1991. DOI: 10.1109/45.84092.

[7] Xilinx. "Xilinx AI Engines and Their Applications". (2020), [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/white_papers/wp506-ai-engine.pdf. (Accessed: 21.12.2021).

[8] A. Ahlander, M. Taveniku, and B. Svensson, "A multiple SIMD approach to radar signal processing", in *Proceedings of Digital Processing Applications (TENCON '96)*, vol. 2, 1996, 852–857 vol.2. DOI: 10.1109/TENCON.1996.608457.

[9] F. Holst and C. Krull, "Classification of radar targets using neural networks on systems-on-chip", M.S. thesis, Computer Science and Engineering, 2021, p. 48.

[10] J. D. Swales and C. B. Feak, *Academic writing for graduate students*. University of Michigan Press, 2012.

[11] M. Schetzen, "Doppler effect", in *Airborne Doppler Radar : Applications, Theory, and Philosophy*. 2006, pp. X–X, ISBN: 1-56347-828-5.

[12] C. Alabaster, "Pulse Radar", in *Pulse Doppler Radar - Principles, Technology, Applications*. SciTech Publishing, 2012, ISBN: 978-1-891121-98-2. [Online]. Available: https://app.knovel.com/hotlink/khtml/id:kt00BID3R1/pulse-doppler-radar-principles/pulse-repetition-frequency.

[13] V. Chen, "Introduction", in *The Micro-Doppler Radar Effect*. 2011, pp. 1–28, ISBN: 978-1-60807-057-2.

[14] A. D. Brown, "AESA Overview", in *Active Electronically Scanned Arrays: Fundamentals and Applications*. 2022, pp. 1–17. DOI: 10.1002/9781119749097.ch1.

[15] R. J. Mailloux, "Phased Arrays in Radar and Communication Systems", in *Phased Array Antenna Handbook, Third Edition*. 2017, pp. 1–64, ISBN: 978-1-63081-029-0.

[16] ——, "Pattern Characteristics of Linear and Planar Arrays", in *Phased Array Antenna Handbook, Third Edition*. 2017, pp. 65–111, ISBN: 978-1-63081-029-0.

[17] A. D. Brown, "Transmit Receive Modules", in *Active Electronically Scanned Arrays: Fundamentals and Applications*. 2022, pp. 103–134. DOI: 10.1002/9781119749097.ch4.

[18] ——, "Beamformers", in *Active Electronically Scanned Arrays: Fundamentals and Applications*. 2022, pp. 135–163. DOI: 10.1002/9781119749097.ch5.

[19] H. Hang and L. Shaobin, "Application of weighting methods adapting to array structure for design of dbf in subarray [radar beamforming]", in *2002 3rd International Conference on Microwave and Millimeter Wave Technology, 2002. Proceedings. ICMMT 2002.*, 2002, pp. 697–700. DOI: 10.1109/ICMMT.2002.1187796.

[20] Y. Li, S. A. Vorobyov, and A. Hassanien, "Robust beamforming for jammers suppression in mimo radar", in *2014 IEEE Radar Conference*, 2014, pp. 0629–0634. DOI: 10.1109/RADAR.2014.6875667.

[21] A. D. Brown, "Aesa architectures", in *Active Electronically Scanned Arrays: Fundamentals and Applications*. 2022, pp. 183–207. DOI: 10.1002/9781119749097.ch7.

[22] Xilinx. "Versal Architecture and Product Data Sheet: Overview". (2022), [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds950-versal-overview.pdf. (Accessed: 07.02.2022).

[23] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture", New York, NY, USA: Association for Computing Machinery, 2019, ISBN: 9781450361378. DOI: 10.1145/3289602.3293906. [Online]. Available: https://doi.org/10.1145/3289602.3293906.

[24] Xilinx. “VCK190 Evaluation Board User Guide”. (2021), [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documentation/boards_and_kits/vck190/ug1366-vck190-eval-bd.pdf. (Accessed: 07.02.2022).

[25] ——, “PetaLinux Tools Documentation, Reference Guide”. (2022), [Online]. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/sw_manuals/xilinx2022_1/ug1144-petalinux-tools-reference-guide.pdf. (Accessed: 02.05.2022).

[26] ——, “Vitis Unified Software Platform Documentation: Embedded Software Development (UG1400)”. (2022), [Online]. Available: https://docs.xilinx.com/r/en-US/ug1400-vitis-embedded/Using-the-Vitis-IDE. (Accessed: 04.07.2022).

[27] “United Nations Charter”. (1973), [Online]. Available: https://www.un.org/en/about-us/un-charter/full-text. (Accessed: 29.06.2022).