Hardware-in-the-Loop Simulation with PYNQ for Radar Applications

Christopher D. Long CHRISTOPHER.D.LONG50.CIV@US.NAVY.MIL
Tyler Brant TYLER.A.BRANT.CIV@US.NAVY.MIL
Lodewijk Brand LODEWIJK.W.BRAND.CIV@US.NAVY.MIL
Naval Information Warfare Center Atlantic, 1 Innovation Dr, Hanahan, SC 29410 USA

Jeffrey D. Ouellette JEFFREY.D.OUELLETTE4.CIV@US.NAVY.MIL
Remote Sensing Division, US Naval Research Laboratory, Washington, DC 20375 USA

Abstract

This work presents a hardware-in-the-loop simulation workflow that integrates GNU Radio with hardware acceleration on a field programmable gate array (FPGA). We specifically leverage the open-source PYNQ (Python Productivity for Adaptive Computing Platforms) framework to demonstrate an interactive development process for accelerating digital signal processing algorithms. We show the workflow by simulating a continuous wave radar where key processing tasks, such as the cross ambiguity function, are offloaded from the GNU Radio host to custom PYNQ accelerators on an FPGA. This approach streamlines the difficult and time-consuming FPGA development cycle, enabling faster design iteration and optimization directly from an interactive Python environment. By co-simulating parts of the radar receiver chain in software and hardware, this hardware-in-the-loop workflow enables direct performance analysis and verification of the GNU Radio to FPGA pipeline. Our work paves the way for an adaptable research and development workflow for emerging digital signal processing applications with commodity hardware, open-source software, and emerging programming paradigms on FPGAs in an interactive Python development environment.

1. Introduction

Modern radar designs are becoming increasingly sophisticated (Thornton et al., 2020; Martone et al., 2021; Flandermeyer et al., 2024), demanding rigorous testing and validation of their digital signal processing (DSP) and corresponding control algorithms. As a consequence of this increased complexity, emerging radar systems require scalable prototyping workflows to enable sufficient experimentation and algorithm refinement during product development. Efficient hardware-in-the-loop workflows offer a competitive advantage to radar developers, enabling the test and evaluation of DSP routines in variety of environments. This paper introduces a hardware-in-the-loop simulation workflow for accelerating radar applications using an FPGA. Our approach is focused on leveraging various open-source software packages to develop and accelerate algorithms for radar.

Hardware-in-the-loop simulation is a key development strategy for emerging radar systems (Zhu et al., 2021; Flandermeyer et al., 2022; Kern et al., 2024). To test and evaluate these emerging systems it is critical to enable high-bandwidth signal processing to achieve the fine-grained range resolution needed for realistic radar use cases. To meet these high-bandwidth requirements, we develop strategies for offloading our radar signal processing to a hardware accelerator. Our approach enables rapid development and testing of various algorithms on emerging platforms. Our work is primarily focused on leveraging AMD-series FPGAs (AMD, 2025b) and associated software. Other radar applications with different vendors also exist including Analog Devices with the ADALM-PHASER (Analog Devices, 2023) kit and Ettus-series software defined radios programmed with RFNoC (Braun et al., 2016).

Our software approach centers around the PYNQ framework (PYNQ Project, 2025), which allows for seamless integration of Python-based software with custom hardware accelerators implemented on AMD ZYNQ System-on-Chip products. The PYNQ project aims to reduce the barrier to entry for software engineers to develop applications leveraging FPGAs to accelerate the execution of various signal processing routines. This algorithm acceleration capability is critical for reducing the latency of time-critical applications such as IoT (Sahu et al., 2022), radar (Mattingly & Metcalf, 2022), and object detection (Kim &

Proceedings of the 15th GNU Radio Conference, Copyright 2025 by the author(s).

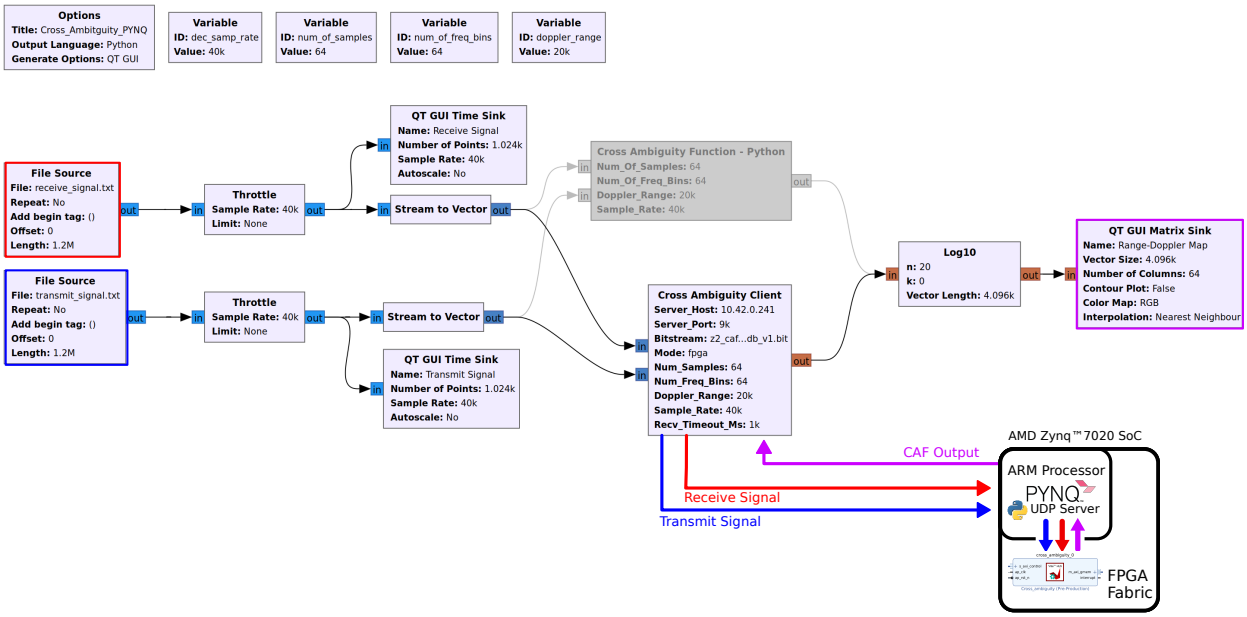

Figure 1. An overview of the developed hardware-in-the-loop GNU Radio flowgraph for accelerating the cross ambiguity function (CAF) for a continuous wave radar. The flowgraph traces the transmit (*blue*) and receive (*red*) signals as they flow through the system accelerated via an FPGA. Our work describes a workflow for the test and evaluation of an accelerated CAF (*e.g.* the "**Cross Ambiguity Client**") implemented on an AMD ZYNQ System-on-Chip. The programmable logic on the ZYNQ board accelerates the calculation of a range-Doppler map which is displayed to the user in real time (*purple*) through a GNU Radio interface.

Choi, 2023). A key component of PYNQ allows engineers to develop inside an interactive Jupyter notebook (Kluyver et al., 2016) for rapid prototyping as opposed to traditional ASIC-style design tools (Smith, 1998). PYNQ provides hooks (*i.e. overlays*) to accelerate Python functions using the FPGA fabric.

The radar routine we will use to demonstrate FPGA acceleration capabilities is the cross ambiguity function (CAF):

$$X(v) = \frac{IFFT\left(FFT(x) * conj\left(FFT\left(y * e^{\frac{2j\pi vm}{Fs}}\right)\right)\right)}{N} \quad \text{for } v = -\frac{R}{2}, -\frac{R}{2} + \frac{R}{M}, \ldots, \frac{R}{2} - \frac{R}{M}, \frac{R}{2} \tag{1}$$

where v is the frequency shift, m is the sample index, F_s is the sample rate, N is the total number of samples, R is the total Doppler extent centered on 0 Hz, M is the number of frequency bins, x is the transmit signal, and y is the receive signal. We choose to use explicit fast Fourier transform (FFT) and inverse-FFT ($IFFT$) notation in Eq. 1 as it maps closely to our hardware accelerated implementation. The CAF is designed to estimate the range and velocity of point targets in a scene illuminated by a continuous-wave radar. Our approach relies on taking this algorithm and targeting various FPGAs through the use of high-level synthesis tools which can translate C/C++ description into

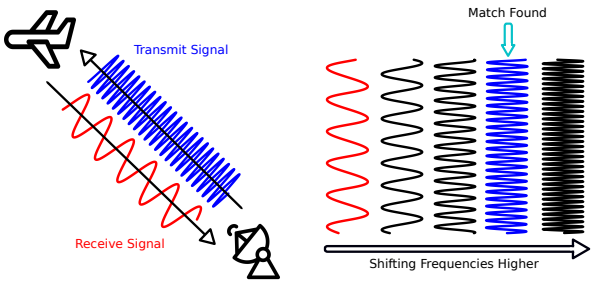

Figure 2. Continuous wave radar scenario illustrating the Doppler shift effect. The transmit signal (blue) is compared to the receive signal (red) across various frequencies to determine target velocity during the calculation of the cross ambiguity function. Since each frequency bin is independent the computation can occur in parallel.

a hardware-optimized code executed in a Python development environment. Throughout this work we iteratively develop the CAF algorithm in Python and then accelerate it on various FPGA hardware.

As shown in Figure 1, the CAF accepts two complex-valued input signals: a transmit and a receive signal. The

Table 1. Hardware Used

Component	Cost (USD)
PYNQ-Z2 Development Board	$200
MPSoC ZCU102 Evaluation Kit	$3,250
Nooelec NESDR SMArt v5 SDR	$40
Diamond Tri Band HT Antenna	$50

CAF involves calculating the cross-correlation between the transmit signal and frequency-shifted copies (*e.g.* Figure 2) of the receive signal, which produces a two-dimensional matrix showing range and Doppler shifts of objects in the scene relative to the radar receiver. This range-Doppler map, where peaks within the map indicate the presence of potential targets, can be integrated over time to discover objects in a scene. This function is well-suited to hardware acceleration due to independent computation across range and Doppler bins.

As part of this effort, we also integrate our accelerated signal processing blocks into GNU Radio, a free and open-source software development toolkit (Blossom, 2004). GNURadio is a collection of signal processing blocks written in C++/Python that are designed to implement various phases of the DSP chain. Libraries associated with GNURadio are popular in the educational and hobbyist communities due to the drag-and-drop functionality implemented in the GNURadio Companion tool and the visualization capabilities provided by off-the-shelf QT blocks. By integrating GNU Radio with our hardware-accelerated DSP cores, we are able to easily visualize where parts of the radar receiver chain are implemented and accelerate specific signal processing blocks on appropriate hardware; moreover, we can also incorporate existing developments from the GNU Radio community into our work. Our proposed workflow, described in Figure 3, affords rapid algorithm development and acceleration in a familiar software development environment. Our workflow emphasizes iterative development, enabling researchers and engineers to quickly prototype, test, and refine their radar systems.

This paper describes the implementation, test, and evaluation, of a hardware-in-the-loop simulation system which includes the design of a hardware-accelerated CAF core on an FPGA. We integrate this accelerated core into GNU Radio via a PYNQ-based driver and network service. We validate our approach against a simulated continuous-wave radar scenario, demonstrating the utility of PYNQ and GNU Radio for evaluating accelerated radar DSP algorithms.

Our work contributes to adaptable research and development workflows for radar applications using commodity hardware, open-source software, and emerging programming paradigms using FPGAs.

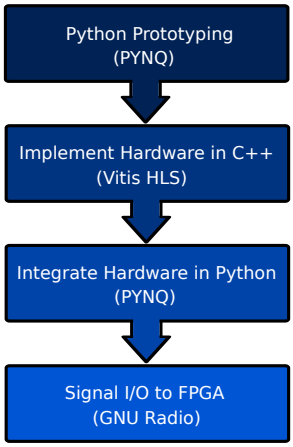

Figure 3. A high-level overview of our algorithm development life cycle. Our workflow aims for an initial development and validation of digital signal processing routines using Python, algorithm acceleration using C/C++ and Vitis HLS, followed by deployment within GNU Radio via the PYNQ framework.

2. Methods

This section describes the implementation of our radar simulation system which combines hardware acceleration on an FPGA with software-defined control in GNU Radio. Each component of the system is designed to prototype, validate, and eventually accelerate the CAF described in Eq. 1. Development was performed on a laptop with an Intel i7-8665U processor with 16GB of RAM and the FPGA/SDR hardware listed in Table 1. The CAF implementations were first accelerated on the PYNQ-Z2 board followed by the ZCU102 using the development workflow discussed in this section.

In the following subsections we will discuss the key software components and their interactions, starting with CAF algorithm design in Python, hardware acceleration in PYNQ, and integration into GNU Radio as shown in Figure 3.

2.1. Python Prototyping (PYNQ)

Our hardware-in-the-loop simulation workflow begins with the development of the radar signal processing steps within an interactive programming environment hosted on the ZYNQ processing system. This stage focuses on rapid prototyping of the core signal processing algorithms using a read-evaluate-print loop within a Jupyter notebook environment. This notebook, hosted on the FPGA's processing system, is accessible via a web browser and IP address over Ethernet. Within the Jupyter notebook we leverage various Python libraries to generate simulated radar signals,

```python
# Shift signal for cross ambiguity
def calculate_shifted_signals(signal, fft_length, num_of_freq_bins, freq_step, sample_rate):
    shifted_signals = np.zeros((num_of_freq_bins, fft_length), dtype=np.complex64)
    for j in range(num_of_freq_bins):
        f_shift = (j - (num_of_freq_bins/2)) * freq_step
        phase_shift = 2 * np.pi * f_shift * np.arange(num_of_samples) / sample_rate
        exponential_value = np.exp(1j * phase_shift)
        shifted_signals[j] = signal * exponential_value
    return shifted_signals

# Python cross correlation implementation
def cross_correlation(transmit_signal, receive_signal):
    transmit_fft = np.fft.fft(transmit_signal)
    receive_fft = np.fft.fft(receive_signal)
    transmit_fft_conjugate = np.conj(transmit_fft)
    signals_fft = transmit_fft_conjugate * receive_fft
    cross_correlation_numpy = np.fft.ifft(signals_fft)
    return cross_correlation_numpy

# Python cross ambiguity implementation
def calculate_cross_ambiguity(
    transmit_signal, receive_signal, num_of_samples, num_of_freq_bins, freq_step, sample_rate):
    cross_ambiguity = np.zeros((num_of_freq_bins, num_of_samples), dtype=np.complex64)
    shifted_signals = calculate_shifted_signals(
        receive_signal, num_of_samples, num_of_freq_bins, freq_step, sample_rate)
    for i in range(num_of_freq_bins):
        cross_correlation_result = cross_correlation(transmit_signal, shifted_signals[i]) / num_of_samples
        cross_ambiguity[i] = cross_correlation_result
    return np.fliplr(cross_ambiguity.real.T)
```

Figure 4. Python implementation of the cross ambiguity function. Code is developed inside a Jupyter notebook in PYNQ.

implement the cross-ambiguity function, and visualize the results.

Specifically, we begin by crafting simulated radar signals using NumPy arrays to represent the transmit and receive signals. We define key parameters (carrier frequency, bandwidth, sampling rate, and modulation scheme) and incorporated simulated target effects into the receive signal: time delay to represent range, Doppler frequency shift to represent the target velocity, and additive noise to model real-world conditions. This allowed us to create a reproducible environment for testing our initial CAF implementation later in our workflow.

Next, we implemented the core logic for the CAF in Python and generate the resulting range-Doppler map, which provides a visualization of the target's range and velocity simulated inside the receive signal. This allows us to quickly verify the correctness of the algorithm, tune scenario parameters, and explore different software designs. We show the Python code developed for this CAF implementation in Figure 4.

This Python-based prototyping phase lays the groundwork for subsequent hardware acceleration on the FPGA's programmable logic. The validated Python implementation serves as a reference model, ensuring that downstream FPGA algorithm acceleration behaves as expected. This initial Python design also provided a clear benchmark against which we can compare the runtime performance of the FPGA-accelerated implementations.

2.2. Implement Hardware in C++ (Vitis HLS)

Following the Python prototyping phase, the next step was to accelerate the CAF using AMD's Vitis High Level Synthesis (HLS) tool (AMD, 2025c). Vitis HLS enables the rapid creation of custom hardware accelerators. By allowing us to describe the cross-ambiguity algorithm in a high-

$$X(v) = \frac{IFFT\left(FFT(x) * conj\left(FFT\left(y * e^{\frac{2j\pi vn}{Fs}}\right)\right)\right)}{N}$$

$$\text{for } v = -\frac{R}{2}, -\frac{R}{2} + \frac{R}{M}, \ldots, \frac{R}{2} - \frac{R}{M}, \frac{R}{2}$$

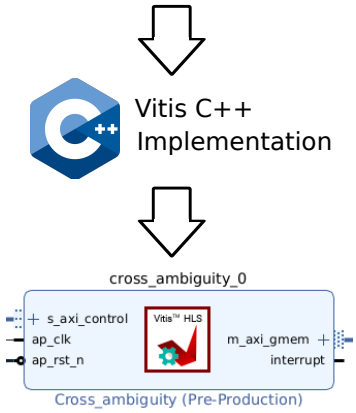

Figure 5. Overview of the tooling required to implement the accelerated cross ambiguity function. The algorithm is initially implemented in Python, converted to C++, and integrated into the Vitis HLS tool to be executed on the FGPA after synthesis in Vivado.

level language like C++, Vitis HLS abstracts away the complexities of traditional hardware design. Our workflow follows a two-step process: first, we port our Python implementation of the CAF into C++ for integration with the Vitis HLS tool. Second, Vitis HLS translates this high-level C++ implementation into a hardware design ready for synthesis and deployment on the FPGA's programmable logic as shown in Figure 5.

To optimize the accelerated CAF core, we leveraged several capabilities of Vitis HLS. First, we employed directives that enabled data flow optimization, allowing Vitis HLS to analyze data dependencies and automatically create a pipeline, to improve throughput. Second, we utilized directives to enable pipelining of computationally intensive loops within the CAF, which increases the processing speed. For mathematical operations like the fast-Fourier transform required in Eq. 1, we leveraged the highly optimized FFT IP Core (AMD, 2025a). In order to take advantage of the ZCU102's additional resources we configured the FFT to use a larger amount of DSP resources as opposed to the PYNQ-Z2 board.

Once the Vitis HLS stage was complete, we integrated the generated CAF IP block into the Vivado design environment (AMD, 2025d) which we used to generate the final bitstream to be loaded onto the FPGA. As shown in Fig-

ure 6, the CAF IP is connected to the ZYNQ processing system via AXI interconnects using Vivado. After configuring these connections we synthesized a bitstream file which we can use to reprogram the FPGA inside PYNQ.

2.3. Integrate Hardware in Python (PYNQ)

After generating the CAF IP core using Vitis HLS and Vivado, the next stage involved integrating it into the PYNQ environment. The PYNQ framework streamlines this integration through Python, taking advantage of the Jupyter Notebook environment running on the processing system of the FPGA.

Within PYNQ, we wrote a Python script to load our custom hardware accelerator onto the FPGA and developed a Python driver function to interact with the accelerated CAF core. This driver utilizes memory-mapped I/O (Figure 7) to access the control and data registers of the IP core, enabling us to configure relevant parameters, transfer input data, and retrieve the processed results of the CAF. The driver abstracts away the low-level hardware details, providing a high-level Python interface for controlling the core.

This new IP core can replace our previous Python-based CAF implementation prototyped in the Section 2.1. This workflow enables the developed hardware-accelerated implementation of the CAF to be accessible from Python. As part of this workflow, we can reuse the same simulated signals and visualization test harnesses initially implemented. This helped us verify that the FPGA-accelerated core produces equivalent results to the software-only implementation.

2.4. Signal I/O to FPGA (GNU Radio)

With the hardware-accelerated CAF core validated, the next step involved preparing for integration with GNU Radio. We achieved this by exposing the PYNQ-hosted CAF driver as a network service using a UDP server.

This UDP server serves as a bridge between GNU Radio and the PYNQ-based hardware. The server listens for UDP packets containing IQ data and system configurations, then uses the Python driver to interact with the accelerated CAF core. The CAF core generates data for a range-Doppler plot which is packaged into UDP packets and sent back to GNU Radio for visualization on the host computer.

This approach provides a flexible architecture where GNU Radio can run on a separate machine, communicating with the PYNQ board over a network. To exercise this architecture we leverage a software defined radio (SDR) to capture the transmit and receive signals. In the following sections we describe our simulation process for creating these transmit and receive signals inside of GNU Radio.

2.5. Transmit Signal Generation

To generate a realistic transmit signal for our radar simulation, we employ a separate GNU Radio flowgraph designed to capture real-world radio signals using an SDR. The flowgraph starts with an RTL-SDR source block, which captures radio signals from a Nooelec NESDR SMArt v5 SDR. This SDR is tuned to a specific frequency (typically an FM radio station) and a low-pass filter removes unwanted out-of-band noise and interference. The filtered signal is then passed to a file sink block, which writes the IQ samples to a file. This file serves as our reference transmit signal for subsequent radar simulation and testing. Recording the signal to a file ensures that the same IQ data is used for all CAF benchmarks, allowing for consistent performance comparisons. By leveraging real-world radio signals, we can create realistic and repeatable test scenarios for our hardware-in-the-loop radar simulation.

2.6. Receive Signal Generation

In order to simulate a realistic receive radar signal for our experiments, we use a GNU Radio flowgraph that captures the effects of the target and stationary clutter. This flowgraph takes the previously recorded transmit signal and processes it to generate a signal representative of a radar return. We leverage the modified signal to benchmark all of our experiments.

First, the recorded transmit signal is fed into a custom Python block implementing a Swerling target model. The Swerling model simulates the fluctuating radar cross-section (RCS) of a point target, which varies randomly over time due to changes in the target's orientation or surface characteristics. The block modulates the amplitude of the signal, to capture this phenomenon.

The signal containing the target is passed through additional Python blocks that impart a time delay and frequency shift as shown in Figure 8. The block simulates the range and velocity of a moving target which is captured by a time delay and frequency shift, respectively. This signal delay captures the time it takes for the signal to travel to propagate to and from the point target and the frequency shift captures Doppler shift caused by the target's motion relative to the radar transmitter and receiver. These values will be recovered by the CAF.

In order to simulate stationary clutter, representing radar returns from other objects in the environment, we employ multiple time-delay blocks with random delay values. These delays simulate radar reflections from stationary objects at different distances from the continuous wave radar.

Finally, the target return and the simulated clutter returns are combined using an "Add" block. This composite signal represents the complete receive signal, containing the target

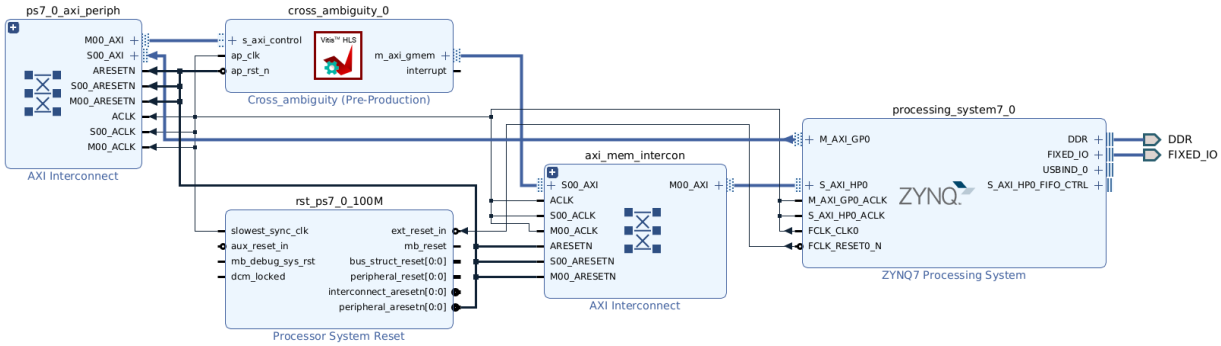

Figure 6. Vivado block diagram containing the cross ambiguity function core created in Vitis HLS. Vivado takes this block diagram and compiles it into bitstream that can be loaded onto the FPGA as an *overlay* within the PYNQ framework. Implementation of the cross ambiguity function as an *overlay* (*i.e.* implemented on the FPGA's programmable logic) provides significant speedup as compared to execution on the ZYNQ processing system.

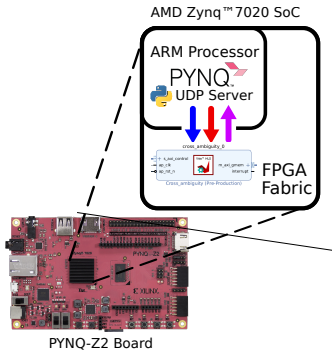

Figure 7. Visualization of where applications are running on the ZYNQ System-on-Chip. The memory-mapped "CAF Core" is running on the programmable logic which enables FPGA-based algorithm acceleration.

Figure 8. GNU Radio flowgraph capturing effects of a moving target on the receive signal.

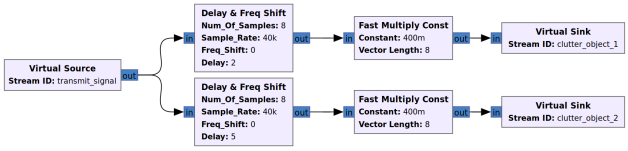

Figure 9. GNU Radio flowgraph capturing effects of stationary clutter on the receive signal.

return and clutter. This final signal is saved to a file, which can then be reused by our main flowgraph as shown in the "File Source" block in Figure 1.

2.7. Cross Ambiguity Client

To integrate the PYNQ-hosted hardware acceleration into GNU Radio, we developed a custom "**Cross Ambiguity Client**" block shown in Figure 1. This block acts as a UDP client within the GNU Radio flowgraph, which is responsible for sending IQ data to the UDP server running on the PYNQ board and receiving the resulting range-Doppler map calculated on the FPGA.

The block is initialized with network parameters (IP address and port), the bitstream name corresponding to the CAF core generated from Vitis HLS and signal processing parameters (number of samples, frequency bins, Doppler range, and sampling rate). Upon initialization, the block establishes a UDP connection with the server, sends a configuration message to initialize the CAF driver on the PYNQ board, and begins logging.

During operation, the block receives complex-valued transmit and receive signal vectors as input. It sends the input vectors to the server as a binary payload via UDP packets, and then waits for the range-Doppler map to be returned. After receiving a valid result, the block reshapes the data and returns it to the GNU Radio flowgraph. The block also includes error handling for timeouts and server errors, and logs performance metrics.

3. Results

This section details the performance evaluation of our FPGA-accelerated CAF within the hardware-in-the-loop radar simulation developed in Section 2. Our experiments

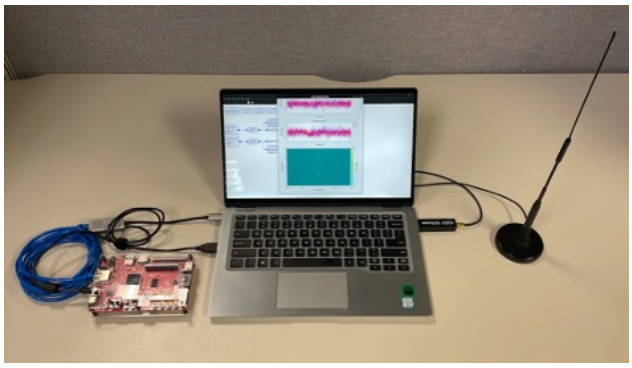

Figure 10. Photograph of our experimental setup for PYNQ-Z2 algorithm acceleration. From left to right: the PYNQ-Z2 boards is running the accelerated cross ambiguity function, the developer laptop is running GNU Radio, and the software defined radio on the right is collecting transmit and receive signals to exercise the system.

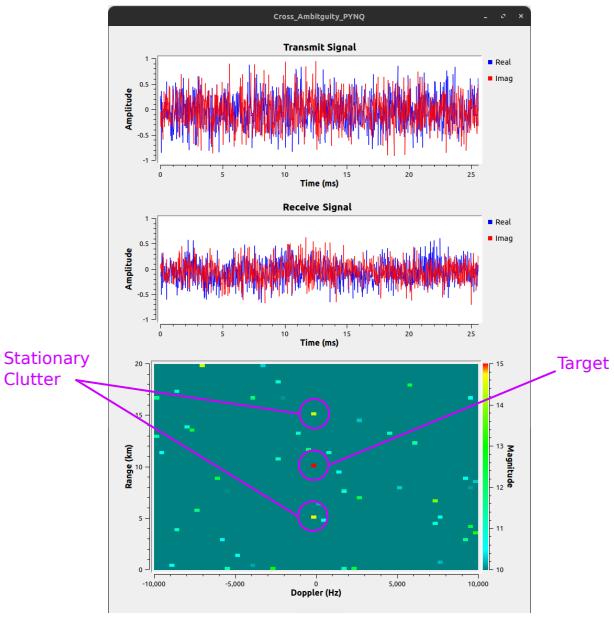

Figure 11. Range-Doppler plot output from GNU Radio. The top of the diagram shows the transmit and receive IQ samples fed into the accelerated cross ambiguity function. The bottom pane displays stationary clutter and target returns on the developer laptop.

focused on how the number of range-Doppler bins affects processing time and compared FPGA performance to the software implementation on a developer laptop as shown in Figure 10. In this section we investigate and quantify the performance benefits of FPGA acceleration, the overhead of data transfer, and the differences in performance between the PYNQ-Z2 and ZCU102 boards as the number of range-Doppler bins increases from 8x8 to 256x256.

In order to collect our results we execute the GNU Radio flowgraph developed in the previous sections. The flowgraph reads pre-recorded transmit and receive signals from file sources, feeds them into a custom Python block, which sends the data to the PYNQ board via UDP. The GNU Radio flowgraph displays a range-Doppler map back to the user. We can see in Figure 11 that the generated flowgraph identifies a target with a peak at zero-Hertz and ten-kilometers and multiple peaks associated with clutter. We leverage this set up during subsequent runtime performance evaluation of the CAF core on different FPGA hardware.

3.1. Performance Evaluation

To establish a baseline, we measured the CAF computation time using Python within GNU Radio on the developer laptop. Next, we evaluated the PYNQ-Z2 and ZCU102 FPGA development boards, measuring the on-board processing time for the CAF core running on the FPGA. Our results are presented in Figure 12. With the exception of the 8x8 bin size, the FPGA implementations consistently outperformed the developer laptop, achieving a 2x performance improvement at 256x256 range-Doppler bins.

Surprisingly, at the 256x256 bin size, the PYNQ-Z2 CAF implementation was faster than the ZCU102 (6.127ms vs 8.085ms). Although the ZCU102 used more DSP resources for its FFT implementation, the reason for this performance difference is unclear. It is possible that Vitis compiler optimizations during synthesis may have played a role, but this warrants further investigation.

In addition to benchmarking the CAF computation time, we measured the latency introduced by both boards due to transmit and receive data transfer. We calculated the average latency by subtracting the average CAF processing time from the round-trip time for data to travel from GNU Radio to the FPGA boards and back to GNU Radio.

The data in Figure 13 suggests increased overhead from data transfer at 128x128 and 256x256 bin sizes, further highlighting the impact of data transfer on latency. The ZCU102 generally performed better across all bins, as indicated by the lower latency values. This result is likely due to the ZCU102's more capable on-board processing system. However, the latency introduced by data transfer is a significant component of our overall runtime for both FPGA implementations in practice; this is particularly true at higher range-Doppler bin counts. This is likely due to Ethernet packet size limitations from our approach. This shortcoming is a consequence of our UDP server architecture and may be resolved by lower latency data transfer methods (*e.g.* PCIe) and is left as a future extension.

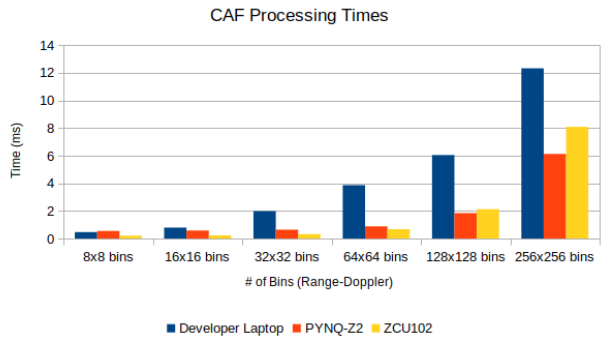

Figure 12. Cross ambiguity function processing times measured on various hardware platforms. Timing results are measured from before the cross ambiguity function is called to right after the range-Doppler data is generated.

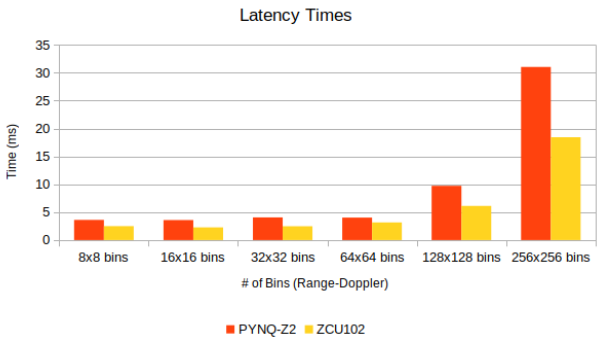

Figure 13. Latency added by signal data transfer for both FPGA platforms while computing the cross ambiguity function.

4. Conclusion

While the on-board processing time of the FPGA implementations on both the PYNQ Z2 and ZCU102 boards consistently demonstrated significant speedups compared to Python-based CAF implementation, our experiments showed that data transfer overhead represented a key factor limiting the overall performance of our hardware-in-the-loop system. Nonetheless, our approach shows the potential for the PYNQ framework to enable algorithm acceleration.

Future extensions of this work could focus on several key areas to enhance system performance. First, the system could be extended to leverage data transfer approaches with lower latency, which would better realize the processing benefits of the FPGA fabric. Additionally, researchers could explore more complex radar algorithms that effectively reduce data transfer costs by performing a larger amount of computation on the FPGA per data transmission. Further improvements could include offloading a greater portion, or even the entirety, of the radar signal processing chain to reside on the FPGA board, thereby reducing penalties incurred by data transfer using a similar PYNQ workflow described in this work. Future work could also investigate performance with a higher number of range-Doppler bins to fully characterize the system's scaling behavior on additional FPGA platforms.

This study presented a hardware-in-the-loop simulation workflow for radar systems which can be used accelerate the development and testing of digital signal processing algorithms for radar. Our work facilitates rapid experimentation by combining GNU Radio and PYNQ to accelerate algorithms on FPGA fabric with a focus on interactive development workflows that are accessible to a broad audience.

References

AMD. Fast fourier transform v9.1 logicore ip. https://docs.amd.com/r/en-US/pg109-xfft, 2025a. Product Guide PG109.

AMD. Pynq-z2 development board. AMD University Program, 2025b. URL https://www.amd.com/en/corporate/university-program/aup-boards/pynq-z2.html. Based on Zynq XC7Z020-1CLG400C SoC, Accessed: August 19, 2025.

AMD. Vitis hls, 2025c. URL https://docs.amd.com/r/en-US/ug1399-vitis-hls. User Guide UG1399.

AMD. Vivado design suite, 2025d. URL https://docs.amd.com/r/en-US/ug910-vivado-getting-started. User Guide UG910.

Analog Devices. EVAL-CN0566-RPIZ: X Band Phased Array Exploration Platform. Evaluation Board Documentation, 2023. URL https://wiki.analog.com/resources/eval/user-guides/circuits-from-the-lab/cn0566. Also known as ADALM-PHASER.

Blossom, Eric. Gnu radio: tools for exploring the radio frequency spectrum. *Linux journal*, 2004(122):4, 2004.

Braun, Martin, Pendlum, Jonathan, and Ettus, Matt. Rfnoc: Rf network-on-chip. In *Proceedings of the GNU Radio Conference*, volume 1, 2016.

Flandermeyer, Shane, Mattingly, Rylee, and Metcalf, Justin. gr-plasma: A new gnu radio-based tool for software-defined radar. In *Proceedings of the GNU Radio Conference*, volume 7, 2022.

Flandermeyer, Shane A, Mattingly, Rylee G, and Metcalf, Justin G. Deep reinforcement learning for cognitive radar spectrum sharing: A continuous control approach. *IEEE Transactions on Radar Systems*, 2:125–137, 2024.

Kern, Nicolai, Schoeder, Pirmin, and Waldschmidt, Christian. Virtually augmented radar measurements with hardware radar target simulators for machine learning applications. *IEEE Sensors Letters*, 8(3):1–4, 2024.

Kim, Victoria Heekyung and Choi, Kyuwon Ken. A reconfigurable cnn-based accelerator design for fast and energy-efficient object detection system on mobile fpga. *IEEE Access*, 11:59438–59445, 2023.

Kluyver, Thomas, Ragan-Kelley, Benjamin, Pérez, Fernando, Granger, Brian, Bussonnier, Matthias, Frederic, Jonathan, Kelley, Kyle, Hamrick, Jessica, Grout, Jason, Corlay, Sylvain, et al. Jupyter notebooks–a publishing format for reproducible computational workflows. In *Positioning and power in academic publishing: Players, agents and agendas*, pp. 87–90. IOS press, 2016.

Martone, Anthony F., Sherbondy, Kelly D., Kovarskiy, Jacob A., Kirk, Benjamin H., Narayanan, Ram M., Thornton, Charles E., Buehrer, R. Michael, Owen, Jonathan W., Ravenscroft, Brandon, Blunt, Shannon, Egbert, Austin, Goad, Adam, and Baylis, Charles. Closing the loop on cognitive radar for spectrum sharing. *IEEE Aerospace and Electronic Systems Magazine*, 36 (9):44–55, 2021. doi: 10.1109/MAES.2021.3072698.

Mattingly, Rylee G and Metcalf, Justin G. Adventures in rfnoc: Lessons learned from developing a real-time spectrum sensing block. In *Proceedings of the GNU Radio Conference*, volume 6, 2022.

PYNQ Project. Pynq: Python productivity for amd adaptive computing platforms, 2025. URL http://www.pynq.io/. Open-source framework for Zynq development.

Sahu, Geet, Seal, Ayan, Yazidi, Anis, and Krejcar, Ondrej. A dual-channel dehaze-net for single image dehazing in visual internet of things using pynq-z2 board. *IEEE Transactions on Automation Science and Engineering*, 21(1):305–319, 2022.

Smith, Douglas J. *HDL Chip Design: A practical guide for designing, synthesizing and simulating ASICs and FPGAs using VHDL or Verilog*. Doone publications, 1998.

Thornton, Charles E, Kozy, Mark A, Buehrer, R Michael, Martone, Anthony F, and Sherbondy, Kelly D. Deep reinforcement learning control for radar detection and tracking in congested spectral environments. *IEEE Transactions on Cognitive Communications and Networking*, 6(4):1335–1349, 2020.

Zhu, Bing, Sun, Yuhang, Zhao, Jian, Zhang, Sumin, Zhang, Peixing, and Song, Dongjian. Millimeter-wave radar in-the-loop testing for intelligent vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 23(8): 11126–11136, 2021.