UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

TRADESPACE ANALYSIS OF HETEROGENEOUS DATA PROCESSING IN ALL-DIGITAL PHASED ARRAY RADARS TO MAKE REAL-TIME PROCESSING POSSIBLE

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

Skyler Garner

Norman, Oklahoma

2026

TRADESPACE ANALYSIS OF HETEROGENEOUS DATA PROCESSING IN ALL-DIGITAL PHASED ARRAY RADARS TO MAKE REAL-TIME PROCESSING POSSIBLE

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY THE COMMITTEE CONSISTING OF

Dr. Mark Yeary, Chair

Dr. Justin Metcalf

Dr. Boonleng Cheong

Acknowledgments

I would like to thank my colleagues at the Advanced Radar Research Center, who have made this research possible. All of your dedication, spirit, and intelligence make the ARRC a wonderful place for research and development that I am proud to be a part of. Engineers such as John Meier, Cody Piersall, Matthew Herndon, James Martin, Boonleng Cheong, Rylee Mattingly, and more inspire me.

I would like to give thanks to my Mom and Dad, who have supported me tremendously always and instilled in me a great value for education. I would also like to thank my Grandparents for being my role models, I aspire to be like them. A special thanks to Devin Thompson, who has supported me educationally, professionally, and as a friend. Another special thanks to Sam Cox for seeing the best in me. Many friends also deserve thanks, such as Tyler Reavis, Zack Mitchell, Cora Defrancesco, Carlos Villanueva, Jacob Sacco, Jacob Marlow, and Jet Vellinga. Without their support this work would not be possible.

Finally, I would like to thank Mark Yeary. He has given me many professional opportunities, from my internship at the Naval Research Lab to my role as an engineer at the ARRC. He also encouraged me to go to grad school and helped immensely with this thesis. I can say that I am where I am today because of Dr. Yeary. Thank you to my committee members Justin Metcalf and Boonleng Cheong. I can tell how much care Dr. Metcalf puts into his classes and students. Dr. Cheong is a brilliant engineer that I am happy to work with and learn from.

Table of Contents

List of Tables

List of Figures

Abstract

All-digital phased array radars are an emerging technology that requires data processing techniques which are scalable, leverage open-architecture software components, and avoid bottlenecks for the real-time operation. Addressing these challenges relies on heterogeneous processing and distributed software development on CPUs, GPUs, and FPGAs, whereby the overarching algorithm processing chain is distributed in a way that begins at the ADC of each element on an array and concludes in actionable data products. An important aspect is not only the speed of the algorithms, but also system design considerations and throughput of packetized data transfer. These packets are routed and processed within a network of computing services. This thesis explores the tradespace of heterogeneous data processing for all-digital phased array radar and develops several key algorithms and presents architectural solutions. These include CPU-based Doppler processing using the FFTW guru interface, GPU based-Doppler processing using cuFFT, and an efficient GPU corner-turn. These algorithms are integrated into a scalable framework for use in real-time on Horus. The results quantify the impact of dataset size, PCIe throughput, and heterogeneous resource allocation on real-time performance, and they highlight system-level constraints that will inform future digital radar architectures. This thesis will demonstrate how heterogeneous computing, combined with a thorough understanding of software architecture, hardware architecture, and implementation algorithms, can address the substantial

throughput requirements of real-time signal processing for all-digital phased array radar.

Chapter 1

Introduction

1.1 Overview

Phased array radars consist of many individual elements that operate coherently. These systems can electronically steer beams, greatly improving overall flexibility. This flexibility enables several application-specific capabilities, including faster volume update rates. Electronically steered beams can scan a volume much faster than mechanical antenna motion. All-digital phased array radars offer several advantages over their analog counterparts. In this context, all-digital means that each element has the capability to convert analog signals to digital. Notable advantages include support for multiple simultaneous receive beams, fully reconfigurable beam shapes, low sidelobes, and flexible adaptive cancellation for interference suppression [1]. Additional advantages include graceful degradation in the presence of element failures.

All-digital phased array radar has grown in popularity as supporting components have improved. Commercial off-the-shelf (COTS) RF and digital electronics have become more capable and less expensive. Improvements in components such as analog-to-digital converters (ADCs) and field-programmable gate arrays (FPGAs), along with continued advancements associated with Moore's law, have made

this technology far more achievable. The increased availability of these components has led to extensive ongoing research on how to implement this technology. One example of this research is the Advanced Radar Research Center's (ARRC's) Horus program. The ARRC has two functional all-digital array radars, Horus-NOAA and Horus-D, as well as several panel-sized carts for testing.

A key drawback of all-digital phased array radar is the volume of data that must be processed. This arises from the fact that every element produces digital samples. This increased data volume is also what enables the system's enhanced flexibility. However, fully exploiting the capabilities of digital phased arrays requires processing this data in real time. Real-time processing is a key requirement that enables informed decisions about how to use the system. For example, the system may detect a dangerous storm and increase the resolution and update rate in its vicinity. Such adjustments enable more accurate and faster weather predictions, as well as higher-fidelity data for meteorological research.

A promising approach to addressing this challenge is heterogeneous computing. Heterogeneous computing refers to the use of multiple processor architectures to solve a computational problem. This approach enables leveraging the strengths of different processor architectures, such as central processing units (CPUs), field-programmable gate arrays (FPGAs), and graphics processing units (GPUs). Several considerations must be evaluated to determine which processes or algorithms are best suited for specific processor architectures. Both the algorithms and the processors must be understood in depth to make such judgments. Engineering effort and development time must also be taken into account.

In this thesis, the tradespace of heterogeneous computing for radar signal processing is examined with respect to increasing throughput to meet the requirements of all-digital phased array radar.

1.1.1 Document Outline

This document is organized into the following chapters:

Chapter 1 provides a literature review of heterogeneous processing for radar systems and background on the corner-turn operation. Next, it presents an overview of the data processing challenges associated with all-digital phased arrays, along with an introductory explanation of data processing in Horus.

Chapter 2 provides an overview of different processing architectures, including CPUs, GPUs, FPGAs, ASICs, and DSPs. It then presents an overview of radar signal processing algorithms with a focus on which architectures excel at each algorithm. Finally, it provides an in-depth discussion of CPU and GPU-based Doppler processing, including how each accomplishes the corner-turn operation. Performance metrics for CPU and GPU based-Doppler latency are presented in Section 2.3.3.

Chapter 3 provides an overview of the software and hardware architectures relevant to radar signal processing. It also includes an in-depth examination of data transfer, which often forms a bottleneck in radar signal processing.

Chapter 4 summarizes the thesis and clearly states its contributions, limitations, and broader impacts. A full list of algorithms implemented on Horus is provided, along with additional opportunities for future research in Section 4.4.

1.1.2 Literature Review

The master's thesis [2] focuses on GPU-accelerated radar signal processing. The combination of the GPU's parallel architecture, its flexibility through high-level languages, and its lower cost compared to FPGAs positions the GPU as a suitable option for radar signal processing. This thesis provides a comprehensive background on GPU architecture, CUDA, and signal representation. The litera-

ture review section examines current requirements and the utilization of GPUs in real-time radar signal processing. The primary applications are found in bistatic radar, particularly passive bistatic radar. Additional applications include phased array radar, pulse-Doppler radar, and MIMO radar. None of the surveyed works focus on all-digital phased arrays, a gap this thesis aims to address. The majority of surveyed papers in the literature review employ a heterogeneous CPU-GPU architecture. Two platforms utilize GPUs exclusively, while only one paper explores a heterogeneous CPU-GPU-FPGA architecture. Zaknoun et al. employ a CPU-GPU architecture to perform an FFT on a wideband radar signal [2]. The emphasis on wideband radar with a high sampling rate demonstrates the GPU's capability for high-throughput processing. A software library is employed to transfer data packets directly to the GPU, bypassing CPU memory staging, which often introduces bottlenecks. This approach represents an impressive solution that may be applicable in future implementations at the ARRC. The implementation achieved a throughput of 18 Gbps to the GPU. The data packets utilized are VITA-49, a format commonly associated with software-defined radio. This packet type is detailed in their thesis on pages 25 and 26. The data is subsequently extracted from the packet and processed within the GPU. This GPU implementation demonstrated a significant improvement in FFT processing speed and stability compared to popular CPU-based FFT libraries. Zaknoun's thesis illustrates the advantages of GPU-based radar signal processing. It further indicates that this approach is gaining popularity within the broader research community for diverse applications. This thesis will demonstrate how heterogeneous computing, combined with a thorough understanding of software architecture, hardware architecture, and implementation algorithms, can address the substantial throughput requirements of real-time signal processing for all-digital phased

array radar.

In the conference paper [3], Rupniewski et al. present a heterogeneous signal processing architecture for an active electronically scanned array (AESA). Their setup resembles Horus, as the array is divided into subarrays, with each subarray functioning as a separate channel. Similarly, Horus processes multiple channels of data concurrently, reflecting a comparable architectural approach. Additional similarities include the use of one data packet per pulse and an overall processing chain comparable to that employed for point-target processing in our system. The paper does not provide an in-depth explanation of the data packet structure, but notes that it consists of two components: a header and a payload. The header contains data relevant to the pointing angle, while the payload is immediately available for processing. The header is directed to the host (CPU-based system), whereas the payload is transferred to the device (GPU processor). Typically, the host provides instructions to the device, making this data partitioning a logical design choice. It is important to note that their data is simulated. The paper describes the integration of an FPGA, GPU, and CPU to perform radar signal processing. In many studies, GPUs are viewed as competitors to FPGAs; however, further research is needed on their collaborative use to achieve radar signal processing objectives. As noted in the paper, GPUs are often less effective for radar systems requiring low latency, such as surveillance radar. This limitation arises from the inherent characteristics of GPU hardware architecture. To achieve peak efficiency, defined here as high throughput in floating-point operations, GPUs require processing large data chunks. Radar types such as SAR and passive radar can arbitrarily segment large datasets for high-throughput processing, whereas surveillance radar requires low latency, necessitating smaller temporal data sizes. The authors targeted a latency of 50 ms and batched multiple CPIs across dif-

ferent elevations. Our all-digital phased array system at the ARRC enables the creation of large datasets, allowing us to leverage GPU efficiency. These larger datasets are not temporally extensive, which maintains high throughput without requiring batching. In their architecture, both the FPGA and GPU are connected to the host system via PCIe. They perform as much processing as possible on the GPU, while the FPGA possibly handles lower-level filtering operations. Remote direct memory access (RDMA) is employed to transfer radar data directly from the FPGA to the GPU, reducing latency by bypassing CPU overhead. The paper highlights the reduced development and debugging time associated with GPU-based algorithms compared to FPGA development. Their data cube comprises three dimensions: spatial elevations, slow-time pulses, and fast-time range samples. The following algorithms are applied to the data: downconversion, filtering, and decimation, which are performed on either the FPGA or GPU. Beamforming, pulse compression, Doppler filtering, and detection are executed on the GPU. Finally, estimation and tracking are performed on the CPU. Beamforming inherently reduces data volume, as the number of beams is smaller than the number of subarrays in their implementation. Beamforming is implemented as a complex matrix multiplication, a process that is highly parallelizable. Pulse compression is similarly highly parallelizable in the frequency domain. Any algorithm that is parallelizable has the potential to achieve exceptional efficiency on GPUs. Overall, the resource allocation strategy presented in the paper is sound, with some notable exceptions. Due to system differences, performing beamforming on the GPU in Horus is not feasible because of the extremely large number of channels. Without the associated data reduction, the overwhelming data volume could not be transferred off the array. The ARRC system also relies more heavily on CPU-based processing. While exclusive GPU-based processing is an interesting concept,

utilizing both CPU and GPU resources significantly increases the available computational capacity, enabling more efficient execution of radar signal processing tasks. Allocating tracking to the CPU is a sound decision, as it involves significant state management and benefits from branch prediction. The paper provides detailed discussion on inter-processor communication within the architecture, particularly between the FPGA and GPU. As noted by the authors, typical radar signal processing algorithms are constrained by memory bandwidth rather than computational complexity. The authors identify several CUDA memory optimization techniques, including tiling, loop unrolling, minimizing thread divergence within a warp, and utilizing shared memory. They further present GPU utilization metrics for each algorithm and provide explanations for variations in efficiency. Their results indicate a throughput exceeding five times real-time performance, achieved with acceptable latency. Overall, the system architecture presented in the paper is highly compelling. Although not identical to the ARRC's Horus system, it shares significant similarities and demonstrates promising results.

The authors in [4] provide a survey of GPU utilization in radar signal processing. The paper identifies GPUs as a key component of heterogeneous signal processing architectures. Compared to FPGAs and DSPs, GPUs offer superior scalability and lower cost, albeit at the expense of higher power consumption. The paper includes a detailed discussion of GPU architecture, which is optimized for compute-intensive, highly parallel tasks. This design prioritizes transistors for data processing rather than data caching and flow control. GPU architecture also provides greater processing stability. Here, stability refers to reduced variation in processing time. This increased stability results from minimal interactions between the GPU and the operating system. The authors emphasize that GPU development is more complex than CPU development, while FPGA development is

significantly more challenging than both. They note that due to PCIe data transfer latency, GPUs are unsuitable for processing tasks requiring sub-millisecond response times. The paper provides several examples and case studies on GPU applications in radar signal processing. One example involves pulse compression in an all-digital phased array high-frequency radar. Pulse compression is performed prior to digital beamforming, a process similar to multiple beam operations in Horus. This example demonstrates a nearly threefold increase in throughput and improved stability through GPU utilization. Finally, they present a case where GPUs achieved an overall fourfold performance gain in weather radar processing, primarily due to interpolation. These implementations illustrate the advantages of GPUs, particularly when handling the substantial data volumes generated by all-digital phased arrays.

The authors in [5] provide a detailed discussion of GPU-based CA-CFAR. The authors describe the development process, beginning with a prototype in MATLAB, followed by an implementation in native C, and ultimately a GPU-based implementation. CUDA was employed for the GPU implementation, accompanied by a detailed explanation of GPU shared memory management. The paper also addresses throughput and latency metrics, using the latency of processing a single RDM to estimate throughput. The authors note that CA-CFAR runtime is constrained by global memory bandwidth, which is exceptionally high on GPUs. A trend observed in the study indicates that GPU throughput increases with larger RDM sizes. The authors attributed this behavior to the overhead associated with data transfers between the host and device (CPU and GPU). As seen in [3], larger data sets allow the GPU to process more data in parallel, increasing the computational efficiency. The authors conclude that for typical dataset sizes, GPU utilization is not worthwhile when compared to CPU processing. However,

given the observed trend of increasing throughput with larger data sets, GPUs may prove beneficial for CA-CFAR in all-digital phased array systems. The authors also note that they were testing CA-CFAR in isolation and implementing more of the radar signal processing pipeline on the GPU may amortize the data transfer cost. Considering technological advancements over the past 14 years, this experiment warrants reevaluation using all-digital phased array data. Multiple RDMs, one for each beam, can be processed concurrently on the GPU, thereby enhancing parallelization.

The authors in [6] discuss real-time signal processing for one-dimensional wideband radar. The paper focuses on airborne and spaceborne SAR, as SAR is a common application of wideband radar. Wideband radar signal processing faces challenges similar to those encountered in all-digital phased array radar. Due to their substantial sampling rates, wideband radars require extremely high throughput. Both wideband radar and all-digital phased array systems require low latency, although the requirement is generally severe for surveillance radar. Given the focus on airborne and spaceborne SAR, these systems impose significantly stricter SWaP (Size, Weight, and Power) requirements. The paper specifically mentions using CPUs, GPUs, DSPs, and ASICs for ground-based radar as an approach to address the challenge of processing large data volumes with low latency. The authors emphasize the importance of employing "standard, modular, extensible, and reconfigurable architecture design methods." Due to the relatively high power consumption of CPUs and GPUs, these components were excluded from their spaceborne prototype. However, their results indicate that the CPU-GPU configuration achieved the lowest processing time. The authors identify the key technology in their system as the creation of "a single granularity processing node." This concept resulted in a prototype SoC integrating multiple processing architectures

with shared memory. This design yielded adequate processing performance and low power consumption. The SoC benefits from combining processor architectures without requiring memory transfers. Data transfer and memory management were key concerns in their design.

The authors in [7] present another study on real-time radar signal processing. The paper focuses on the evolving requirements of automotive radar systems. The emergence of autonomous driving technology has increased the need for automotive radar systems with a wider field of view. This requirement was addressed by adding more channels, enabling the radar to perform beamforming for an expanded field of view. However, increasing the number of channels also raises the required processing throughput. This must be achieved under strict latency constraints; in their study, the latency requirement was set at 50 ms. Their system employed a heterogeneous architecture, distributing processing tasks between CPUs and two Radar Processing Units (RPUs). According to [8], the RPU is an integrated circuit (ASIC) that performs non-coherent and coherent integration, CFAR, and local maximum detection. Processing tasks are allocated based on the strengths of each architecture. This design enables the ASICs to handle range-Doppler and angle processing using FFTs, while CPUs focus on statistical and post-processing tasks. In their design, RPUs and CPUs share a common radar memory, with processors using DMA to transfer data to and from this memory. The paper demonstrates how heterogeneous architectures can address challenges associated with increased throughput requirements. The use of shared memory parallels the approach in [6], offering an effective solution to data transfer challenges. The deployment of multiple RPUs and CPUs underscores the importance of parallel processing, as depicted in Fig. 3 of their paper. Each processor is assigned distinct tasks to maximize computational resource utilization.

1.1.3 Corner-Turn Background

The corner-turn has long been a challenging operation in radar signal processing. As data streams into the system, the range dimension is the fastest-changing. This layout is natural because samples acquired from the ADC are placed sequentially in memory. This organization is also convenient for early processing stages, such as pulse compression. Processing data in contiguous memory regions provides a significant advantage because accessing contiguous memory is substantially faster. Radar signal processing is often constrained by memory bandwidth. However, later stages operate across other dimensions, most notably Doppler processing, which operates across pulses. Accessing data in this range-dominant matrix layout requires retrieving each sample from a computed memory location, greatly slowing both memory reads and writes. Other algorithms, such as adaptive processing and CFAR, also operate across dimensions other than range and may require access across multiple dimensions simultaneously. The corner-turn is the process of reorienting the data layout to make it suitable for the next stage of processing. More technically, the corner-turn is a memory operation that copies and transposes a two-dimensional matrix of data, preparing it for an FFT across the range cells (i.e., the slow-time dimension) to form the range-Doppler map.

The corner-turn has been studied for many years because it is a central component of radar signal processing. Many techniques have been proposed to improve corner-turn performance, as increases in memory read and write throughput have progressed more slowly than transistor scaling predicted by Moore's law. Izumi et al. note that the corner-turn, which is part of their SAR processing chain, becomes a bottleneck in parallel algorithms due to high cache-miss rates [9]. As discussed in the CPU section, the cache is the fastest memory available to the processor but is limited in size. These high cache-miss rates arise from accessing

non-contiguous data and lead to poor performance, especially in parallel algorithms. In their work, the authors introduced a method they called the Square Block Corner-turn Technique. This method relies on blocking regions of the data matrix to improve the cache-hit rate. The block size depends on the algorithm and on several cache-memory properties, such as line size and cache hierarchy. Blocking the matrix increases the cache-hit rate because an entire block can be loaded into the cache at once. One limitation of blocking is that, in UNIX systems, processor assignment may change during execution. For this reason, the authors recommend using smaller block sizes to avoid assuming control of the entire cache. Meisl et al. provide an overview of SAR data processing, including the corner-turn within the context of SAR [10]. Synthetic aperture radar shares similarities with all-digital phased array radar, particularly in the need to process very large data sets. They emphasize that corner-turns are a potential source of inefficiency because the transpose operation is costly. They also note that memory operations can be a major bottleneck, particularly during corner-turn operations.

Several recent articles address corner-turn techniques; here, we discuss two from the last few years. Gigleux et al. discuss the corner-turn problem in the context of FPGA-based processing [11]. In systems that utilize an FPGA for pulse compression and Doppler processing, the same issue persists. This arises because these systems typically read from and write to external memory, and switching processing dimensions necessitates non-contiguous accesses. Their solution was to detect targets after pulse compression and before Doppler processing, a step they call predetection. After this predetection step, they perform random accesses only to memory regions likely to contain targets, reducing the overall amount of data that must be read out of order. This approach has caveats related to clutter and probability of detection. Because detecting before Doppler filtering

reduces SNR, the predetection false-alarm probability must be set higher, thereby capturing more data that must subsequently be transposed during the corner-turn. A second drawback concerns clutter. Beams with high clutter power can lead to excessive predetections, particularly because clutter is attenuated only during the Doppler processing step. Accordingly, the authors recommend this technique for low-clutter environments, such as space rendezvous and upper-atmosphere detect-and-avoid radar. Cochrane et al. present an FPGA-based signal processor for FMCW radar and spectroscopy [12]. The authors optimize for very low SWaP to enable spaceborne, airborne, and portable deployments. With a focus on low SWaP, they implemented the corner-turn to use no more memory than is required to store a single frame of data. They achieved this using a read-first, write-second strategy, which they refer to as patterned memory addressing. In their implementation, each read is paired with a write to the location associated with the read sample. This method resulted in the highest latency of their entire processing chain.

1.2 Data Creation

All-digital phased array radars are pushing the limits of modern data and signal processing techniques. Overcoming these challenges requires new approaches to distributed data processing, rather than relying on a single computing node to produce actionable data. Consequently, heterogeneous processing is essential, as evidenced by [3], [13], [14], particularly as digital radar architectures continue to flourish [15]–[20].

Modern and next-generation radar systems that are digital at every element, such as the Horus program at the Advanced Radar Research Center (ARRC), have the ability to generate an enormous amount of data [21]–[23]. For example, with

a sampling rate of 125 MSPS, a sample size of complex64, and 1,664 elements on a populated array, the system can theoretically produce upwards of 1.6 terabytes per second. The complex64 data type consists of two IEEE 754 single-precision binary floating-point numbers (float32s): One real and one imaginary component per sample.

The amount of data produced by radar systems is increasing faster than computers can process, given the decline of Moore's Law. The advantages of phased array radar rely heavily on digital signal processing. These strengths mostly come from the processing of multiple beams simultaneously and the subsequent dynamic allocation of phased array resources. Timely decision-making is critical to effectively utilize phased array radar resources. Without reliable real-time processing, increasing the number of beams will only exacerbate latency and delay decision-making.

1.2.1 Shaping Data

To begin, the data are arranged into a processing cube organized along the dimensions of fast-time, slow-time, and beam cluster, ordered from the fastest-changing index to the slowest. For our arrays, the fastest changing index is the rightmost index, with decreasing speed toward the leftmost index. This is "row-major" because most processing is done in C++. In computing, "row-major" order is a method for storing multidimensional arrays in linear storage such as random access memory [24]–[26]. In brief, we have followed the IEEE Std. 754 for binary floating-point arithmetic for bit assignments. On Horus, the sampling size chosen for processing is 32-bit floating-point numbers, for a total size of 64 bits for each complex sample. This is converted from the 16-bit floating-point numbers produced by the FPGAs (e.g., we use Intel Arria 10 GX). The conversion is done

Figure 1.1: Example of Horus forming four spatial beams.

because most modern CPUs do not have ALUs that support 16-bit floats, and almost all GPUs can use 32- or 64-bit floats. This conversion also adds additional dynamic range before pulse compression. Our full data matrix for a single pulse is as follows:

$$\text{BeamGroup} \times \text{Beam} \times \text{Sample} \tag{1.1}$$

Alternatively, if combining multiple pulses in a CPI:

$$\text{Pulse} \times \text{BeamGroup} \times \text{Beam} \times \text{Sample} \tag{1.2}$$

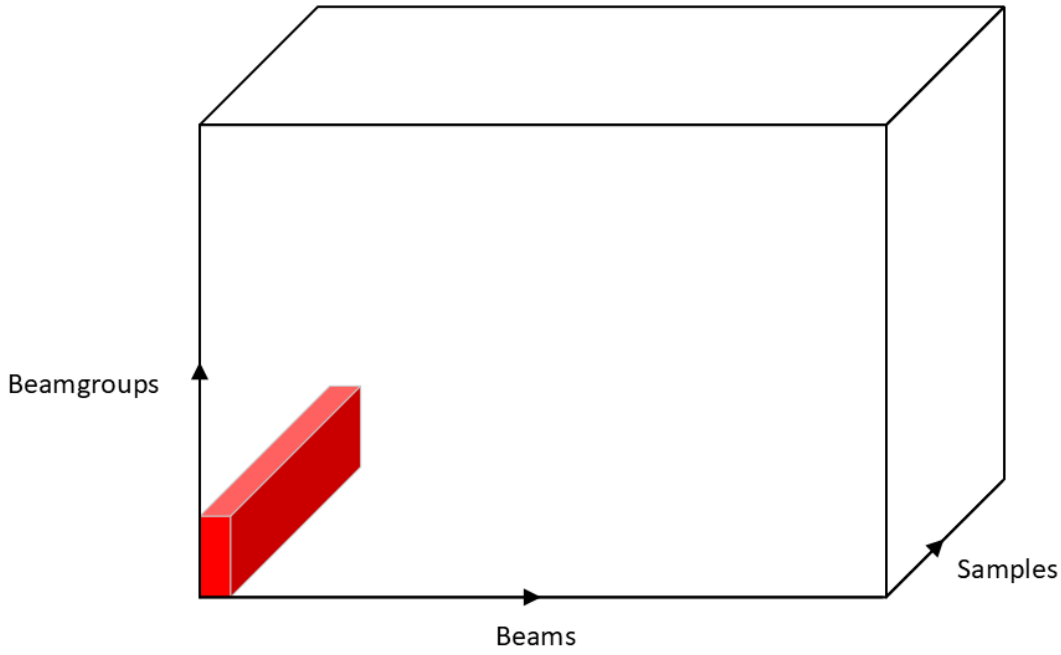

Figure 1.2: Representation of a pulse as a data cube.

This data aggregation is shown as a cube in Fig. 1.2. A four-dimensional object cannot be represented here; conceptually, this corresponds to a tesseract. Alternatively, this can be visualized as multiple data cubes over time, corresponding to pulses. The beam group and beam dimensions are unique to the phased array

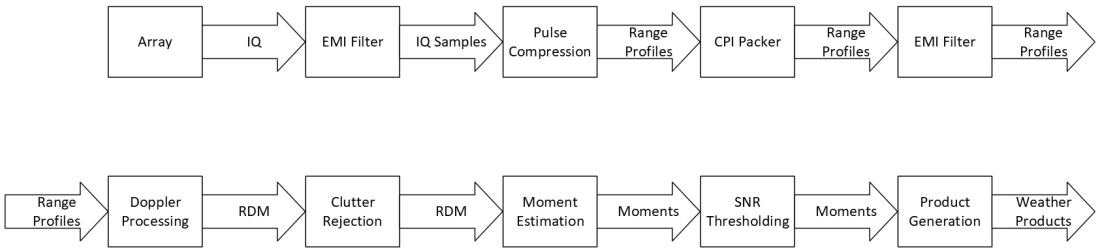

Figure 1.3: Algorithms being implemented for Horus weather processing.

data cube, in contrast to a parabolic dish. The beam dimension represents beams in space. For example, multiple receive beams in a fan configuration to image a volume, such as in Fig. 1.1. The beam group dimension sorts which elements should be paired together to create the spatial beams. An example of this is the horizontal and vertical polarizations used in weather processing. Another example of beam groups is the use of sub-panels for finer backend beamforming.

These extra dimensions quickly increase the required throughput for a signal processing chain to keep up with real-time. Along with the radar IQ data, there are important metadata that must be considered. These metadata include pulse repetition frequency (PRF), sampling frequency, and details of the transmitted waveform. Because phased array radars are highly flexible, many parameters can change rapidly, and processing must adapt accordingly.

1.2.2 Processing Chains

All-digital phased arrays require changes to the general radar signal processing chain. Dish radars and even analog phased array radars typically process only one beam at a time. An exception is dual polarization radars, which process two beams simultaneously. One beam is polarized horizontally and one vertically, as the differences in the return power between the polarizations provide information about atmospheric conditions.

The processing chain in Fig. 1.3 is a high-level overview of the planned signal processing algorithms for Horus radars. There is a different chain between point-target and weather processing. Incorporating flexibility into the radar chain is crucial to using all-digital phased arrays.

First scan updates are provided to the system by the operator. These settings define the desired behavior of the radar system. This includes setting the PRF, waveform, array beamforming characteristics, and other settings. The array takes these settings and distributes them to all elements on the radar. Pulses are beamformed and transmitted. The radar then listens for return signals.

At this point, the array beamforms the data toward the selected directions specified in the scan settings. Beamforming crucially reduces the amount of IQ data that must be transferred from the array. This operation reduces the data rate provided that the number of receive beams is less than the total number of elements. If the number of beams exceeds the number of elements on a system as large as Horus, the data rate becomes impractically high.

Another stage where the data changes shape is the backend beamforming step, which can increase or decrease the number of beams depending on the application. For example, breaking the array into subarrays and then beamforming their outputs can produce more spatial beams than the number of subarrays. Because the sizes of subarrays and number of beams are arbitrary, there could be any amount of data expansion or reduction.

Chapter 2

Heterogeneous Computing

2.1 Overview

One approach to increasing signal processing throughput is heterogeneous computing. Heterogeneous computing involves combining different types of compute architectures to enhance performance. Some examples of these compute architectures include multicore CPUs, FPGAs, GPUs, ASICs, and DSPs. The ARRC primarily focuses on CPUs and FPGAs, while Horus and future projects are expected to leverage powerful GPUs for specific tasks. Certain kinds of processors are better suited for particular tasks.

Heterogeneous computing approaches the computational problem holistically. A thorough understanding of the key features of different compute architectures, combined with deep knowledge of the problem space, enables acceleration of compute-intensive applications. One such compute-intensive application is radar signal processing, which will be discussed later in this text. However, this approach significantly increases design and implementation complexity. This trade-off is necessary to keep pace with the data rates produced by all-digital phased-array radars.

Understanding the problem is essential for selecting the appropriate compute

architecture. For radar signal processing, specific processing algorithms will be examined independently. A deep understanding of available compute architectures is necessary to map algorithms to specific processors. Algorithm classification typically involves mapping them to patterns of computation and memory access. In the following sections, we will analyze each processor along with its strengths and limitations. According to Amdahl's law, it is crucial to accelerate only those components that benefit the most [27], [28]. Amdahl's law states that the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is utilized. An example would be a task that takes 50 milliseconds to compute. If 20 milliseconds of the task cannot be parallelized while the remainder can, the minimum achievable latency is 20 milliseconds. Therefore, a process must exhibit substantial parallelism to justify acceleration on specialized hardware. Employing multiple compute architectures generally increases complexity and development time; therefore, such decisions must be made wisely.

Several drawbacks accompany the use of multiple heterogeneous processors. The most significant drawback is the need to transfer data between separate processors. Even if an algorithm executes faster on a GPU, the combined transfer and processing time must outperform CPU execution for GPU utilization to be worthwhile. For reference, Horus is using servers with PCIe 4.0, which gives a theoretical throughput of 2 GB per second per lane. The system has 8 lanes for the FPGA card, and GPUs often have 16 lanes. Another drawback is development time. Using a definition of computational efficiency that includes development effort, developer time is an important part of overall efficiency. Programming an FPGA in Verilog or a GPU using CUDA is generally far more time-consuming than developing on a CPU in C++ or Python. Therefore, heterogeneous comput-

ing should be employed only where it provides substantial benefit.

2.1.1 Central Processing Unit (CPU)

CPUs have the longest history among these processor architectures. They have existed since the advent of stored-program computers and have evolved significantly over time. Their physical dimensions have decreased dramatically, with manufacturing tolerances approaching the nanometer scale. CPUs have evolved into general-purpose processors, enabling mass production for a wide range of tasks. As CPUs improved, they encountered fundamental limitations. One such limitation is heat dissipation, as increasing transistor density concentrates power in a smaller area. Another limitation arises from data synchronization; as clock rates increase, maintaining coherence within a single large CPU becomes problematic. The solution found to help with both of these problems is multicore processing. It is also essential to consider how memory interacts with CPUs. Initially, memory and processor speeds were comparable, but processors soon outpaced memory read and write speeds. This called for another solution: the cache. A CPU cache is a small, high-speed memory located close to the processor. It reduces memory access latency and stores data most frequently used by processes.

All of these past issues have come to create the modern CPU. The modern CPU is fast, versatile, and ubiquitous for good reason. Nearly all systems rely on the CPU in some capacity. If there is only one processor a system can choose due to constraints, it will almost certainly be the CPU. In many architectures, the CPU also facilitates data transfer between other processors. CPUs come in many shapes and sizes for different tasks. For example, a server CPU differs significantly from a microcontroller. Even after selecting a CPU for a specific algorithm, there are many CPU configuration choices remaining. CPUs feature high memory band-

width; for instance, modern DDR4 RAM operating at 3200 MHz can transfer up to 3.2 gigabytes per second per lane. Memory bandwidth is a dominating factor for many algorithms because it cannot be parallelized. According to Amdahl's law, which states that processing speed is limited by non-parallelizable or serial components, memory transfer becomes a crucial consideration.

CPUs offer high single-thread performance and efficient branch prediction. This makes them ideal for algorithms that cannot leverage parallel processing and those requiring frequent decision-making. CPUs support the widest range of libraries and programming languages, making them the most convenient platform for algorithm prototyping. With their integrated caches, CPUs are particularly effective for algorithms that maintain state or rely on stored data from previous computations. Figure 2.1 shows a simplified layout of a CPU and its memory. This layout can be compared and contrasted with that shown in Fig. 2.2. A CPU typically has fewer cores, with many transistors dedicated to control logic and caching. Through complex flow-control mechanisms and large cache structures, the CPU is able to avoid expensive memory-access costs. Here, control refers to the control unit, which converts instructions into hardware-level signals and incorporates optimizations such as pipelining and branch prediction.

From a radar signal processing perspective, we will examine each processor type and their advantages and weaknesses. CPUs are very useful for radar signal processing. They offer strong performance while remaining versatile and cost-effective. A useful guideline is that algorithms requiring complex decision-making are well-suited for CPU processing. This is because CPUs are highly flexible and designed for general-purpose computing. Programming on CPUs typically offers the fastest development cycle, an important consideration in system design. This advantage stems from the wide availability of languages and libraries compatible

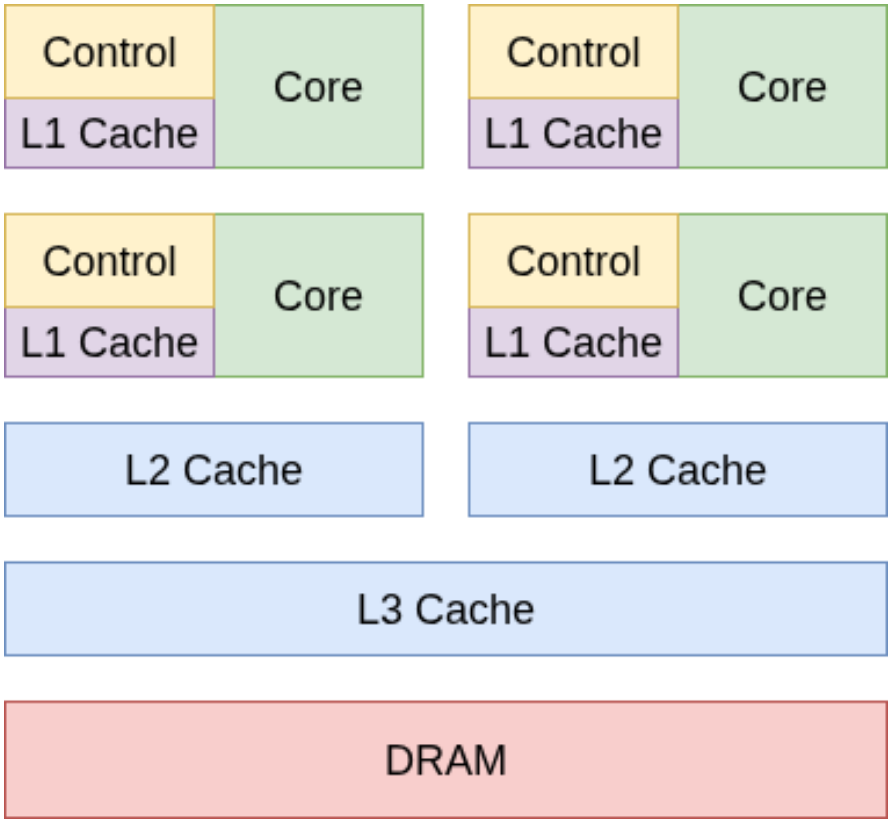

Figure 2.1: Diagram of CPU memory hierarchy and cores, adapted from the CUDA C Programming Guide [29].

with CPUs. This makes CPUs ideal for services that are complex or primarily single-threaded, such as tracking algorithms, which often benefit from CPU-based processing. More use cases for CPUs include adaptive clutter rejection and fine parameter estimation. Drawbacks of CPUs include a limited number of cores, lack of customization, and reduced efficiency in highly parallel processing tasks.

2.1.2 Graphics Processing Unit (GPU)

GPUs were initially designed to accelerate computer graphics, as their name suggests. The first integrated circuits for graphics acceleration appeared in arcade machines during the 1970s. At the time, GPUs provided a cost-effective alternative to using expensive RAM for buffering display content. In the 1980s, an increasing number of computer systems adopted dedicated graphics chips, particularly in personal computers. The 1980s also marked the introduction of the first large-scale integration GPU chip. The 1990s witnessed a significant increase in real-time processing for 3D video games. This demand drove rapid advancements in GPU development. In fact, the term "GPU" was coined by Sony for the 32-bit graphics processor in the original PlayStation, released in 1994. Notably, the release of OpenGL in 1992 made GPU software development and utilization significantly easier.

Eventually, GPUs were recognized for applications beyond video processing. Over time, GPUs evolved into specialized tools for a wide range of applications. This shift began in the 2000s, when NVIDIA released GPUs for personal computers that significantly outperformed previous generations. Research began into how to use these devices for applications such as machine learning, oil exploration, scientific image processing, linear algebra, statistics, 3D reconstruction, and stock options pricing. Nvidia introduced CUDA in 2007, creating an easy-

to-use API for programmers to have more control of the GPU. The widespread adoption of CUDA accelerated research into GPU-based computing across many applications. The 2010s brought significant advancements in both consumer and industrial GPUs. Today, GPUs are integrated into nearly every device with a display. Smartphones, laptops, desktops, car dashboards, gaming consoles, and even some radar systems have GPUs.

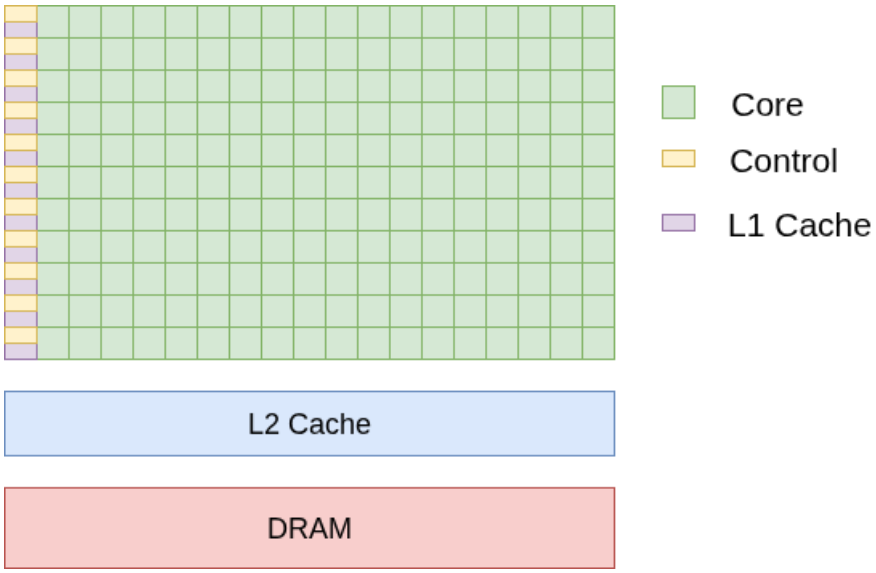

Figure 2.2: Diagram of GPU memory hierarchy and cores, adapted from the CUDA C Programming Guide [29].

Figure 2.2 provides another example of processor resources. As shown, a GPU contains many more cores than a CPU. Multiple cores share a single control unit and L1 cache, forming what is known as a Streaming Multiprocessor (SM). An SM is the fundamental processing unit within NVIDIA GPUs and is roughly analogous to a CPU core. An SM consists of CUDA cores (ALUs), control structures organized into warps, and an L1 cache that is shared among the cores, referred to as shared memory. The GPU's design dedicates more resources to raw data processing by employing many cores to achieve higher floating-point operations per second (FLOPS). This design can offset expensive memory-access latencies,

but doing so requires highly parallel algorithms that can fully exploit the GPU architecture.

GPUs contain significantly more cores than CPUs but operate at lower clock rates and offer less versatility. This means GPUs rely heavily on parallel processing. GPU programming requires more specialized knowledge than CPU programming but is generally considered much less complex than FPGA development. GPUs excel at performing large numbers of repetitive operations that can be handled by their arithmetic logic units (ALUs). GPUs specialize in floating-point operations, and their performance is typically measured in FLOPS (floating-point operations per second). For most GPUs, 32-bit floating-point numbers, known as floats, offer the highest performance. They can also perform operations on 64-bit floating point numbers, or doubles, with reduced performance. For example, NVIDIA's A100 can achieve 19.5 teraflops with 32-bit floats and 9.7 teraflops with 64-bit doubles. Internal memory bandwidth is extremely fast in GPUs. This high bandwidth enables their massively parallel architecture to load data efficiently. For example, NVIDIA's A100 offers a memory bandwidth of approximately 2 terabytes per second. This should not be confused with the PCIe transfer speed used to move data into GPU memory. PCIe transfer speeds are generally much lower than internal memory bandwidth. CUDA provides access to numerous high-performance mathematical libraries, including cuBLAS and cuFFT. These libraries make GPU programming practical for implementing a wide range of algorithms. Drawbacks of GPUs include data transfer, high energy consumption and cost, limited memory capacity, and their rigid architecture restricts them to a narrow set of specialized tasks.

2.1.3 Field Programmable Gate Array (FPGA)

FPGAs are relatively recent, with the first models introduced in the mid to late 1980s. They are made of programmable logic gates and interconnects. Early FPGAs contained thousands of gates, but by the 1990s and 2000s, they scaled to millions. This increase in functionality was accompanied by growing demand as industries increasingly adopted FPGAs. Their highly parallel architecture initially attracted military applications, but FPGAs later expanded to domains such as medical imaging and hardware acceleration. For example, Microsoft integrated FPGAs into its search engine and AI research infrastructure during the 2010s.

FPGAs are another method of processing to be discussed. FPGAs are very fast and highly customizable. They are generally more expensive and demand extensive development time and specialized expertise. FPGAs should therefore be used for critical low-latency applications. In an all-digital radar system, the primary candidate for FPGA implementation is beamforming all elements immediately after sampling.

2.1.4 Application-Specific Integrated Circuit (ASIC)

ASICs are specialized digital circuits designed for dedicated functions. Unlike general-purpose circuits and compute architectures, ASICs are tailored for specific tasks. Early ASIC designs employed gate arrays, which interconnected basic logic gates during manufacturing. With technological advancements, ASICs became faster and more compact. A contemporary example in radar signal processing is the monolithic microwave integrated circuit (MMIC). MMICs enable large-scale production of phased-array digital processing systems, thereby reducing costs. ASIC development entails a high barrier to entry due to substantial design expenses and factory tooling costs. When demand is sufficiently high, these costs

become justifiable and may even result in performance gains. ASICs offer the highest performance, with lower latency than FPGAs, albeit at the expense of complete lack of reconfigurability. FPGAs and ASICs share a common origin in gate array technology and exhibit similar design principles. Both are programmed or implemented using hardware description languages (HDLs), such as VHDL or Verilog. Consequently, they are suited for similar computational tasks, though their economic implications differ significantly. ASICs are advantageous when production volumes are high or when lower latency than FPGAs can provide is required.

ASICs are not planned for use in the Horus system and are rarely employed at the ARRC. This is primarily because programmability is essential for research hardware, such as with FPGAs, CPUs, and GPUs. FPGAs are generally more cost-efficient for lower production volumes as well. MMICs are commonly utilized in all-digital phased-array systems where production volumes are high.

2.1.5 Digital Signal Processor (DSP)

DSPs are specialized microprocessors optimized for computationally intensive operations, such as Texas Instruments' C6000 series or Analog Devices' SHARC family. DSPs represent one of the earliest dedicated architectures for real-time signal processing [30]. Much of the traditional role of DSP processors has been absorbed by FPGAs due to their flexibility, reconfigurability, and parallelism [31]. Modern FPGAs incorporate dedicated DSP blocks, often numbering in the hundreds or thousands, that implement the arithmetic operations traditionally performed by standalone DSP processors. DSPs are typically programmed in C or assembly, which generally results in shorter development cycles compared to FPGAs. As microprocessors, DSPs can handle branching operations, but they also present

several limitations. Due to their lower clock rates relative to CPUs, DSPs rely on specialized parallelism, such as SIMD units and optimized memory hierarchies, to sustain real-time performance. DSPs are well-suited for applications with lower data rates; however, all-digital phased-array systems operate at extremely high data rates. Consequently, DSPs were not selected for implementation in the Horus system.

2.2 Radar Signal Processing

A thorough understanding of radar signal processing is essential for the effective implementation of heterogeneous computing. Transforming raw IQ samples into usable data products involves multiple processing stages. The choice of algorithms depends heavily on the specific application. For instance, algorithms employed in weather radar, point-target radar, and synthetic aperture radar differ significantly. Moreover, techniques within a single application can vary substantially across different systems. Given the vast number of potential algorithms, this section focuses exclusively on the most widely adopted algorithms.

2.2.1 Pulse Compression

Pulse compression is employed in nearly all radar signal processing chains. It leverages the transmitted waveform to focus the energy from the pulse in fast time, or range. There are many waveforms used for different properties, for now we will focus on linear frequency modulated (LFM) waveforms. As illustrated in Fig. 2.3, the waveform's frequency varies over time. Because radar is an active sensing system, the transmitted waveform can be used to identify returns from the radar itself.

Processing this algorithm requires either time domain convolution or conver-

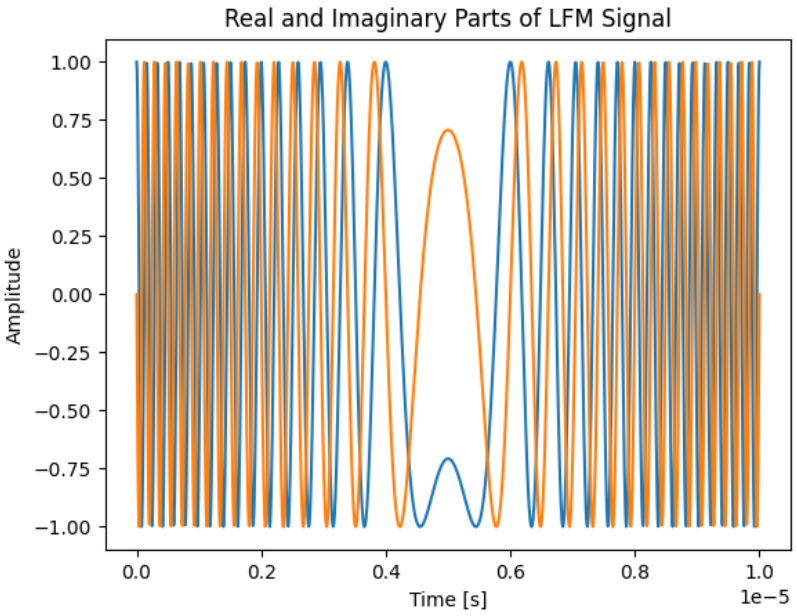

Figure 2.3: Simulated LFM waveform.

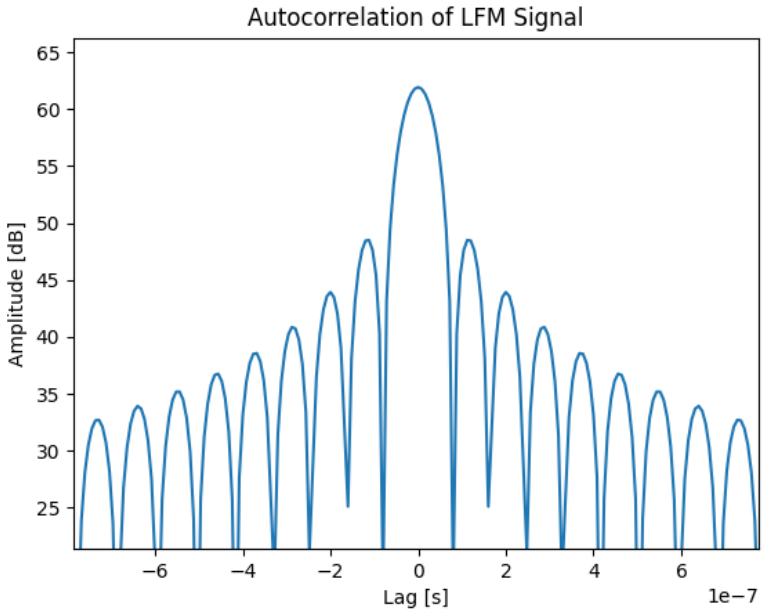

Figure 2.4: Simulated autocorrelation of an LFM waveform.

sion to the frequency domain followed by multiplication. Generally, frequency domain processing is significantly faster, although its efficiency depends on the number of waveform and range samples. The Fast Fourier Transform (FFT) is employed to convert to the frequency domain. This processing step must be applied to each range profile. In a phased-array system, each pulse may correspond to multiple spatial beams and beam groups, each associated with a distinct range profile.

This algorithm is well-suited for execution on either CPUs or GPUs. Both architectures can perform this algorithm with relatively low latency. This algorithm forms the foundation for subsequent processing steps. CPUs handle this algorithm efficiently; however, resource utilization depends significantly on the number of beams processed concurrently. Typically, the CPU spawns a new thread for each range profile, consuming a portion of the available cores. GPUs are massively parallel; however, transferring the data onto the GPU is often the dominating factor in overall latency and throughput.

2.2.2 Doppler Processing

Doppler processing varies across applications. In point-target processing, a coherent processing interval (CPI) is employed to estimate phase shifts between pulses, or slow time. This value is obtained by applying an FFT in slow time, which concentrates the pulse-to-pulse energy into Doppler frequencies that correspond to radial velocity, as shown in Equation 2.1.

$$F_D = \frac{2v}{c} F_0 = \frac{2v}{\lambda_0} \tag{2.1}$$

Where F_D is the Doppler frequency, v is radial velocity, c is the speed of light, F_0 is the carrier frequency, and λ_0 is the wavelength. Since each range bin requires

its own FFT, Doppler processing involves numerous transforms.

Doppler processing is a highly parallelizable process. The number of FFTs required scales with the product of range samples, spatial beams, and beam groups. Consequently, the compute architectures of GPUs are ideally suited for this algorithm. CPU implementations remain reasonably efficient because the FFT size is relatively small, typically equal to the number of pulses. As a result, each FFT executes quickly, allowing threads to process multiple transforms sequentially. Due to lower memory transfer overhead and fast execution, CPUs are not a bad choice for this algorithm. The performance for this algorithm depends heavily on the CPU resources allocated to the task.

Weather processing uses a different Doppler processing technique known as pulse-pair processing. This method avoids a costly FFT by assuming that the Doppler-frequency spread follows a Gaussian random distribution. Pulse-pair processing compares the phase of consecutive samples on a range-gate by range-gate basis. This approach directly estimates the primary moments of the Doppler spectrum, namely the mean velocity and the spectrum width. The spectrum width represents the statistical spread of the weather spectrum. This process does not require an FFT but instead relies on per-gate vector operations. Because this computation is generally not computationally expensive, it is generally not worthwhile to offload it to a GPU. The CPU's fast clock rate, SIMD capabilities, and direct access to the memory where the samples reside make it an efficient target for pulse-pair processing.

2.2.3 Beamforming

Beamforming is the algorithm that enables a phased array to steer beams in space. Beamforming can be applied during both transmit and receive; however, this dis-

cussion focuses on receive beamforming. Beamforming can be implemented using various techniques and remains an active area of research. The simplest approach to beamforming involves delaying and summing the signals from each array element. This delay is often implemented as a phase shift at a fixed frequency. An alternative is true time delay, which maintains beam accuracy across frequency variations. True time delay is computationally more demanding than phase shifting. Phase shifting requires a complex multiplication for each data sample of every array element. In contrast, true time delay requires processing through an FIR filter, which involves convolution. Additionally, storing FIR filters in real time imposes significant memory requirements.

Although these algorithms are relatively simple, they must process the full volume of data. Each channel, two channels per element in a dual-polarization all-digital phased array radar, generates a stream of data that must be processed. Subsequent algorithms depend on the data reduction achieved through beamforming. Beamforming consolidates numerous element channels into a manageable set of beams. Minimizing latency in beamforming is critical due to its frequent use across the array. Beamforming is a fundamental operation in all phased array systems. FPGAs provide the necessary specialization and low latency for beamforming tasks. Their high configurability allows FPGAs to operate near the radar front ends without difficulty. This capability enables FPGAs to process large volumes of data by distributing computations across the array. Low latency further enhances FPGA suitability for real-time beamforming. GPUs offer the parallelism needed to achieve high throughput for this task, but their performance is constrained by data transfer requirements. GPUs may be well-suited for backend beamforming, where the reduced data volume results in lower data transfer demands.

2.3 Heterogeneous Computing in Radar Signal Processing

Heterogeneous computing for radar signal processing involves using the most applicable compute architecture to accomplish the algorithm. This section will overview some heterogeneous real time processing algorithms that were implemented and lessons learned. Because of the lower development time for algorithms on CPUs and GPUs, as well as their applicability, they are focused on in this section. All algorithms mentioned here are implemented to work efficiently on digital phased array radar, meaning that along with range and pulse dimensions we will also have an arbitrary number of beams to process all at once.

Using GPUs for highly parallelizable algorithms is where the performance of the GPU is best. Because of the large size of the data, coming in large part from multiple receive beams, the efficiency of the data transfer is increased. CPU-based algorithms are often developed first for prototyping, and then optimized to run more smoothly. All algorithms discussed in this section are optimized and written in C++.

2.3.1 CPU-Based Doppler Processing

A CPU-based Doppler processing algorithm was developed to operate on Horus. The algorithm's objective is to generate Range-Doppler Maps for each beam received from preceding services. As previously noted, the IQ data must be properly aligned to ensure compatibility with the algorithm. The dimensions of the input data for the service must be ordered as: number of beam groups, number of spatial beams, number of pulses, and number of samples. The data is structured such that the number of samples is the fastest changing dimension, optimizing pulse compression. Pulse compression is performed across range samples for each beam group, spatial beam, and pulse. This data arrangement simplifies and accelerates

processing, as the data remains contiguous in memory. Consequently, the algorithm requires only two parameters for frequency domain conversion: the number of samples in a range swath and the number of FFTs to be performed. However, this layout presents a notable drawback.

Doppler processing requires operations across pulses, meaning an FFT must be performed for each beam group, spatial beam, and range bin. If the pulse dimension is not the fastest-changing, performance degradation occurs. This issue can be addressed either by reorganizing the data in memory or by enhancing the FFT algorithm to stride across dimensions. Experiments with data reorganization, or reshaping, revealed that the performance cost of copying memory was excessively high. Reshaping requires striding through the data to extract elements in the pulse dimension and copying them into a new matrix of equivalent size. Subsequently, the data must be copied into the FFT algorithm for processing, a step that is unavoidable. This approach is suboptimal because it introduces additional data copies, a primary obstacle to optimization. In theory, enhancing the FFT algorithm to stride through the data and copy it directly into its processing routine would eliminate intermediate steps. This modification would eliminate the need for an intermediate matrix, thereby conserving both time and memory resources. This led to the exploration of a lesser-known and harshly documented FFTW feature: the "guru interface."

FFTW is a software library designed to perform discrete Fourier transforms, or DFTs. The acronym FFTW stands for Fastest Fourier Transform in the West, which is partially proven by its widespread adoption within the scientific community. According to the FFTW documentation, "The 'guru' interface to FFTW is intended to expose as much as possible of the flexibility in the underlying FFTW architecture." It allows one to compute multidimensional "vectors" (loops) of mul-

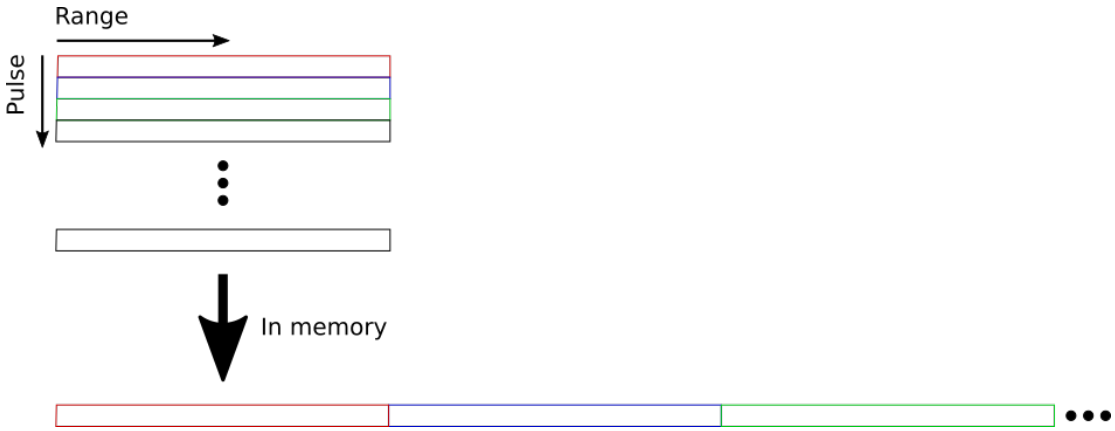

Figure 2.5: Layout of one range pulse matrix. Each combination of beam and beam group has one.

tidimensional transforms, where each vector/transform dimension has an independent size and stride. One can also use more general complex-number formats, e.g. separate real and imaginary arrays." While this appears to be a promising solution, it introduces additional complexity. The implementation requires striding through each image to group samples by pulse, as well as striding across images through two higher dimensions of beams. Furthermore, the large number of range bins necessitates performing as many FFTs as computational resources allow. Leveraging the flexibility of the guru interface successfully achieved this objective. After extensive troubleshooting, this solution was implemented and demonstrated significantly higher throughput compared to the copy-based approach. This implementation underscores the practical utility of the guru interface in optimizing multidimensional FFT operations.

The algorithm also utilizes key metadata: the sampling frequency and the pulse repetition interval. While not essential for the core algorithm, these metadata are required to construct the range and Doppler axes. The general equations for frequency sample spacing and bin indexing are provided below. The frequency

resolution is defined by the sampling frequency f_s and number of samples N.

$$\Delta f = \frac{f_s}{N} \tag{2.2}$$

The frequency value f_n for bin n is calculated as:

$$f_n = \Delta f \cdot n, \quad n = 0, 1, 2, ..., N-1 \tag{2.3}$$

Once the frequency reaches half the sampling rate, the spectrum wraps around to represent negative frequencies. Consequently, the higher frequencies are shifted to the left, forming the Doppler axis from $-\frac{fs}{2}$ to $\frac{fs}{2}$. This operation, known as an FFT shift, reorders the frequency components to produce a frequency axis centered on zero Hertz. This frequency axis relates directly to Equation 2.1, where each bin value f_n corresponds to a specific Doppler frequency F_D. Consequently, the sampling rate (PRF) determines the maximum frequency the FFT can resolve, which in turn limits the unambiguous velocity of the radar.

2.3.2 GPU-Based Corner-Turn

The publications from 1.1.3 outline the corner-turn problem and survey several proposed solutions. A common feature across these implementations is a deep understanding of processor memory. For example, Izumi et al. accelerated their algorithm by optimizing for CPU cache behavior [9]. In recent years, solutions have increasingly targeted application-specific constraints. Because our ground-based weather and point-target processing involves high clutter power, the technique in [11] is not suitable for our operating environment. Similarly, our priorities are throughput and latency rather than aggressive SWaP minimization as in [12]; therefore, that approach is not aligned with our objectives. Returning to the more

fundamental work, blocking or tiling as described in [9] is applicable to our case. A compute architecture with higher effective memory bandwidth would be ideal. Accordingly, we apply a GPU to this problem.

As discussed in Section 2.1.2, GPUs have very high theoretical memory bandwidth. This bandwidth is enabled by a wide memory bus with many data lanes, specialized high-speed memory, and massive parallelism that supports SIMD execution. The memory diagram in Figure 2.2 illustrates how this memory hierarchy is utilized. The full row including the cores, shared memory, and control is known as a Streaming Multiprocessor (SM). Within an SM, accesses to L1 cache and shared memory have very low latency relative to off-chip, or global, memory. Shared memory is software-controlled memory within the SM that all cores within the SM can access with low latency. By contrast, accesses that traverse the off-chip memory such as L2 or DRAM, are much slower than on-chip accesses in shared memory or L1. Effective use of SIMD requires coalesced, contiguous memory access patterns, which helps sustain high bandwidth. A clear understanding of the processor's memory architecture enables a GPU design that leverages high effective memory bandwidth to perform the corner-turn efficiently.

To perform the corner-turn operation, the data must first be loaded into the GPU. To increase throughput, the entire data set is transferred at the beginning, and pointers to specific images are computed for each corner-turn. Transferring larger data sets increases overall throughput because it reduces memory-transfer overhead across more data [5] [3]. Each pulse-by-range section is referred to as an image in this context. A separate image is formed for each beam-group and spatial-beam combination. The objective is to convert these into range-by-pulse images. A dedicated function was implemented to transpose a single image, and it is invoked for each image in the data tensor. Here, tensor refers to a multi-

dimensional array, an appropriate abstraction for all-digital phased array radar given the inherently high dimensionality. These tensors are implemented using the C++ Eigen library, which facilitates efficient data manipulation. Eigen is a widely used library for linear algebra. The transpose function accepts the following inputs: a pointer to the input data, the image width and height, and a pointer specifying the output location. The input data pointer is computed for each beam-group and spatial-beam combination prior to being passed to the function. The image dimensions are determined by radar configuration parameters, specifically the number of range samples and pulses. Finally, the output pointer directs the transposed data to the memory region designated for FFT Doppler processing.

The corner-turn function includes an additional parameter: the tile size. This parameter is defined at compile time and determines how many samples are loaded into shared memory. After copying contiguous blocks from global memory into shared memory, each tile is transposed locally. The copy from global memory is fast because accesses are contiguous and coalesced; the GPU memory system is designed for this access pattern. The local transpose is fast because it executes entirely in on-chip shared memory, avoiding non-contiguous global-memory accesses (i.e., of the conventional CPU). Each thread relocates one sample to its transposed position, maximizing the parallelism of GPUs. Finally, the transposed data in shared memory is written back to the location allocated by the FFT plan in global memory. This process is outlined in Fig. 2.6, where the original matrix has range as the fastest changing dimension. The data are copied into the shared memory tile, transposed locally, and written out to global memory in the correct index. A key point is that the transposed data is also written contiguously, further accelerating the corner-turn. Multiple tiles are transposed concurrently,

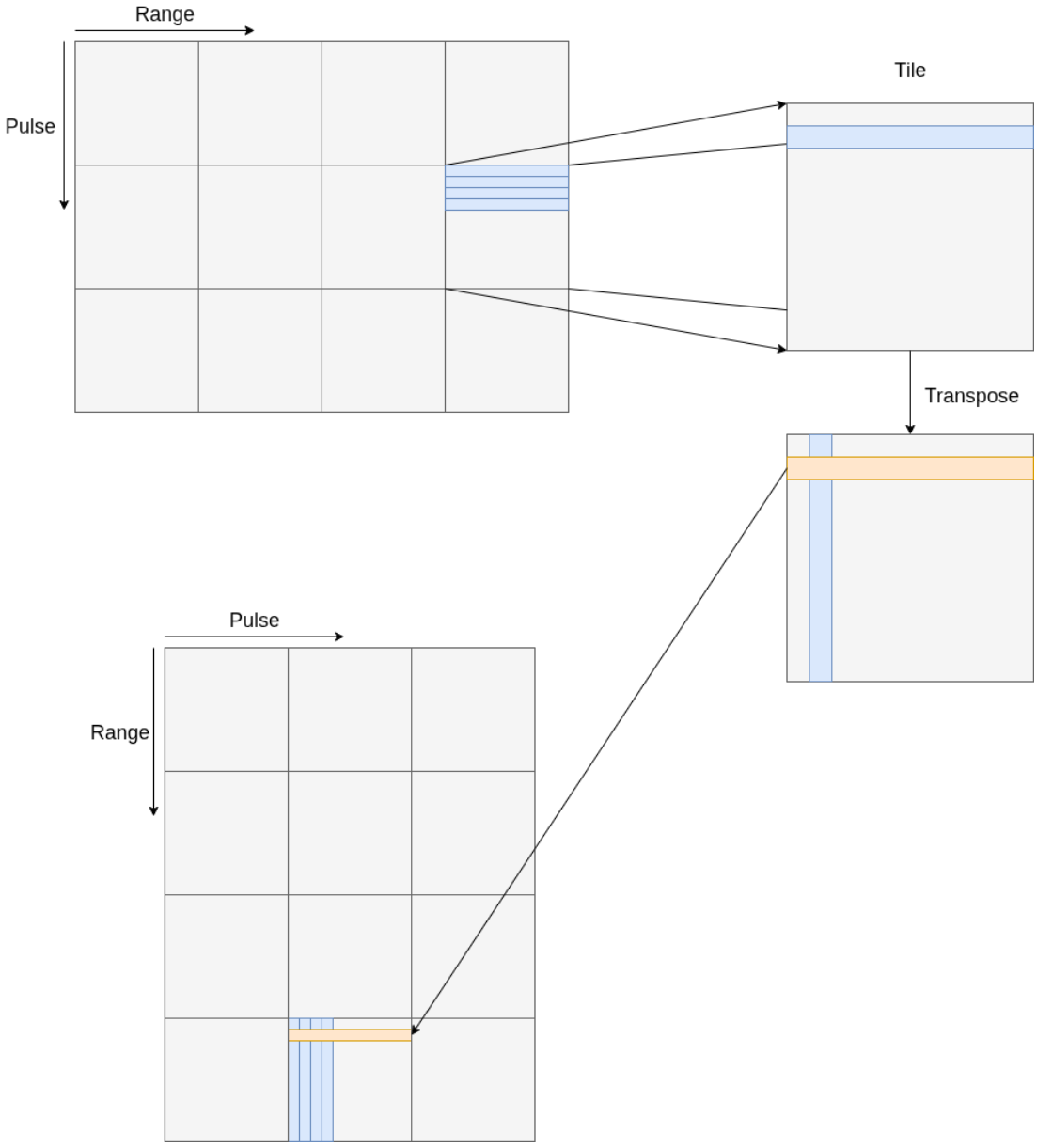

Figure 2.6: Diagram of transposing with tiles of memory.

further increasing parallelism. Each thread performs a bounds check on its assigned index to handle cases where the tile size does not evenly divide the number of bins or pulses. Because most memory operations now involve contiguous, coalesced blocks, the transpose algorithm performs only marginally slower than a direct copy. Achieving such efficient corner-turn performance within Doppler processing highlights the potential of heterogeneous computing for radar signal processing. The same technique can be applied to optimize algorithms that include a two-dimensional FFT, which typically requires a transpose between the row and column transforms.

2.3.3 GPU-Based Doppler Processing

The GPU-based Doppler processing implementation encounters challenges similar to those of its CPU counterpart. To leverage the strengths of the GPU's distinct hardware architecture, an alternative approach was adopted. As previously discussed, two key points are relevant: First, software development on GPUs is generally more complex due to more limited library support. Second, GPUs have very high internal memory bandwidth. NVIDIA's cuFFT is a GPU-based FFT library with interfaces designed to closely resemble those of FFTW. It is part of a suite of libraries developed by NVIDIA to make GPUs more accessible for software development. Regarding the first point, cuFFT lacks a highly flexible interface such as FFTW's guru interface, which can accommodate more complex data layouts. However, cuFFT provides a function named cufftPlanMany, which enables numerous concurrent FFT computations. This function requires the input data to be pre-aligned with a fixed stride across dimensions. In other words, the pulse dimension must be the fastest changing dimension to ensure compatibility. Typically, FFT libraries operate by creating a computation plan. The

plan must be configured with details about the DFT, including its size, input and output data locations in memory, the number of transforms, and any stride between datasets. Since input and output locations in memory are predefined, data must be explicitly copied into and out of these memory buffers. Regarding the second point, the necessity of copying data into the GPU presents an opportunity to exploit its high internal memory bandwidth.

The proposed approach involves copying the data into the GPU in its original dimensions and then rearranging it to align the data for processing. However, effectively utilizing the GPU's high memory bandwidth proved challenging. A naive implementation would retrieve individual samples from global memory at the required stride and copy them sequentially into the output matrix. However, accessing non-contiguous locations in global memory significantly reduces throughput compared to retrieving blocks of contiguous samples. To mitigate this, shared memory integrated into the GPU architecture can be utilized. As on-chip memory located close to the executing threads, shared memory offers significantly faster access times than global memory. Accessing shared memory can be orders of magnitude faster than accessing global memory. Its shared nature enables threads within a block to collaborate efficiently on data-intensive tasks. Section 2.3.2 explains the corner-turn operation on the GPU in depth.

At this stage, the radar IQ data resides on the GPU and has undergone the corner-turn operation. The data is now prepared for further processing to generate the service output. Because the data is aligned with the pulse dimension as the fastest-changing dimension, the FFT plan can be configured. In practice, the FFT plan is established prior to these operations. FFT plan initialization incurs a setup cost; however, Horus rarely changes its Volume Coverage Pattern (VCP). Consequently, a new plan is created only when no existing FFT plan is available or

when the incoming data dimensions differ from those specified in the current plan. The plan requires information on the number of FFTs to perform and the size of each transform. Next, the pointer to the device, or GPU, memory is provided to the FFT plan. The FFT plan is also configured to perform an in-place FFT, in which the input memory is overwritten by the output. A blocking function is then invoked to ensure the GPU completes the FFT operation before processing continues. Finally, the processed data is copied from device memory back to host, or CPU, memory. The data is then packaged into a Babeldict for output, along with the generated range and Doppler axes.

Table 2.1: Table of processing time for Doppler processing for a dataset size of $2 \times 16 \times 64 \times 1,354$ samples.

Operation	GPU time (µs)	CPU time 1 core (µs)	CPU time 4 core (µs)	CPU time 8 core (µs)
Copy in	1,807	1,950	1,846	1,879
Computation	317	4,741	1,493	1,295
Copy out	1,767	NA	NA	NA
Packet creation	14,711	16,493	16,284	15,232
Total	18,602	23,184	19,623	18,406
Total excluding packet	3,891	6,691	3,339	3,174

Table 2.1 presents performance metrics for both the CPU and GPU implementations of the Doppler processing algorithm. Multiple CPU tests were performed using different numbers of processing cores. These CPU cores are utilized by the FFTW library to perform the Doppler processing. Both implementations must move data into a processing buffer: the CPU copies data into an FFTW buffer, whereas the GPU copies data into device memory. The next stage is computation, which consists solely of performing the many FFTs required for each range bin and beam. Finally, the copy out stage applies only to the GPU, as the CPU can use its processing buffer directly for packet creation. Once the data have been

copied out of the GPU, they are packaged into a Babeldict and forwarded to the next stage of the processing chain.

Table 2.2: Table of processing time for Doppler processing for a dataset size of $2 \times 4 \times 128 \times 15,625$ samples.

Operation	GPU time (µs)	CPU time 8 core (µs)
Copy in	13,884	26,450
Computation	1,441	9,077
Copy out	13,318	NA
Total	28,643	35,527

Timing measurements were taken before and after each block, and the difference between these timestamps yields the latency for that block. These measurements were collected while processing a large volume of data and were averaged to obtain the results shown in Tables 2.1 and 2.2. The first few measurements for each case were discarded to allow each processor to warm up. Here, warm-up refers to initializing the FFT plan and allowing the processors to allocate resources such as caches and branch predictors. Two tables are provided to illustrate the performance of each implementation for different data sizes. The first example in Table 2.1 uses 2 beam groups and 16 spatial beams, with 64 pulses and 1,354 fast-time samples, resulting in 32 range-pulse matrices of 64 $\times$ 1,354 complex samples. The second example in Table 2.2 uses 2 beam groups and 4 spatial beams, with 128 pulses and 15,625 fast-time samples, resulting in 8 range-pulse matrices of 128 $\times$ 15,625 complex samples. The FFT length is given by the number of pulses, while the number of FFTs to be performed per range-pulse matrix is given by the number of fast-time samples. The number of FFTs is increased multiplicatively with the number of range-pulse matrices.

The discrepancy between the copy in and copy out latencies on the GPU is explained by the GPU-based corner-turn. Once the data are moved onto the GPU,

they are immediately transposed directly into the FFT buffer. This operation was surprisingly fast, taking on average only 40 µs longer than the copy out step for the smaller dataset and 570 µs for the larger matrices. One drawback of this approach is increased GPU memory usage: the dataset must reside on the GPU twice simultaneously, once for the initial copy and once for the FFT buffer. A noteworthy result is that the host-side copy into the CPU buffer took longer than the host-to-device copy into the GPU. The CPU-side copy step could likely be reduced through multithreaded memory operations, but this also illustrates how high PCIe transfer rates can be during large transfers. The smaller dataset contains a total of 2,772,992 samples per CPI, and with 8 bytes per complex sample, this corresponds to 22.2 MB per CPI. Dividing this by the average copy out time yields a throughput of 12.55 GB/s. Applying the same calculation to the larger dataset gives 9.61 GB/s. This result was unexpected, as the larger dataset should, in principle, have less overhead and therefore achieve higher throughput. However, the reduced throughput may be explained by the use of pageable host memory. Because the larger dataset span many more host-memory pages, it is more likely to be physically fragmented. This fragmentation increases overhead for device-to-host transfers, as the DMA engine must process a larger scatter-gather list instead of streaming the data into a single contiguous region, thereby reducing effective copy throughput.

Another noteworthy result is the time required to construct the packet. This outcome was unexpected but is understandable in hindsight. Although the various packet types will be discussed later in the thesis, this latency can be reduced. Flexible buffer structures have been useful during development, but hardening these packet formats can reduce latency. Notably, packet creation on the CPU took longer on average, which may indicate that copying data out of the FFTW

buffer was slower than expected.

Overall, the GPU demonstrates strong performance for the all-digital phased-array radar use case. For the large dataset, the GPU implementation operates 19.4% faster than the CPU implementation. This increase in throughput results from the GPU-based corner-turn and the use of many batched transforms. The difference in computation time between the GPU and CPU is substantial, and continued improvements in data transfer speeds will further expand the achievable performance. In contrast, the smaller dataset operates 22.6% slower on the GPU, indicating that larger datasets are generally required for GPU acceleration to be effective.

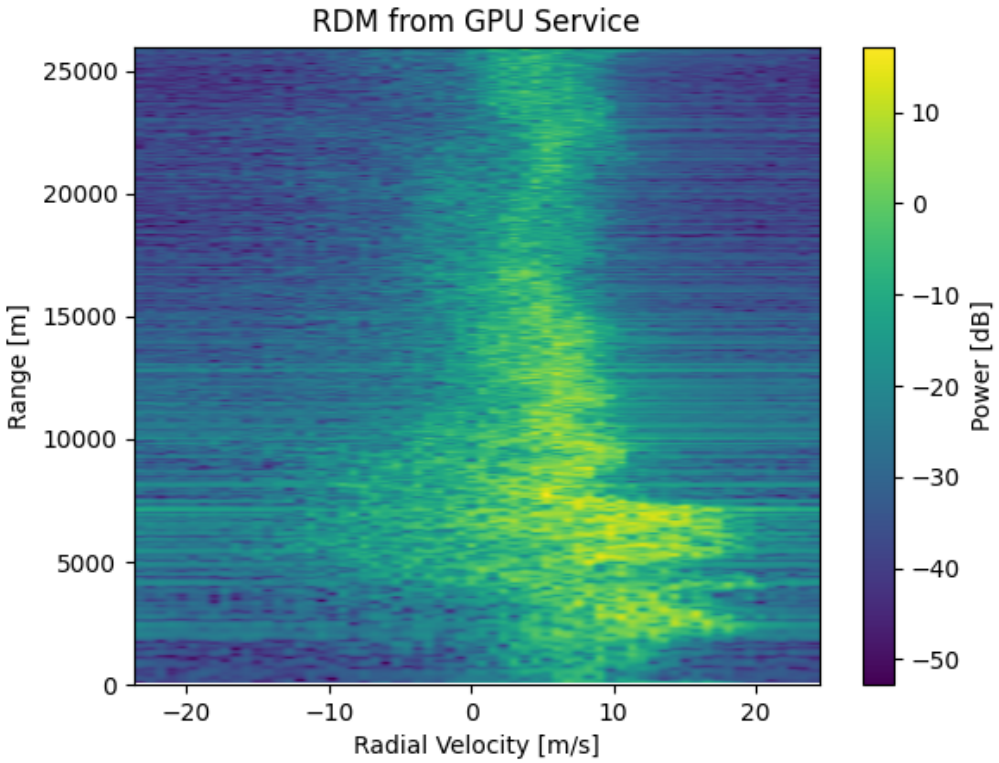

Figure 2.7: Range-Doppler map of a storm processed on the GPU-based Doppler processing service.

To illustrate the performance on high-volume datasets, figure 2.7 shows the outputs of the GPU-based Doppler processing algorithm. This figure represents the smaller dataset used to benchmark the performance of the CPU- and GPU-based algorithms. Focusing on the weather data, the FFT size is 64, corresponding to the number of pulses per CPI. For this dataset, the GPU performs 43,328 total transforms (calculated from 32 range-pulse matrices $\times$ 1,354 complex samples). During this scan the radar operated with 26 beamgroups, organized as 2 dual-pol $\times$ 13 panel subarrays, which were beamformed into 2 dual-pol beamgroups and 16 spatial beams via the CPU-based backend beamformer. This configuration results in a net increase in data volume prior to Doppler processing, necessitating an efficient parallel implementation. While Figure 2.7 displays a single range-Doppler matrix, it is one of 32 maps generated per CPI for each beamgroup and spatial beam combination. Parallelization is achieved by utilizing many cores to handle relocation during the GPU-based corner-turn and batching transforms through the `cufftPlanMany` function.

Chapter 3

Data Processing System Design

Alongside the algorithms themselves, peripheral design decisions have a significant impact on the overall signal processing chain. These decisions include selecting appropriate hardware for algorithm execution, determining a software architecture for the processing chain, and identifying advantageous points for data transfer. The system design has substantial implications for how the algorithms and software are developed. The design is often binding; making substantial changes after implementation requires considerable engineering effort. For these reasons, it is essential to understand the considerations that inform the system's design.

3.1 Software Architecture

An important consideration is determining the most suitable software architecture for a given system. Selecting a software architecture is among the earliest and most consequential decisions, as altering the architecture later requires significant effort. Software architecture comprises design decisions that define the overall system structure. Careful planning of software architecture is essential, as each approach presents distinct advantages and disadvantages. This is an important tradespace to consider for radar signal processing systems.

Although there is no shortage of architectures and opinions on them, this dis-

cussion focuses on two commonly utilized architectures for radar signal processing. The first is the pipe and filter architecture, in which each service or algorithm is connected sequentially. The next is pub/sub, or publish-subscribe architecture. In this architecture, publishers broadcast messages that are routed to subscribed services. The following sections examine each architecture in detail, highlighting their respective advantages and disadvantages.

3.1.1 Pipe and Filter Architecture

This architecture consists of a basic structure in which pipes and filters form the signal processing chain. Pipes primarily serve to transmit data between filters, though they may also fulfill additional functions. Filters are the components responsible for processing the data within the architecture. A key advantage of this architecture is that, provided developers understand the expected input data format, services can be straightforward to implement and utilize. Filters function as black boxes, abstracting internal operations from the end user and thereby promoting encapsulation and modularity. This modularity facilitates research and development, enabling rapid substitution of filters to evaluate alternative algorithms. Figure 3.1 illustrates a basic pipe and filter architecture.

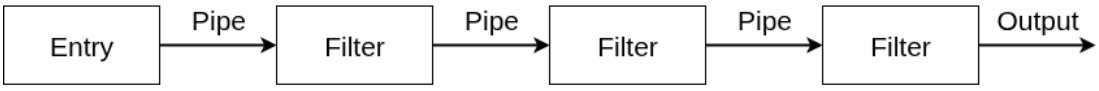

Figure 3.1: Simple pipe and filter architecture.

From a radar signal processing perspective, the pipe and filter architecture is well-suited for implementing simple processing chains. However, a notable caveat

is that the architecture can become significantly more complex when incorporating adaptive processing techniques. Within this framework, filters correspond to radar signal processing algorithms such as pulse compression, Doppler processing, and moment estimation. For example, Figure 1.3 demonstrates how the pipe and filter architecture aligns naturally with radar processing chains. The implementation of pipes can extend beyond simple data routing, enabling additional functionality. For instance, pipes may buffer or queue data when a particular processing stage operates at a slower rate. Another example involves distributing data across multiple filters in separate processing chains. A practical application of data splitting is in phased array radar systems, where the same pulses are processed for both weather moment estimation and point-target detection.

It is important to recognize complications associated with the pipe and filter architecture. The pipe and filter architecture tends to be monolithic in nature. A failure in any individual filter can halt the entire processing chain. When pipes become complex, system resources, such as CPU cores and memory, that could otherwise support algorithmic processing are instead consumed by data transfer operations. In radar signal processing, a major concern is the architectural complexity introduced by adaptive radar techniques. This complexity arises from a ripple effect, where processing steps must adapt dynamically in response to incoming data. This phenomenon, known as cascading complexity, often emerges when filters modify the data in ways that affect downstream processing. To mitigate complexity, it is essential to design filters as modular components with clearly defined roles in the processing pipeline.

3.1.2 Publish-Subscribe (Pub/Sub) Architecture

Publish-subscribe (Pub/Sub) is another software architecture commonly employed in radar signal processing systems. In this model, publishers generate and transmit messages, while subscribers receive them. The messages are a key consideration, and are organized into categories known as topics. This architecture enhances subsystem independence, as publishers and subscribers operate without direct knowledge of one another, thereby increasing system flexibility. It is also highly scalable, allowing additional publishers and subscribers to be integrated with minimal impact on existing components. Furthermore, messages can be delivered asynchronously, which can lead to improved system responsiveness if properly leveraged. Figure 3.2 illustrates a basic example of message routing in a publish-subscribe architecture.

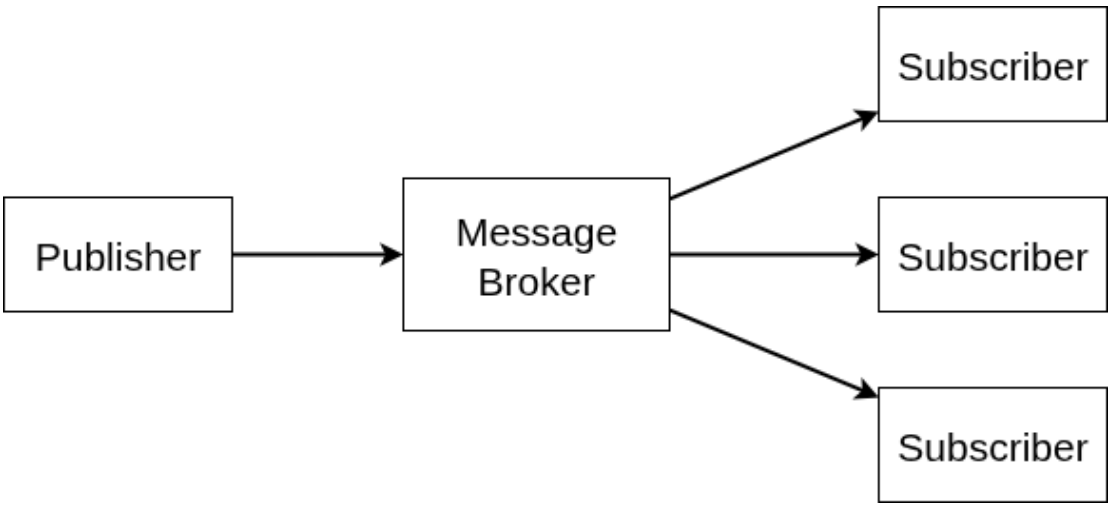

Figure 3.2: Basic publish-subscribe architecture.

From a radar signal processing perspective, algorithms exchange both data and critical metadata among themselves. Each algorithm subscribes only to messages it can process, eliminating the need for direct awareness of other algorithms. Algorithm independence is particularly valuable in research environments, as it

facilitates rapid development and experimentation. Nevertheless, each algorithm should be designed for maximum encapsulation, with clearly defined objectives. Scalability is straightforward, as publishers and subscribers can be added or removed without disrupting the overall system. The primary drawback of this architecture is its inherent complexity. Debugging can be challenging, as message origins and delivery times are often difficult to trace. This characteristic renders the architecture non-deterministic, as variations in the timing of asynchronous messages can lead to different results. Developing testing infrastructure is also more difficult, requiring modeling of multiple inputs and outputs in addition to handling asynchronous behavior. Without careful upfront planning, this architecture can quickly devolve into a complex and difficult to manage web of interactions. Finally, depending on the complexity of the message broker, significant processing resources may be consumed for message routing.

3.1.3 Conclusions

For the Horus platform, a pipe-and-filter architecture was adopted. The primary motivation for this choice was the emphasis on achieving high data throughput. By avoiding the overhead of a message broker, messages are transmitted with higher throughput and lower latency compared to a Pub/Sub architecture. The pipe and filter architecture is well-suited to this context, as most algorithms depend on the output of the preceding algorithm. Another key factor was the simplicity of the architecture. Given the large number of services to implement, simplifying the design of each block significantly reduces development time. The implementation of each algorithm is less complex in the pipe and filter architecture for a few reasons. First, we can make stronger assumptions about the input data and the required output. The data remains in order because there are no asynchronous

messages. Additionally, data routing is implicit; each service forwards its output to the next stage in the chain automatically. Second, debugging is considerably easier in the pipe and filter architecture. Because the data flows linearly, any issues that arise are deterministic. Such issues are easily repeatable in testing. Finally, algorithm testing is simplified due to the absence of asynchronous messages and the implicit nature of message routing. The pipe and filter architecture requires significantly fewer test cases compared to a Pub/Sub architecture.

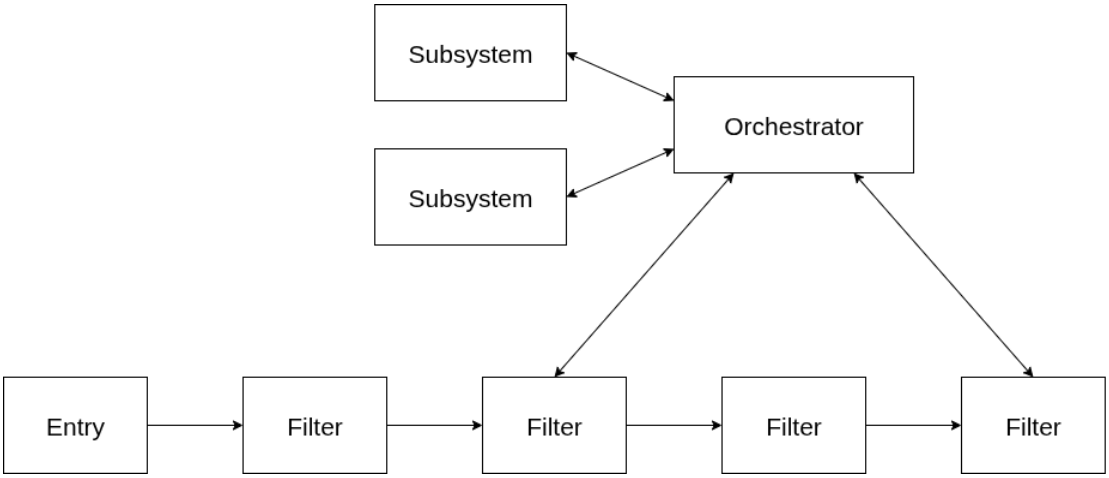

Figure 3.3: Horus Data Processing Architecture.

While the preceding discussion applies to radar IQ data, the primary data stream, certain services still require high-level information to operate. Most algorithms require only the data dimensions and other low-level metadata available directly from the packets. For example, Doppler processing illustrates this. It operates using only the PRF and the number of pulses, range gates, and beams. However, certain processes require additional high-level information, such as scan settings. One example is the backend beamformer, which requires configuration details about the array setup for beamforming. It must have access to the beamforming matrix to apply to the data, along with the azimuth and elevation angles of the beams. To facilitate this, we utilize the orchestrator. The orchestrator

comprises a set of programs and libraries that enable communication between the various components of Horus. Essentially, it functions as a Pub/Sub architecture for command, control, status, and logging. The design philosophy is to employ the pipe and filter architecture for all radar data, while delegating non-data-related tasks to the orchestrator. It is important to note that modifying high-level data may introduce asynchronous behavior; therefore, such changes should be made only when necessary. Figure 3.3 provides an overview of the architecture currently employed in Horus.

3.2 Data Transfer

An important consideration beyond the speed of the algorithms is the throughput and latency of data transfer. Data transfer occurs for several reasons within the system. One reason is inter-service communication, which may occur between services running on either the same or different processors. Since services may be implemented on heterogeneous processor types, data must occasionally be transferred across processor boundaries. In general, minimizing inter-processor transfers is desirable to reduce latency. When multiple processes or algorithms are executed sequentially on a single processor, latency overhead is significantly reduced. Nevertheless, as discussed in the heterogeneous processing section, data transfer may be justified when it enables substantial performance gains.

A common reason for data transfer is communication between services hosted on separate servers. For inter-server communication, throughput is determined by the bandwidth of the network links connecting the servers. For example, Horus utilizes 100-gigabit network links. These links can be readily upgraded to 400-gigabit connections through hardware enhancements. As multithreaded processing scales to meet computational demands, data transfer becomes increasingly

critical due to the reliance on distributed resources.

3.2.1 Data to be Moved

The Advanced Radar Research Center has developed a custom serialization library named Babeldict. Babeldict is built upon Google's FlexBuffers, a flexible serialization library. FlexBuffers differs from FlatBuffers, another widely used serialization library, in that it does not require a predefined schema. A schema defines a fixed packet format used to encode and decode data within the buffer. The absence of a schema introduces a degraded performance because of the associated overhead. However, this trade-off is justified by the increased flexibility provided by the schema-less packet format. Schema-less buffers are particularly advantageous for research and development on the Horus platform. FlexBuffers eliminates the need for parsing and memory copying, as data can be accessed directly without intermediate structures. The Babeldict library enables efficient transport and storage of free-form radar data.

In summary, Babeldicts offer fast transmission and read performance, but require relatively more time for creation. Babeldicts are immutable; therefore, any modification requires the creation of a new instance. Transfer efficiency was a core design goal of Babeldict, making it well-suited for the high-volume data movement required by the system.

In general, data transfer should be managed by a dedicated thread, separate from the signal processing algorithms. Isolating data transfer into its own thread is beneficial, as the transfer logic remains largely consistent across services. Offloading data movement from the processing threads enhances throughput by allowing algorithms to focus exclusively on computation. Managing data transfer independently of core algorithm logic also accelerates development and reduces

the likelihood of implementation-specific bugs.

3.2.2 Transferring Data Within a Processor

On the same server, data is typically transferred by referencing memory internal to the system. The implementation details vary slightly depending on the processor type. For CPU-based processors, pointers to RAM can be passed between algorithms. These pointers can be passed through queues to ensure data arrives in order and only when the receiving algorithm is ready. Transmission Control Protocol (TCP) ports are often used for inter-process communication.

3.2.3 Data Changing Operations

The shape of the data may vary across different stages of radar signal processing. This variation must be considered to minimize data transfer whenever possible. Reducing data volume prior to transfer is beneficial, as moving large amounts of data introduces latency. Beamforming is one such example. Beamforming can significantly reduce data volume during algorithm execution, which directly impacts throughput and resource utilization. In an all-digital phased array radar system, beamforming reduces data volume by a factor equal to the ratio of the number of beams to the number of channels. For example, using four spatial beams and two polarizations results in a total of eight beams. To perform this operation, data from every array element must be included at the input. On the Horus platform, this corresponds to 1,664 channels, derived from 13 panels with 64 dual-polarization channels per panel. In this example the data rate is reduced to 0.48% of the original value.

Controlling the data rate effectively is crucial for radar signal processing. One operation specifically designed to manage data size and throughput is digital down

conversion (DDC). This process operates on the sampled signal and directly reduces, or decimates, it to decrease the data volume. Decimation involves two steps: first, filtering out high-frequency components of the signal; second, downsampling by selecting every N-th sample. This approach can be advantageous in scenarios such as reducing throughput, minimizing stored data, or expanding range gates for weather-related processing. Another example of data transformation occurs during detection, where only detected targets are passed for further processing. Transforming the full range-Doppler matrix into a list of detections at specific ranges and velocities results in substantial data reduction. While this data reduction decreases transfer requirements, it does not inherently reduce processing time. Even with low data rates, the complexity of tracking algorithms mean they can still be a bottleneck in the radar system. For pulse compression and Doppler filtering, the data size usually remains the same. Upsampling is also possible, though it is rarely employed, as it typically does not enhance radar performance and comes at the cost of more data transfer.

3.2.4 PCIe Throughput

Radar signal processing chains have historically been constrained by memory bandwidth [3]. The current Horus hardware connects to GPUs via a PCIe 4.0 x16 interface. PCIe 4.0 has a theoretical speed of 16 GT/s per lane. This metric is expressed in gigatransfers per second because PCIe uses 128b/130b line encoding: for every 128 bits of data, 2 bits are overhead, resulting in an effective throughput of 15.75 Gb/s, or approximately 1.97 GB/s, per lane. Because an x16 link consists of 16 lanes operating in parallel, the total theoretical one-direction bandwidth is 31.52 GB/s. The evolution of PCIe technology is summarized in Table 3.1.

Table 3.1: Table of per-lane PCIe throughput.

Version	Year	Encoding	Transfer rate	Throughput
1.0	2003	8b/10b	2.5 GT/s	0.250 GB/s
2.0	2007	8b/10b	5.0 GT/s	0.500 GB/s
3.0	2010	128b/130b	8.0 GT/s	0.985 GB/s
4.0	2017	128b/130b	16.0 GT/s	1.969 GB/s
5.0	2019	128b/130b	32.0 GT/s	3.938 GB/s
6.0	2022 [1]	242B/256B	64.0 GT/s	7.563 GB/s
7.0	2025 [2]	242B/256B	128.0 GT/s	15.125 GB/s
8.0	2028 [3]	242B/256B	256.0 GT/s	30.250 GB/s

The specifications using 242B/256B encoding employ four-level pulse-amplitude modulation, which enables higher throughput at the cost of a higher bit-error rate. This increased bit-error rate is mitigated through low-latency forward-error correction, which uses 6 of the additional 14 bytes. The remaining bytes are used for general header information.

$$N_b = \frac{B}{2 f_s S}, \tag{3.1}$$

Using the beam-bandwidth product, we can calculate the number of beams that can be transferred to the GPU at different sampling frequencies. Here, N_b represents the number of beams, B is the available GPU memory bandwidth, f_s is the sampling frequency, and S is the number of bytes per sample. The expression has the divide by two because the PCIe interface must transfer data both into and out of the GPU. In this work, S = 8bytes, representing a complex 32-bit floating-point sample. Using the theoretical PCIe 4.0 x16 bandwidth of 31.52 GB/s, we can solve for the number of beams that can be transferred at various DDC sampling frequencies.

[1] First implemented in 2025.
[2] Specification released in 2025.
[3] Planned.

Table 3.2: Number of beams from sampling frequency for PCIe 4.0 x16.

Sampling frequency	Number of beams
125.000 MS/s	15.76
62.500 MS/s	31.52
31.250 MS/s	63.04
15.625 MS/s	126.08

Table 3.2 shows the number of beams corresponding to each sampling frequency. These values represent an acceptable number of beams to process at these sampling frequencies. Horus most often uses fewer than 32 beams at lower sampling frequencies for weather applications. Further work is needed to measure the actual throughput, as it is expected to be lower than the theoretical maximum. As transfer speeds increase, as shown in Table 3.1, heterogeneous computing over PCIe becomes increasingly advantageous by improving throughput and reducing latency. If sufficient bandwidth is available, it may also be possible to run multiple concurrent processes on the GPU.

Chapter 4

Conclusion

4.1 Summary

In brief, new strategies must be devised to keep up with the volume of data produced by all-digital phased-array radars. These strategies include heterogeneous computing, parallel processing, and effective resource management. Heterogeneous computing requires a deep understanding of both the algorithms being implemented and the characteristics of different high-performance processor architectures. Parallel processing is well suited to the challenges of all-digital array processing, as processing must be performed for each beam group and individual beam. Resource management is critical, as memory and compute cores are limited and must be allocated to achieve both low latency and high throughput.

These three topics are tightly interconnected: heterogeneous computing is most effective for embarrassingly parallel workloads, it introduces additional resources that must be managed, and parallel processing further increases the utilization of available compute resources. Targeting algorithms to specific processors is highly advantageous but adds complexity to the overall system. Given the importance of low-latency and high-throughput processing in all-digital phased-array applications, this added complexity is often a worthwhile cost. With continued

research and development, this cost will diminish, enabling a broader range of applications to benefit from this approach.

The exploration of the tradespace of heterogeneous data processing for all-digital phased array radar led to innovation in both corner-turn and Doppler processing. First, the challenges of data generation in all-digital phased array systems are analyzed, particularly the large throughput requirements imposed by multiple simultaneous receive beams. Implementing heterogeneous solutions requires a deep understanding of processing architectures as well as radar signal processing algorithms. Using this knowledge, key algorithms were developed, including Doppler-processing implementations for both the CPU and GPU. The corner-turn implementation for the GPU-based Doppler processing algorithm proved particularly interesting. Leveraging the architecture of the GPU allowed the corner-turn operation to be executed with exceptional efficiency. This efficiency was achieved while simultaneously copying the data into the device buffer required for the Doppler processing FFT. The observed speed-up is strongly dependent on dataset size; larger datasets are processed more efficiently on the GPU.

Beyond the performance of individual algorithms, examining the broader system design is also crucial. Different software architectures are compared, and the rationale for selecting a modified pipe-and-filter architecture is presented. Finally, data transfer is examined, as it often forms a bottleneck in radar signal processing and can significantly influence system-design decisions. These design decisions are incorporated into the Horus data processing system, enabling real-time product generation.

4.2 Contributions, Limitations, and Broader Impacts

This thesis provides several key contributions to all-digital phased array data processing. First, it presents a complete characterization of the heterogeneous tradespace for the Horus system, including a detailed comparison of CPU, GPU, and FPGA roles within the processing chain. Second, radar signal processing algorithms were developed, optimized, and benchmarked across heterogeneous architectures. These include CPU-based Doppler processing using the FFTW guru interface, a highly efficient GPU corner-turn that approaches device copy performance, and GPU-based Doppler processing using cuFFT. Together, these algorithm implementations demonstrate performance that can run in real-time and quantify performance for different architectures for different dataset sizes. Finally, system-level insights such as software architecture for radar signal processing, hardware architecture design for signal processing, and data transfer and its relation to beam-bandwidth product help inform future digital radar architectures.

The GPU-based corner-turn is particularly well-suited for high-bandwidth architectures, such as all-digital phased arrays and SAR, where data volume often exceeds traditional processing limits. While integrated here to enable real-time Doppler processing, this kernel is valuable to any processing stage requiring data reorganization. GPU acceleration of Doppler processing and other algorithms is critical for closing the link between high-throughput sampling and real-time decision making. These stages represent the primary computational bottleneck on the CPU-based algorithms because of the substantial processing and strided memory patterns. Consequently, the tile-based corner-turn is a critical component for systems that require high-throughput, real-time radar signal processing.

This work also has several limitations. GPU acceleration is effective only for

sufficiently large datasets, as smaller workloads fail to amortize memory-transfer and kernel-launch overheads. Future work regarding PCIe throughput should evaluate RDMA-based or pinned memory approaches as these algorithms have a slower than expected memory throughput. PCIe bandwidth remains a fundamental bottleneck, and heterogeneous designs must carefully account for the cost of moving data between processors. Transferring data from the aperture to the backend processors is one example of this limitation and can constrain system performance.

Broader impacts extend beyond the Horus system. The design patterns, heterogeneous strategies, and data transfer analyses developed here generalize to other all-digital phased arrays where similar throughput and latency constraints arise. As next-generation digital arrays continue to increase channel counts, spatial beams, and sampling rates, the techniques demonstrated in this thesis—including GPU acceleration of highly parallel operations, containerized algorithm deployment, and explicit modeling of data transfer bottlenecks—provide a foundation for scalable, real-time radar processing in future operational systems.

4.3 Implemented Algorithms

The current system relies on a combination of FPGAs and CPUs to process the received data. After the data are sampled in the transceivers, they are routed to the FPGAs for beamforming. FPGAs are well suited for this operation because it requires low latency and must sustain the full throughput of the array. Another advantage of implementing beamforming on the FPGAs is that it drastically reduces the data rate. This reduced data rate facilitates transferring the data off the array for back-end processing. The FPGAs perform beamforming systolically, meaning each FPGA is connected in sequence to the next. A data packet is passed

from the first FPGA in the chain through each subsequent device until it reaches the last. As the packet moves through the chain, each FPGA contributes its own data to the accumulated sum. The FPGA-to-FPGA interconnect must provide low latency and high throughput for this architecture to function effectively. The final beamformed result is then transferred off the array to the backend servers, where it is either further processed or stored.

Many algorithms have been implemented for execution on CPUs. All algorithms are written to operate with an arbitrary number of spatial beams, beam groups, pulses, and range samples. They use multithreading where appropriate to achieve higher throughput through parallel processing. The first algorithm in the back end is the FPGA translator. It converts the lower-level data packet into higher-level fields. The most notable conversion occurs in the data itself, where 16-bit floating-point samples are expanded to 32-bit floating-point samples. At this stage, the data are also organized into the aforementioned dimensions. Other examples of conversions include translating clock-cycle counts into timestamps, translating digital down-converter settings, and interpreting waveform-generator settings.

The next algorithm in the processing chain is the pulse compressor. Pulse compression requires access to the transmitted waveform; however, when the algorithm was first developed, there were issues with the digital down-conversion of the transmitted waveform. This limitation required the introduction of an additional service, the CPI meta injector, named after the CPI meta packets it produces. A CPI meta packet is produced at the beginning of each CPI and contains the transmitted waveform along with other metadata associated with that CPI. This algorithm, which is now defunct, injected the correct waveform into the data stream after the FPGA translator and before the pulse compressor. It operated by

selecting the appropriate waveform from a table of stored options that varied by pulse length and bandwidth. The pulse compressor is based on RadarKit's pulse compression implementation. RadarKit is an open-source C library for radar signal processing in weather applications, mainly developed by the ARRC's Boonleng Cheong. RadarKit's pulse compressor achieves high performance through the use of multithreading, Single Instruction/Multiple Data (SIMD) operations, and the FFTW library. SIMD enables the same operation to be applied simultaneously to multiple samples, thereby increasing throughput. A popular SIMD implementation, Advanced Vector Extensions (AVX), can perform the same operation on eight 32-bit floating-point values or four 64-bit floating-point values simultaneously. Multithreading enables parallel processing by dividing each dataset into pulses that can be processed concurrently. Because pulse compression must be applied to every spatial beam, beam group, and pulse, this parallelization significantly improves overall throughput and latency. RadarKit's pulse-compression implementation also supports Progressive Pulse Compression (PPC), which can help recover signals that lie within the radar's blind range [32].

The next process batches an entire CPI for subsequent stages of processing. This service is referred to as the CPI packer. The CPI packer aggregates incoming packets, each containing a single pulse, into a complete CPI. Pulses vary in size based on the number of beam groups, spatial beams, and range gates. This is accomplished by copying pulses into a large pre-allocated buffer and storing that buffer in a Babeldict. The buffer is allocated using information from the first pulse of the CPI. Additional important metadata are also extracted from the first pulse and stored in the same Babeldict for use in later processing stages. Interestingly, this block was once a performance bottleneck in the Horus data-processing chain. It was optimized by increasing the size of the memory copies, thereby reducing

overhead. This again illustrates that memory operations frequently limit radar signal-processing performance and therefore must be carefully optimized.

The next process in the chain is Doppler processing. This algorithm operates across pulses and can accommodate any number of spatial beams, beam groups, range samples, and pulses. In principle, the algorithm is straightforward: it performs an FFT across each spatial beam, beam group, and range sample. The FFT size is set to the number of pulses in the CPI, which is typically small, on the order of 64 pulses. However, the number of FFTs that must be executed is substantial, dominated by the product of the number of range samples and the number of beams. The large number of required transforms, combined with the need to reformat the data using a corner-turn, motivated the development of a GPU-based corner-turn and Doppler processing algorithm, as outlined in Sections 2.3.2 and 2.3.3. An in-depth examination of CPU-based Doppler processing is provided in Section 2.3.1.

The backend beamformer was developed to beamform beam groups, such as panel-level subarrays, in real time. This capability became necessary as subarrays were incorporated into some VCPs. The algorithm was initially implemented in Python for rapid prototyping and, once validated, was rewritten in C++ for improved performance. The beamforming operation consists of a large matrix multiplication between the beam table specified by the VCP and the dataset containing all subarrays. Because the VCP must supply the beam table to the algorithm, a dedicated communication path was created for this purpose, as described in Section 3.1.3. Beamforming performance was accelerated using Intel's Math Kernel Library (MKL) and its integrated multithreading support for linear algebra operations. This optimization enabled the C++ implementation to run 1.8× faster while using only 0.125× as many CPU cores. Interestingly, this algo-

rithm, along with the CPI packer and the pulse compressor, is order independent, meaning these stages can be arranged in any sequence. Below, Fig. 4.1 illustrates multiple elevation beams to image a storm.

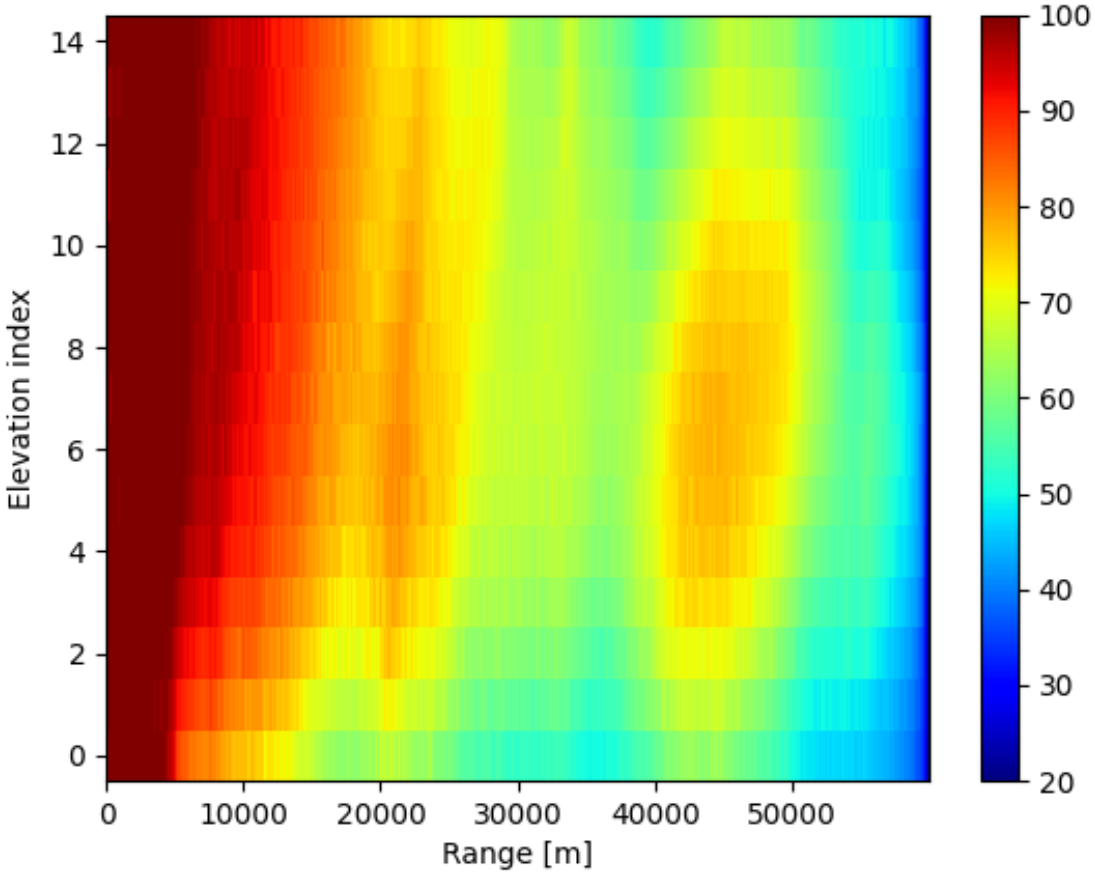

Figure 4.1: Returned power from a storm.

The product generator is a process used for real-time weather processing. This service also relies heavily on RadarKit. It takes in beamformed and pulse compressed data, organizes them into rays or CPIs, and forwards them to RadarKit to generate all dual-polarization radar products. The product generator is essential for producing real-time weather products and is highly optimized through the use of multiple CPU cores and extensive SIMD operations. Recently, Gaussian Model Adaptive Processing (GMAP) was implemented in RadarKit for use within the

product generator. This algorithm filters clutter from the received signal without degrading the weather returns. This operation requires an FFT across pulses, similarly to Doppler processing. Figure 4.2 shows dual-pol radar moments and Fig. 4.3 shows the effectiveness of GMAP on clutter.

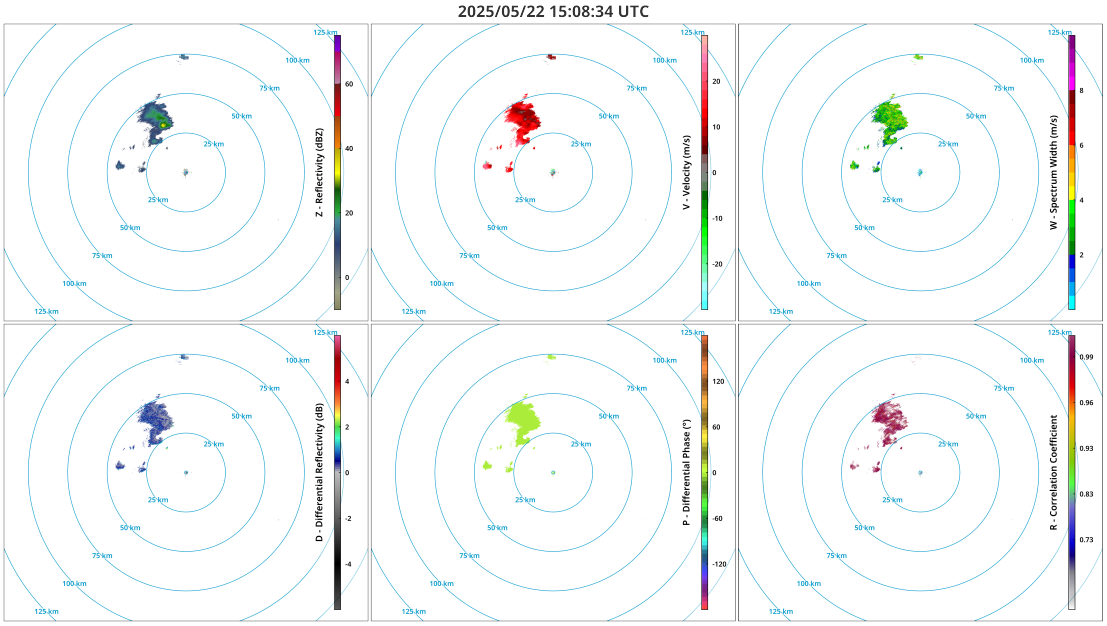

Figure 4.2: Weather products of a storm.

There are also services that act as the entry and egress points for the processing system. One such entry point is the array interface, which reads data transferred to the server via DMA. Another entry point service is the Babel file player, which can read saved Babeldicts from past deployments or tests. This service is particularly useful for measuring service throughput. It first loads all replayed data into memory, eliminating the slower transfer rate associated with disk storage. There are two primary egress points: outputting to a GUI and a service to save data to disk. The GUI service connects to a Python interface to provide real-time data visualization for the user.

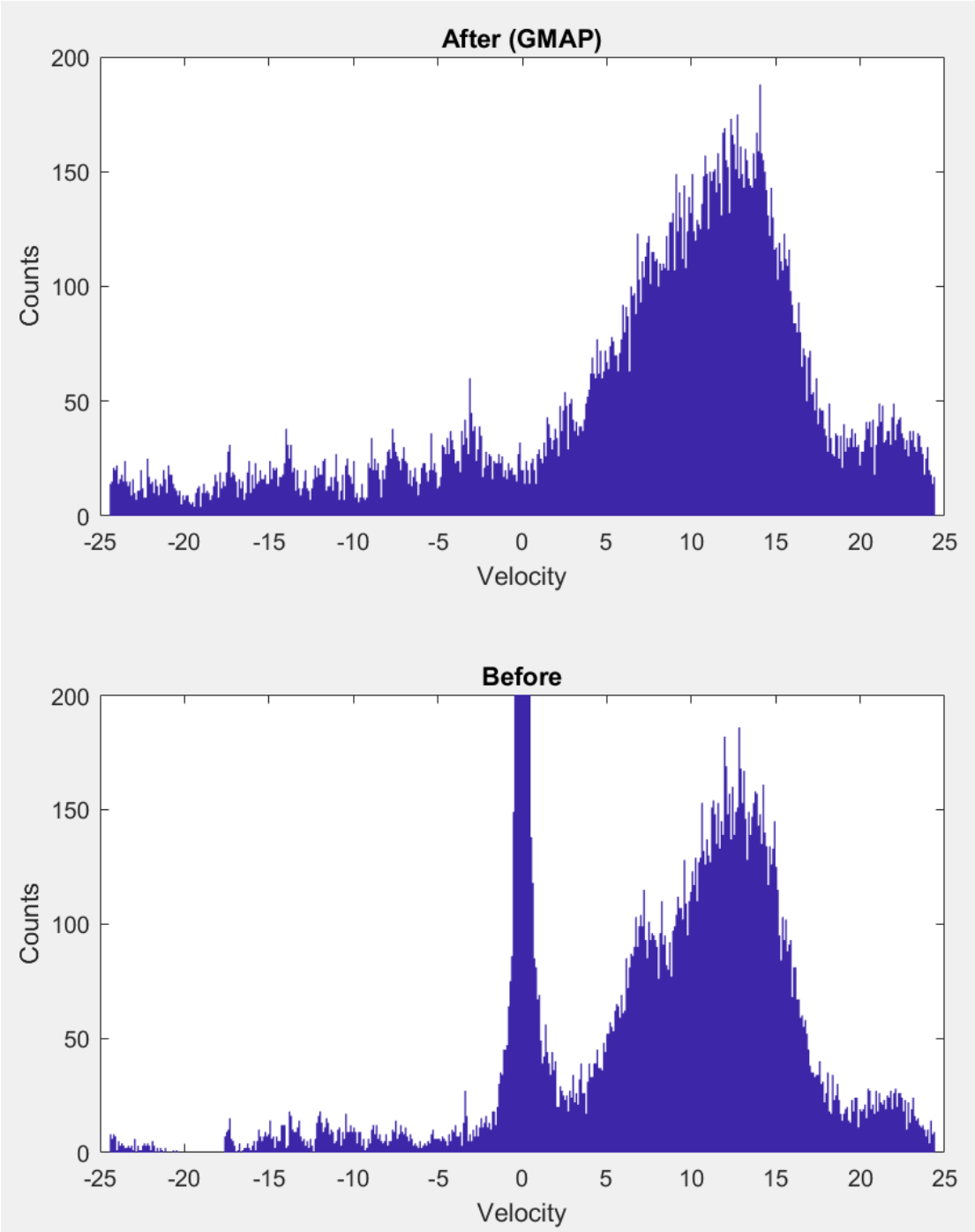

Figure 4.3: After and before GMAP of a storm.

4.4 Ongoing Development

As the ARRC continues research and development in the area of all-digital phased array radars, we look to improve many aspects our systems. One improvement is exploring the use of heterogeneous computing, specifically combining CPUs, GPUs, and FPGAs. These compute architectures are suited very well to radar signal processing. The use of different compute architectures to their strengths will allow us to increase real-time throughput, giving us more information. This information can be used to make decisions on how to use the amazing flexibility of all-digital phased array radar.

The goal I want to build towards is closing the loop. This idea would allow us to increase resolution on severe storms, allowing for advanced warning accuracy and invaluable data for research. Temporal resolution could be increased by steering more beams to the target, range sampling by increasing the bandwidth for promising beams, and azimuth resolution by focusing beams tightly in the area. Increasing these resolutions puts considerable strain on throughput, increasing bandwidth or the number of spatial beams directly increases throughput needs. For that reason, increased resolutions can not be used in every direction but must be focused on the most important targets available. The flexibility of the all-digital phased array radar depends on real-time processing.

Some near term goals are to test and optimize the weather processing chain for real time processing this spring. This goal is very close with the help of RadarKit and a lot of work from my team. Intensive tests in the field are needed to be able to rely on the processing chain during high-stakes deployments. Being able to use the weather processing chain in real time will be a huge boon to the system which has primarily relied on post-processing.

Farther term goals to increase throughput would include moving more services

to the GPU. A prime target for this would be to implement backend beamforming and Doppler processing on the GPU. These tasks are very parallelizable. Beamforming is directly a matrix multiplication which is highly parallelizable as the resulting matrix elements are calculated independently. Doppler processing, especially for multiple beams, is highly parallelizable as seen in sections 2.3.1 and 2.3.3. The FFT itself is parallelizable, which will help accelerate the longer range-gate length transforms. All of these would be excellent algorithms to target to the GPU architecture, and can be placed next to each other in the processing chain. Being next to each other, the cost of data transfer to and from the GPU is lessened by the acceleration of computation of all the algorithms.

There are many more improvements to be made to the processing chain. Optimization of memory operations on CPUs, optimizing packet creation, implementing new algorithms, throughput aware processing, and scalable on-array processing are more areas of interest. This field is in an exciting time for research and development.

References

[1] S. H. Talisa, K. W. O'Haver, T. M. Comberiate, M. D. Sharp, and O. F. Somerlock, "Benefits of digital phased array radars," *Proceedings of the IEEE*, vol. 104, no. 3, pp. 530–543, 2016. DOI: 10.1109/JPROC.2016.2515842.

[2] K. Zaknoun, "Real-time, GPU-accelerated processing of digital radar signal data," Master's Thesis, Department of Computing, M.S. thesis, University of Turku, Turku, Finland, 2024. [Online]. Available: https://www.utupub.fi/bitstream/handle/10024/178763/Kimi_Zaknoun_Real-Time_GPU-accelerated_Processing_of_Digital_Radar_Signal_Data.pdf.

[3] M. Rupniewski, G. Mazurek, J. Gambrych, M. Nalecz, and R. Karolewski, "A real-time embedded heterogeneous GPU/FPGA parallel system for radar signal processing," in *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress*, IEEE, 2016, pp. 1189–1197.

[4] R. S. Perdana, B. Sitohang, and A. B. Suksmono, "A survey of graphics processing unit (GPU) utilization for radar signal and data processing system," in *2017 6th International Conference on Electrical Engineering and Informatics (ICEEI)*, 2017, pp. 1–6. DOI: 10.1109/ICEEI.2017.8312430.

[5] C. J. Venter, H. Grobler, and K. A. AlMalki, "Implementation of the CA-CFAR algorithm for pulsed-doppler radar on a GPU architecture," in *2011 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, IEEE, 2011, pp. 1–6. DOI: 10.1109/AEECT.2011.6132514. [Online]. Available: https://ieeexplore.ieee.org/document/6132514.

[6] T. Long, Y. Li, W. Zhang, *et al.*, *Wideband Radar*. Springer, 2022, ISBN: 978-981-19-7561-5. DOI: 10.1007/978-981-19-7561-5. [Online]. Available: https://link.springer.com/book/10.1007/978-981-19-7561-5.

[7] J. Yoon *et al.*, "Real-time radar signal processing for automotive applications using heterogeneous architectures," in *Proceedings of the IEEE Radar Conference*, 2023, pp. 1–6. DOI: 10.1109/RadarConf.2023.xxxxx.

[8] M. Khalid *et al.*, "Design of radar processing units for real-time applications," in *Proceedings of the International Radar Symposium*, 2017, pp. 1–5. DOI: 10.1109/IRS.2017.xxxxx.

[9] H. Izumi, K. Sasaki, K. Nakajima, and H. Sato, "An efficient technique for corner-turn in SAR image reconstruction by improving cache access," in *Proceedings 16th International Parallel and Distributed Processing Symposium*, 2002, 6 pp–. DOI: 10.1109/IPDPS.2002.1015471.

[10] P. Meisl, M. Ito, and I. Cumming, "Parallel synthetic aperture radar processing on workstation networks," in *Proceedings of International Conference on Parallel Processing*, 1996, pp. 716–723. DOI: 10.1109/IPPS.1996.508137.

[11] B. Gigleux, G. Beaugendre, E. Chaumette, and F. Vincent, "Solving the corner-turn problem with an improved radar range-doppler detection architecture," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 61, no. 5, pp. 15 047–15 056, 2025. DOI: 10.1109/TAES.2025.3572873.

[12] C. J. Cochrane, K. B. Cooper, S. L. Durden, R. Rodriguez Monje, and R. J. Dengler, "An FPGA-based signal processor for FMCW doppler radar and spectroscopy," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 58, no. 8, pp. 5552–5563, 2020. DOI: 10.1109/TGRS.2020.2967212.

[13] L. Newmeyer, D. Wilde, B. Nelson, and M. Wirthlin, "Efficient processing of phased array radar in sense and avoid application using heterogeneous computing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, IEEE, 2016, pp. 1–8.

[14] A. Shan, "Heterogeneous processing: A strategy for augmenting Moore's law," *Linux Journal*, vol. 2006, no. 142, p. 7, 2006.

[15] J. S. Herd and M. D. Conway, "The evolution to modern phased array architectures," *Proceedings of the IEEE*, vol. 104, no. 3, pp. 519–529, 2015.

[16] W. H. Weedon and R. D. Nunes, "Low-cost wideband digital receiver/exciter (drex) technology enabling next-generation all-digital phased arrays," in *2016 IEEE International Symposium on Phased Array Systems and Technology (PAST)*, IEEE, 2016, pp. 1–5.

[17] W. Chappell and C. Fulton, "Digital array radar panel development," in *2010 IEEE International Symposium on Phased Array Systems and Technology*, IEEE, 2010, pp. 50–60.

[18] J. A. Haimerl, B. Hudson, G. P. Fonder, and D. K. Lee, "Overview of the large digital arrays of the space fence radar," in *2016 IEEE International Symposium on Phased Array Systems and Technology (PAST)*, IEEE, 2016, pp. 1–8.

[19] K. A. Steiner and M. B. Yeary, "A 1.6-GHz sub-Nyquist-sampled wideband beamformer on an RFSoC," *IEEE Transactions on Radar Systems*, vol. 1, pp. 308–317, 2023. DOI: 10.1109/TRS.2023.3294796.

[20] T. Hoffmann, C. Fulton, M. Yeary, *et al.*, "IMPACT—A common building block to enable next generation radar arrays," in *2016 IEEE Radar Conference (RadarConf)*, IEEE, 2016, pp. 1–4.

[21] C Fulton, R Palmer, M Yeary, *et al.*, "Horus: A testbed for fully digital phased array radars," *Microwave Journal*, vol. 63, no. 1, pp. 20–36, 2020.

[22] R. D. Palmer, M. B. Yeary, D. Schvartzman, *et al.*, "Horus–a fully digital polarimetric phased array radar for next-generation weather observations," *IEEE Transactions on Radar Systems*, 2023.

[23] M Yeary, R Palmer, C Fulton, J Salazar, and H Sigmarsson, "Update on an S-band all-digital mobile phased array radar," in *2021 IEEE Radar Conference (RadarConf21)*, IEEE, 2021, pp. 1–4.

[24] A. Leist and A. Gilman, "A comparative analysis of parallel programming models for C++," *Proceedings of ICCGI*, pp. 121–127, 2014.

[25] D. H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on computers*, vol. 100, no. 12, pp. 1145–1155, 1975.

[26] A. Almurayh *et al.*, "Improved matrix multiplication by changing loop order," *Mobile Information Systems*, vol. 2022, 2022.

[27] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008. DOI: 10.1109/MC.2008.209.

[28] U. Verner, A. Mendelson, and A. Schuster, "Extending Amdahl's law for multicores with turbo boost," *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 30–33, 2017. DOI: 10.1109/LCA.2015.2512982.

[29] NVIDIA Corporation, *CUDA C programming guide*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Accessed: 2026-01-19, 2026.

[30] D. Ibrahim and A. Davies, "The evolution of digital signal processors," in *2019 6th IEEE History of Electrotechnology Conference (HISTELCON)*, 2019, pp. 25–29. DOI: 10.1109/HISTELCON47851.2019.9040130.

[31] S. Sadeghi, "Classifying FPGA technology in digital signal processing: A review," *International Journal of Engineering & Technology Sciences*, vol. 10, pp. 1–10, 2024, Article ID: IJETS-2408302112908. [Online]. Available: https://jms.procedia.org/archive/IJETS_29/procedia_2024_2024_ijets-2408302112908.pdf.

[32] C. M. S. Aquino, B. Cheong, and R. D. Palmer, "Progressive pulse compression: A novel technique for blind range recovery for solid-state radars," *Journal of Atmospheric and Oceanic Technology*, vol. 38, no. 9, pp. 1599–1611, 2021. DOI: 10.1175/JTECH-D-20-0164.1.

Appendix A

List of Acronyms and Abbreviations

ADC	Analog-to-Digital Converter
AESA	Active Electronically Scanned Array
ALU	Arithmetic logic unit
API	Application Programming Interface
ARRC	Advanced Radar Research Center
ASIC	Application-Specific Integrated Circuit
AVX	Advanced Vector Extensions
COTS	Commercial Off-The-Shelf
CPI	Coherent Processing Interval
CPU	Central Processing Unit
DDC	Digital Down-Converter
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FLOPS	Floating Point Operations Per Second
FMCW	Frequency Modulated Continuous Wave
GMAP	Gaussian Model Adaptive Processing
GPU	Graphics Processing Unit

GUI	Graphical User Interface
HDL	Hardware Description Language
LFM	Linear Frequency Modulated
MKL	Math Kernel Library
$MMIC$	Monolithic Microwave Integrated Circuit
PAR	Phased Array Radar
PFA	Probability of False Alarm
PPC	Progressive Pulse Compression
PRF	Pulse Repetition Frequency
$PubSub$	Publish-Subscribe
RDM	Range-Doppler Map
$RDMA$	Remote Direct Memory Access
RPU	Radar Processing Unit
$SIMD$	Single Instruction, Multiple Data
SM	Streaming Multiprocessor
SNR	Signal-to-Noise Ratio
$SWaP$	Size, Weight, and Power
TCP	Transmission Control Protocol
VCP	Volume Coverage Pattern