Real-Time, GPU-accelerated Processing of Digital Radar Signal Data

Master of Science in Technology
Thesis
University of Turku
Department of Computing
Software Engineering
2024
Kimi Zaknoun

Supervisors:
Prof. Ville Leppänen
Dr. Panu Lähdekorpi

UNIVERSITY OF TURKU
Department of Computing

Kimi Zaknoun: Real-Time, GPU-accelerated Processing of Digital Radar Signal Data

Master of Science in Technology Thesis, 63 p.
Software Engineering
June 2024

This thesis investigates the utilization of Graphics Processing Units (GPUs) for real-time processing of digital radar signal data. The motivation is to leverage the massive parallel computing capability of the GPU for computationally intensive tasks involved in digital radar signal processing.

A real-time, GPU-accelerated radar signal processing solution is implemented on a heterogeneous platform by utilizing CUDA and various NVIDIA libraries. The solution is tested using high bitrate digitized wideband signal data. The solution's performance is then evaluated through a series of experiments focusing on key data processing stages.

The experimental evaluation demonstrates that the GPU significantly enhances data processing speeds in the preprocessing and FFT stages by achieving real-time performance for wideband radar signals. However, a bottleneck in the CPU-side post-processing is also identified, indicating the need for further optimization or offloading it to the GPU. The findings suggest that while GPUs offer substantial improvements in radar signal processing, the specific task division between GPU and CPU requires careful consideration to fully exploit the potential of GPU acceleration.

This work seeks to contribute to the advancement of GPU-accelerated computing by highlighting the potential to take advantage of GPUs in real-time digital signal processing for radar applications.

Keywords: real-time, digital, radar, signal-processing, DSP, graphics processing unit, GPU, GPGPU, wideband, RF, IF, CUDA

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Research objectives 2

1.3 Thesis outline 2

2 Background and theory **4**

2.1 Graphics processing unit 4

2.2 General-purpose computing on GPUs 7

2.3 CUDA as a GPGPU programming model 7

2.3.1 Kernels 8

2.3.2 Threads, blocks, and grids 8

2.3.3 Memory hierarchy 9

2.4 Field-programmable gate array 10

2.5 Real-time digital signal processing 11

2.5.1 Sampling 13

2.5.2 Constraints 14

2.6 I/Q data 14

2.7 Fourier transform 17

2.7.1 Discrete Fourier transform 18

2.7.2 Fast Fourier transform 18

2.8 Amplitude spectrum 20
2.9 Window functions 21
2.9.1 Rectangular window 22
2.9.2 Blackman–Harris window 22
2.10 Software-defined radio 23
2.11 VITA-49 24

3 Literature review **27**
3.1 Methodology 27
3.2 Findings 30
3.2.1 Radar type 31
3.2.2 Signal processing chain 32
3.2.3 Processing platform 36
3.2.4 Performance 38
3.2.5 GPU models 42
3.3 Conclusion 43

4 Design and implementation **45**
4.1 Description 45
4.2 Software architecture 46
4.2.1 NVIDIA DOCA 47
4.2.2 Optimizations 48
4.3 Signal processing chain 48
4.3.1 Preprocessing 48
4.3.2 cuFFT 49
4.3.3 Post-processing 50
4.3.4 Maximum amplitude spectrum 51
4.4 Hardware platform 52

5 Experimental evaluation **54**

5.1 Experimental setting 54

5.1.1 Test signal characteristics 54

5.1.2 Performance tests 55

5.2 Performance 55

5.2.1 Packet reception rate 55

5.2.2 Processing time 55

5.2.3 FFTW3 comparison 56

6 Conclusion **60**

6.1 Limitations 60

6.2 Future work 61

6.3 Summary 61

References **64**

List of Figures

2.1 Data flow between the CPU and a discrete GPU at a high-level. Figure adapted from [2]. 5

2.2 Comparison between CPU and GPU architectures at a high-level. Figure adapted from [5]. 6

2.3 Illustration of a CUDA C program's compilation process. Figure adapted from [6]. 8

2.4 A typical FPGA architecture at a high-level. Figure adapted from [9]. 11

2.5 I/Q signal representation as a helix. Figure taken from [16]. 16

2.6 Complex signal representation as a phasor diagram. 17

2.7 Data flow diagram for a $N = 8$ decimation-in-time radix-2 DFT decomposition into two $N/2$ DFT computations and their combination. Figure adapted from [13]. 20

2.8 An example plot of the Blackman–Harris window function. 23

2.9 VITA-49 template for the signal data packet, adapted from [24]. . . . 26

3.1 Literature review process used in this thesis. 28

4.1 The solution's data flow in stages illustrated at a high-level. 47

4.2 Hardware-level set-up for the system. 52

5.1 FFTW3 single-threaded performance measurements. 57

5.2 cuFFT performance. 58

5.3 Batched FFTW3 performance measurements. 59

List of Tables

3.1 Total number of articles retrieved from each database. 30

3.3 List of radar types implemented in the papers. 32

3.4 Different processing platforms used in the papers. 37

3.5 Different GPU models used in the papers. 42

3.2 List of papers used in this literature review. 44

4.1 Expected data sizes for one-dimensional transforms in cuFFT. Table adapted from [43]. 50

4.2 Hardware system parameters of the receiver platform. 52

4.3 Specifications of the NVIDIA A40 GPU [45]. 53

5.1 Specifications of the test signal. 55

5.2 Processing times of the signal chain for $N = 4096$ and batch size 8160. 56

5.3 Processing time on the host side FFTW for one single-threaded C2C FFT where $N = 4096$. 57

5.4 Processing time for single-threaded C2C FFT with batch size 8160 and $N = 4096$. 58

1 Introduction

1.1 Motivation

Over the past decades, graphics processing units (GPUs) have shifted from pure graphics rendering tasks to having become widely used in the acceleration of computationally demanding applications through offloading tasks from the central processing unit (CPU). Radar signal and data processing impose heavy real-time performance demands. Traditionally, digital signal processing for radars has been implemented using specialized hardware such as the digital signal processor (DSP) or the field-programmable gate-array (FPGA). FPGAs for this context are complex to design, offer low flexibility, and are relatively costly compared to GPUs. Meanwhile, many key signal processing algorithms such as the fast Fourier transform are inherently parallelizable, making them ideal to compute on parallel architectures such as the GPU. The GPU has the potential to combine the performance of the FPGA and the programming flexibility offered by high-level programming languages such as CUDA, and thus be suitable for a real-time software-defined radio (SDR) system.

This confluence of factors makes it an attractive proposition to investigate the suitability of the GPU in the context of real-time digital radar signal processing, which is what this master's thesis attempts to do. In this thesis, a solution for real-time data processing on the graphics processing unit (GPU) of wideband radar signals is developed. The solution's performance is then tested and evaluated using

digitized wideband signal data.

It is worth noting that GPUs exist in both integrated and discrete forms. In this thesis, we focus specifically on the discrete GPU and the solution is built and evaluated on a discrete graphics processing unit.

1.2 Research objectives

The primary objective of this master's thesis is to investigate the performance of real-time digital radar signal processing on a GPU by designing and developing a proof-of-concept solution and evaluating its performance. A high-end GPU (NVIDIA A40) along with NVIDIA SDKs and libraries are utilized to process high-bitrate, I/Q sampled digital signal data. Additionally, the thesis aims to assess the current adoption of GPUs in real-time digital signal processing for radars, achieved through a review of existing academic literature on the topic.

The research questions for this master's thesis are as follows:

RQ1: What is the current state of adoption of GPUs in real-time radar signal data processing?

RQ2: What is the performance of the GPU in real-time processing of wideband radar signal data?

RQ1 is answered through a literature review of previously published academic work, while RQ2 is answered through developing and testing a real-time radar signal data processing application on a high-end GPU.

1.3 Thesis outline

The rest of this thesis is organized as follows. Chapter 2 provides an in-depth overview of the theoretical and background concepts essential to understanding the thesis subject. It begins with an introduction to graphics processing units (GPUs)

and then goes into the concept of general-purpose computing on GPUs. The chapter also introduces NVIDIA's CUDA programming model. After this, real-time digital signal processing is covered. The concepts of In-phase (I) and Quadrature (Q) data in signal representation are also discussed, along with the fundamentals of Fourier transforms. Additionally, window functions are covered. Finally, the chapter examines software-defined radios (SDR) and the ANSI/VITA 49.2 standard.

Chapter 3 consists of a literature review of previous academic work in the area of real-time radar signal processing on GPUs. The findings cover areas such as the various radar types, signal processing chains, processing platforms, optimizations, and performance outcomes that have been implemented in the literature. The chapter concludes with a summary of findings and answers RQ1.

Chapter 4 covers the implementation of the solution developed in this thesis. It begins with an overview of the solution's general structure. The software architecture is discussed as well as the high-level flow of data in the signal processing chain. Then, different stages of the data processing are covered in more in detail.

Chapter 5 presents and analyzes the results from the experiments. First an explanation of the experimental setting and the characteristics of the test signal used are given. Then, the performance results are presented and analyzed.

Finally, the thesis is concluded in Chapter 6.

2 Background and theory

This chapter introduces the relevant theoretical and background information pertaining to the thesis subject.

2.1 Graphics processing unit

A *graphics processing unit* (GPU) is a specialized processor for accelerating computer graphics rendering. A graphics card is a device for wherein the graphics processing unit is the main component. GPUs exist in both integrated and in discrete forms.

In a typical architecture, the GPU is a co-processor that is connected to a 'host', which is usually a conventional central processing unit (CPU). The host (CPU) and co-processor or 'device' (GPU) communicate through a PCI express (PCIe) interface [1]. The PCI express bandwidth frequently becomes a bottleneck. Therefore, it is crucial to offload work to GPUs only when the performance gains outweigh the associated offload costs [1]. The Figure 2.1 illustrates this set-up and the data flow across PCIe between the host (CPU) and the device (GPU).

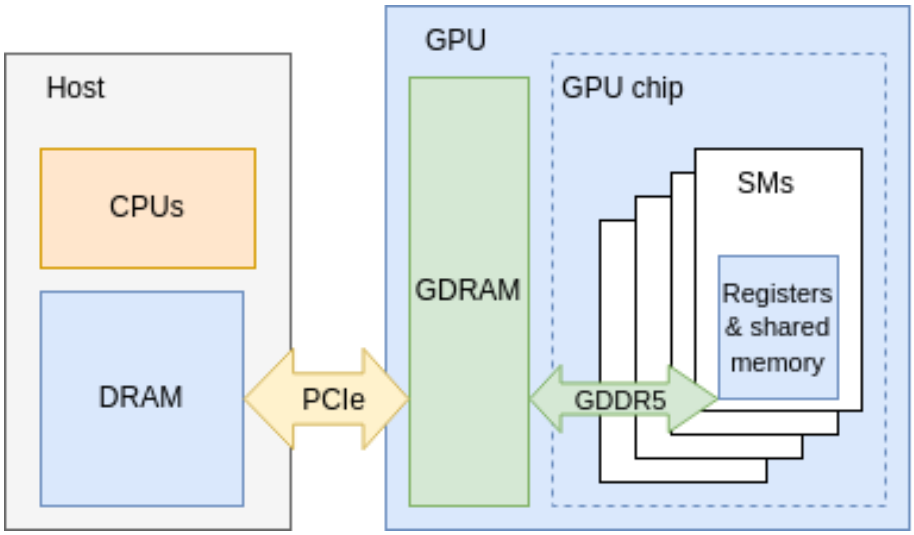

Figure 2.1: Data flow between the CPU and a discrete GPU at a high-level. Figure adapted from [2].

Architecture

GPUs tend to increase in complexity every new generation. It is necessary to gather a sufficient architectural understanding of the GPU in order to optimize software for performance [3].

The hardware architecture of a graphics processing unit (GPU) differs fundamentally from that of a central processing unit (CPU). These differences are a result of their different purposes and design philosophies [1]. The CPU architecture focuses on low-latency computations and high versatility for fast serialized execution. A CPU can efficiently execute a wide range of complex operations, but it can handle only a limited number of concurrent operations. Meanwhile, GPU architecture emphasizes high throughput via parallel execution of operations, which enables handling many more simple operations simultaneously [4]. The GPU compensates for slower individual thread performance with significantly higher overall throughput [5]. The Figure 2.2 illustrates the difference between the CPU and GPU architectures at a high level.

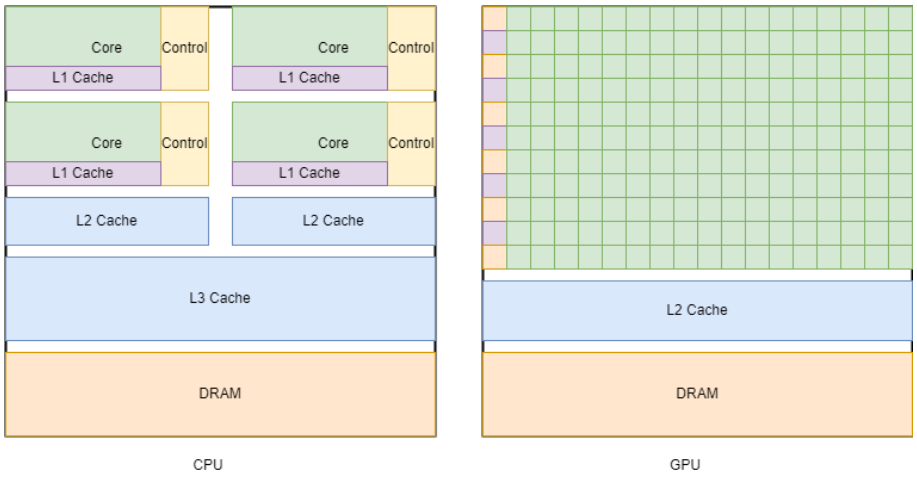

Figure 2.2: Comparison between CPU and GPU architectures at a high-level. Figure adapted from [5].

A basic GPU architecture is composed of two primary components: the memory block and the compute block. The compute block consists of caches and streaming multiprocessors (SMs), which are dedicated to performing computations. Within the SMs are some specialized units, including floating-point cores and special function units. The GPU's memory hierarchy is distributed among these components [4].

A GPU has several processor clusters (PCs), each containing multiple streaming multiprocessors (SMs). Each SM is equipped with a level-1 (L1) cache, which includes both instruction and data caches directly connected to its cores. An SM uses its dedicated L1 cache and a shared level-2 (L2) cache before accessing data from the Graphics Double Data Rate (GDDR) memory. This architecture is designed so as to mitigate memory latency issues. Unlike a CPU, a GPU operates with fewer and smaller memory cache layers because a larger proportion of its transistors are allocated to computation rather than data retrieval [4].

2.2 General-purpose computing on GPUs

General-purpose computing on graphics processing units (GPGPU) refers to the use of GPUs in computations traditionally performed by CPUs.

In recent decades, GPUs have become widely used beyond graphics rendering, e.g., in signal processing, scientific computing, deep learning, and cryptocurrency mining. Their parallel architecture makes GPUs particularly well-suited for *embarrassingly parallel* workloads. Additionally, the rise of general-purpose computing on GPUs has been facilitated by the development of programming tools, with NVIDIA's CUDA (Compute Unified Device Architecture) being the most widely adopted [6].

In terms of floating-point calculation throughput (measured in floating-point operations per second or FLOP/s), modern GPUs are about one order of magnitude more performant than modern CPUs. In terms of memory bandwidth, modern GPUs are similarly about one order of magnitude faster than modern CPUs. This is because GPUs must be able to move large amounts of data quickly in and out of Dynamic Random Access Memory (DRAM) in graphics rendering tasks [1].

2.3 CUDA as a GPGPU programming model

NVIDIA's *CUDA* (Compute Unified Device Architecture) is a parallel computing platform and C/C++ API for general-purpose computing on GPUs. CUDA programs include code that executes on a host (the CPU) and kernels that operate on one or more devices (GPUs) [1]. CUDA was initially released in 2006 and it is a proprietary and closed-source platform.

The host code is compiled using a standard compiler (e.g., GCC), while the device code is initially converted to an intermediate device language, after which it is translated into a device-specific binary code. Figure 2.3 illustrates the CUDA program compilation process.

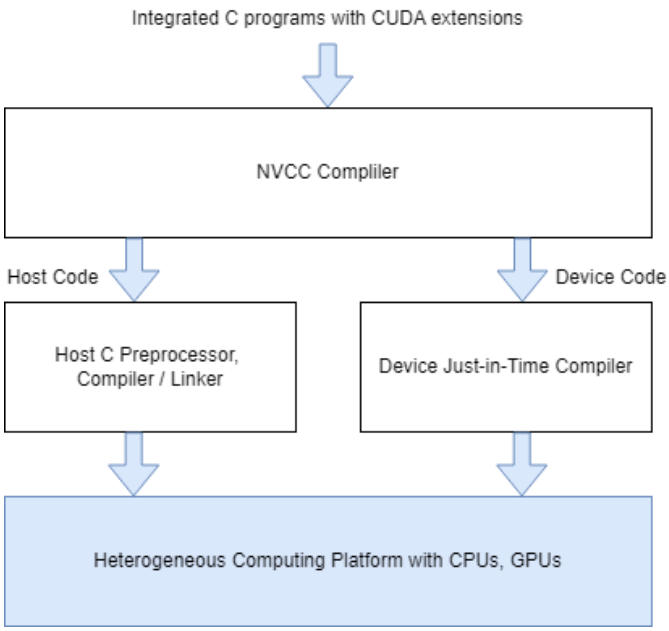

Figure 2.3: Illustration of a CUDA C program's compilation process. Figure adapted from [6].

2.3.1 Kernels

A kernel is essentially a function that is executed N times in parallel by N different CUDA threads. When a kernel function is called, it is executed by a large number of threads on the device. In the host code, a kernel function is called using the extended function call syntax `kernelFunc<<blocksPerGrid, threadsPerBlock>>(args);`. The collection of threads generated by a kernel launch is collectively referred to as a grid. Once all threads in a kernel have completed, the corresponding grid terminates, and the host resumes executing the CPU code [1].

2.3.2 Threads, blocks, and grids

CUDA has a three-level execution hierarchy that consists of threads, blocks, and grids [1]. A 'thread' is the smallest basic execution unit in CUDA. A 'block' is a group of threads with a shared memory, while a 'grid' is a group of one or more

blocks. There exists additionally an optional level of hierarchy called a thread block cluster [5].

Threads can be synchronized at the block level to ensure that they reach the same point in the kernel before any continue. However, there is no built-in mechanism in CUDA to synchronize threads across different blocks at the grid level [6].

When a kernel is launched, the number of blocks and threads per block is defined by the programmer. At runtime, each block is executed by only one streaming multiprocessor (SM). One SM can execute multiple blocks in a concurrent fashion [6].

When a block has been assigned to an SM, it is divided into thread units called 'warps' that are processed together. A warp is the unit of thread scheduling in SMs [6]. SMs manage the warps and schedule them based on resource availability.

2.3.3 Memory hierarchy

In CUDA, each thread has its own private registers and also a memory space called the 'local memory'. The local memory is used for thread-private variables that do not fit into the register space, but it has a higher latency than register memory [1].

Additionally, each block has its own memory space, called the 'shared memory', which is accessible by all the threads within the block. However, the memory space of one block is not visible to other blocks. Registers and shared memory are on-chip memories, meaning they are accessible at very high speed and in a highly parallel manner [1].

Additionally, the kernel also has a memory space, accessible by any thread in any block, which is called the 'global memory'. The global memory is the largest memory space and has the highest latency [1]. The local memory is actually placed in global memory in sections for each thread and thus has a similar latency [6].

2.4 Field-programmable gate array

A *field-programmable gate array* (FPGA) is a type of programmable integrated circuit or 'microchip'. FPGAs can be programmed in the 'field' – meaning their internal functionality is not determined by the manufacturer – to perform a wide variety of tasks. Some FPGAs may only be programmed once and are referred to as *one-time programmable*, while others may be programmed and reprogrammed indefinitely [7][8].

An FPGA architecture typically consist of configurable logic blocks (CLBs) arranged into a grid that implement logic functions. These logic functions are connected by programmable routing. FPGAs also have I/O blocks at the periphery of the grid that make off-chip connections and that are connected to the logic blocks through the routing interconnect [7]. Figure 2.4 illustrates a typical FPGA architecture on a high-level. The FPGA is similar to the GPU in that it has a parallel architecture.

FPGAs in signal processing

FPGAs are particularly usable in high-performance digital signal processing (DSP) applications. This is due to their inherent hardware parallelism that allows for processing large quantities of data and pipelining of dataflow. FPGAs also perform operations under strict latency constraints and are easy to integrate with commonly used modules such as ADCs [10].

In signal processing tasks where the sampling rate is high (e.g., above a few MHz) or where the data rate is high (e.g., above few tens of Mbytes/s), an FPGA will be a better choice than a traditional DSP chip. This is because when the sample rate is high, a DSP chip has to perform many data transfer operations without loss, as the DSP chip uses shared resources like memory buses [11].

However, FPGAs also have downsides both generally and specifically relating to

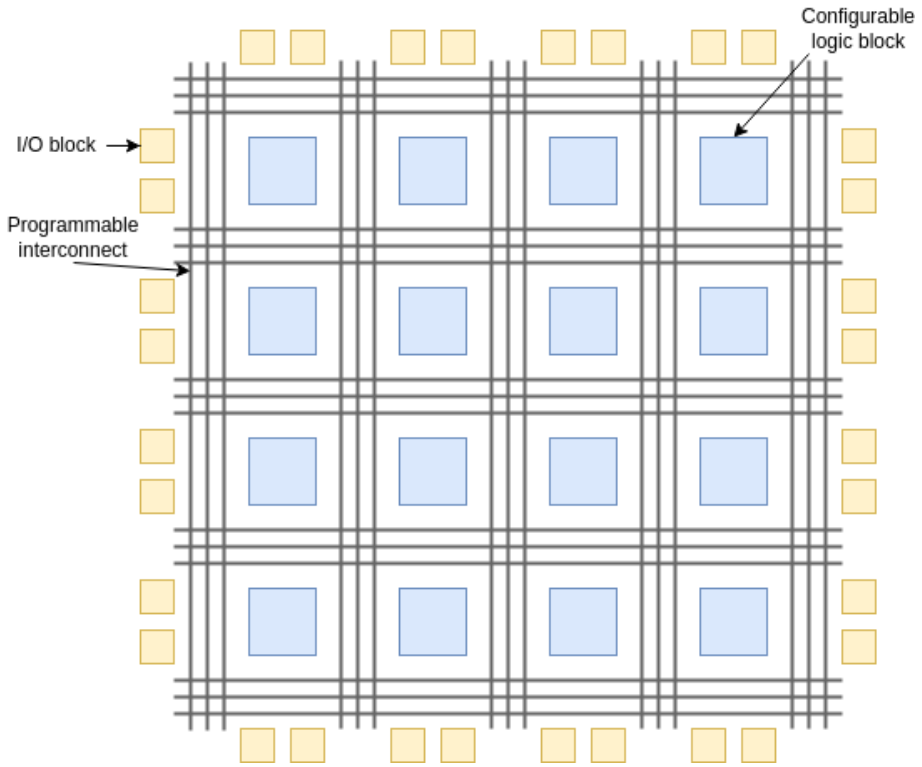

Figure 2.4: A typical FPGA architecture at a high-level. Figure adapted from [9].

digital signal processing applications. They tend to be more complex to design and program, requiring detailed knowledge of e.g., hardware design and VHDL, while a DSP or a GPU can be programmed e.g., using C and CUDA respectively. The longer FPGA development cycle increases development costs and time-to-market and is thus typically a more expensive choice than a DSP [11].

Given the limitations of dedicated DSP processors and FPGAs, the GPU may be an alternative in some high-performance digital signal processing tasks.

2.5 Real-time digital signal processing

In signal processing, a *signal* is some function $f(t)$, real or complex, of a time variable t that conveys information about some phenomenon [12], e.g., the state or behavior of a physical system. At the physics level, a signal is some voltage, current,

or electromagnetic wave that carries information over time. Signals can be classified into two categories: continuous analog signals and discrete digital signals. All digital signals are by definition discrete (in time or in amplitude) and all continuous signals (in time and in amplitude) are by definition analog.

An analog signal is continuous and its domain is the real line (or some interval thereof), $t \in \mathbb{R}$ [12]. Analog signals have infinite resolution of amplitude values and can be processed using analog electronics devices (such as amplifiers, modulators, and filters).

A signal is discrete if its signal is represented by a sequence of numbers $x = \{x[n]\}$ where n is the set of all integers ($-\infty < n < \infty$) and $x[n]$ is the nth sample in the sequence [13]. A discrete signal is defined at a particular set of time instances and it can be represented as a sequence of numbers that have a range of values. Discrete signals are often generated by periodic sampling of some continuous-time signal. In such a case, the nth sample of the sequence is equal to the value of the analog signal at a time $t = nT$:

$$x[n] = x_a(nT) \tag{2.1}$$

where T is the sampling period and $x_a(t)$ is the sampled analog signal [13].

Digital signal processing (DSP) is the digital representation of signals as well as, e.g., the analysis, modification, storing, transmission, and information extraction from these signals [14].

A DSP application is said to be either non-real-time or real-time. Real-time refers to the system's correctness depending on the outputs and on the timeliness of said outputs. In other words, the response time to an event must occur within a predetermined amount of time. In practice this means that processing of each signal or data point must be completed before the next data point arrives. Failure to meet the deadlines can result in system failure or bad performance. There are

also latency constraints, referring to the time from input to processed output, which must be sufficiently low for the system's purpose. Finally, a real-time DSP system must be deterministic [13].

2.5.1 Sampling

In signal processing, *sampling* is the act of turning a continuous-time signal into a discrete-time signal [12]. *Periodic sampling* consists of recording the value $S(t)$ of some signal at regular intervals T (i.e., the *sampling period*). An ideal sampler has a sampling period of T, where f_s is the sampling frequency in hertz (Hz) [13]. Mathematically, this is given as:

$$T = \frac{1}{f_s} \tag{2.2}$$

According to the Nyquist-Shannon sampling theorem, the sampling frequency f_s must be at least twice the maximum frequency component f_M (also called the *bandwidth*) in the analog signal. Mathematically, this is given as:

$$f_s \geq 2 \cdot f_M \tag{2.3}$$

This minimum sampling rate or frequency f_s is called the *Nyquist rate* [12]. If a sample rate below the Nyquist rate is used, information about the signal will be lost. This will cause a phenomenon known as *aliasing* to occur, leading to artifacts and distortion in the reconstructed signal.

In practice, sampling is done, e.g., by using an analog-to-digital converter (ADC). This process, called A/D conversion or digitization, consists of sampling (digitization in time) and quantization (digitization in amplitude) [13].

2.5.2 Constraints

A significant limitation in real-time applications of digital signal processing is the system's bandwidth [15]. The maximum rate at which an analog signal can be sampled is determined by the system's processing speed.

In sample-by-sample processing, each output sample is generated before a new input sample is presented to the system. This means that the time delay between the input and output for must be less than one sampling interval (T seconds) [15].

A requirement in real-time DSP is that the signal processing time t_p and output time t_o must be less than the sampling period T for it to be able to complete the processing before a new sample is received. Mathematically, that is given as [15]:

$$t_p + t_o < T \tag{2.4}$$

This is a hard constraint that limits the highest processable frequency in the sample-by-sample approach. This constraint can be expressed mathematically as:

$$f_M \leq \frac{f_s}{2} \leq \frac{1}{2}(t_p + t_o) \tag{2.5}$$

where f_M is the highest frequency signal that can be processed [15].

It is possible to increase the real-time bandwidth by using faster digital signal processors, simplified algorithms, and multiple or multi-core processors. The above equation also shows that real-time bandwidth can be increased by reducing the I/O operations.

2.6 I/Q data

In-phase (I) and *quadrature* (Q) components are concepts in the field of signal processing. They refer to two sinusoidal signals with the same frequency, which are phase-shifted by ±90 degrees (or $\pm\frac{\pi}{2}$ radians).

A time-varying signal can be expressed mathematically as:

$$y(t) = Asin(2\pi ft + \phi) \tag{2.6}$$

where A is the amplitude, f is the frequency and ϕ is the phase [16].

As covered earlier, in digital signal processing, a signal is represented by sampling it at some set time intervals. These samples are represented by a point in the complex plane, which includes two components: the in-phase (I) component and the quadrature (Q) phase component. On this plane, every (I, Q) tuple of values corresponds to a sample of the time-varying signal $y(t)$. The sequence of these values is represented as (I_t, Q_t) where $t = 1, 2, ..., n$ [16].

The *real* part of the complex representation is the in-phase component (I) and the *imaginary* part is the quadrature component (Q). By convention, the cosine part is the in-phase component and the sine part is the quadrature component. The quadrature component then has a relative phase shift of $\pm\frac{\pi}{2}$.

The I/Q representation is useful because it offers more detailed information than what can be obtained by simply using a sequence of time-domain samples from the signal. The limitations of the latter approach are that determining the signal's frequency is, in that case, impossible and, further, it is difficult to exactly determine the signal's peak amplitude from the time-varying data. The I/Q representation solves these issues by conceptualizing the signal as a helix in three dimensional space instead of as samples from a time-varying curve [16]. In other words, the I/Q representation preserves the amplitude and phase information of the signal. The Figure 2.5 illustrates I/Q signal representation as a helix.

I/Q data can also be viewed as a 'phasor diagram', which involves plotting complex numbers as vectors. Figure 2.6 illustrates this. A complex number consists of a real and imaginary portion, usually expressed in the form $a + bi$ (where i is the imaginary unit $i^2 = -1$), and it has a magnitude and a phase [12]. The magnitude can be calculated as $A = \sqrt{a^2 + b^2}$ while the phase can be calculated as $\phi = \tan^{-1}\frac{b}{a}$.

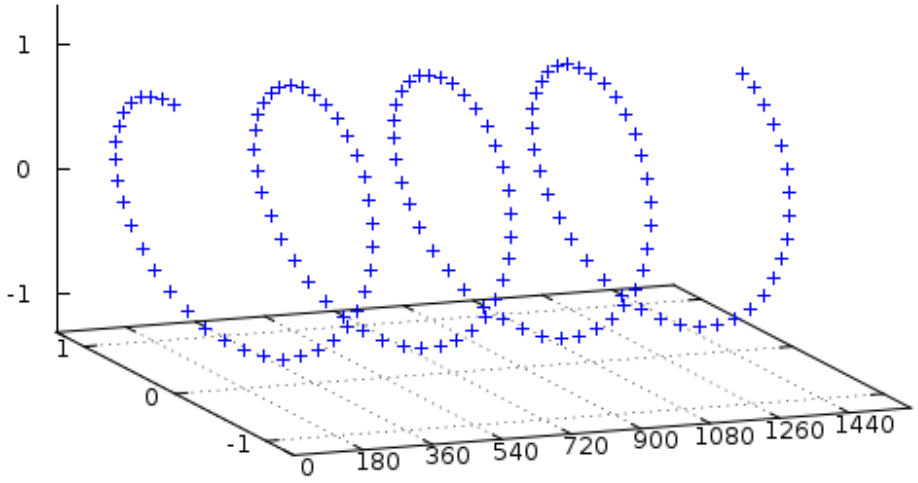

Figure 2.5: I/Q signal representation as a helix. Figure taken from [16].

A *phasor* is a complex number representing the amplitude and phase of a sinusoidal function [12]. I/Q sampling is the form of sampling used by, e.g., software-defined radios (SDRs).

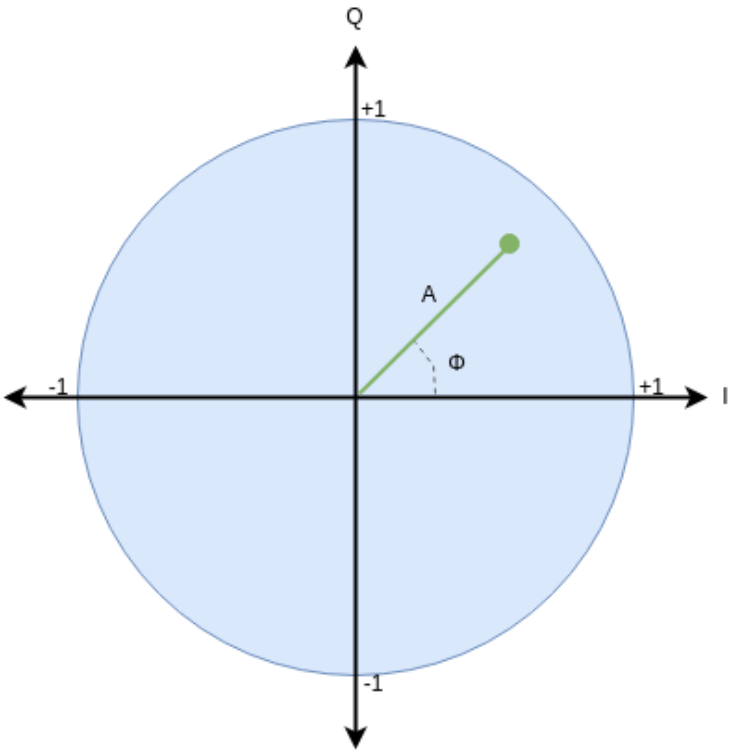

Figure 2.6: Complex signal representation as a phasor diagram.

2.7 Fourier transform

The *Fourier transform* (FT) is an integral transform of a function from time or space domain to the frequency domain [12].

The Fourier transform of a continuous time signal $f(t)$ is given by [12]:

$$F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt, -\infty \leq \omega \leq \infty \tag{2.7}$$

The Fourier transform is often implemented with a discrete version of the transform, called the discrete Fourier transform (DFT). This, in turn, is generally implemented with a more efficient algorithm known as the fast Fourier transform (FFT) [12]. In signal processing, the Fourier transform is fundamental for analyzing the frequency components of signals. It can be used, e.g., in spectral analysis to extract features that can be used for target identification.

2.7.1 Discrete Fourier transform

The *discrete Fourier transform* (DFT) maps a function from, e.g., its time domain into its frequency domain representation. The DFT is used to represent a reasonable approximation of a signal for which only a finite sample exists [12]. The DFT of a finite-duration signal $x(n)$ of length N can be defined mathematically as [15]:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-i(\frac{2\pi}{N})kn}, k = 0, 1, ..., N-1, \tag{2.8}$$

where k is the frequency index and $X(k)$ the kth DFT coefficient. The DFT can also be expressed as [15]:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, k = 0, 1, ..., N-1, \tag{2.9}$$

where

$$W_N^{kn} = e^{-i(\frac{2\pi}{N})kn} = cos(\frac{2\pi kn}{N}) - isin(\frac{2\pi kn}{N}), 0 \leq k, n \leq N-1$$

in which the W_N^{kn} parameters are called the *twiddle factors* [15].

To compute each coefficient $X(k)$ in the DFT, approximately N complex multiplications and additions are performed. Thus, approximately N^2 such operations are required to compute N samples of $X(k)$ for $k = 0, 1, ..., N-1$ [15]. As such, the DFT is often difficult to use in practical applications due to its high computational time complexity of $O(n^2)$.

2.7.2 Fast Fourier transform

The *fast Fourier transform* (FFT) is an algorithm that reduces the computational time complexity of the discrete Fourier transform (DFT) from $O(n^2)$ down to $O(n \log n)$ [12]. Various different implementations of the FFT have been developed. The first and most well-known FFT algorithm is the Cooley-Tukey algorithm from 1965

[17], which was a major breakthrough in signal processing at the time. FFT implementations can be broadly classified into *decimation-in-time* (DIT) and *decimation-in-frequency* (DIF) methods. These refer to the specific methods of dividing the original DFT computation into smaller parts. DIT algorithms divide the original sequence into smaller subsequences based on their indices in the time domain. These are then recursively processed. DIF algorithms consider the full sequence and then recursively divide it based on frequency components.

The twiddle factor W_N^{kn} is a *periodic* function as its values repeat over a specific interval. It is also said to be *symmetric* because of the twiddle factor's conjugate symmetry property. The periodicity and symmetry of the twiddle factor can be exploited to eliminate some of the mathematical operations involved in the calculation of the DFT mentioned earlier, giving us the FFT algorithm and reducing computation [13][15]. As mentioned, the FFT is based on the fundamental idea of decomposing the computation of the DFT of a sequence of length N into smaller DFTs that are then combined to form the N-point transform. The specific manner in which this is implemented results in the different variations of FFT algorithms that exist [15].

Perhaps the simplest and most common form of the Cooley-Tukey FFT is a radix-2 decimation-in-time. It involves dividing a DFT of size N into two interleaved DFTs of size $N/2$. First the even-indexed inputs are computed and then the odd-indexed inputs, after which the results are combined. This is then performed recursively [13]. Figure 2.7 shows the diagram for a $N = 8$ decimation-in-time radix-2 decomposition of a DFT, where $G[k]$ designates the 4-point DFT of the even-indexed points and $H[k]$ designates the odd-indexed points.

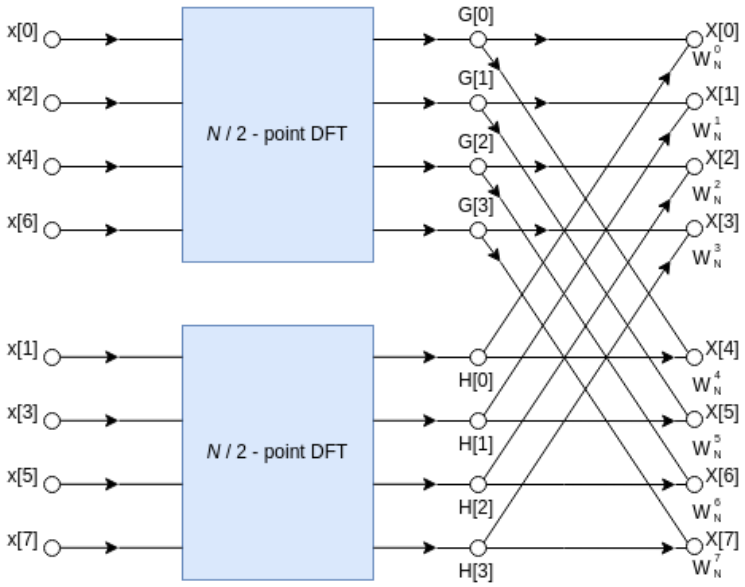

Figure 2.7: Data flow diagram for a $N = 8$ decimation-in-time radix-2 DFT decomposition into two $N/2$ DFT computations and their combination. Figure adapted from [13].

2.8 Amplitude spectrum

The *amplitude spectrum*, often also called the *magnitude spectrum*, is a representation of the magnitudes of the various frequency components present in a signal. It shows how the amplitude of each frequency component varies with frequency.

The DFT coefficient $X[k]$ is a complex variable and may be expressed in polar form as [15]:

$$X[k] = |X[k]|\, e^{j\phi[k]} \tag{2.10}$$

The absolute amplitude $|A[k]|$ for each frequency component or bin k can be calculated as [15]:

$$|A[k]| = \sqrt{\Re(X[k])^2 + \Im(X[k])^2} \tag{2.11}$$

where $\Re(X[k])$ and $\Im(X[k])$ are the real and imaginary parts of $X[k]$, respectively. This operation gives the amplitude or magnitude spectrum.

2.9 Window functions

One of the properties of the DFT is that it assumes that the signal that it is applied upon is periodic (i.e., a time series of length N repeating itself in a cyclic manner). In other words, it expects that the next value after N samples is zero. Consequently, if the frequency of the sinusoidal input signal is not an exact multiple of the frequency resolution, meaning it is not perfectly aligned with the center of a frequency bin, this assumption does not hold. As a result, the DFT will interpret a discontinuity between the last sample and the first sample. This discontinuity causes the power to spread across the entire spectrum [18].

Window functions are mathematical functions that are zero-valued outside of some predefined interval [19]. A window function typically begins at zero, peaks at the center of the time series, and then decreases to zero. Mathematically, it is defined as a vector of real numbers $\{w_j\}, j = 0 \ldots N-1$ [18].

Window functions are applied on the signal in the time domain before the DFT computation. I.e., the time series x_j is multiplied with the window. This gives an input of $x'_j = x_j \cdot w_j$ to the DFT. This effectively removes the previously discussed discontinuity phenomenon. Numerous window functions have been developed. When choosing between them, there is typically a trade-off between the width of the resulting peak in the frequency domain, amplitude accuracy, and the rate at which spectral leakage decreases into other frequency bins [18].

2.9.1 Rectangular window

A situation where no windowing is applied is called the *rectangular window*. Consequently, it is also sometimes called *no window*. It is given by [18]:

$$w_j \equiv 1$$

When using the rectangular window function, the normalization of DFT samples is achieved by dividing the magnitudes by the number of samples N.

2.9.2 Blackman–Harris window

The *Blackman–Harris* window function belongs to a family of window functions defined by a sum of cosine terms. Various characteristics can be optimized by adjusting the number and coefficients of said terms. The Blackman–Harris window's benefits are minimal spectral leakage as well as reasonable bandwidth and amplitude error. It is ideal for detecting small sinusoidal signals that are close in frequency to large signals thanks to its small sidelobes. It can also serve as a versatile general-purpose window for applications requiring a high dynamic range, as long as amplitude accuracy for sinusoidal signals is not crucial [18].

The four-term periodic Blackman-Harris window function $w(n)$ can be defined as [20]:

$$w(n) = a_0 - a_1 \cos\left(\frac{2\pi n}{N}\right) + a_2 \cos\left(\frac{4\pi n}{N}\right) - a_3 \cos\left(\frac{6\pi n}{N}\right) \tag{2.12}$$

where $0 \leq n \leq N-1$ and the coefficients are:

$$a_0 = 0.35875, \quad a_1 = 0.48829, \quad a_2 = 0.14128, \quad a_3 = 0.01168$$

with N being the total number of points in the window.

The Figure 2.8 illustrates a Blackman–Harris window function of size $N = 4096$.

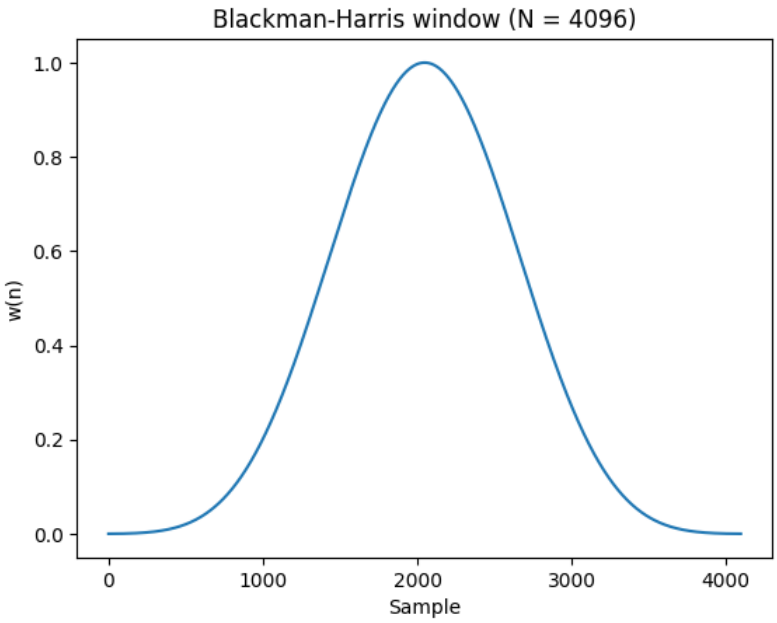

Figure 2.8: An example plot of the Blackman–Harris window function.

Normalization

When using the Blackman-Harris window function, the normalization of DFT samples is performed by dividing the magnitudes by the sum S of the window samples w_i [18]:

$$S = \sum_{i=0}^{N-1} w_i \tag{2.13}$$

2.10 Software-defined radio

The SDR Forum and the Institute of Electrical and Electronic Engineers' (IEEE) P1900.1 group has given the following definition to *software-defined radio* (SDR): "a radio in which some or all of the physical layer functions are software defined" [21]. A *radio* is any device that transmits or receives wireless signals in the radio frequency (RF) part of the electromagnetic spectrum, typically considered to be from 3 kHz to

300 GHz [22]. *Software-defined* refers to the use of software processing to implement operating (but not control) functions [21].

Digital processing has become very common in communications systems. Contemporary signal processing systems have developed in a direction where the majority of baseband functionality is implemented in software. An SDR system is typically complex and performs several tasks simultaneously to enable a seamless transmission and reception of data [21].

In an SDR, the radio communications functions that are traditionally implemented in hardware – e.g., the modulation and demodulation of signals – are instead implemented by software. An SDR processes all incoming and outgoing RF signals in digital form and has little or no analog hardware. Instead, SDRs may incorporate modules like FPGAs, Analog-to-Digital and Digital-to-Analog Converters, and digital signal processors (DSP) as part of the system.

2.11 VITA-49

In communication systems, efficient and standardized data transport protocols are essential for ensuring interoperability and optimal performance. This section provides an overview of the VITA-49 standard, which is foundational to the solution implementation discussed later in this thesis.

VITA (formerly VMEbus International Trade Association) is a business association and a standards development organization that works on, among other things, establishing standards for the wireless industry [23].

The *ANSI/VITA 49.2* standard is a packet-based protocol for conveying digitized signal data. It is part of the VITA Radio Transport (VRT) family of standards. VRT defines a message protocol for interoperability between radio frequency (RF) systems and related equipment by defining the packet framework that describes the spectrum with respect to its physical properties. The protocol is independent of

manufacturer equipment type, system type, and it does not prescribe any specific hardware architecture or software framework. It is designed to provide a standardized way to transport digitized radio signals over a network [24]. VITA-49 is an application layer protocol in the OSI model. This means it is not restricted to any specific underlying transport method and can operate over various lower layer protocols within the OSI model.

Signal Data Packet

The Figure 2.9 illustrates a VITA-49 signal data packet's structure. Only the header and data payload fields are mandatory, as per the standard [24]. A signal data packet transports digitized samples of an RF, IF, or baseband signal in a standard format across any type of link [24].

The signal data packet's payload consists of a contiguous sequence of an integer number of samples from some signal data stream. The payload field is not strictly defined, and hence a wide range of data types are supported, including up to 32-bit for In-phase (I) and Quadrature (Q) data each (i.e., 64-bit complex in total). Thus, the data's format can be optimized depending on the specific use case's requirements and constraints [24].

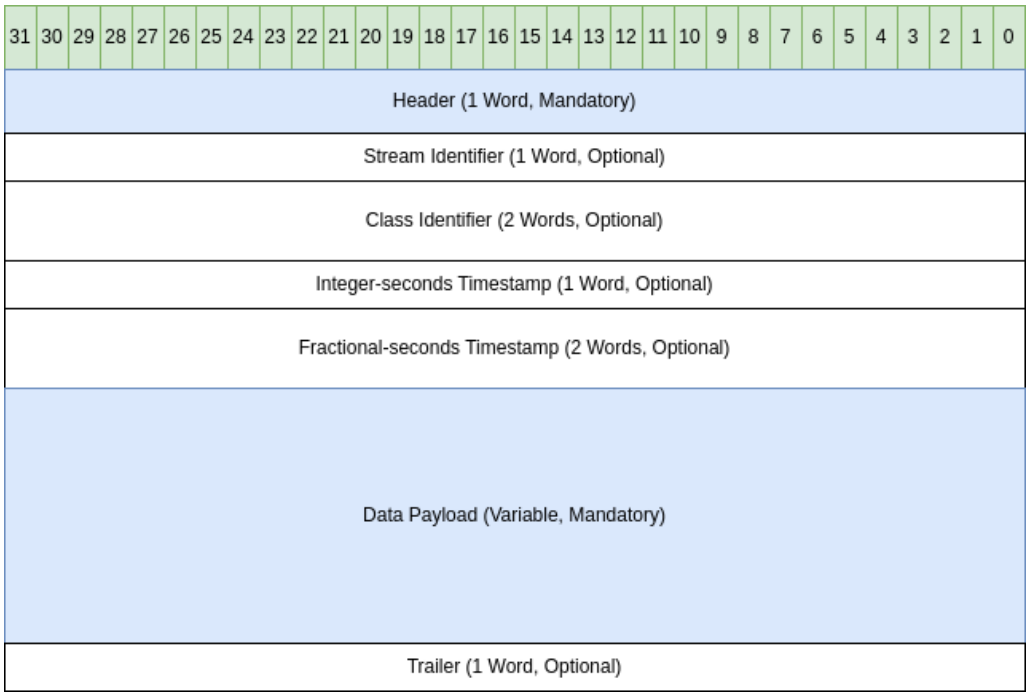

Figure 2.9: VITA-49 template for the signal data packet, adapted from [24].

3 Literature review

To establish an understanding of the current state of the research topic, a review of the existing literature on real-time, digital radar signal data processing on GPUs was conducted. The aim was to understand and provide an overview of the current state of affairs and recent developments in this area. The focus was to understand the state of adoption of GPUs in real-time radar signal data processing, the known challenges and performance bottlenecks, what signal processing tasks are typically performed on the GPU, and the level of performance typically achieved in this context.

3.1 Methodology

To find the papers for this literature review, three databases were utilized: Google Scholar, ACM Digital Library, and IEEE Xplore. The searches were conducted in April of 2024. These databases were chosen for their extensive coverage of technological and scientific research articles.

The inclusion criteria for selecting papers were as follows:

- Paper is written in the English language
- Peer-reviewed article or conference paper, or technical report by an established organization, or an industrial research paper or report
- Topic concerns the application of GPU technology in real-time digital signal

processing of radar signals

- Published in the last ten years (i.e., between 2014 and 2024)
- Has a similar digital signal processing implementation on the GPU as the one developed in this thesis
- Contains empirical or experimental results of interest related to e.g., performance and latency

The following search strategy was implemented: three databases were searched for papers using specific queries. These databases were Google Scholar, IEEE Xplore, and ACM Digital Library. The resulting sets of papers from IEEE Xplore and ACM Digital Library were screened based on their entire contents, while the papers on Google Scholar, due to their much larger number, were screened based on title and abstract first. Then, a subset of the aforementioned was further evaluated based on their entire contents. Finally, out of all of the databases, a selection of eleven papers made it to the final literature review. Figure 3.1 illustrates the implemented methodology. If a paper was encountered in one database, any subsequent appearances in other searches were omitted.

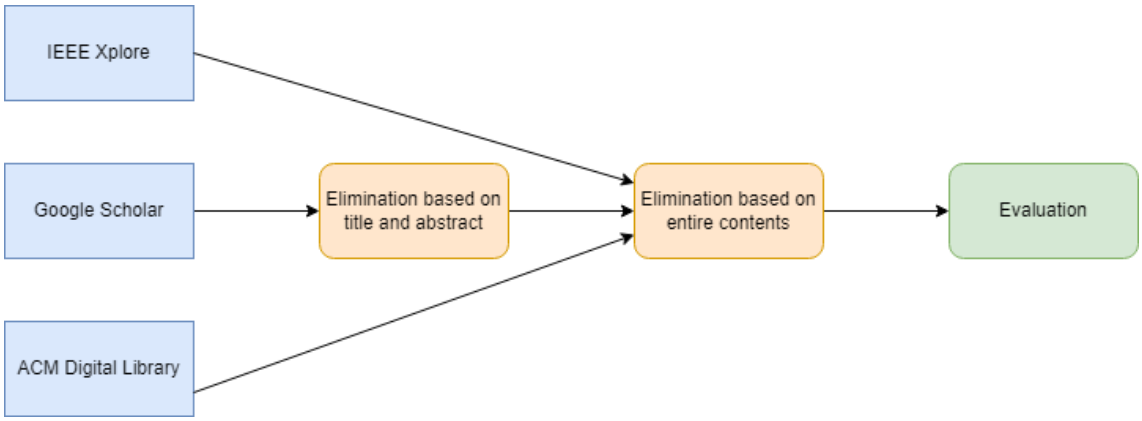

Figure 3.1: Literature review process used in this thesis.

The initial Google Scholar query *("real-time" OR "real time") AND ("radar signal processing" OR "radar data processing") AND (GPU OR "graphics processing unit" OR GPGPU OR CUDA)* with the date range 2014 - 2024 yielded a large number of articles. The search query was further refined to narrow the scope and to exclude irrelevant papers, such as medical literature. The Google Scholar search with the query *("real-time" OR "real time") AND ("radar signal processing" OR "radar data processing") AND (GPU OR "graphics processing unit" OR GPGPU OR CUDA) AND ("passive radar" OR "synthetic aperture radar") AND -medical -"neural network" -weather* and date range 2014 - 2024 yielded 217 results. These articles were then screened by title and abstract for relevance. Papers most relevant to this thesis contained similar digital radar signal processing implementations on the GPU and empirical results relating to performance (e.g., in throughput or latency). Ultimately, four of these articles made it to the final literature review evaluation.

The IEEE Xplore search with the query *("real-time" OR "real time") AND ("radar signal processing" OR "radar data processing") AND (GPU OR "graphics processing unit" OR GPGPU OR CUDA)* and date range 2014 - 2024 yielded 35 results. These papers were all evaluated based on their content for relevance to this thesis. Ultimately, five of these articles made it to the final literature review evaluation.

The ACM Digital Library search with the same query *("real-time" OR "real time") AND ("radar signal processing" OR "radar data processing") AND (GPU OR "graphics processing unit" OR GPGPU OR CUDA)* and the same date range yielded 33 results. These were also evaluated based on their content for relevance. Ultimately, one of these articles made it to the final literature review evaluation.

The Table 3.1 displays the search results with the aforementioned queries and shows how many articles were taken from each database for the final literature review.

Repository	Number of results	Final evaluation
Google Scholar	217	4
IEEE Xplore	35	5
ACM Digital Library	33	1

Table 3.1: Total number of articles retrieved from each database.

3.2 Findings

Based on the findings, there has been a modest but increasing amount of peer-reviewed, published research on the utilization of GPUs for real-time radar signal processing tasks. The variation in the research tends to be in the type of radar signal processing tasks or algorithms examined, the specific type of radar in question, and comparisons between devices such as GPUs and CPUs. A majority of the implementations seem to have been done using CUDA and most have employed a heterogeneous CPU-GPU platform.

The Table 3.2 at the end of this chapter lists the final set of papers included in this literature review. The chosen set of papers will next be discussed individually and their findings analyzed thematically. As mentioned earlier, these papers were chosen for their similarity to the solution developed in this thesis, in terms of software implementation and performance evaluation.

3.2.1 Radar type

The selected papers on the application of GPUs in digital radar signal processing tasks contains applications with different types of radar.

Zhao et al. discussed the implementation and evaluation of GPU parallel processing techniques for signal processing in passive bistatic radar (PBR) systems [25]. Kong et al. discussed real-time signal processing on the GPU for a general phased array radar [2]. Li et al. presented a GPU-accelerated, software-defined radar system [27]. Liu et al. implemented a MIMO radar parallel simulation system based on a heterogeneous CPU-GPU platform [26]. Malanowski et al. implemented a real-time passive coherent location (PCL) radar, which are a special case of passive bistatic radar [30]. Zhang et al. also implemented real-time, entirely GPU-based signal processing for a passive bistatic radar (PBR) system [31]. Dong et al. researched real-time signal processing on a CPU-GPU based heterogeneous platform for a phased array radar [29]. Deng et al. implemented a real-time high frequency radar signal processing system on the GPU [33]. Rupniewski et al. implemented real-time signal processing for a phased array pulse-Doppler radar on a heterogeneous GPU-FPGA platform [32]. Cai et al. implemented various general adaptive pulse correction (APC) algorithms of a pusle-Doppler radar on the CPU and GPU [28].

The Table 3.3 summarizes the different radar types that were implemented in the papers. In total, there were four implementations of the phased array radar type, three implementations of the passive bistatic radar type, two software-defined radars, and one pulse-Doppler radar in the selected papers. However, this is merely a small subset of different radar types that exist in civilian and military use.

Paper	Radar type
Zhao et al. [25]	passive bistatic radar (PBR)
Kong et al. [2]	phased array radar
Li et al. [27]	software-defined radar
Liu et al. [26]	multiple-input multiple-output (MIMO) radar
Malanowski et al. [30]	passive coherent location (PCL) radar
Dong et al. [29]	phased array radar
Deng et al. [33]	high-frequency software-defined radar
Rupniewski et al. [32]	phased array pulse-Doppler radar
Cai et al. [28]	pulse-Doppler radar
Zhang et al. [31]	passive bistatic radar (PBR)

Table 3.3: List of radar types implemented in the papers.

3.2.2 Signal processing chain

In signal processing, the *signal processing chain* refers to the series of stages through which a signal passes to be processed and transformed. Each stage involves specific operations designed to modify or analyze the signal to extract useful information, improve quality, or prepare it for further processing or transmission.

Kong et al. implemented a generic signal processing chain for a phased array radar, which included the following steps in sequential order: preprocessing, digital beam forming, pulse compression, Doppler spectrum estimation, and post-processing. They used baseband radar I/Q data that they obtained from a phased array radar. The input data was multidimensional (referred to as a 'data cube'). Each of the nodes in the signal processing chain involved highly parallelizable matrix manipulations (e.g., Fourier beam forming). The authors used the optimized cuFFT and cuBLAS CUDA libraries in their implementation. In the work, the data cube

was at each stage of the signal processing chain larger than the number of cores available on the GPU, and according to the authors this meant that they were able to maximize the computational power of the GPU [2]. However, they did not report performance in FLOP/s and compare it to the theoretical maximum output of their GPUs to support this.

Li et al. implemented a software-defined radar solution on a heterogeneous CPU-GPU platform. Their signal processing chain consists of, among other things, cross-ambiguity function (cross-correlation), CLEAN algorithm, constant false alarm rate (CFAR), as well as Doppler shift and range for spectrogram generation. The tasks performed on the GPU specifically are in cross-correlation: complex conjugate, fast Fourier transform, multiplication of received signal by complex conjugate, and inverse fast Fourier transform. Everything else, e.g., CLEAN algorithm, CFAR, and spectrogram generation, was performed on the CPU. The authors used batched two-dimensional FFTs on the GPU [27].

Liu et al. proposed a time division MIMO radar signal processing implementation on a CPU-GPU heterogeneous platform. In their solution the signal processing chain consisted of dechirp processing, moving target indication (MTI), moving target detection (MTD), and digital beamforming. On the GPU, the following parts were performed: windowed MTI, MTD, phase compensation, and beamforming, while the CPU mostly did the preprocessing. The authors combined windowed MTI, MTD and Doppler phase compensation into one module for processing to improve performance and to meet the real-time requirements. The authors used the cuFFT library for computing the two-dimensional FFTs. They also optimized by using a CUDA stream architecture. Further, the authors also implemented multi-threading on the CPU-side with OpenMP [26].

Zhao et al. implemented passive bistatic radar signal processing on a heterogeneous CPU-GPU platform, which consisted of implementing the following signal pro-

cessing chain on the GPU: clutter suppression (ECA-B algorithm), range-Doppler processing, and cell averaging constant false alarm rate (CA-CFAR) processing. However, the authors did not implement a CUDA stream architecture for their solution. The authors also used the cuFFT library [25].

Malanowski et al. implemented a passive coherent location radar system that processes DVB-T and FM signals as so-called illuminators of opportunity. It was also a heterogeneous solution based on CPU-GPU cooperation. The signal processing chain consisted of the following steps: beamforming, filtering, cross-ambiguity calculation, acceleration filtering, CA-CFAR detection, low level extraction, object extraction, as well as tracking and localization. Of these, everything except for the object extraction as well as tracking and localization was done on the GPU. The authors also used the cuBLAS and cuFFT libraries [30].

Zhang et al. implemented a passive bistatic radar system with the following signal processing chain: adaptive filtering, range-Doppler estimation, and CA-CFAR detection. The authors also used the cuBLAS and cuFFT libraries [31].

Dong et al. implemented a phased array radar solution on a heterogeneous CPU-GPU platform. In their solution, the CPU was responsible for the overall processing flow, while the GPU was responsible for executing the computationally intensive parts of the algorithms. The authors also noted that in addition to considering the parallelizability of a given module in the radar signal processing algorithm, one must also consider whether the pre- and post-processing tasks of the module are also computationally intensive, as the data copying between host and device can be a performance bottleneck. Interestingly, the authors implemented a pipelined architecture on the host side to optimize multi-core CPU usage [29].

Deng et al. proposed a high frequency (HF) radar system inspired by the software-defined radar concept. By SDR it is meant that the signal is digitized at an early stage in the processing and most operations are performed digitally by

software. The implemented signal processing chain consisted of pulse compression, Doppler processing, digital beamforming, and generalized sidelobe canceller. The authors tried implementing the HF radar using a CPU-only approach with the Intel Math Kernel Library (MKL). However, this implementation had bad stability (i.e., heavy fluctuation in the processing times) and could not meet the real-time requirement. The authors then decided to implement a GPU-based approach, which yielded a better performance and very little fluctuations. The authors also made use of the cuBLAS and cuFFT libraries [33].

Rupniewski et al. proposed a heterogeneous radar processing system consisting of field-programmable gate array (FPGA) and GPU, and CPU. In their solution, the FPGA is used a signal generator and the GPU is used as the primary signal processing unit at each stage. The authors employed NVIDIA's GPUDirect RDMA (remote direct memory access) to transfer signal data from the FPGA into GPU memory, bypassing CPU overhead. The following part of the signal processing chain could be done on either FPGA or GPU: downconversion using the digital product detector algorithm, filtering, and decimation. The processing chain solely on the GPU-side consisted of beamforming, pulse compression (i.e., match filtering), Doppler filtering, and detection. The authors also used the cuBLAS and cuFFT libraries [32].

Cai et al. implemented adaptive pulse correction (APC) signal processing algorithms such as reiterative minimum mean-square error (RMMSE) and RMMSE based on matched filter outputs (MF-RMMSE). Their latency performances were then compared on the CPU and the GPU. The authors also used the cuBLAS and cuFFT libraries [28].

3.2.3 Processing platform

Two distinct approaches were encountered in the literature related to the processing platforms of choice. Either a GPU-only approach was used or a so-called heterogeneous platform was used (i.e., one where the CPU and the GPU work together).

Kong et al. implemented phased array radar signal processing algorithms entirely on the GPU. To further reduce processing time, the authors suggested pipelining multiple GPUs in a way that each node in the signal processing chain is implemented on one single GPU, though they did not implement this suggestion [2]. Another GPU-only approach was done by Cai et al. [28].

Li et al. implemented their SDR solution on a heterogeneous CPU-GPU platform, with a rather complex task division (the GPU was only partly responsible for the signal processing chain) [27]. Liu et al. also implemented their MIMO radar system on a heterogeneous CPU-GPU platform. The authors implemented a parallel acceleration of the CPU based on OpenMP technology in combination with CUDA stream operation on the GPU. This was done so that the CPU and GPU can be perform in parallel simultaneously and in order to improve the processing speed and meet the real-time requirements [26]. Zhao et al. also implemented a heterogeneous CPU-GPU system [25], as did Malanowski et al. [30], Zhang et al. [31], Dong et al. [29], and Deng et al. [33].

Rupniewski et al. implemented a heterogeneous platform using CPU, GPU, and FPGA. However, the FPGA's role was to act as an artificial signal simulator, meaning the actual signal processing was here also mainly done on the GPU. This setup highlighted the potential of the GPU working alongside the FPGA in a radar signal processing system and not necessarily replacing it altogether, which is in part what the authors sought to demonstrate [32].

One of the known performance pitfalls in a heterogeneous setup is the potential time loss in memory copies having to be made back and forth between host (CPU)

and device (GPU), which was noted in Li et al. [27]. Another related problem is the requirement of high data throughput. Utilizing RDMA to transfer the signal data directly into GPU memory via PCIe is a way to cope with this and bypass the CPU overhead, as demonstrated by Rupniewski et al. [32].

Based on the literature, digital radar signal processing can be performed entirely on the GPU (including pre- and post-processing) without needing the CPU. However, whether this is feasible or the ideal design varies on a case-by-case basis. The most common implementation in the literature, however, was a heterogeneous CPU-GPU system, where GPU-acceleration was used for the parallel, computationally intensive signal processing algorithms, while the CPU controls the overall flow of the program, and also performs some kind of pre- and post-processing of the data.

Table 3.4 lists the processing platform used in each paper. In total, there were two GPU-only platforms and eight heterogeneous systems, of which seven were CPU-GPU and one was CPU-GPU plus FPGA.

Paper	**Processing platform**
Kong et al. [2]	GPU-only
Cai et al. [28]	GPU-only
Li et al. [27]	heterogeneous CPU-GPU
Liu et al. [26]	heterogeneous CPU-GPU
Zhao et al. [25]	heterogeneous CPU-GPU
Malanowski et al. [30]	heterogeneous CPU-GPU
Zhang et al. [31]	heterogeneous CPU-GPU
Dong et al. [29]	heterogeneous CPU-GPU
Deng et al. [33]	heterogeneous CPU-GPU
Rupniewski et al. [32]	heterogeneous CPU-GPU and FPGA

Table 3.4: Different processing platforms used in the papers.

3.2.4 Performance

Kong et al. used the time it took to complete the entire signal processing chain for each input data cube as the benchmark figure for their performance analysis. In at least half of the cases, the GPU ran faster than the corresponding CPU. In their real-time processing case analysis, the real-time requirement was to process an input data cube in less than 63 ms. The GPU met the real-time requirement in all of the cases, while the CPU only met it when the number of beams formed was low (less than 13) [2]. From their results, it can be observed that the CPU processing time increased almost linearly with number of beams, while the GPU processing time remained almost flat.

Li et al. compared the processing time of cross-correlation on CPU and GPU with various data sizes. In their results, the processing time of the GPU was much faster than that of the CPU irrespective of block number or batch length (even though the CPU processing was optimised for matrix multiplication). It was observed that the processing time of the GPU-based method was two to five times faster than the CPU-based method. The authors also conducted processing rate tests at 20 MHz (CPU and GPU) and 80 MHz (GPU only). At 20 MHz the CPU processing time was 1329.3 ms, which was close to the overall processing time of the GPU at 80 MHz (1389.5 ms). The GPU at 20 MHz had an overall time of 654.4 ms which was less than half of the CPU-only processing time. The GPU unsurprisingly had a much lower FFT and IFFT processing time. Although the GPU approach had additional download and upload processes (data transfers to and from CPU memory), it was still faster than the CPU-only processing overall [27].

Liu et al. compared GPU and CPU performance for MIMO radar signal processing algorithms. As the amount of data increased, the processing time for CPU (MATLAB) increased significantly. At data volume 200 million, the GPU-based MIMO radar signal processing algorithm was 100 times faster than the CPU pro-

cessing method. The authors also proposed an improvement to the time division MIMO radar signal processing algorithm, which sped up the processing time of the original algorithm by approximately 50 ms on the GPU. Finally, the authors used OpenMP-based stream parallel processing on the CPU side to further improve efficiency and system parallelism. In these results, the advantages of stream acceleration processing appeared as data volume increased. At data volume 200 million, the stream acceleration processing method was approximately 20 ms faster than the already improved GPU processing method. Comparatively, the MATLAB version took 19.807 seconds, which was approximately 150 times slower [26].

Zhao et al. tested a CPU serial algorithm and GPU parallel algorithm for processing passive bistatic radar signals. The execution time for each of the three algorithms (ECA-B, RD-processing, 2D-CA-CFAR) and the total processing time for the entire chain were measured. ECA-B duration was 0.122 seconds on the CPU while it was 0.009 on the GPU. RD-processing was 12.490 seconds on the CPU while it was 0.344 seconds on the GPU. 2D-CA-CFAR duration was 6.440 seconds on the CPU while it was 0.026 on the GPU. For the whole algorithm, the CPU time was 19.052 seconds and 0.379 seconds on the GPU. The speedups were: 14.36 times for ECA-B, 36.31 for RD-processing, 247.69 for 2D-CA-CFAR, and 50.34 for the overall signal chain. [25].

Malanowski et al. measured the execution time of their DVB-T and FM signal processing implementation. For FM signals, the authors reported processing times on the GPU of 3 ms for beamforming, 3 ms for filtering, 23 ms for crossambiguity, and 26 ms for CACFAR with an overall time of 55 ms. This compared to MATLAB (CPU), where the times were 170 ms, 5000 ms, 40000 ms, 7000 ms respectively and a total time of 52170 ms (calculated here as it was not reported). However, it is also worth noting that the authors used single-precision floating-point numbers to achieve better performance, while the MATLAB comparison used double precision. The

authors also reported a linear relationship between the execution and the number of beams with the slope coefficient below 1.0 [30].

Zhang et al. measured execution time on different platforms to evaluate the speedups of their GPU implementation. The CPU implementation was coded both in C and MATLAB. Overall, the MATLAB version was always the slowest, with the C version being marginally quicker. The quickest GPU time (139 ms) was as much as 69.48 times faster than the slowest CPU time (9658 ms) [31].

Dong et al. reported their processing time in and compared it to a MATLAB implementation. The CPU-GPU platform achieved processing times of 29.91 µs for HRRP, 98.21 µs for pulse accumulation, 93.72 µs for target detection, and 7.35 µs for objective to measure. This in comparison to 599.679 ms, 20.497 ms, 39.396 ms, 3.78, respectively, on the MATLAB side. Compared to the MATLAB implementation, the CPU-GPU heterogeneous platform was up to over 20049 times faster, and it was able to fully meet the real-time requirements [29].

Deng et al. reported the time cost of pulse compression on the GPU and on Intel MKL. With varying batch count, the processing time on the GPU ranged from 1.12s to 1.41s and remained relatively flat. Meanwhile, on the Intel MKL the processing time ranged from 2s to 4.5s, exhibiting worse performance and more variability. The GPU implementation was faster and had better stability than the Intel MKL-based CPU version [33]. Interestingly, the GPU implementation was in this case only two to four times faster.

Rupniewski et al. were able to achieve a throughput from the FPGA into GPU memory (RDMA) of around 3.6 GB/s, which was slightly below the theoretical limit. The authors reported a computational performance of 250 GFLOP/s for the downconversion, filtering, and decimation procedures. For the pulse compression the authors employed the `cufftExecC2C()` function from the cuFFT library with batching. They achieved a performance of 50 GFLOP/s for the FFT. According

to the authors, this low figure was mainly caused by the small length of their FFT transforms. The overall performance of the entire signal processing chain was reported to be 120 GFLOP/s, which was enough to meet the real-time requirement of the radar system in question. Further, according to the authors, it is not possible to utilize all of the theoretical computational power of a GPU in radar signal processing due to latency constrains [32].

Cai et al. compared performance times between CPU and GPU platforms. The GPU platform demonstrated significant acceleration, achieving up to 4.5 times speedup for Matched Filter and up to 10 times speedup for RMMSE when data size was sufficiently large. However, the authors also noted a memory size limitation on the GPU which prevented further gains [28]

Based on the findings, the achieved GPU speedup can be an order of magnitude or more (even up to multiple orders of magnitude [26][29]) superior. Many papers also noted that a CPU-based approach is not suitable for real-time processing [30][2] even with considerable effort [33]. However, such comparisons can be rather trivial as it is well established by now that GPUs outperform CPUs in massively parallel tasks. Additionally, such figures are naturally contingent on the specific CPU and GPU models used. Unfortunately, only one of the papers reported their GPU solution's performance in terms of FLOP/S. This is relevant to know as it captures how much of the theoretical maximum computational output of the GPU has been exploited. Based on these findings, it can be said that performance comparisons between the CPU and the GPU in radar signal processing has been done extensively in the literature. A more valuable comparison would be direct performance tests between the GPU and, e.g., another parallel architecture such as the FPGA. However, such a comparison falls outside the scope of this thesis.

3.2.5 GPU models

The papers specified the GPU model(s) utilized and typically highlighted some key specifications. The GPUs span a range of generations and performance levels. While some papers utilized high-performance models, such as the Tesla V100 (H. Dong et al. [29]), the Tesla T4 (G. Liu et al. [26]), and the RTX 3090 (X. Zhao et al. [25]), others used mid-tier GPUs, such as the GTX 750 Ti (F. Kong et al. [2]), the RTX 2060 (W. Li et al. [27]), and the GTX 660 (W. Deng et al. [33]).

Table 3.5 lists the GPU model(s) used in each paper. Most of the models listed belong to earlier GPU generations given the age of some of the papers. The most comparable model in terms of performance and specifications is the RTX 3090, used by Zhao et al. [25], which, like the A40 GPU used in this thesis, is designed on top of NVIDIA's Ampere microarchitecture.

Paper	**GPU model(s)**	**Release year(s)**
Kong et al. [2]	GTX 750 Ti, GTX TITAN Black	2014
Li et al. [27]	RTX 2060	2019
Liu et al. [26]	Tesla T4	2018
Zhao et al. [25]	RTX 3090	2020
Malanowski et al. [30]	GTX TITAN	2014
Zhang et al. [31]	GT 620M, Quadro 600, Tesla C2075	2012, 2010, 2011
Dong et al. [29]	Tesla V100	2017
Deng et al. [33]	GTX 660	2012
Rupniewski et al. [32]	Tesla K20	2012
Cai et al. [28]	GTX TITAN Z	2014

Table 3.5: Different GPU models used in the papers.

3.3 Conclusion

Based on the literature review findings, research question one (RQ1) can be answered as follows.

There is a growing body of research showcasing the successful integration of GPUs into radar signal processing systems. Most of these implementations utilize CUDA and are based on heterogeneous platforms combining CPUs and GPUs, though GPU-only approaches also appear in the literature. The adoption of GPUs differs across the various radar systems discussed in the literature, including phased array radars, MIMO radars, software-defined radars, and pulse-Doppler radars. However, the radar signal processing implementations encountered in the literature also remain relatively narrow, and there is room for exploring GPU-acceleration in more sophisticated radar systems.

Throughout the literature, the advantages of using GPUs for radar signal processing are consistently demonstrated. The primary benefits GPU-acceleration in this context are significantly faster data processing speeds, higher throughput, and improved real-time performance compared to CPU-based systems. GPUs can accelerate commonly used radar signal processing algorithms by up to several orders of magnitude when compared to the CPU. These factors could enable more efficient and capable radar systems at a cheaper cost compared to traditional devices (FPGAs and DSPs). However, there is a lack of direct comparisons and analysis of the choice between GPUs and its actual alternatives (FPGAs and DSPs) in the literature. Furthermore, some performance bottlenecks remain, such as data transfer latencies between the CPU and GPU.

The future of GPU usage in digital radar signal processing thus looks promising. The ease of higher-level CUDA programming and new software frameworks as well as advances in GPU hardware can be expected to drive further interest and adoption.

Author(s)	**Title**	**Published in**	**Year**	**Citations**
F. Kong et al.	"Real-time radar signal processing using GPGPU (general-purpose graphic processing unit)" [2]	*Radar Sensor Technology XX*	2016	3
X. Zhao et al.	"GPU-Accelerated Signal Processing for Passive Bistatic Radar" [25]	*Remote Sensing*	2023	0
G. Liu et al.	"MIMO Radar Parallel Simulation System Based on CPU/GPU Architecture" [26]	*Sensors*	2022	10
W. Li et al.	"Design of high-speed software defined radar with GPU accelerator" [27]	*IET Radar, Sonar & Navigation*	2022	2
J. Cai et al.	"Acceleration of advanced radar processing chain and adaptive pulse compression using GPGPU" [28]	*Proceedings of the 24th High Performance Computing Symposium*	2016	2
H. Dong et al.	"Research on parallel architecture design of radar real-time signal processing based on CPU-GPU heterogeneous platform" [29]	*IET International Radar Conference (IET IRC 2020)*	2020	0
M. Malanowski et al.	"Effective implementation of passive radar algorithms using general-purpose computing on graphics processing units" [30]	*2015 Signal Processing Symposium (SPSympo)*	2015	2
P. Zhang et al.	"Real-time signal processing for FM-based passive bistatic radar using GPUs" [31]	*2014 19th International Conference on Digital Signal Processing*	2014	5
M. Rupniewski et al.	"A Real-Time Embedded Heterogeneous GPU/FPGA Parallel System for Radar Signal Processing" [32]	*2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing*	2016	5
W. Deng et al.	"High frequency radar signal processing based on the parallel technique" [33]	*IET International Radar Conference 2015*	2015	4

Table 3.2: List of papers used in this literature review.

4 Design and implementation

This chapter presents the solution design and implementation. The solution's general structure is presented and some important areas are described in more detail.

4.1 Description

As seen in the literature review, there has been previous research into GPU-accelerated radar signal data processing applications. The results have tended to show that a GPU-based approach brings a significant performance improvement over a CPU-only approach. This has usually been attributed to the GPU's high parallel computation capacity and memory bandwidth advantage over the CPU. However, a known limiting factor in the GPU-based approach is the cost incurred from data transfers over PCIe (i.e., between device and host, as illustrated in the Figure 2.1) [34].

Real-time GPU data processing typically involves a CPU-centric approach, wherein the CPU plays a central role in orchestrating the network interface controller (NIC) to transfer the signal data into GPU memory. However, in this approach the CPU can become a bottleneck [35]. In this solution, NVIDIA's DOCA GPUNetIO library is used, which is a solution to this problem at the receiving end. It works by enabling direct data transfers between network interfaces and GPUs. This solution makes it possible to receive packets directly into GPU memory via RDMA over PCIe, i.e., without staging copies through CPU memory.

In a heterogeneous CPU-GPU platform such as the one utilized in this solution,

the CPU is responsible for controlling the overall flow of the program. The GPU receives the network packets into its memory and performs the preprocessing as well as the key signal processing algorithms such as the FFT. The program has a sequential execution flow, meaning that the output of one process is used as the input of a latter process along the processing chain. The radar signal processing task that is implemented in this solution is to find the maximum amplitude spectrum of the input signal.

4.2 Software architecture

The Figure 4.1 illustrates the high-level data flow of the application from beginning to end. Initially, the application configures the network interface card to receive packets. NVIDIA's DOCA and GPUNetIO are utilized to manage packet flows and interface with the GPU. The network packets are transferred from the NIC directly into GPU memory. The incoming packets are stored in what is referred to as a 'cyclic buffer' in GPU memory. Everything thus far is based on an NVIDIA example of DOCA and GPUNetIO usage [35] and everything henceforth is a custom build. Also in the receiving CUDA kernel, each VITA-49 data packet has its I/Q data samples extracted. This extraction is done by parsing the VITA-49 data packet structure to locate the start of the data payload field (i.e., the beginning of the I/Q data segment), as shown in the Figure 2.9, and then copying it into a designated cuFFT data structure located in global GPU memory for batch FFT processing. This is also where the I/Q samples are byte-swapped and type-converted into single-precision, floating-point numbers. The application collects a predetermined amount of I/Q data samples from n packets into a batch. Once a full batch of signal samples has been stored in the buffer, a device flag is set to indicate that the batch is ready for FFT processing. When this flag is checked by the host as true, it triggers a one-dimensional, complex-to-complex, batched FFT execution. The output of the

FFT is an array of complex numbers. Each element represents the amplitude and phase of a particular frequency component of the input signal. Once the FFT computation is complete, the result is post-processed. This involves transferring the buffer to the host, calculating the magnitude of each complex number obtained from the FFT transform to generate the magnitude spectrum, as well as normalizing the values and converting them into logarithmic scale. While doing these operations, the maximum magnitudes are kept track of so as to find the maximum amplitude spectrum of the signal.

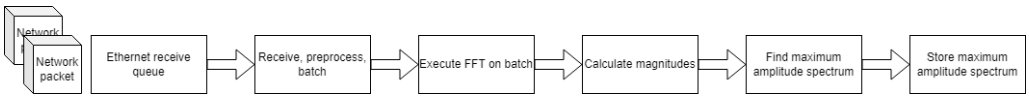

Figure 4.1: The solution's data flow in stages illustrated at a high-level.

4.2.1 NVIDIA DOCA

NVIDIA's *DOCA* (Data Center-on-a-Chip Architecture [36]) is a software development kit (SDK) and runtime environment for data centers. It enables developers to create applications that interface with NVIDIA BlueField DPUs (data processing units) and network interface cards (NICs) [37].

GPUNetIO

GPUNetIO is a library within DOCA, which enables remote direct memory access (RDMA) operations between NVIDIA GPUs and DPUs or NICs [38]. Direct memory access (DMA) enables a hardware device to access main system memory (i.e., DRAM or Dynamic Random Access Memory) without the involvement of the CPU. Remote direct memory access (RDMA) allows direct memory access (DMA) from one computer's memory to another's through a RDMA-capable network without the involvement of either machine's operating system or CPU [39]. Some popu-

lar RDMA protocols are InfiniBand and RoCE (RDMA over Converged Ethernet). RDMA technology is widely used in, e.g., high-performance computing (HPC) environments and data centers to improve data transfer speeds and to reduce latency.

4.2.2 Optimizations

Here, some of the optimizations performed in the solution are covered. Due to time constraints, the amount of optimization performed was limited.

Memory reuse: Memory reuse is implemented to avoid wasting memory space and to make efficient use of memory resources. The cuFFT library requires the allocation and freeing of address space. These are time-consuming operations which can even exceed the time it takes to perform the FFT operation itself [40]. Thus, address space is allocated only on the first call to the cuFFT library and only free it once all cuFFT calls are complete (i.e., once the program is stopped). This results in the subsequent FFT computations being slightly quicker than the first FFT execution.

4.3 Signal processing chain

4.3.1 Preprocessing

In order to be able to perform fast Fourier transform using NVIDIA's cuFFT library, the data must be formatted into a form and layout required by said library.

A VITA-49 data packet's payload field, as shown in Figure 2.9 earlier, consists of a contiguous sequence of n number of, in this case, 16 bit (I) + 16 bit (Q) I/Q data samples from a signal data stream. These signal data samples are extracted from the VITA-49 data packet payloads. Each of these signed integer samples is then byte-swapped, and converted into single-precision floating-point (FP32) values in round-to-nearest-even mode using the `__int2float_rn` CUDA function. While doing this,

the floating-point numbers are packed into a `cufftComplex` data structure, located in global GPU memory (using `cudaMalloc()`), in a layout required by the cuFFT-library for one-dimensional, single-precision, complex-to-complex FFT computation. This is done in a batched manner, i.e., by taking the c samples from n VITA-49 data packets each at a time, in order to enable batch FFT processing. The four-term Blackman–Harris window function discussed in Subsection 2.9.2 is applied to each data sample. The function is defined over a specific frame size; 4096 in this case, as that is the length of each individual FFT. The windowing is applied across the batch for each 4096-sample frame. The signal data being transmitted within the VITA-49 frames is a baseband signal, meaning that it is centered around zero frequency on the frequency axis. Baseband signals are original signals that have not been modulated to different frequencies for transmission [41].

4.3.2 cuFFT

The CUDA Fast Fourier Transform library or *cuFFT* is an NVIDIA library for GPU-accelerated FFT computing. It is based on FFTW3 (a collection of C routines for DFT computing) [42].

cuFFT supports various different types of inputs and configurations, and it can be used to compute complex-to-complex, complex-to-real, as well as real-to-complex transformations. The Table 4.1 lists the required data layout for different transform types. cuFFT kernels are optimized for input sizes of the form $2^a \times 3^b \times 5^c \times 7^d$. Performance will be better for smaller prime factors; as such, powers of two are the fastest to compute. Transforms can be made in half (16-bit floating point), single (32-bit floating point), and double (64-bit floating point) precision, and they can be either one, two, or three dimensional. Additionally, computation of multiple transforms simultaneously is supported in what are known as batched transforms. Batching results in better performance due to increased parallelism [43]. The cuFFT

library automatically chooses the optimal kernel configuration based on the specified FFT plan and GPU hardware [42][43].

When using cuFFT, one must create a *plan*, which is a type of execution strategy. It involves specifying the dimensions and configuration of the data to be transformed, as well as additional features pertaining to the FFT that the user can define. It is generally advisable to use the `cufftPlanMany` variant, as is done in this solution, due to there being negligible performance loss for a single transform and performance gain for batched transforms [44]. cuFFT plans are reusable, meaning they do not have to be destroyed and recreated every time for a new transform, as long as the desired transform remains the same. This improves performance by reducing the overhead associated with plan creation and destruction.

Importantly, the execution of the cuFFT library functions is initiated from the host side. In this solution, the function call to perform the Fast Fourier Transform (FFT) is carried out using `cufftExecC2C`.

FFT type	**Input data size**	**Output data size**
Complex-to-complex	x `cufftComplex`	x `cufftComplex`
Complex-to-real	$\left\lfloor \frac{x}{2} \right\rfloor + 1$ `cufftComplex`	x `cufftReal`
Real-to-complex	x `cufftReal`	$\left\lfloor \frac{x}{2} \right\rfloor + 1$ `cufftComplex`

Table 4.1: Expected data sizes for one-dimensional transforms in cuFFT. Table adapted from [43].

4.3.3 Post-processing

The output of a one-dimensional, single-precision, complex-to-complex cuFFT transform is a `cufftComplex` array of the same size as its input [43] containing complex values. At this point, it is still in device global memory. Once the FFT calculation is finished, the buffer holding the complex values is transferred to the host using

`cudaMemcpyDeviceToHost`. Once transferred, these values are stored in the host's RAM as a float array allocated with `malloc`. This FFT output array represents the signal in the frequency domain. Each complex number in the output corresponds to a different frequency component of the input signal. These values in the FFT output array are commonly referred to as *frequency bins*. The input signal's magnitude spectrum can be derived from the output by calculating the magnitude of each complex number.

The magnitudes of the complex values are calculated on the host, as described by the Formula 2.11 in Section 2.8, generating the magnitude or amplitude spectrum. Another manipulation that is performed on the output is normalization, where each element in the output array is divided by the sum of the window samples, as described by the Formula 2.13 in Subsubsection 2.9.2:

For $N = 4096$ and with the Formula 2.13, this sum S evaluates to approximately:

$$\sum_{n=0}^{4095} w(n) \approx 1469.44$$

Further, the values are converted into logarithmic scale (decibel) using the following formula [15]:

$$Y_{dB}[k] = 10log_{10}(|X[k]|^2) = 20log_{10}(|X[k]|) \tag{4.1}$$

This is done as the logarithmic scale provides a more understandable sense of the signal's frequency content, especially with signals that have a wide dynamic range.

4.3.4 Maximum amplitude spectrum

As described earlier, the goal of this solution was to find the maximum amplitude spectrum for the input signal. Determining the maximum amplitude spectrum is done by finding the peak magnitude value for each frequency bin k across multiple FFTs. This process results in a maximum amplitude spectrum, where each frequency

bin contains the maximum value observed for that frequency component across all segments.

4.4 Hardware platform

The Figure 4.2 illustrates the system's hardware configuration. A Linux computer acts as the transmitter, sending digitized signal data in network packets over a 100 GbE LAN connection at approximately 18 Gbps to the receiver, a Linux computer equipped with a discrete GPU. Both the transmitter and receiver utilize Mellanox NICs. The Table 4.2 provides the hardware specifications of the receiver.

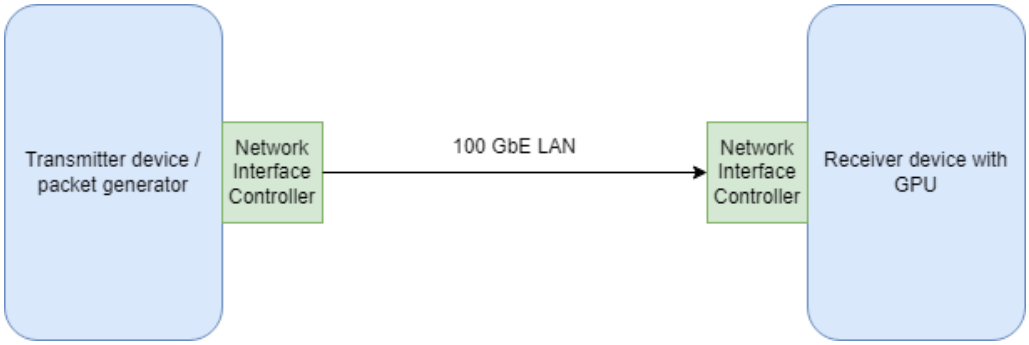

Figure 4.2: Hardware-level set-up for the system.

Component	**Specifications**
CPU	Intel Xeon Gold 8 core 3.7GHz Processor
RAM	128GB
GPU	NVIDIA A40 48GB
NIC	Mellanox ConnectX-6 Dx

Table 4.2: Hardware system parameters of the receiver platform.

GPU model

The specific GPU model used to test the digital radar signal processing solution is the NVIDIA A40 data center GPU. The Table 4.3 describes the relevant hardware specifications of this GPU model.

While the NVIDIA A40 has impressive theoretical FP32 performance (37.4 trillion floating-point operations per second), such a high value can be challenging to achieve in practice [32]. A high degree of GPU resource utilization requires high computational intensity and optimized memory access patterns from the implementation. There are also additional considerations about the closed-source nature of CUDA and its libraries. For example, the actual FFT performance will tend to vary depending on the specific workload characteristics, such as the FFT transform length [32].

Model name	NVIDIA A40
GPU architecture	Ampere
Peak FP32 TFLOP/s	37.4
CUDA cores	10,752
RAM	48GB GDDR6
Memory bandwidth	696 GB/s
Interconnect interface	PCIe Gen4: 64 GB/s

Table 4.3: Specifications of the NVIDIA A40 GPU [45].

5 Experimental evaluation

5.1 Experimental setting

The goal was to transfer a digitized 500 MHz signal, sampled at 700MSPS sample rate and 32-bit depth, over the local area network (LAN) to the GPU. The theoretical bitrate R_b can be calculated using the following formula:

$$R_b = f_s \cdot n_b \cdot N_c \tag{5.1}$$

where f_s is sample rate, n_b is bit depth, and N_c is number of channels.

In this case, the Formula 5.1 would give the following theoretical bitrate:

$$700000000 \cdot 32 \cdot 1 \approx 22\,\text{Gbps}$$

However, in the test environment, the highest bitrate that was measured over the network was approximately 18 Gbps. This can be explained by limitations on the data send side.

5.1.1 Test signal characteristics

The implementation was tested using a synthetic digital signal generated based on a real signal. The Table 5.1 lists the test signal's key characteristics.

Signal type	Synthetic, pulsed, periodic
Bandwidth	500 MHz
Sample rate	700 MSPS
Noise characteristics	White noise
Waveform shape	Sinusoidal

Table 5.1: Specifications of the test signal.

5.1.2 Performance tests

To evaluate the solution's performance, the time duration at two points of the signal processing chain was measured. A total of $n = 10$ trial runs were conducted, with each run lasting for $t = 10$ seconds. The measured points in the signal processing chain were the preprocessing stage, the complex-to-complex FFT stage, and the post-processing stage, which were described in Subsections 4.3.1, 4.3.2, and 4.3.3, respectively. The mean value and standard deviation were then calculated for each stage.

5.2 Performance

5.2.1 Packet reception rate

Network packets were successfully received at a rate of approximately 18 Gbps from the NIC into GPU memory via DMA over PCIe.

5.2.2 Processing time

The Table 5.2 presents the processing time measurements for the three stages. It is evident that, while the GPU-side preprocess and FFT calculation are fast and meet the real-time requirements, the CPU-side post-processing fails to do so, and acts as

a bottleneck. This issue persists even though the FFT output has been transferred to the CPU's random access memory (RAM). Upon investigation, it was found that a significant portion of the processing time is consumed by the logarithmic operation. Excluding this operation reduces the post-processing time to approximately 0.16 seconds, which, although faster, is still significantly slower than the GPU-side processing. To ameliorate the post-processing performance bottleneck, a multi-threading approach, inspired by Liu et al. [26], was also attempted using OpenMP. However, this approach resulted in even worse processing performance for reasons that remain unclear. More sophisticated CPU-side performance-tuning and analysis would have been necessary to have the post-processing stage meet the real-time requirements. For instance, using a lookup table for logarithmic computations could result in a substantial speedup. Implementing SIMD (Single Instruction, Multiple Data) could also be a worthwhile approach. Alternatively, a logarithmic approximation method could be employed, trading some accuracy for improved performance. However, these approaches were not explored due to time constraints in this thesis.

Processing stage	**Mean duration (SD)**
Preprocess	39 ms (0.067 ms)
C2C FFT	21 µs (1.3 µs)
Post-process	0.41 s (0.004 s)

Table 5.2: Processing times of the signal chain for $N = 4096$ and batch size 8160.

5.2.3 FFTW3 comparison

As described in Subsection 4.3.2, the cuFFT library is based on FFTW3. Therefore, an analogous single-precision C2C FFT transform using FFTW3 was implemented on the CPU side, utilizing the same machine (described in Table 4.2) and input signal, so as to compare cuFFT performance against it. The processing time was

again measured $n = 10$ times for $t = 10s$.

In these results, the FFTW3 implementation was markedly slower than the much larger (in terms of elements), batched cuFFT implementation. The Table 5.3 lists the FFT execution times for the single-threaded, non-batched (utilizing `fftw_plan_dft_1d()`) execution of size $N = 4096$. It is also noticeable that the CPU side FFT processing has more variability in its time measurements, a phenomenon also noted in the literature [33].

Processing stage	**Mean duration (SD)**
C2C FFT	0.041 s (6.1 µs)

Table 5.3: Processing time on the host side FFTW for one single-threaded C2C FFT where $N = 4096$.

The Figure 5.2 displays the measured cuFFT processing times, while the Figure 5.1 displays the single-threaded FFTW3 processing times.

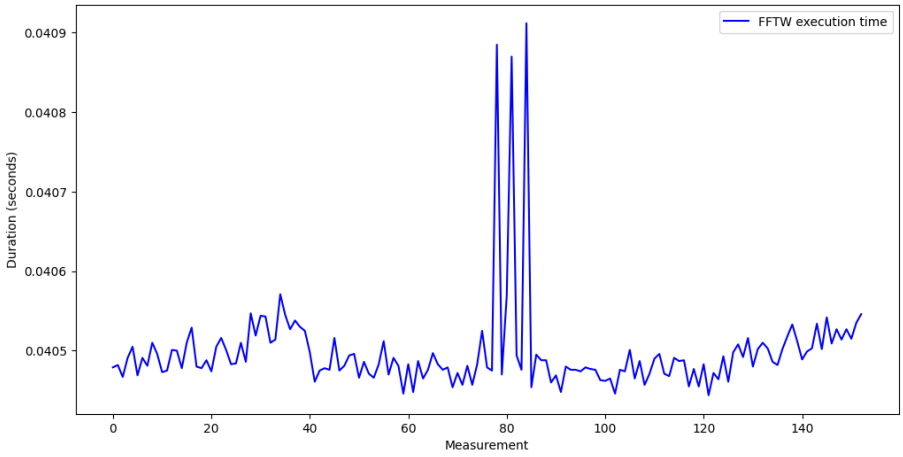

Figure 5.1: FFTW3 single-threaded performance measurements.

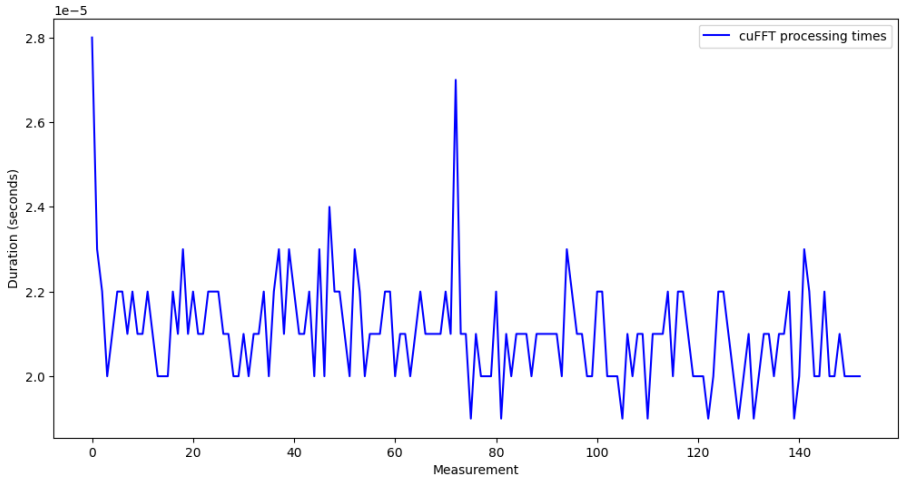

Figure 5.2: cuFFT performance.

Moreover, a batched version of FFTW3 (utilizing `fftw_plan_many_dft()`) with the same batch size was evaluated to directly compare with the cuFFT implementation used in the solution. The performance results, shown in Table 5.4, reveal that the CPU-based FFTW3 implementation is significantly slower than the GPU-accelerated cuFFT transform, by a factor of approximately 10^4. These measurements are illustrated in Figure 5.3. In the batched case, the high variability of the CPU-side processing times is again evident.

Processing stage	**Mean duration (SD)**
C2C FFT	0.237 s (0.0002 s)

Table 5.4: Processing time for single-threaded C2C FFT with batch size 8160 and $N = 4096$.

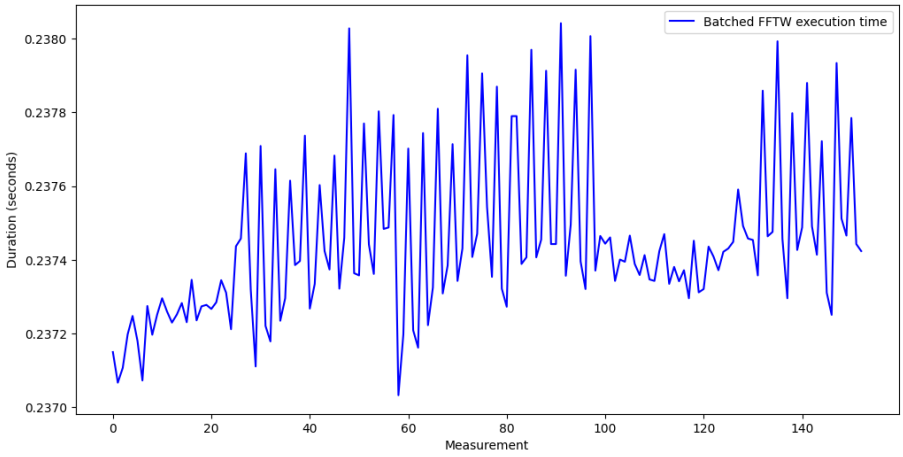

Figure 5.3: Batched FFTW3 performance measurements.

6 Conclusion

This chapter brings this master's thesis to a conclusion. In this work, a software-defined, real-time digital radar signal processing application was developed using the C and CUDA programming languages. The application was evaluated on a high-end NVIDIA A40 GPU by processing digitized wideband signal data.

6.1 Limitations

Many aspects of this work can be improved upon and extended. From a software engineering perspective, adopting a streaming architecture instead of a sequential one would improve performance by overlapping data reception and processing. There are also performance gains to be unlocked from optimizing the code and rethinking e.g., the task divisions. Some processing stages could be split into multiple separate kernels. Due to time constraints, not much time was spent on optimizing the solution. Post-processing could be implemented on the GPU side, which would improve performance and prevent the CPU from acting as a bottleneck.

Another aspect that was not discussed in this thesis is the problem of numerical accuracy. cuFFT has a higher numerical error than, e.g., Intel's Math Kernel Library (MKL) [46].

Additionally, the *cuFFT Device Extensions* (cuFFTDx) library, which enables the execution of FFT within custom CUDA kernels, could be utilized in place of the regular cuFFT. This approach could reduce latency by eliminating the need to

signal the host (CPU) when to initiate the FFT execution function [47].

The implemented radar signal processing chain is a simplified example of what might typically be involved in real-world signal data processing of a passive radar system. As such, the implementation is limited in its scope and complexity.

The system was tested using known signal data with specific characteristics. A test using multiple channels was not able to be implemented and instead a simulated approach was taken. Further, the solution was tested on one single NVIDIA GPU model, meaning reproducibility in terms of performance is not necessarily guaranteed on other GPUs with different hardware specifications or architecture.

6.2 Future work

Future work in this area could include testing radar signal processing algorithms on multi-GPU platforms as well as performing multi-channel processing. A very interesting topic would be direct comparisons between the GPU and the FPGA in various radar signal processing tasks. Investigating the power efficiency and performance trade-offs between these hardware platforms would also be valuable. Finally, implementing the GPU into more sophisticated modern radar systems could be investigated.

6.3 Summary

Research Question 1 (RQ1) was answered in Chapter 3, and Research Question 2 (RQ2) was answered in Chapter 5. Let us revisit these questions and summarize their answers.

RQ1: What is the current state of adoption of GPUs in real-time radar signal data processing?

There is a growing body of research demonstrating the successful integration of

GPUs into radar signal processing systems, predominantly using CUDA on heterogeneous platforms combining CPUs and GPUs. This adoption varies across different radar systems. However, the scope of GPU-accelerated radar signal processing implementations remains relatively narrow, indicating potential for further exploration in more advanced radar systems.

The literature highlights the advantages of GPUs in radar signal processing. These are significantly faster data processing speeds, higher throughput, and improved real-time performance compared to CPU-based systems. GPUs can enhance commonly used radar signal processing algorithms by up to several orders of magnitude compared to CPUs. Despite these benefits, the literature lacks direct comparisons between the GPU and its actual alternatives (notably the FPGA). Additionally, some performance bottlenecks are noted, particularly data transfer latencies between CPU and GPU.

To conclude, the future of GPU usage in digital radar signal processing appears promising. The ease of higher-level CUDA programming, new software frameworks, and advancements in GPU hardware are factors that are likely to drive further adoption.

RQ2: What is the performance of the GPU in real-time processing of wideband radar signal data?

The solution's evaluation focused on the GPU's performance in handling a digitized 500 MHz radar signal sampled at 700 MSPS with a 32-bit depth. The aim was to assess the GPU's ability to process this data in real-time, particularly during the preprocessing, complex-to-complex Fast Fourier Transform (C2C FFT) stage and the subsequent post-processing stage.

The preprocessing and C2C FFT stages on the GPU were found to be very efficient, meeting the real-time processing requirements. However, the post-processing stage on the CPU was significantly slower and became a bottleneck, particularly due

to the logarithmic operation involved. Attempts to optimize post-processing using multi-threading (OpenMP) did not yield better performance.

The GPU-side FFT performance was also compared against a CPU-side equivalent implementation in FFTW3. It was found that the GPU-accelerated cuFFT significantly outperformed the CPU-based FFTW3.

The GPU demonstrated excellent performance in the real-time processing of wideband radar signal data. The major limitation was identified in the CPU-side post-processing, which failed to keep pace with the GPU's processing speed. Addressing this bottleneck would likely require more advanced optimization techniques and possibly offloading more processing tasks to the GPU or using a more efficient optimization strategy on the CPU.

References

[1] H. Kim, R. Vuduc, S. Baghsorkhi, J. Choi, and W.-m. Hwu, *Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)*. Springer Cham, 2012, ISBN: 978-3-031-00609-8. DOI: 10.1007/978-3-031-01737-7.

[2] F. Kong, Y. R. Zhang, J. Cai, and R. D. Palmer, "Real-time radar signal processing using GPGPU (general-purpose graphic processing unit)", in *Radar Sensor Technology XX*, K. I. Ranney and A. Doerry, Eds., International Society for Optics and Photonics, vol. 9829, SPIE, 2016, p. 982 914. DOI: 10.1117/12.2222282. [Online]. Available: https://doi.org/10.1117/12.2222282.

[3] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, *Dissecting the nvidia volta gpu architecture via microbenchmarking*, 2018. eprint: arXiv:1804.06826.

[4] K. Adámek, J. Novotný, J. Thiyagalingam, and W. Armour, "Efficiency near the edge: Increasing the energy efficiency of ffts on gpus for real-time edge computing", *IEEE Access*, vol. 9, pp. 18 167–18 182, 2021. DOI: 10.1109/ACCESS.2021.3053409.

[5] NVIDIA, *Cuda c programming guide*, Accessed 2023-10-27. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[6] W.-m. W. Hwu, D. B. Kirk, and I. El Hajj, *Programming Massively Parallel Processors: A Hands-on Approach*, 4th. Morgan Kaufmann, 2022, ISBN: 978-0-323-91231-0.

[7] U. Farooq, Z. Marrakchi, and H. Mehrez, *Tree-based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization*. Springer New York, NY, 2012, ISBN: 978-1-4614-3593-8. DOI: 10.1007/978-1-4614-3594-5.

[8] C. Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Newnes, 2004, ISBN: 978-0750676045.

[9] S. Brown and J. Rose, "Fpga and cpld architectures: A tutorial", *IEEE Design & Test of Computers*, vol. 13, no. 2, pp. 42–57, 1996. DOI: 10.1109/54.500200.

[10] N. Eddy, "Fpga based digital signal processing for beam instrumentation - applications & techniques", in *Beam Instrumentation Workshop (BIW12)*, CERN, 2012. [Online]. Available: https://accelconf.web.cern.ch/BIW2012/talks/weap02_talk.pdf.

[11] M. Yasir, *Fpga or dsp processor - parameters to make the right choice*, Accessed: 2024-02-02, Dec. 2011. [Online]. Available: https://www.fpgarelated.com/showarticle/21.php.

[12] P. Laplante, *Comprehensive Dictionary of Electrical Engineering* (Electrical Engineering Handbook). Springer Berlin Heidelberg, 1999, ISBN: 9783540648352.

[13] A. Oppenheim and R. Schafer, *Discrete-Time Signal Processing: Pearson New International Edition*. Pearson Education, 2013, ISBN: 9781292038155.

[14] R. Brice, "4 - digital signal processing", in *Newnes Guide to Digital TV (Second Edition)*, R. Brice, Ed., Second Edition, Oxford: Newnes, 2003, pp. 85–111, ISBN: 978-0-7506-5721-1. DOI: https://doi.org/10.1016/B978-075065721-1/50005-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780750657211500050.

[15] S. M. Kuo, B. H. Lee, and W. Tian, *Real-Time Digital Signal Processing: Fundamentals, Implementations and Applications*. John Wiley & Sons, 2013, ISBN: 978-1-118-41432-3.

[16] M. Ramasubramanian, C. Banerjee, D. Roy, E. Pasiliao, and T. Mukherjee, "Exploiting spatio-temporal properties of i/q signal data using 3d convolution for rf transmitter identification", *IEEE Journal of Radio Frequency Identification*, vol. 5, no. 2, pp. 113–127, 2021. DOI: 10.1109/JRFID.2021.3051901.

[17] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series", *Mathematics of Computation*, vol. 19, no. 90, 1965.

[18] G. Heinzel, A. Rüdiger, and R. Schilling, *Spectrum and spectral density estimation by the discrete fourier transform (dft), including a comprehensive list of window functions and some new at-top windows*, Feb. 2002. [Online]. Available: https://hdl.handle.net/11858/00-001M-0000-0013-557A-5.

[19] E. W. Weisstein, *CRC Concise Encyclopedia of Mathematics*, 2nd. Chapman and Hall/CRC, 2002. DOI: 10.1201/9781420035223.

[20] F. Harris, "On the use of windows for harmonic analysis with the discrete fourier transform", *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, 1978. DOI: 10.1109/PROC.1978.10837.

[21] T. F. Collins, R. Getz, D. Pu, and A. M. Wyglinski, *Software-Defined Radio for Engineers*. Analog Devices, Inc., 2018, ISBN: 978-1-63081-457-1.

[22] R. Tell *et al.*, "Ieee std c95.2(tm)-2014", IEEE Standards Association, Tech. Rep., 2014.

[23] VITA, *About vita*, Accessed 2024-04-01. [Online]. Available: https://www.vita.com/AboutUs.

[24] ANSI/VITA, "Vita radio transport (vrt) standard for electromagnetic spectrum: Signals and applications", VITA, Standard 49.2, 2017.

[25] X. Zhao, P. Liu, B. Wang, and Y. Jin, "Gpu-accelerated signal processing for passive bistatic radar", *Remote Sensing*, vol. 15, no. 22, 2023, ISSN: 2072-4292. DOI: 10.3390/rs15225421. [Online]. Available: https://www.mdpi.com/2072-4292/15/22/5421.

[26] G. Liu, W. Yang, P. Li, G. Qin, J. Cai, Y. Wang, S. Wang, N. Yue, and D. Huang, "Mimo radar parallel simulation system based on cpu/gpu architecture", *Sensors*, vol. 22, no. 1, 2022, ISSN: 1424-8220. DOI: 10.3390/s22010396. [Online]. Available: https://www.mdpi.com/1424-8220/22/1/396.

[27] W. Li, C. Tang, S. Vishwakarma, K. Woodbridge, and K. Chetty, "Design of high-speed software defined radar with gpu accelerator", *IET Radar, Sonar & Navigation*, vol. 16, no. 7, pp. 1083–1094, 2022. DOI: https://doi.org/10.1049/rsn2.12244. eprint: https://ietresearch.onlinelibrary.wiley.com/doi/pdf/10.1049/rsn2.12244. [Online]. Available: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/rsn2.12244.

[28] J. Cai, Y. R. Zhang, F. Kong, and L. Li, "Acceleration of advanced radar processing chain and adaptive pulse compression using gpgpu", in *Proceedings of the 24th High Performance Computing Symposium*, ser. HPC '16, Pasadena, California: Society for Computer Simulation International, 2016, ISBN: 9781510823181. DOI: 10.22360/SpringSim.2016.HPC.008. [Online]. Available: https://doi.org/10.22360/SpringSim.2016.HPC.008.

[29] H. Dong, C. Zheng, and W. Tian, "Research on parallel architecture design of radar real-time signal processing based on cpu-gpu heterogeneous platform", in *IET International Radar Conference (IET IRC 2020)*, vol. 2020, 2020, pp. 323–329. DOI: 10.1049/icp.2021.0781.

[30] K. Szczepankiewicz, M. Malanowski, and M. Szczepankiewicz, "Effective implementation of passive radar algorithms using general-purpose computing on

graphics processing units", in *2015 Signal Processing Symposium (SPSympo)*, 2015, pp. 1–5. DOI: 10.1109/SPS.2015.7168277.

[31] P. Zhang, Y. Wu, J. Wang, and J. Qiao, "Real-time signal processing for fm-based passive bistatic radar using gpus", in *2014 19th International Conference on Digital Signal Processing*, 2014, pp. 536–540. DOI: 10.1109/ICDSP.2014.6900723.

[32] M. Rupniewski, G. Mazurek, J. Gambrych, M. Nałęcz, 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 (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, 2016, pp. 1189–1197. DOI: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP-SmartWorld.2016.0182.

[33] C. Zhang, Q. Yang, and W. Deng, "High frequency radar signal processing based on the parallel technique", in *IET International Radar Conference 2015*, 2015, pp. 1–4. DOI: 10.1049/cp.2015.1246.

[34] Y. Go, M. A. Jamshed, Y. Moon, C. Hwang, and K. Park, "APUNet: Revitalizing GPU as packet processing accelerator", in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, Mar. 2017, pp. 83–96, ISBN: 978-1-931971-37-9. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/go.

[35] NVIDIA, *Doca gpunetio*, Accessed 2023-10-27. [Online]. Available: https://docs.nvidia.com/doca/sdk/doca+gpunetio/index.html.

[36] SmartNICs Summit, *Smartnic definitions*, Accessed 2024-04-15. [Online]. Available: https://smartnicssummit.com/terminology/.

[37] I. Burstein, "Nvidia data center processing unit (dpu) architecture", in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–20. DOI: 10.1109/HCS52781.2021.9567066.

[38] M. Girondi, M. Scazzariello, G. Q. Maguire, and D. Kostić, "Toward gpu-centric networking on commodity hardware", in *Proceedings of the 7th International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys '24, Athens, Greece: Association for Computing Machinery, 2024, pp. 43–48. DOI: 10.1145/3642968.3654820. [Online]. Available: https://doi.org/10.1145/3642968.3654820.

[39] R. Dulong, "Towards new memory paradigms : Integrating non-volatile main memory and remote direct memory access in modern systems", Theses, Institut Polytechnique de Paris ; Université de Neuchâtel, Dec. 2023. [Online]. Available: https://theses.hal.science/tel-04426035.

[40] T. Yang, Q. Xu, F. Meng, and S. Zhang, "Distributed real-time image processing of formation flying sar based on embedded gpus", *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 15, pp. 6495–6505, 2022. DOI: 10.1109/JSTARS.2022.3197199.

[41] J. Rutenbeck, *Tech Terms: What Every Telecommunications and Digital Media Professional Should Know*, 3rd. Routledge, 2006, ISBN: 9780240807577.

[42] M. Denham, J. Areta, and F. G. Tinetti, "Synthetic aperture radar signal processing in parallel using gpgpu", *The Journal of Supercomputing*, vol. 72, no. 2, pp. 451–467, Feb. 2016, ISSN: 1573-0484. DOI: 10.1007/s11227-015-1572-z. [Online]. Available: https://doi.org/10.1007/s11227-015-1572-z.

[43] NVIDIA, *Cufft api reference*, Accessed 2024-03-27. [Online]. Available: https://docs.nvidia.com/cuda/cufft/index.html.

[44] D. Střelák and J. Filipovič, "Performance analysis and autotuning setup of the cufft library", in *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, ser. ANDARE '18, Limassol, Cyprus: Association for Computing Machinery, 2018, ISBN: 9781450365918. DOI: 10.1145/3295816.3295817. [Online]. Available: https://doi.org/10.1145/3295816.3295817.

[45] NVIDIA, *Nvidia a40 datasheet*, 2022. [Online]. Available: https://images.nvidia.com/content/Solutions/data-center/a40/nvidia-a40-datasheet.pdf.

[46] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors", in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12. DOI: 10.1109/SC.2008.5213922.

[47] NVIDIA Corporation, *Nvidia cufftdx documentation*, Accessed: 2024-06-21. [Online]. Available: https://docs.nvidia.com/cuda/cufftdx/index.html.