Acceleration of signal processing in modern radar based on GPU platform

Tran Van Anh*, Ha Huy Dung

Institute of Defense Equipment, Academy of Military Science and Technology, 17 Hoang Sam, Nghia Do, Hanoi, Vietnam.
*Corresponding author: anhstran87@gmail.com
Received 24 Dec. 2025; Revised 02 Mar. 2026; Accepted 11 May 2026; Published 25May 2026.
DOI: https://doi.org/10.54939/1859-1043.j.mst.111.2026.60-70

ABSTRACT

Modern radar systems aiming for high resolution, wide bandwidth, and real-time processing of large data volumes pose significant computational challenges to traditional signal processing approaches. Implementations based on CPUs, FPGAs (Field-Programmable Gate Array), or dedicated DSPs (digital signal processors) often fail to provide sufficient throughput and computational resources for intensive tasks such as matched filtering, fast Fourier transforms (FFTs), Doppler processing, digital beamforming, and synthetic aperture radar (SAR) image formation. To address these limitations, this paper proposes the use of graphics processing units (GPUs) as an acceleration platform for radar signal processing algorithms by exploiting the massive parallelism inherent in GPU architectures. The paper further presents performance measurements and evaluations conducted on representative radar datasets using various signal processing algorithms. The results demonstrate that GPU-based implementations can achieve speedups ranging from tens to hundreds of times compared to MATLAB-based CPU implementations. These findings indicate that GPU-accelerated signal processing is a promising solution for meeting real-time processing requirements in modern radar systems. In addition, computational complexity analysis and numerical accuracy validation between CPU and GPU implementations are provided to ensure the correctness and scientific rigor of the reported performance improvements.

Keywords: Radar; DSP; GPU Processing; GPU Acceleration; CUDA Programming.

1. INTRODUCTION

Radar signal processing pipelines have evolved significantly in recent years, incorporating increasingly sophisticated algorithms to support advanced functionalities such as high-resolution imaging, adaptive detection, and multi-dimensional parameter estimation. In response to these challenges, radar processing platforms have gradually transitioned from homogeneous, general-purpose processors toward heterogeneous computing paradigms that combine multiple types of processing units. Within this context, graphics processing units (GPUs) have attracted considerable attention due to their massive parallelism, high memory bandwidth, and rapid evolution driven by the consumer and high-performance computing markets. Previous studies have demonstrated the potential of GPUs for accelerating individual radar processing tasks, such as FFTs, pulse compression, and Doppler estimation [1–3]. However, a comprehensive evaluation of GPU-based acceleration across multiple radar algorithms and representative datasets remains an open research topic.

GPUs are designed with massively parallel architectures, originally intended for graphics rendering but now widely adopted in scientific computing. Compared to CPUs, which consist of a small number of powerful cores optimized for sequential operations, GPUs contain thousands of lightweight cores capable of performing parallel computations efficiently [4]. This architectural advantage aligns well with radar algorithms such as matched filtering, fast Fourier transforms (FFTs), Doppler processing, and image formation in synthetic aperture radar (SAR), all of which are inherently parallelizable [5–7]. Leveraging GPUs, researchers have reported significant

performance improvements, enabling real-time or near-real-time radar signal processing in operational scenarios.

Over the past decade, multiple studies have demonstrated GPU acceleration across various radar applications. For instance, passive bistatic radar implementations on GPU using CUDA Toolkit (Compute Unified Device Architecture) have achieved substantial speedups in clutter suppression, range–Doppler processing, and constant false alarm rate (CFAR) detection [8]. High-frequency surface wave radar has also benefited from GPU-based FFT acceleration for real-time signal processing [9]. In the context of synthetic aperture radar, both airborne and automotive SAR imaging pipelines have been successfully deployed on GPUs, reducing image formation time to the order of milliseconds [10-11]. Similarly, software-defined radar (SDR) platforms integrated with GPU accelerators have enabled high-speed and flexible implementations of pulse compression and Doppler processing [12].

In addition to standalone GPU solutions, heterogeneous CPU–GPU frameworks have been investigated to balance latency and throughput, achieving over two-fold acceleration compared to CPU-only systems [13]. More recently, embedded GPUs have been applied to unmanned aerial vehicle (UAV) platforms for real-time SAR imaging, highlighting the feasibility of deploying advanced radar processing algorithms on compact and power-constrained systems [14]. These advancements suggest that GPU-based computing not only enhances processing speed but also broadens the scope of radar applications, making sophisticated algorithms such as space–time adaptive processing (STAP), interference suppression, and compressed sensing feasible in real-time scenarios.

Despite the significant body of work on GPU acceleration for individual radar processing modules, most existing studies focus on either specific algorithms or highly optimized embedded platforms. A systematic evaluation that jointly considers algorithmic computational complexity, architectural characteristics of consumer-grade GPUs, and numerical validation against conventional CPU-based implementations remains limited. In particular, the relationship between theoretical asymptotic complexity and practical runtime behavior on parallel GPU architectures is not thoroughly analyzed in existing literature.

Therefore, this work aims to bridge this gap by providing a structured experimental evaluation of multiple representative radar processing stages on a unified GPU platform. The study integrates complexity analysis, architectural-level performance interpretation, and numerical accuracy validation to provide a comprehensive assessment of GPU-based acceleration in modern radar signal processing.

2. SIGNAL PROCESSING IN RADAR AND GPU SOFTWARE DEVELOPMENT

2.1. Introduction of signal processing system in radar

Figure 1 shows the block diagram of the signal processing module in the radar station using the linear frequency modulation signal LFM. The raw data of 2 channels I, Q from the ADC is stored in RAM as SHM_Raw_Data.dat memory. Then, the raw data are copied from the CPU to the GPU and a series of processing algorithms are performed on the GPU including:

- Algorithm for converting data from 16-bit integer to floating-point data;
- Algorithm for interference canceller;
- Algorithm for digital pulse compression;
- Doppler processing algorithm;
- CFAR detection processing algorithm.

For each processing stage, the data are copied from the GPU to the CPU to display and check the correctness of the data. The management of the test data is performed in the "Debug data manager" test data management module and is stored in RAM as a SHM_Debug_Data.dat file.

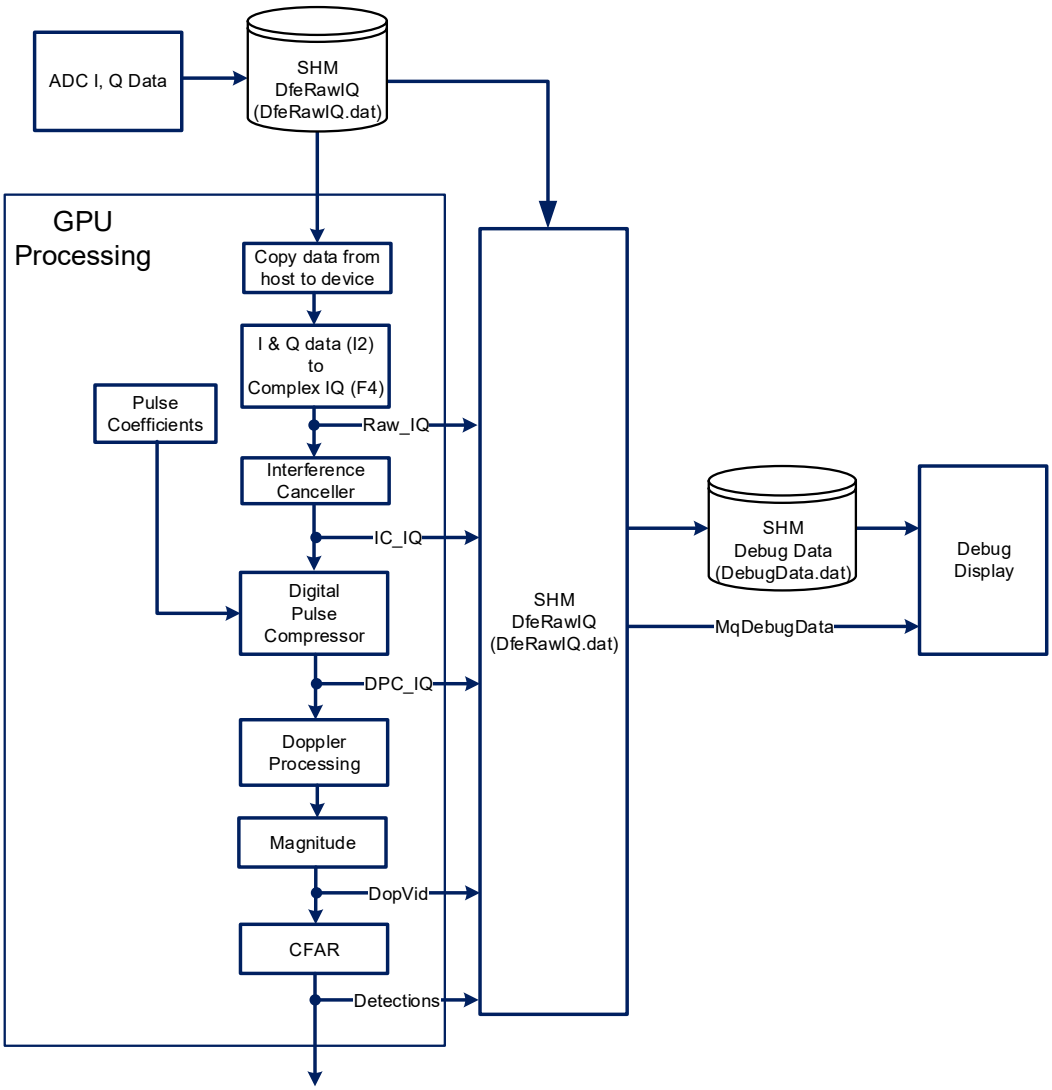


Figure 1. *Block diagram of DSP in radar using LFM.*

2.2. Software development in GPU

To increase the processing performance of the already optimized MATLAB only processor, we investigate the use of GPU technology. Our strategy for GPU software development was to use commercially available optimized libraries where possible. The modern CUDA runtime and compute architecture allows for extended flexibility and fine tuning in addition to an advanced assortment of optimized libraries, such as cuBLAS, cuFFT, etc. The runtime also supports an interface to OpenGL, which, in combination with GLUT, allows for the development of customized visualization interfaces.

2.2.1. Introduction of CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA. It allows developers to use GPUs for general-purpose computing by writing programs in languages such as C, C++, Fortran, and Python. By exploiting the massive parallelism of GPUs, CUDA enables significant performance acceleration for computationally intensive tasks in areas like scientific computing, signal processing, machine

learning, and real-time data processing.

Advantages of CUDA:

- Asynchronous copying;
- Streams and concurrency;
- Global Memory Coalescing;
- Locked Memory;
- CUDA Events.

Disadvantages of CUDA

- Development is more error prone and time consuming.
- Device side memory is unmanaged, which can complicate development and make code more error prone.
- Additional handling of host to device data transfers
- Knowledge of GPU architecture and CUDA runtime required.

2.2.2. Processing task scheduling & streams in CUDA

This paragraph describes the problem we encountered by describing how a typical GPU work package is broken up and scheduled for execution on the GPU. A GPU has at least one data copy engine and one kernel engine.

Data copy engine: The The primary task of the data copy engine is to move data to/from the GPU device. It is important to note that the copy engine is not used to move around or copy data on the GPU device. The hardware interface for data transfer on today's devices is mainly PCIe (Peripheral Component Interconnect Express). PCIe is a full duplex interface that allows for simultaneous data copy to & from a GPU device.

Kernel engine: The kernel execution engine is the only entity on the GPU that schedules & executes code through the CUDA runtime.

A stream in CUDA is a sequence of operations that execute on the device in the order in which they are issued by the host code. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, they can even run concurrently.

All device operations (kernels and data transfers) in CUDA run in a stream. When no stream is specified, the default stream (also called the "null stream") is used. The default stream is different from other streams because it is a synchronizing stream with respect to operations on the device.

GPU scheduling & execution: A typical GPU work package consists of the following three tasks:

- Copy data from host to GPU,
- Execute some GPU functions (kernels),
- Copy data from GPU back to host.

Detailed scheduling and synchronizing of the streams needs to be done to ensure that all the functions to perform have valid data and that data is not overwritten before it is completely processed. To effectively deploy processing algorithms on GPUs, it is essential to evaluate their computational complexity, memory access patterns, data transfer overhead between CPU and GPU, and real-time latency constraints. A systematic assessment of the applicability of GPUs within the radar signal processing chain provides critical insights into achievable performance gains, scalability, and energy efficiency. Such an evaluation plays a vital role in hardware design and selection, ensuring that GPU-based architectures meet both performance and real-time requirements of advanced radar systems. Section 3 provides a comprehensive performance

evaluation of GPU-based implementations in comparison with CPU-based approaches for classical signal processing algorithms in modern radar systems.

3. EVALUATIONS & PERFORMANCE MEASUREMENTS

To quantitatively assess the effectiveness of GPU acceleration in the modern radar signal processing chain, this section presents a comprehensive evaluation framework and corresponding performance measurements. The evaluation focuses on key computationally intensive radar processing stages, which are representative of real-world radar workloads. Performance is analyzed in terms of execution time, throughput, and acceleration factor relative to CPU-based implementations, while also considering data transfer overheads and resource utilization. The presented results provide objective insights into the practical benefits and limitations of deploying GPU-based solutions for real-time and high-resolution radar systems, thereby supporting informed hardware design and algorithm deployment decisions.

3.1. Input simulation data

To thoroughly assess the performance and computational behavior of the proposed algorithms, a dedicated input simulation dataset was constructed. The dataset is organized as a three-dimensional array of size 16384 × 128 × 2, where 16384 corresponds to the number of range bins after range compression, 128 represents the burst size typically associated with one coherent processing interval, and 2 denotes the number of consecutive bursts included for processing and performance comparison.

Each sample in the dataset is complex-valued and stored in floating-point format, capturing both in-phase and quadrature components with high numerical precision. This representation enables realistic modeling of amplitude fluctuations, phase histories, and noise-like characteristics commonly observed in wideband radar signals.

The selected dimensions reflect practical system parameters found in modern high-resolution radar modes, ensuring that the simulation accurately emulates real-world data volumes and processing loads. This design also allows meaningful evaluation of memory access patterns, parallelization efficiency, and throughput when implementing the algorithms on GPU-based architectures.

3.2. Methods and simulation tools

The performance evaluation is carried out by implementing the three radar processing algorithms on both MATLAB and CUDA, and subsequently comparing their execution times across the two computing platforms. The three core algorithms considered in this study include:

- Pulse compression using a Non-Linear Frequency Modulated (NLFM) waveform;
- Doppler coherent integration for velocity estimation;
- Amplitude extraction, which involves identifying the peak value among Doppler resolution cells and computing the logarithmic magnitude of the detected peak.

This comparative analysis enables a quantitative assessment of the computational efficiency achieved through GPU acceleration relative to traditional CPU-based processing.

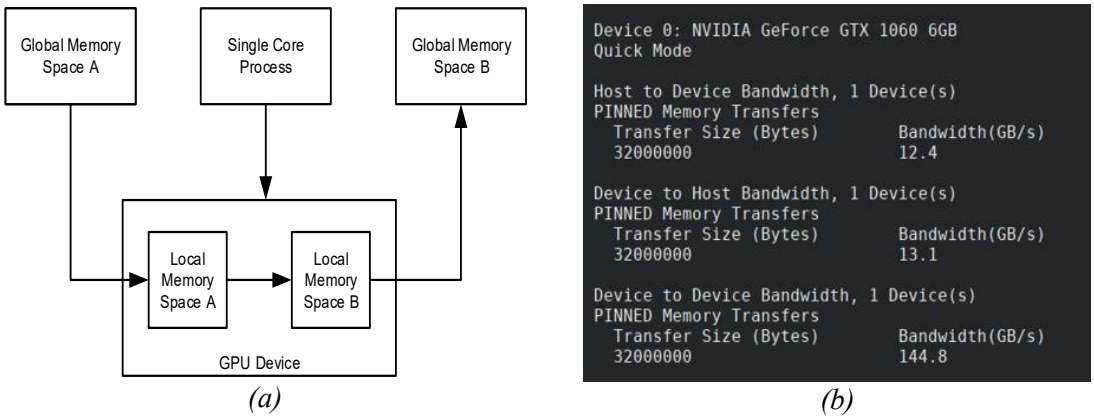
The hardware platform used for algorithm implementation consists of a workstation equipped with an NVIDIA GTX 1660 GPU for CUDA-based processing and an Intel Xeon E5-2686 v4 CPU for MATLAB execution. This configuration provides a representative comparison between GPU-accelerated computation and conventional multi-core CPU processing. The parameters of graphics card are shown in Figure 2. The host interface between the CPU and GPU is PCIe Gen3 x16, providing a theoretical maximum bandwidth of 15.76 GB/s.

Figure 3 illustrates the test setup used to measure the GPU's memory bandwidth. Two 64 MB shared memory blocks and two 64 MB local memory blocks were created, with random data

generated and stored in shared memory A. Bandwidth was then measured across three stages of data transfer, as depicted in the block diagram. The test programs were implemented in C/C++, compiled with GCC, and executed on a single CPU core. The CPU was responsible for loading the CUDA kernel onto the GPU device, after which data movement was managed by DMA engines through the PCIe interface, requiring minimal CPU intervention during transfer.

```
Device 0: "NVIDIA GeForce GTX 1060 6GB"
  CUDA Driver Version / Runtime Version          12.0 / 12.0
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 6072 MBytes (6367412224 bytes)
  (010) Multiprocessors, (128) CUDA Cores/MP:    1280 CUDA Cores
  GPU Max Clock rate:                            1772 MHz (1.77 GHz)
  Memory Clock rate:                             4004 Mhz
  Memory Bus Width:                              192-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        98304 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 4 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
```

Figure 2. *Parameters of GPU Graphics Card GTX 1660.*



(a) *(b)*

Figure 3. *Graphics card GTX 1660 bandwidth test, (a): block diagram and (b): test results.*

3.3. Simulation results and comments

Tables (1), (2), and (3), together with Figures (4), (5), and (6) illustrate the execution time of the aforementioned algorithms when executed on CUDA-GPU and MATLAB-CPU platforms with the number of simulation runs for each algorithm is set to 10. The average results show that CUDA achieves a speedup of 35.6 for digital pulse compression, 25.9 for Doppler processing, and 20.2 for MagMaxLog calculation. All test data are structured as three-dimensional matrices defined by range, burst length, and number of bursts. The input data format consists of complex numbers with two floating-point components (FP32 I and Q).

The results demonstrate that, for the same algorithm, execution on a GPU using CUDA provides substantially higher computational speed compared with MATLAB, even though MATLAB functions are generally well optimized. Overall performance improves further when

functions are fully implemented in CUDA, as GPU efficiency benefits not only from parallel processing but also from high-bandwidth data transfers between host and device. These advantages become even more pronounced when the GPU supports multiple copy engines.

Table 1. *MATLAB and GPU MagMaxLog processing time.*

Simulation index	**MagMaxLog Processing**		
	Processing Time, MATLAB, s	**Processing Time, GPU, s**	**Processing Time Gain, CUDA/MATLAB**
1	0.019089	0.000914	20.9
2	0.016556	0.000863	19.2
3	0.018058	0.000874	20.7
4	0.015548	0.000883	17.6
5	0.015868	0.001054	15.1
6	0.017706	0.000868	20.4
7	0.018866	0.000864	21.8
8	0.016992	0.000862	19.7
9	0.020128	0.000868	23.2
10	0.021151	0.000865	24.4
Average	**0.017996**	**0.000892**	**20.2**

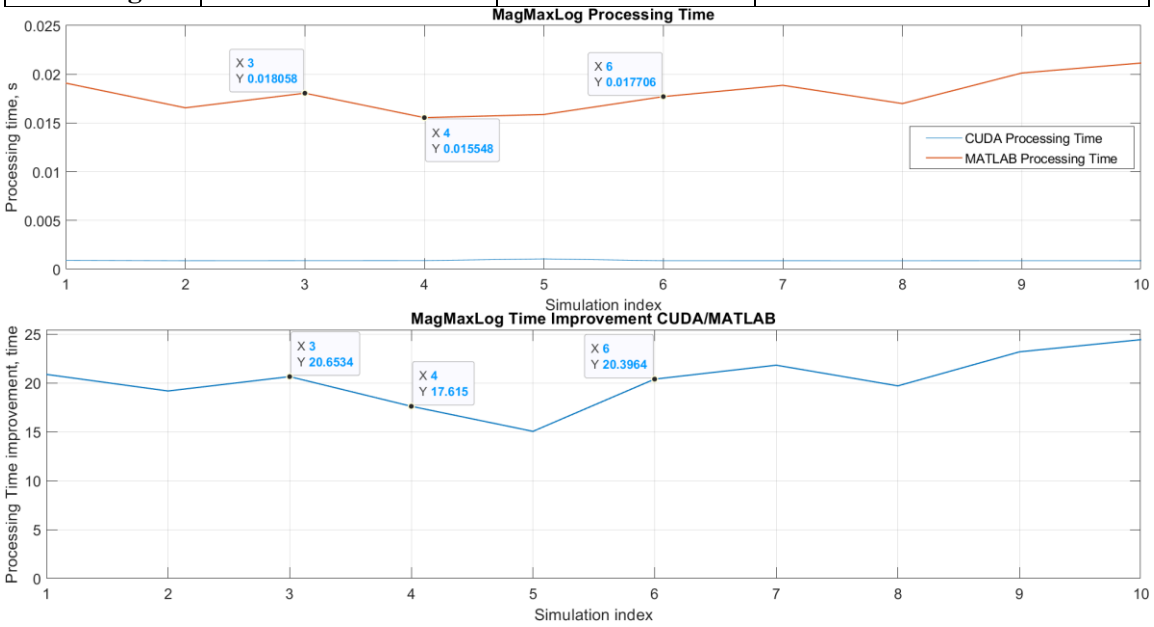


Figure 4. *Comparison of execution speed between MATLAB and CUDA for the MagMaxLog algorithm.*

Table 2. *MATLAB and GPU Doppler processing time.*

Simulation index	**Processing Time, MATLAB, s**	**Processing Time, GPU, s**	**Processing Time Gain, CUDA/MATLAB**
1	1.0962	0.0428	25.6
2	1.0510	0.0445	23.6
3	1.2602	0.0424	29.7

4	1.1941	0.0428	27.9
5	1.0340	0.0425	24.3
6	1.0819	0.0427	25.3
7	1.0434	0.0426	24.5
8	1.0864	0.0423	25.7
9	1.0533	0.0429	24.6
10	1.2268	0.0435	28.2
Average	**1.1127**	**0.0429**	**25.9**

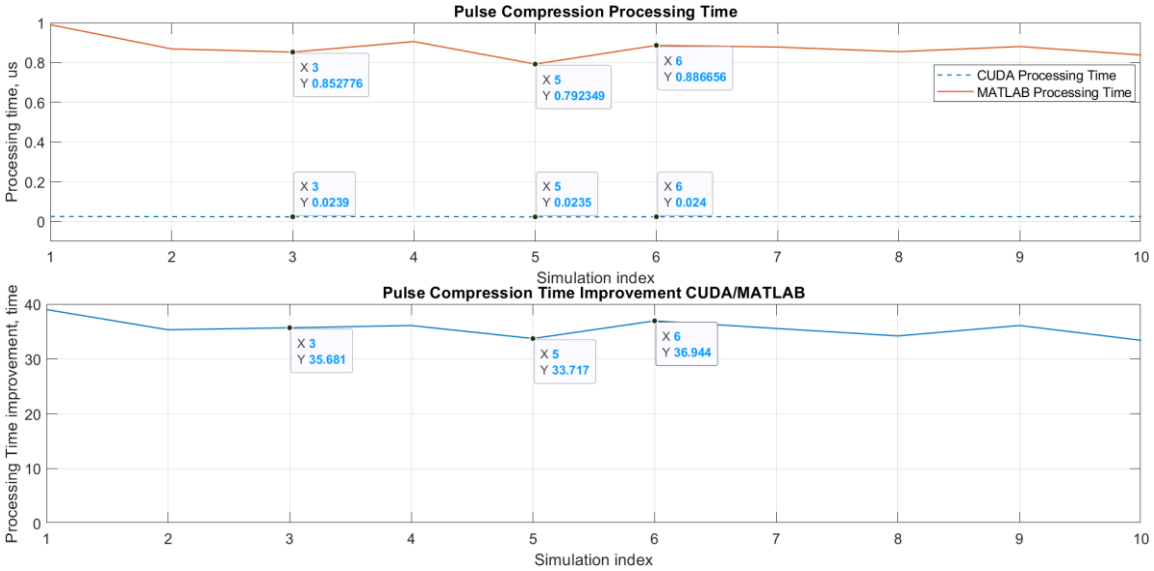


Figure 5. *Comparison of execution speed between MATLAB and CUDA for the Pulse Compression algorithm.*

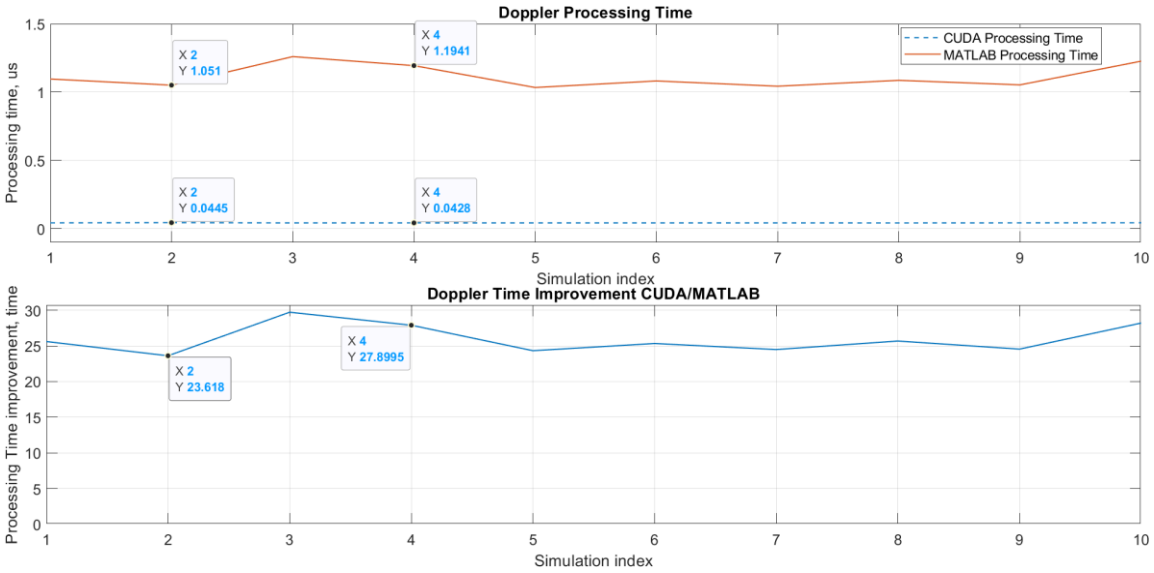


Figure 6. *Comparison of execution speed between MATLAB and CUDA for the Doppler Processing algorithm.*

Table 3. *MATLAB and GPU Pulse Compression processing time.*

Simulation index	Processing Time, MATLAB, s	Processing Time, GPU, s	Processing Time Gain, CUDA/MATLAB
1	0.9907	0.0254	39.0
2	0.8688	0.0246	35.3
3	0.8527	0.0239	35.7
4	0.9057	0.0251	36.1
5	0.7923	0.0235	33.7
6	0.8866	0.0240	36.9
7	0.8781	0.0247	35.5
8	0.8548	0.0250	34.2
9	0.8807	0.0244	36.1
10	0.8381	0.0251	33.4
Average	**0.8749**	**0.0246**	**35.6**

3.4. Computational complexity and accuracy validation

3.4.1. Computational complexity analysis

The core radar signal processing stages considered in this study, including pulse compression and Doppler processing, are implemented using FFT-based methods. For an input signal of length N, FFT-based pulse compression has a computational complexity of O(NlogN). Doppler processing, which involves FFT operations across slow-time samples, exhibits a similar computational order.

For a dataset of size $\mathrm{N}_r \times \mathrm{N}_p \times \mathrm{N}_b$, where N_r, N_p and N_b denotes the number of range bins, number of pulses (number of Doppler bins or burst size) within a coherent processing interval (CPI) and number of bursts, the overall computational complexity scales are calculated as following:

- For pulse compression algorithm:
  - Number of FFTs: $N_{FFT} = \mathrm{N}_p * \mathrm{N}_b$
  - Computational complexity for each FFT: $O(\mathrm{N}_r log_2 \mathrm{N}_r)$
  - Overall computational complexity: $O(\mathrm{N}_p * \mathrm{N}_b * \mathrm{N}_r log_2 \mathrm{N}_r)$
- For Doppler coherent processing:
  - Number of FFTs: $N_{FFT} = \mathrm{N}_r * \mathrm{N}_b$
  - Computational complexity for each FFT: $O(\mathrm{N}_p log_2 \mathrm{N}_p)$
  - Overall computational complexity: $O(\mathrm{N}_r * \mathrm{N}_b * \mathrm{N}_p log_2 \mathrm{N}_p)$
- The total computational complexity of the combined processing chain including pulse compression and Doppler processing can be expressed as:

$$O\left(\mathrm{N}_r * \mathrm{N}_b * \mathrm{N}_p * \left(log_2 \mathrm{N}_r + log_2 \mathrm{N}_p\right)\right)$$

Although theoretical complexity analysis indicates that the pulse compression stage has a higher asymptotic computational order due to the larger FFT size ($\mathrm{N}_r \gg \mathrm{N}_p$), practical execution time does not depend solely on asymptotic complexity.

In GPU implementations, performance is strongly influenced by memory access patterns, kernel configuration, FFT size optimization, and hardware utilization efficiency. In the present experimental results, Doppler processing exhibits longer execution time than pulse compression. This behavior can be attributed to the execution of a large number of smaller FFT operations along the slow-time dimension, which may result in lower GPU efficiency compared to large contiguous FFT operations in pulse compression.

Therefore, while pulse compression dominates in theoretical computational complexity, actual runtime performance depends on hardware-level optimization and data layout characteristics.

This complexity analysis confirms that the considered radar processing algorithms are computationally intensive and therefore well suited for parallel execution on GPU architectures. The inherent independence of FFT butterfly operations further enables effective exploitation of GPU thread-level parallelism. It should be emphasized that the objective of this study is to evaluate computational acceleration characteristics under controlled experimental conditions, rather than to provide a complete system-level optimization or hardware co-design analysis.

3.4.2. Accuracy validation between CPU and GPU implementations

To ensure that the reported acceleration does not compromise algorithmic correctness, the outputs of the GPU-based CUDA implementations are validated against the corresponding MATLAB CPU results.

The validation is performed using the mean squared error (MSE) metric between the complex-valued outputs obtained from both platforms. Experimental results indicate that the MSE values remain on the order of 10^{-6} to 10^{-7}. These discrepancies are attributed to floating-point rounding differences between CPU and GPU arithmetic units and do not affect the radar signal processing outcomes.

The results confirm that the GPU-based implementations preserve numerical accuracy while achieving significant computational speedup.

4. CONCLUSIONS

In this paper, we have presented an implementation of key radar signal processing algorithms on Graphics Processing Units (GPUs) using the CUDA programming model. The study has highlighted the significant advantages of GPU acceleration, particularly its ability to exploit massive parallelism for high-throughput computation, which is essential in modern radar systems with ever-increasing data rates and real-time constraints. The comparative experiments demonstrated that CUDA-based implementations achieve substantial speedups over conventional MATLAB-based processing, confirming the effectiveness of GPU acceleration for radar applications.

Despite these benefits, several challenges remain. CUDA programming requires careful memory management and algorithm adaptation to fully leverage GPU architectures, and performance gains depend strongly on hardware specifications and kernel optimization strategies. Furthermore, portability across different platforms and the learning curve associated with CUDA development can limit its adoption in some scenarios.

Overall, the results suggest that GPU acceleration is a powerful and practical approach for modern radar signal processing, offering remarkable improvements in execution speed while maintaining algorithmic accuracy. The complexity analysis further confirms the suitability of GPU architectures for computationally intensive radar signal processing tasks. Future work will focus on optimizing more advanced algorithms, exploring portability across heterogeneous computing platforms, and integrating GPU-based solutions into complete radar processing pipelines.

REFERENCES

[1]. Kong, Fanxing, Yan Rockee Zhang, Jingxiao Cai, and Robert D. Palmer. *"Real-time radar signal processing using GPGPU (general-purpose graphic processing unit)"*. Radar Sensor Technology XX, Vol. 9829, pp. 311-317, (2016).

[2]. Venter, Christian Jacobus. *"Software-defined pulse-doppler radar signal processing on graphics processors"*. PhD diss., University of Pretoria, (2014).

[3]. Yu, Xining, Yan Zhang, Ankit Patel, Allen Zahrai, and Mark Weber. *"An implementation of real-time phased array radar fundamental functions on a DSP-focused, high-performance, embedded computing platform"*. Aerospace, vol. 3, no. 3, p. 28, (2016).

[4]. NVIDIA. *"CUDA C Programming Guide"*. (2023).

[5]. Jin, Xingxing, and Seok-Bum Ko. *"GPU-based parallel implementation of SAR imaging"*. 2012 International Symposium on Electronic System Design (ISED), pp. 125-129, (2012).

[6]. Rupniewski, Marek, Gustaw Mazurek, Jacek Gambrych, Marek Nałęcz, and Rafał Karolewski. *"A real-time embedded heterogeneous GPU/FPGA parallel system for radar signal processing"*. 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), pp. 1189-1197, (2016).

[7]. Benson, Thomas M., Ryan K. Hersey, and Edwin Culpepper. *"GPU-based space-time adaptive processing (STAP) for radar"*. 2013 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1-6, (2013).

[8]. Zhao, Xinyu, Peng Liu, Bingnan Wang, and Yaqiu Jin. *"Gpu-accelerated signal processing for passive bistatic radar"*. Remote Sensing, vol. 15, no. 22, p. 5421, (2023).

[9]. Liu, Hang. *"Signal parallel processing in high frequency surface wave radar based on CUDA"*. International Conference on Electronic Information Engineering and Computer Science (EIECS 2022), vol. 12602, pp. 171-176, (2023).

[10]. Hoffmann, Marcel, Theresa Noegel, Christian Schüßler, Lars Schwenger, Peter Gulden, Dietmar Fey, and Martin Vossiek. *"Implementation of real-time automotive SAR imaging"*. 2023 20th European Radar Conference (EuRAD), pp. 327-330, (2023).

[11]. Yang, Tao, Xinyu Zhang, Qingbo Xu, Shuangxi Zhang, and Tong Wang. *"An embedded-gpu-based scheme for real-time imaging processing of unmanned aerial vehicle borne video synthetic aperture radar"*. Remote Sensing, vol. 16, no. 1, p. 191, (2024).

[12]. Li, Wenda, Chong Tang, Shelly Vishwakarma, Karl Woodbridge, and Kevin Chetty. *"Design of high-speed software defined radar with GPU accelerator"*. IET Radar, Sonar & Navigation, vol. 16, no. 7, pp. 1083-1094, (2022).

[13]. Zhao, Min, Qianshun Zou, Bing Sun, Youbin Song, and Jing Ma. *"CPU+ GPU architecture radar real-time signal processing method based on signal description technology"*. IET Conference Proceedings CP874, vol. 2023, no. 47, pp. 2365-2369, (2023).

[14]. Bu, Zirong, Lijun Wang, and Huijie Zhu. *"Research on GPU Parallel Acceleration of Efficient Coherent Integration Processor for Passive Radar"*. International Conference on Artificial Intelligence for Communications and Networks, pp. 415-422, (2021).

TÓM TẮT

Tăng tốc xử lý tín hiệu trong ra đa hiện đại trên nền tảng GPU

Các ra đa hiện đại hướng tới tính năng có độ phân giải cao, băng thông rộng và khả năng xử lý dữ liệu lớn thời gian thực đang đặt ra những thách thức tính toán đáng kể cho các phương pháp xử lý tín hiệu truyền thống. Các phương pháp triển khai dựa trên CPU, FPGA hoặc bộ xử lý tín hiệu số chuyên dụng (DSP) thường không đáp ứng đủ thông lượng và tài nguyên cần thiết cho các tác vụ có cường độ tính toán cao như lọc phối hợp, biến đổi Fourier nhanh (FFT), xử lý Doppler, tổ hợp búp sóng số và tạo ảnh ra đa khẩu độ tổng hợp (SAR). Nhằm khắc phục những hạn chế này, bài báo đề xuất sử dụng bộ xử lý đồ họa (GPU) như một công cụ tăng tốc cho các thuật toán xử lý tín hiệu ra đa trên cơ sở tận dụng kiến trúc song song quy mô lớn của GPU. Bài báo cũng đưa ra các đo đạc, đánh giá hiệu năng trên các tập dữ liệu ra đa tiêu biểu với các thuật toán xử lý khác nhau. Kết quả cho thấy sử dụng GPU cho phép tăng tốc độ xử lý từ hàng chục lần đến hàng trăm lần so với trên CPU. Điều này thể hiện giải pháp đề xuất xử lý tín hiệu trên GPU đảm bảo khả năng xử lý thời gian thực trong các đài ra đa hiện đại.

Từ khoá: Ra đa; Xử lý tín hiệu số; CUDA; GPU.