MIMO Radar Hardware Acceleration with Enhanced Resolution

Eric Pitre[1], *Vincent* Roberge[1], *Joey* Bray[1], and *Mostafa* Hefnawi[1]

[1]Royal Military College of Canada, Department of Electrical and Computer Engineering, Canada

Abstract. This paper proposes a method for accelerating an enhanced resolution 3D (range, velocity, and azimuth) Multiple Input Multiple Output (MIMO) radar on a Graphics Processing Unit (GPU). Implementation of MIMO radars and the investigation of their performance are of interest to the research community. However, the MIMO mode of operation increases the computational requirements of the radar system and seldom permits real-time operation without performance compromises. Current methods for achieving reasonable frame rates include reducing the scope of the radar (i.e., limiting the number of dimensions, the field of view, or the ranges of interest), choosing efficient but coarse algorithms (i.e., the FFT for range, velocity, and bearing estimation), or offloading the computation on task specific hardware, DSP, or FPGA. The proposed framework enables real-time operation of the MIMO radar by performing the signal processing on a GPU without compromising the radar coverage, while replacing the widely used 3D FFT with an enhanced resolution alternative. The proposed framework is tested on a Frequency Modulated Continuous Wave (FMCW) MIMO radar using 8 transmitters, 8 receivers, and having a Coherent Processing Interval (CPI) of 256 chirps. The parallel implementation of the enhanced resolution signal processing yields an acceleration of 453.6x when compared to sequential execution on CPU.

1 Introduction

Multiple Input Multiple Output (MIMO), in the context of radar, can be interpreted as a different way of scanning a Field of View (FOV). Instead of relying on mechanically steering an antenna or adjusting the phase of an array to transmit and receive in a particular direction, the MIMO radar illuminates the entire FOV at once and performs the beamforming on receive using multiplexing techniques [1]-[2]. Although the computational demands are greater, the MIMO radar can simultaneously evaluate echoes across the FOV without having to go through the lengthy scanning process required by classical surveillance radars [1], [3].

1.1 Motivation

Using Software Defined Radios (SDRs), a MIMO radar was built and tested at the Royal Military College of Canada (RMC) in 2021 which could operate in different multiplexing modes and yielded a relatively high refresh rate when compared to similar works [3]. However, the signal processing time was still significant, taking approximately 1.25 seconds using a 3D Fast Fourier Transform (FFT). The radar took 308 milliseconds to collect the 256-chirp Coherent Processing Interval (CPI), meaning that the signal processing takes 4.1 times longer that the acquisition time, slowing the radar considerably [3]. Additionally, range and angular measurement were taken in [3] to quantify the performance of the of radar. In general, the resolutions were larger than predicted in theory.

This work aims to realize a high refresh rate MIMO radar which leverages the computational power of parallel processing, improving the radar in [3]. Additionally, the FFT-based range and azimuth estimations are replaced by other algorithms to increase the quality of the radar images. The goal is to implement an algorithm which generates high resolution images in less than 1 second per frame. Table 1 shows the resolution measurements of the radar in [3].

Table 1. Measurement Results, summarized from [3]

Type	Desired Resolution	Measured Resolution
Range	0.75 m	1.75 m
Velocity	0.1 m/s	0.2 m/s
Azimuth	2°	10.5°

2 Problem Formulation

In MIMO radar, multiplexing techniques are used so that N_{tx} transmitters can simultaneously emit their waveforms. The echoes are sampled and sorted by N_{rx} receivers which generate a virtual array containing $N_{tx} \times N_{rx}$ elements [1]-[2]. Figure 1 shows a 2×4 MIMO Uniform Linear Array (ULA) configuration where 4 receivers demultiplex the echoes from 2 transmitters and generate 8 virtual elements. Note that in the figure, $\varphi = \beta d \sin(\theta)$ where θ is measured from broadside.

MIMO radar prototypes have been implemented [3]–[7] which all take advantage of the virtual array, made possible by multiplexing the transmitter waveforms. Prototypes have varying levels of complexity either in their dimensionality (range, azimuth, elevation, and velocity) and the number of receiving and transmitting antennas. A common issue for MIMO radar prototypes, with the exception of [4] and [7], are the slow refresh rates due to the extra required computations. The signal processing execution times of [3]–[6] vary from 1.25 seconds to 11 seconds which may not be considered real-time depending on the application. The radar prototypes in [4] and [7] are only 2D radars (range and azimuth) where the required signal processing is more manageable. Despite this, [4] offloaded the computation tasks on DSP and FPGA to minimize the signal processing delay.

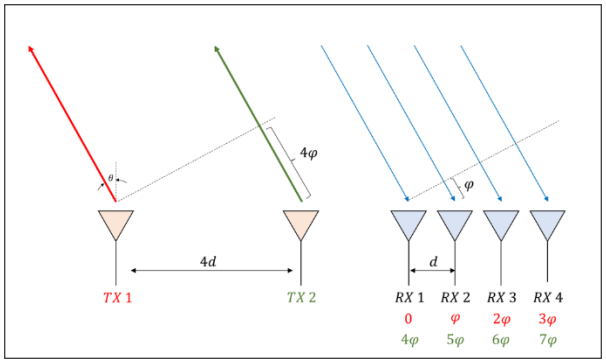

Fig. 1. MIMO Radar Virtual Array, adapted from [2]

2.1 Existing Signal Processing

In [3], the Frequency Modulated Continuous Wave (FMCW) radar uses a chirp time of 4.096 µs sampled at 250 MHz for a total of 1024 samples per chirp. With a CPI of 256 chirps and 64 virtual array elements, the collected data cube contains $1024 \times 256 \times 64$ or ~16.7M complex samples. The 3D FFT (or any signal processing alternative) is performed on the data cube in order to extract the target range, velocity, and azimuth information. Following this, the 3D data cube is compressed into two 2D images to display the range-bearing and range-doppler graphs.

The 3D FFT is simply a sequence of 1D FFTs across the three dimensions of the cube. The range FFT is performed for each virtual channel and each chirp (total of 16,284 FFTs of size 1024). Since only the *positive* ranges are of interest, only half of the range FFT outputs are kept (data cube now has a size of $512 \times 256 \times 64$). Following this, the doppler FFT is performed for all range cells and all virtual channels (total of 32,768 FFTs of size 256). Finally, the FFT beamformer is performed across all range-doppler cells (total of 131,072 FFTs of size 64). Once the 3D FFT is completed, the range-doppler and range-bearing images are generated by compressing the data cube. The output of a range-doppler pixel, for example, is the power sum across all bearings of the range-doppler bin index, as expressed in (1). To generate the entire range-doppler image, (1) needs to be performed for all combinations of range and doppler bin indices.

$$Out[r,d] = \sum_{b=0}^{N_b-1} |x[r,b,d]|^2 \qquad (1)$$

where:

Out = output pixels of range-doppler image
r = range bin index
d = doppler bin index
b = bearing bin index
N_b = number of bearing bins
x = 3D data cube samples

Figure 2 shows the radar frame computation steps, as done on a sequential CPU. Figure 3 illustrates the computation of a range-doppler pixel of the *Cube Compression* function.

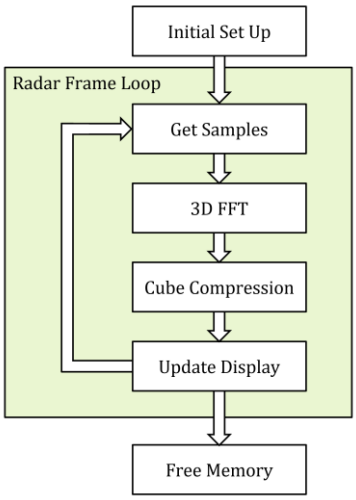

Fig. 2. Radar Frame Computations Steps using 3D FFT method

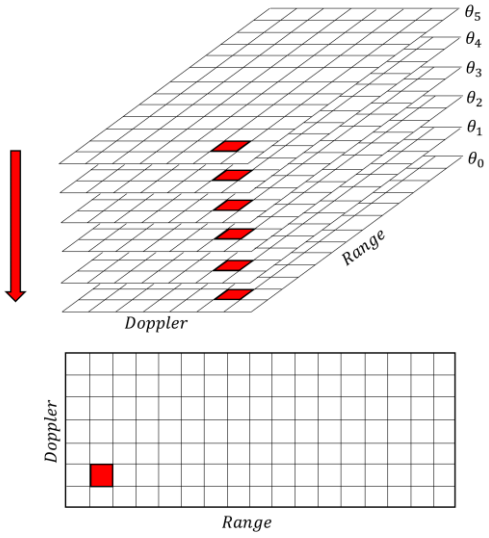

Fig. 3. Visualisation of Cube Compression for Range-Doppler Image

3 GPU Architecture

Graphics Processing Units (GPU) were optimized for gaming applications where developers needed to increase the throughput of operations to generate high quality graphics [8]. Since the computations required to generate image frames are highly independent, a large amount of them could be performed concurrently in parallel [8]. The GPU is able to run several hundred to several thousand of threads simultaneously, and is composed of several building blocks containing Streaming Multiprocessors (SM) [5]-[6]. These SMs have the necessary components to perform highly threaded computations such as Streaming Processors (SP), Special Function Units (SFU), and Shared Memory. The SM is interconnected to a Global Memory which receives and transfers data to and from the host computer [8]. Figure 4 shows the architecture of a typical GPU.

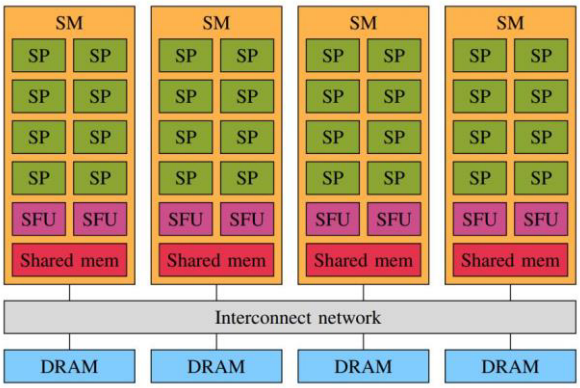

Fig. 4. GPU Architecture, reproduced from [10]

With Compute Unified Device Architecture (CUDA), a programmer can code sequential instructions which are to be executed on the CPU, and *kernels* which are functions that run in parallel on the GPU [9]. When a kernel is called from the CPU, threads are launched on the GPU which will execute the functions. Threads are organized in blocks and are mapped to SMs, which schedule and perform the parallel computations of up to 1024 threads per block [9]. All threads have a unique ID which locates them within the grid (all launched blocks form the grid).

4 Proposed Algorithm

In order to have more control over the algorithm and its parallel implementations, the existing GNU Radio 3D FFT-based signal processing chain implemented in [3] was entirely replaced by a C++ program.

4.1 Range Estimation

To increase the resolution of the range estimates, the FFT is replaced by the Chirp Z Transform (CZT) [11]. The CZT is a frequency analysis tool which is typically used as a *zoom* of the FFT. By selecting the frequency band of interest $f_0 \rightarrow f_1$, the CZT uses all N samples to generate an N-point DFT from f_0 to f_1. The CZT can be expressed as a convolution in time, which is then performed as a multiplication in the frequency domain [8]. Due to this, the FFT can be used for the transformations and its efficiency can be leveraged to reduce the execution time of the CZT. Figure 5 illustrates the CZT algorithm executed as a fast convolution, where $W = \exp(-j2\pi\Delta f)$

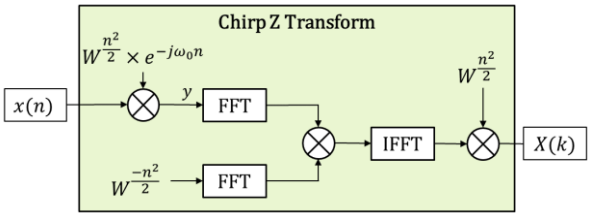

Fig. 5. Fast CZT Algorithm: Block Diagram

4.2 Azimuth Estimation

The Bartlett DOA, or traditional beamformer, is used as a replacement for the angular FFT. Although it is not a *high-resolution* DOA estimator, the calculations in (2) need to be performed for all angles of interest. Where an N-point spatial FFT provides N equally spaced frequency outputs, the Bartlett DOA performs the computation for M angles. Generally, M equally spaced angles are selected to generate the steering vectors [12], which contain coefficients $a[\theta, n] = e^{-j\beta d \sin(\theta_0 + n\Delta\theta)}$.

$$P[\theta] = \boldsymbol{a}[\theta]^H \boldsymbol{R}_x \boldsymbol{a}[\theta] \qquad (2)$$

where:

θ = steering angle
$P[\theta]$ = Bartlett power spectrum, for given angle
$\boldsymbol{a}[\theta]$ = steering vector, for given angle
$\boldsymbol{R}_x$ = estimated correlation matrix
$[\ \ ]^H$ = complex transpose

Since the radar only provides a single snapshot for each range-doppler cells, the matrix multiplications in (2) can be simplified as (3) where x is the receiver input for N array elements:

$$P[\theta] = \left| \sum_{n=0}^{N-1} a[\theta, n] \cdot x[n] \right|^2 \qquad (3)$$

4.3 Parallelization

The main parallelization opportunities lie within the sheer size of the data cube where each operation (range, velocity, and azimuth estimation) needs to be performed tens of thousands of times per radar frame. In the proposed algorithm, the final size of the processed cube is larger than the one for the existing solution. The CZT enables the use of all range samples, and the Bartlett DOA was configured to cover a FOV of $-90°$ to $90°$ with $1°$ increments. The proposed data cube has the following dimensions: $1024 \times 256 \times 180$ or $\sim$47.2M complex samples. Due to the new size, the *Cube Compression* function will also take longer to execute.

Note that before any signal processing can be done, the raw data cube samples must be transferred from the CPU to the GPU. This step takes time and must be considered when measuring the acceleration.

4.3.1 Chirp Z Transform

As shown in Figure 5, the CZT can be done in five steps. Three of these steps are just element-by-element multiplication and the other two steps are the FFT and the IFFT. Algorithm 1 shows the pseudocode associated with the execution of the parallel CZT. Lines 1, 3, and 5 are dedicated kernels which perform the multiplication steps where each launched thread maps to a sample of the 3D data cube. Lines 2 and 4 are the FFT and IFFT steps of the algorithm which use cuFFT plans. The plans are executed across the entire data cube in the *range* direction.

Algorithm 1. Parallel CZT on GPU

```
1:   Multiply_Kernel_1( )
2:
3:   // Do FFT
4:   cufftExecZ2Z(FORWARD)
5:
6:   // Fast Convolution
7:   Multiply_Kernel_2( )
8:
9:   // Do IFFT
10:  cufftExecZ2z(INVERSE)
11:
12:  // Final Multiply
13:  Multiply_Kernel_3( )
14:
15:  // End of Parallel CZT Algorithm
```

4.3.2 Bartlett DOA

Since the Bartlett DOA can be expressed as (3) for the single snapshot case, parallelization becomes quite natural. For each angle of interest, two kernels must be created: one which performs multiplication of the input samples with the steering vector, and another which performs the sum across N samples for each range-doppler combination. Algorithm 2 shows the parallel GPU implementation of the Bartlett DOA. Line 1 indicates that the subsequent steps are performed for all angles of interest. For a FOV of 180 degrees with increments of 1 degree, the parallel *Multiply* and *Sum* kernels (line 4 and 7) are performed 180 times sequentially. Note that the *Multiply* kernel of Algorithm 2 is different than the kernels from the CZT.

Algorithm 2. Bartlett DOA on GPU

```
1:   for all angles of interest {
2:
3:         // Perform element by element multiplication
4:         Multiply_Kernel( )
5:
6:         // Perform Sum across channel direction
7:         Sum_Kernel( )
8:   }
9:
10:  // End of Parallel Bartlett DOA
```

Figure 6 illustrates the execution of Algorithm 2 for a given angle θ. Note that the multiplication of the steering vectors is performed across the entire data cube. The cube is then compressed into a 2D sheet using the *Sum* kernel and stored in the output cube at the correct bearing location.

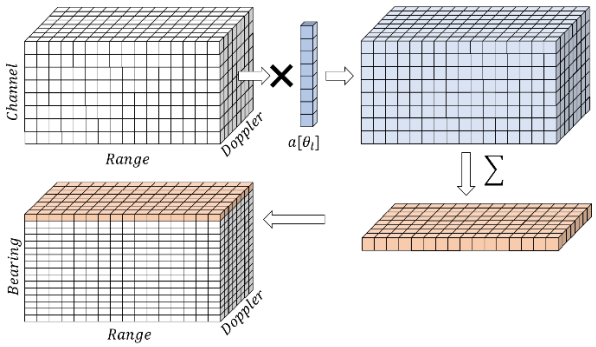

Fig. 6. GPU Implementation of Bartlett DOA

4.3.3 Cube Compression

As discussed in Section 2.1, the Cube Compression algorithm must generate the pixel values for the range-doppler and range-bearing images, which consists of performing many power sums. Since the calculation of a pixel is completely independent from the other pixels, each of them can be computed in parallel. Algorithm 3 shows the *Cube Compression* kernel, which is called form the CPU. In this kernel, each thread is mapped to a pixel of the range-doppler or range-bearing image. If the range-doppler image contains A pixels and the range-bearing image contains B pixels, a total of $A + B$ threads should be launched. Line 1 seeks to assign the thread ID to a variable. Lines 4 and 12 evaluate if the thread belongs to the range-doppler or the range-bearing image. Lines 7-8 is the power sum across all bearings, and lines 15-16 is the power sum across all doppler bins. The pixel outputs are stores in line 10 and 18 for the range-doppler and range-bearing images respectively.

Algorithm 3. Cube Compression Kernel

```
1:   id = thread ID within grid
2:
3:   // Do Range-Doppler Calculation
4:   if (id < A){
5:         find range and doppler bin indexes r and d
6:         temp = 0
7:         for all bearings b{
8:               temp = temp + |cube[r, d, b]|^2
9:         }
10:        out_d[r, d] = temp
11:  }
12:  if (id ≥ A and id < A + B){
13:        find range and bearing bin indexes r and b
14:        temp = 0
15:        for all doppler bins d{
16:              temp = temp + |cube[r, d, b]|^2
17:        }
18:        out_b[r, b] = temp
19:  }
20:
21:  // End of Cube Compression Kernel
```

4.3.4 GPU Implementation

Finally, the parallel GPU solution is inserted into the program. Once the radio samples have been collected, the raw data cube must be transferred to the GPU. From there, 3D signal processing is performed followed by Cube Compression. The range-doppler and range-bearing images are then transferred back to the CPU for display, completing the radar frame. Figure 7 shows the proposed implementation on GPU.

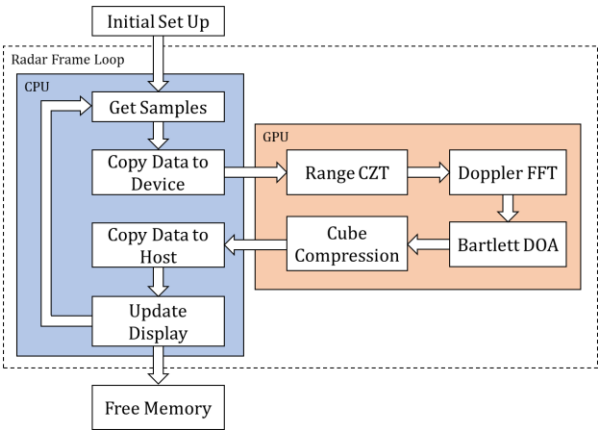

Fig. 7. Implementation of Proposed Algorithm

4.3.5 Parallel Work Efficiency

The parallel implementation of the proposed method is work efficient, where no additional computations on the GPU is required to execute the proposed algorithms. CuFFT is one of the most efficient FFT libraries available for the GPU. However, as with all GPU kernels, additional time is required to transfer the between the host and the device. To mitigate this overhead, the proposed solution limits the transfer of data to the beginning and the end of the radar frame calculations.

5 Simulation and Results

A simulated target echo environment was created to generate reflections for M targets with individual characteristics (i.e., range, speed, bearing, heading, RCS, etc.). To compare the image quality between the existing 3D FFT method and the proposed method, four targets with similar characteristics were simulated. The simulator outputs a raw data cube for the 8×8 MIMO radar with 256 chirps, each containing 1024 complex samples. This data cube is then processed to produce the radar images. As seen from Figure 8, the existing FFT method is not fine enough to resolve all four targets within the range-bearing image. The targets are clearly visible, however, when using the proposed algorithm.

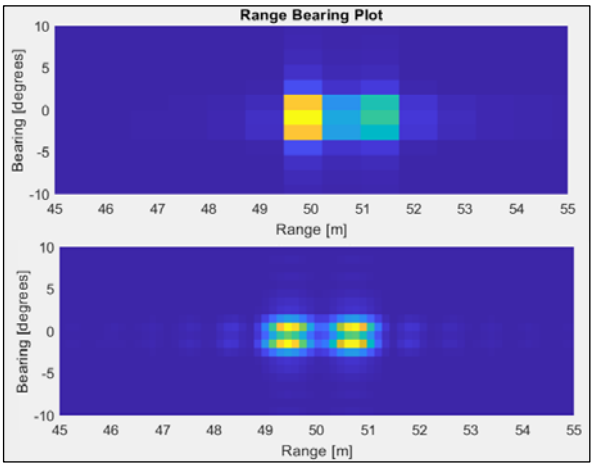

Fig. 8. Resolution Comparison: (above) 3D FFT Method (below) Proposed Algorithm

For speed up measurements, the proposed algorithm was performed sequentially, and in parallel on GPU. The workstation in fitted with two Intel Xeon Gold 5120 Processors and a NVIDIA GeForce RTX 2080 Ti GPU. The execution times for the proposed algorithm, the *Cube Compression*, and the memory transfer times are compared and displayed in Table 2. When implementing the proposed algorithm on GPU, an acceleration of 453.7x is achieved, compared to sequential execution on the CPU. The total execution time of the signal processing on GPU is 407.8 ms. For comparison, the existing 3D FFT method (including *Cube Compression*) takes approximately 3 seconds when done sequentially. In [3], GNU Radio somewhat leveraged the multiple cores of the CPU and was able to execute the 3D FFT and *Cube Compression* in 1.25 seconds.

Table 2. Execution Times and Speed Up

Process Name	Execution Time (ms)		Speed Up
	Sequential	GPU	
Proposed Algorithm	171,752	339.5	506x
Cube Compression	13,268	8.64	1536x
Memory Copy		59.7	
Overall	**185,020**	**407.8**	**453.7x**

6 Conclusion

In this paper, a parallel implementation of MIMO radar signal processing was proposed, which replaced the range and azimuth FFTs with the CZT and the Bartlett Beamformer respectively. By using a step-by-step approach which requires the computations to be performed simultaneously on the entire data cube, parallel execution on the GPU yielded a speed up of 453.7x when compared to the single-threaded sequential CPU implementation. The aim of producing high quality MIMO radar images in less than 1 second was achieved.

References

[1] J. Bergin and J. R. Guerci, *MIMO Radar Theory and Application*. Boston, MA, USA: Artech House, 2018.

[2] S. Sun, A. P. Petropulu, and H. V. Poor, "MIMO Radar for Advanced Driver-Assistance Systems and Autonomous Driving: Advantages and Challenges," *IEEE Signal Process. Mag.*, vol. 37, no. 4, pp. 98–117, Jul. 2020, doi: 10.1109/MSP.2020.2978507.

[3] R. T. Gilpin, "Real-Time Multiple Input Multiple Output (MIMO) Radar Using Sotware Defined Radio," Royal Military College of Canada, Kingston Ontario, 2021.

[4] W. Wang, D. Liang, Z. Wang, H. Yu, and Q. Liu, "Design and Implementation of a FPGA and DSP Based MIMO Radar Imaging System," vol. 24, no. 2, Art. no. 2, 2015.

[5] L. Sit, B. Nuss, S. Basak, M. Orzol, W. Wiesbeck, and T. Zwick, "Real-Time 2D+ velocity Localization Measurement of a Simultaneous- Transmit OFDM MIMO Radar using Software Defined Radios," Oct. 2016.

[6] A. Ganis, "Architectures and Algorithms for the Signal Processing of Advanced MIMO Radar Systems - CORE," Universita' Degli Studi Di Udine, 2018. Accessed: Mar. 09, 2022. [Online]. Available: https://core.ac.uk/display/195748925?recSetID=

[7] A. Figueroa, N. Joram, and F. Ellinger, "A fully modular, distributed FMCW MIMO radar system with a flexible baseband frequency," in *2021 IEEE Radar Conference (RadarConf21)*, May 2021, pp. 1–6. doi: 10.1109/RadarConf2147009.2021.9455174.

[8] D. B. Kirk and W. W. Hwu, *Programming Massively Parallel Processors*, Third. Cambridge, MA, USA: Elsevier, 2017.

[9] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming," in *2008 IEEE Hot Chips 20 Symposium (HCS)*, Aug. 2008, pp. 40–53. doi: 10.1109/HOTCHIPS.2008.7476525.

[10] G.-J. van den Braak, B. Mesman, and H. Corporaal, "Compile-time GPU memory access optimizations," in *Modeling and Simulation 2010 International Conference on Embedded Computer Systems: Architectures*, Jul. 2010, pp. 200–207. doi: 10.1109/ICSAMOS.2010.5642066.

[11] P. Rajmic, Z. Prusa, and C. Wiesmeyr, "Computational cost of Chirp Z-transform and Generalized Goertzel algorithm," in *2014 22nd European Signal Processing Conference (EUSIPCO)*, Sep. 2014, pp. 1004–1008.

[12] T. Yong and C. Wang, "Echo DOA Based High-resolution Target Location," in *2019 IEEE International Conference on Signal, Information and Data Processing (ICSIDP)*, Dec. 2019, pp. 1–4. doi: 10.1109/ICSIDP47821.2019.9173194.