

# **IMPLEMENTATION OF DIGITAL BEAMFORMER ON GPUs**

## ***ECE 612 Real Time Embedded Systems***

**ROHITH KUMAR KALISETTI**

**G01410445**

### **INTRODUCTION**

The field of digital beamforming, particularly when applied using General-Purpose Graphics Processing Units (GPGPUs), marks a significant evolution in array signal processing. In this domain, GPUs, traditionally used for rendering graphics, are repurposed for processing the data received from arrays of sensors. The primary goal of a beamformer is to align or phase-align signals from a targeted direction, enabling coherent signal addition. This selective amplification enhances the signal from the desired direction while diminishing those from others, thereby increasing the signal's clarity and effective range. This directionality is crucial in environments where the potential for interference is high, such as in densely populated urban areas or in radar operations where discerning between objects close together is essential. By improving the directionality of the signal, beamforming significantly enhances the clarity and reach of the communication, ensuring that the transmission is both powerful and precise.

The application of CUDA in beamforming processes brings a transformational change in how these computations are performed. GPUs, with their thousands of small, efficient cores capable of handling multiple operations simultaneously, are ideally suited to the parallelizable nature of beamforming tasks. This parallel processing capability allows beamforming operations, which involve large-scale computations to determine and adjust the phases and amplitudes of multiple signals, to be executed more rapidly and efficiently than ever before.

The application of GPUs in this context is not just about leveraging their computational might but also their architecture, which is inherently suited for parallel data processing. Unlike traditional CPUs, GPUs excel in handling multiple operations simultaneously, making them ideal for the computationally intensive task of beamforming. This involves processing large volumes of data from numerous sensors and synthesizing this information into a coherent output, often in real-time. The transition from using specialized hardware like DSPs or ASICs to more versatile, programmable platforms such as GPUs is driven by the CUDA technology developed by NVIDIA, which has democratized access to parallel computing capabilities, thus broadening the potential for innovative approaches in digital beamforming.

### **THEORETICAL BACKGROUND**

Beamforming is a signal processing technique employed in antenna arrays for directional signal transmission and reception. By manipulating the phase and amplitude of the signal at each antenna element, the array can create a pattern of constructive and destructive interference in the wavefronts. This interference pattern allows the beam to be "steered" toward specific directions without physically moving the antennas. The core principle behind beamforming is the phase shift—altering the phase of signals in a way that the signals at the receiver combine constructively, thereby enhancing the signal power in the desired direction, while signals that arrive out of phase combine destructively, reducing noise and interference from other directions.

The implementation of beamforming can be categorized into two approaches: analog and digital. Analog beamforming is performed using hardware like phase shifters, which adjust the signal phase at each antenna continuously. Digital beamforming, however, uses digital signal processing techniques to multiply the incoming signals by complex weights that adjust their amplitude and phase. The digital approach offers more flexibility and precision, allowing for multiple beams to be formed simultaneously and adjusted dynamically.

The mathematical backbone of digital beamforming is the Fast Fourier Transform (FFT), an algorithm that efficiently computes the Discrete Fourier Transform (DFT) of a sequence. FFT is invaluable in beamforming for converting signals from time domain to frequency domain and vice versa. This conversion is crucial for implementing frequency-domain beamforming, where the array's response is manipulated in the frequency domain to achieve the desired directional patterns. By using FFT, the computational complexity of performing DFT is dramatically reduced, making real-time signal processing feasible even when dealing with large data sets and complex operations typical in modern beamforming applications.

## MATHEMATICAL DESIGN FOR BEAMFORMER

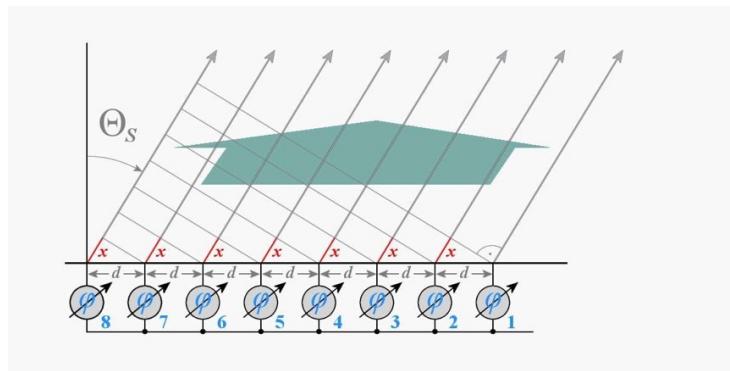


Figure 1 Phased Array Antennas

### Phase Delay ( $\theta$ )

For each antenna in the array, the phase delay is calculated to determine how the signal should be manipulated to achieve beam steering.

The formula used is:  $\theta = n(\text{antenna}) \times D(\text{delay}) \times T(\text{sine wave}) \times 2\pi$  Where:

- Antenna is presumably the index  $n$  of each antenna in the array.
- $D(\text{delay})$  is the delay per element, which might be determined by the antenna spacing and the desired beam direction.
- $T$  (sine wave) represents the period of the sine wave, which is inversely related to the carrier frequency  $f_0$ .

### Signal Representation

The signal at each antenna is represented by a sinusoidal function influenced by the phase delay.

The sine wave is represented as:  $\sin(\omega t + \theta)$

Where:  $\omega$  is the angular frequency corresponding to the carrier frequency  $f_0$  such that  $\omega = 2\pi f_0$ , t is time.

$\theta$  is the phase delay as described above. Antenna Array as shown on the above diagram.

**Elements:** A series of antennas (1, 2, 3, ...) arranged in a linear or planar array. Each element receives incoming signals and emits phase-shifted signals to form a collective beam in a specific direction.

### Signal Processing

**Digital Beamforming:** Digital signal processors (DSPs) apply calculated time delays (phase shifts) to the signals received or sent by each antenna element.

**Data Modulation:** A binary data signal (represented by "1 0 1") modulates the carrier signal to effectively turn each antenna element on or off.

### Phase Shift Calculation

**Equation:** The phase shift  $N_F$  for each element is given by

$N_F = n \times t \times f_0$ , where: n: Element index or identifier in the array. t: Time factor or period corresponding to the desired delay.  $f_0$ : The carrier frequency.

By adjusting  $N_F$ , the beam can be steered. The phase shift causes constructive and destructive interference, resulting in a focused beam in a particular direction.

### Modulation and Transmission

**Signal Generation:** Each antenna element generates a signal as  $\sin(2\pi f_0 t + N_F)$

**Multiplication:** The generated signal is multiplied by the data to encode information. 1 allows transmission (on), and 0 suppresses it (off).

### Steering Logic

**Calculations:** Determines the required phase shift for each antenna element based on the desired beam direction using the relationships:  $\sin(\theta) = D^x$  or  $\sin(\theta) = p^x$ .

Here, x represents the horizontal displacement for beam steering, and D or p denotes the distance related to the antenna array geometry.

**Control:** A controller adjusts the phase shift in real time to change the direction of the beam or to track a target.

**FFT/IFFT:** Fast Fourier Transform and Inverse FFT algorithms for frequency-domain processing if needed.

**Phase Shift Algorithms:** Calculates the precise phase delay for each antenna element based on the desired direction of the beam.

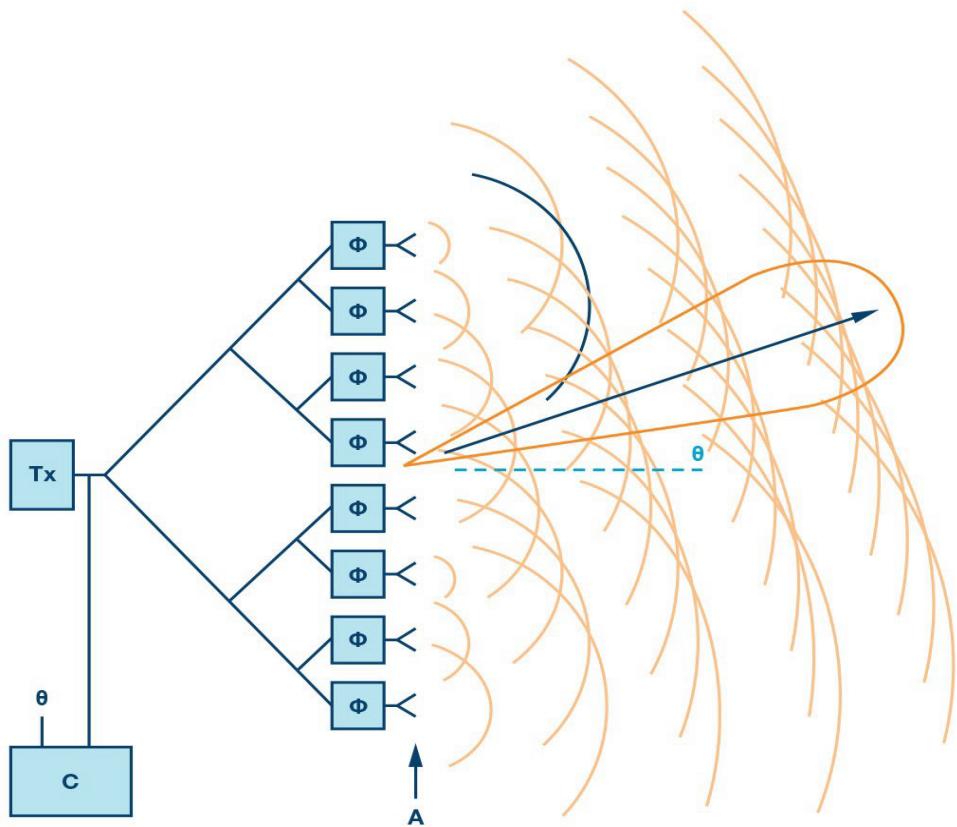


Figure 2 Phase Array Antennas where Beam is Steered in One Direction

## CUDA AND GPU COMPUTING

The application of GPUs in signal processing has transformed the capabilities of many systems, from radar signal processing to audio processing and beyond. GPUs excel in handling the vast number of computations required for signal processing algorithms due to their parallel architecture.

In signal processing, many operations, such as Fourier transforms, convolutions, and matrix multiplications, can be decomposed into independent tasks that can be executed concurrently. GPUs are inherently good at handling these tasks due to their hundreds of cores, which can compute thousands of operations per clock cycle. This makes GPUs particularly useful for real-time signal processing, where large volumes of data must be processed quickly to derive actionable insights or responses.

Furthermore, CUDA's support for fast Fourier transforms (cuFFT), which is an optimized implementation of the FFT algorithm, is crucial for many signal processing tasks. cuFFT allows for efficient transformation of signals between time and frequency domains, a process that is central to many forms of digital signal processing and is critical for implementing frequency-domain beamforming.

By combining the theoretical aspects of beamforming with the technical capabilities of modern GPUs and the CUDA programming model, it is possible to achieve significant improvements in both performance and efficiency. This synergy not only enhances the capabilities of traditional beamforming applications but also

opens up new possibilities in terms of complexity and scalability of signal processing tasks, paving the way for advanced array technologies in various high-demand sectors.

## ENVIRONMENTAL SETUP TO RUN THE PROGRAM

### Part 1: Setting Up Visual Studio for CUDA Development

#### Step 1: Install Visual Studio

- Download and install Visual Studio from the [official Microsoft website](#). For CUDA development, Visual Studio.
- During installation, select the "Desktop development with C++" workload.

#### Step 2: Install CUDA Toolkit

- Download the appropriate version of the NVIDIA CUDA Toolkit from the [NVIDIA website](#).
- Run the installer and ensure you select the Visual Studio Integration option. This step integrates CUDA with Visual Studio, allowing you to create and manage CUDA projects.

### Part 2: Using CUDA on a Hopper Cluster Step 1: Log In to the Cluster

- Use an SSH client to connect to the Hopper cluster.
- Replace 'username' and 'hopper.yourinstitution.edu' with your actual username and the cluster's address.

#### Step 2: Load CUDA Module

- Most clusters use module-based software management. To load CUDA, use **Module load cuda**
- Check the loaded modules to confirm CUDA is active: **Module listmodule**

### Part 3: Write and Compile the Code Step 1: Transfer Your CUDA Code

- Write your CUDA code on your local machine.
- Use SCP or an SFTP client to transfer your files to the cluster.

#### Step 2: Write a Makefile

- Create a Makefile in your project directory to simplify the compilation process. Here's an example Makefile for a CUDA project:

For this project the make file is given as:

```
beamformer : beamformer.cu
            nvcc -o beamformer beamformer.cu -lcu1
```

### Step 3: Compile and Run on the Cluster

- *Cat Makefile*
- *Make Beamformer.cu* Compilation and Execution:
- *./Beamformer <num\_antennas> <carrier\_frequency> <@me> <D\_delay>*
- *nvcc -o beamformer beamformer.cu*

- `./beamformer 1024 2.4e9 0.001 0.000001`

## CODE OVERVIEW

This CUDA program is designed for signal processing in an antenna array, particularly for beamforming tasks which combine signals from multiple antennas to direct a beam towards a target direction.

**Program Structure:** Detailed breakdown of the main components of the program, including the purpose of each part.

**Signal Generation:** signals are generated with specific phase delays and data modulation using CUDA kernels. Here is the code snippet for this.

**Include Necessary Libraries:** Standard input-output operations, CUDA runtime, cuFFT library, mathematical functions, and timing functions are included.

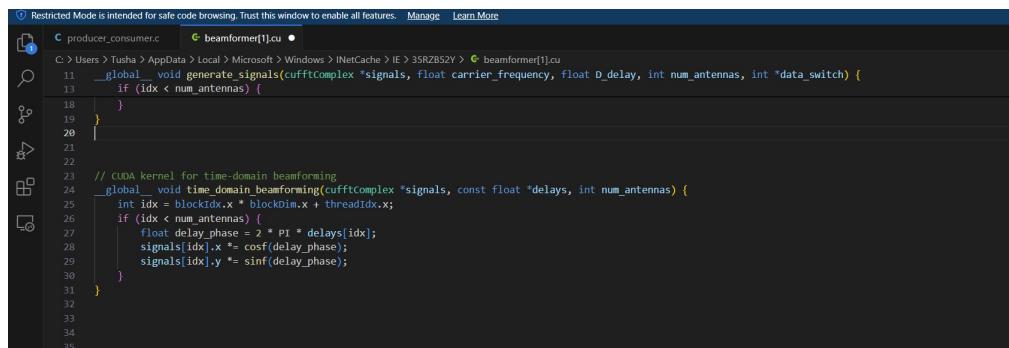
**Define Constants:** PI is defined to handle mathematical computations involving circles and periodic functions.

```
C:\> Users > Tusha > AppData > Local > Microsoft > Windows > INetCache > IE > 35RZB52Y > beamformer[1].cu
1 #include <iostream>
2 #include <cuda_runtime.h>
3 #include <cuFFT.h>
4 #include <cmath>
5 #include <vector>
6 #include <chrono>
7
8 #define PI 3.14159265358979323846
9
10 // CUDA kernel for signal generation with phase delays
11 __global__ void generate_signals(cufftComplex *signals, float carrier_frequency, float D_delay, int num_antennas, int *data_switch) {
12     int idx = blockIdx.x * blockDim.x + threadIdx.x;
13     if (idx < num_antennas) {
14         // Generate signal with phase delay and apply data switch
15         float phase = idx * D_delay * 2 * PI;
16         signals[idx].x = cosf(2 * PI * carrier_frequency + phase) * data_switch[idx];
17         signals[idx].y = sinf(2 * PI * carrier_frequency + phase) * data_switch[idx];
18     }
19 }
20
```

### CUDA Kernels:

- **generate\_signals:** This kernel generates modulated signals with phase delays.
- **time\_domain\_beamforming:** kernel applies time-domain beamforming techniques.
- **frequency\_domain\_beamforming:** This applies complex weights in the frequency domain to signals that have been transformed by FFT.

Here is the code snippet:



```
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
producer_consumer.c beamformer[1].cu
C:\> Users > Tusha > AppData > Local > Microsoft > Windows > INetCache > IE > 35RZB52Y > beamformer[1].cu
11 __global__ void generate_signals(cufftComplex *signals, float carrier_frequency, float D_delay, int num_antennas, int *data_switch) {
12     int idx = blockIdx.x * blockDim.x + threadIdx.x;
13     if (idx < num_antennas) {
14
15
16
17
18
19
20
21
22
23 // CUDA kernel for time-domain beamforming
24 __global__ void time_domain_beamforming(cufftComplex *signals, const float *delays, int num_antennas) {
25     int idx = blockIdx.x * blockDim.x + threadIdx.x;
26     if (idx < num_antennas) {
27         float delay_phase = 2 * PI * delays[idx];
28         signals[idx].x *= cosf(delay_phase);
29         signals[idx].y *= sinf(delay_phase);
30     }
31
32
33
34
35
```

```

6
7 // CUDA kernel for frequency-domain beamforming
8 __global__ void frequency_domain_beamforming(cufftComplex *freq_domain_signals, const cufftComplex *beamforming_weights, int num_antennas) {
9     int idx = blockIdx.x * blockDim.x + threadIdx.x;
10    if (idx < num_antennas) {
11        // Apply the beamforming weight for each antenna in the frequency domain
12        freq_domain_signals[idx].x = freq_domain_signals[idx].x * beamforming_weights[idx].x;
13        freq_domain_signals[idx].y = freq_domain_signals[idx].y * beamforming_weights[idx].y;
14        freq_domain_signals[idx].x = freq_domain_signals[idx].x * beamforming_weights[idx].y;
15        freq_domain_signals[idx].y = freq_domain_signals[idx].y * beamforming_weights[idx].x;
16    }
17 }
18

```

**Error Handling:** The macros and inline functions are defined to handle errors from CUDA and cuFFT operations, ensuring robust error checking throughout the code.

```

// Error check macros for CUDA and cuFFT
#define gpuErrorCheck(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true) {
    if (code != cudaSuccess) {
        std::cerr << "gpuError: " << cudaGetErrorString(code) << " " << file << " " << line << std::endl;
        if (abort) exit(code);
    }
}

#define cufftErrorCheck(ans) { cufftAssert((ans), __FILE__, __LINE__); }
inline void cufftAssert(cufftResult code, const char *file, int line, bool abort=true) {
    if (code != CUFFT_SUCCESS) {
        std::cerr << "cufftError: " << static_cast<int>(code) << " " << file << " " << line << std::endl;
        if (abort) exit(code);
    }
}

```

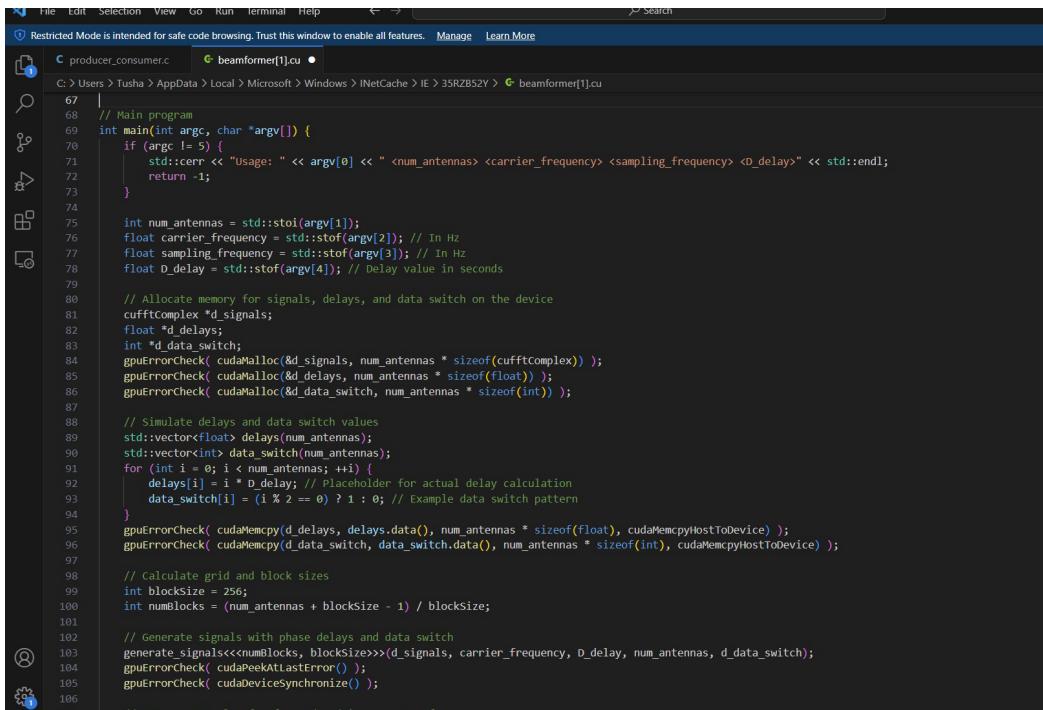
## Implementation Details

**Memory Management:** In the main function the strategies used for memory allocation, transfer between host and device, and deallocation.

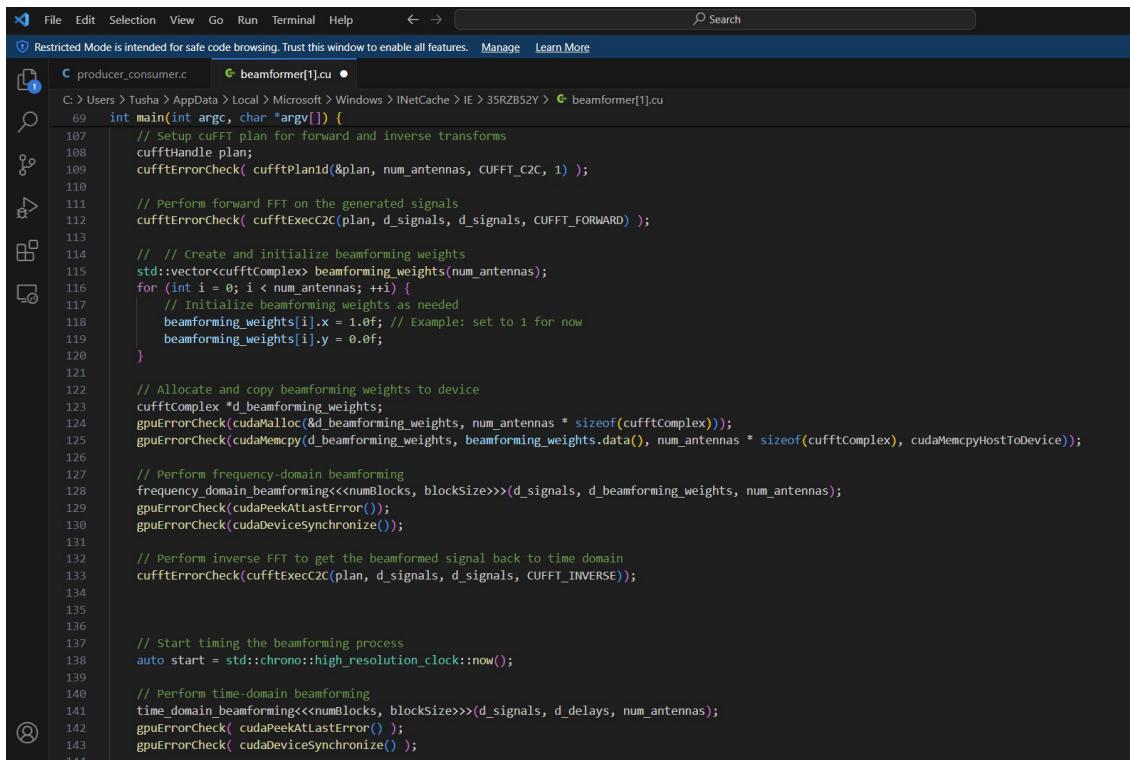
**Parallelization Strategy:** Analysis of grid and block sizes, and how tasks are divided among threads.

In the main function the code checks command-line arguments for proper usage.

- It parses input parameters for number of antennas, carrier frequency, sampling frequency, and delay per element.
- Now it allocates memory on the GPU for signals, delays, and data switching flags.
- After allocating memory it initializes data for delays and switching flags.
- Configures CUDA kernel launches with appropriate grid and block sizes.
- Then executes the signal generation kernel to modulate initial signals.
- Establishes an FFT plan and performs forward and inverse FFT for frequency-domain processing.
- Executes beamforming in both time-domain and frequency-domain.
- Now it measures the performance of time-domain beamforming operations using high-resolution clocks.
- Gives the outputs the processed signals for each antenna and cleans up resources to prevent memory leaks.



```
C:\Users\Tusha\AppData\Local\Microsoft\Windows\INetCache\IE>35RZB52Y> beamformer[1].cu
67
68 // Main program
69 int main(int argc, char *argv[]) {
70     if (argc != 5) {
71         std::cerr << "Usage: " << argv[0] << " <num_antennas> <carrier_frequency> <sampling_frequency> <d_delay>" << std::endl;
72     return -1;
73 }
74
75     int num_antennas = std::stoi(argv[1]);
76     float carrier_frequency = std::stof(argv[2]); // In Hz
77     float sampling_frequency = std::stof(argv[3]); // In Hz
78     float D_delay = std::stof(argv[4]); // delay value in seconds
79
80     // Allocate memory for signals, delays, and data switch on the device
81     cufftComplex *d_signals;
82     float *d_delays;
83     int *d_data_switch;
84     gpuErrorCheck( cudaMalloc(&d_signals, num_antennas * sizeof(cufftComplex)) );
85     gpuErrorCheck( cudaMalloc(&d_delays, num_antennas * sizeof(float)) );
86     gpuErrorCheck( cudaMalloc(&d_data_switch, num_antennas * sizeof(int)) );
87
88     // Simulate delays and data switch values
89     std::vector<float> delays(num_antennas);
90     std::vector<int> data_switch(num_antennas);
91     for (int i = 0; i < num_antennas; ++i) {
92         delays[i] = i * D_delay; // Placeholder for actual delay calculation
93         data_switch[i] = (i % 2 == 0) ? 1 : 0; // Example data switch pattern
94     }
95     gpuErrorCheck( cudaMemcpy(d_delays, delays.data(), num_antennas * sizeof(float), cudaMemcpyHostToDevice) );
96     gpuErrorCheck( cudaMemcpy(d_data_switch, data_switch.data(), num_antennas * sizeof(int), cudaMemcpyHostToDevice) );
97
98     // Calculate grid and block sizes
99     int blockSize = 256;
100    int numBlocks = (num_antennas + blockSize - 1) / blockSize;
101
102    // Generate signals with phase delays and data switch
103    generate_signals<<numBlocks, blockSize>>(d_signals, carrier_frequency, D_delay, num_antennas, d_data_switch);
104    gpuErrorCheck( cudaPeekAtLastError() );
105    gpuErrorCheck( cudaDeviceSynchronize() );
106
```



```
C:\Users\Tusha\AppData\Local\Microsoft\Windows\INetCache\IE>35RZB52Y> beamformer[1].cu
69 int main(int argc, char *argv[]) {
107     // Setup cuFFT plan for forward and inverse transforms
108     cufftHandle plan;
109     cufftErrorCheck( cufftPlan1D(&plan, num_antennas, CUFFT_C2C, 1 ) );
110
111     // Perform forward FFT on the generated signals
112     cufftErrorCheck( cufftExecC2C(plan, d_signals, d_signals, CUFFT_FORWARD) );
113
114     // Create and initialize beamforming weights
115     std::vector<cufftComplex> beamforming_weights(num_antennas);
116     for (int i = 0; i < num_antennas; ++i) {
117         // Initialize beamforming weights as needed
118         beamforming_weights[i].x = 1.0f; // Example: set to 1 for now
119         beamforming_weights[i].y = 0.0f;
120     }
121
122     // Allocate and copy beamforming weights to device
123     cufftComplex *d_beamforming_weights;
124     gpuErrorCheck( cudaMalloc(&d_beamforming_weights, num_antennas * sizeof(cufftComplex)) );
125     gpuErrorCheck( cudaMemcpy(d_beamforming_weights, beamforming_weights.data(), num_antennas * sizeof(cufftComplex), cudaMemcpyHostToDevice) );
126
127     // Perform frequency-domain beamforming
128     frequency_domain_beamforming<<numBlocks, blockSize>>(d_signals, d_beamforming_weights, num_antennas);
129     gpuErrorCheck( cudaPeekAtLastError() );
130     gpuErrorCheck( cudaDeviceSynchronize() );
131
132     // Perform inverse FFT to get the beamformed signal back to time domain
133     cufftErrorCheck( cufftExecC2C(plan, d_signals, d_signals, CUFFT_INVERSE) );
134
135
136     // Start timing the beamforming process
137     auto start = std::chrono::high_resolution_clock::now();
138
139     // Perform time-domain beamforming
140     time_domain_beamforming<<numBlocks, blockSize>>(d_signals, d_delays, num_antennas);
141     gpuErrorCheck( cudaPeekAtLastError() );
142     gpuErrorCheck( cudaDeviceSynchronize() );
143 }
```

```
C:\ File Edit Selection View Go Run Terminal Help ← → Search
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
producer_consumer.c beamformer[1].cu •
C:\Users\Tusha\AppData\Local\Microsoft\Windows\INetCache\IE>35RZBS2Y> beamformer[1].cu
69 int main(int argc, char *argv[])
70 {
71     // Initialize CUDA context
72     gpuErrorCheck( cudaSetDevice(0) );
73
74     // Perform inverse FFT to get the beamformed signal back to time domain
75     cufftErrorCheck( cufftExecC2C(plan, d_signals, d_signals, CUFFT_INVERSE) );
76
77     // Stop timing the beamforming process
78     auto end = std::chrono::high_resolution_clock::now();
79     std::chrono::duration<double, std::milli> elapsed = end - start;
80     std::cout << "Time-domain beamforming took " << elapsed.count() << " ms.\n";
81
82     // Copy the beamformed signals back to host
83     std::vector<cufftComplex> beamformed_signals(num_antennas);
84     gpuErrorCheck( cudaMemcpy(beamformed_signals.data(), d_signals, num_antennas * sizeof(cufftComplex), cudaMemcpyDeviceToHost) );
85
86     // Process output signals as required
87     // For example, print the real part of the first 10 beamformed signals
88     for (int i = 0; i < std::min(10, num_antennas); ++i) {
89         // std::cout << "Antenna " << i << ": Signal = " << beamformed_signals[i].x << " + " << beamformed_signals[i].y << "j" << std::endl;
90     }
91
92     // Cleanup
93     cufftErrorCheck( cufftDestroy(plan) );
94     gpuErrorCheck( cudaFree(d_signals) );
95     gpuErrorCheck( cudaFree(d_delays) );
96     gpuErrorCheck( cudaFree(d_data_switch) );
97
98     return 0;
99 }
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
```

## Resource Management and Cleanup

Properly deallocates all dynamically allocated GPU memory and destroys cuFFT plans to ensure there are no resource leaks.

## Execution Flow

- The program begins by setting up the environment and processing inputs.
- It proceeds to generate and process signals using both beamforming techniques.
- Outputs are calculated and printed, providing insights into the final signal states postbeamforming.
- The program concludes by cleaning up all allocated resources, ensuring a clean termination.

## RESULTS

Time domain output for 100 antennas on an array of a beamformer.

```
[tgarimel@gpu01 Project612]$ ./beamformer 100 1000 10000 0.001
Time-domain beamforming took 0.025478 ms.
PROBLEMS ② OUTPUT DEBUG CONSOLE TERMINAL PORTS
Antenna 0: Signal = 4409.24 + 590.76j
Antenna 1: Signal = 224.357 + 347.697j
Antenna 2: Signal = 136.171 + 140.063j
Antenna 3: Signal = 99.6895 + 82.282j
Antenna 4: Signal = 79.8881 + 55.0355j
Antenna 5: Signal = 67.3929 + 39.1056j
Antenna 6: Signal = 58.7366 + 28.6385j
Antenna 7: Signal = 52.3644 + 21.1435j
Antenna 8: Signal = 47.4425 + 15.4567j
Antenna 9: Signal = 43.456 + 11.0189j
Antenna 10: Signal = 40.1717 + 7.42353j
Antenna 11: Signal = 37.4696 + 4.37614j
Antenna 12: Signal = 35.0529 + 1.79748j
Antenna 13: Signal = 33.0863 + -0.485275j
Antenna 14: Signal = 31.0363 + -2.52982j
Antenna 15: Signal = 29.3484 + -4.27932j
Antenna 16: Signal = 27.8383 + -5.8778j
Antenna 17: Signal = 26.3874 + -7.39684j
Antenna 18: Signal = 25.0286 + -8.79448j
Antenna 19: Signal = 23.7726 + -10.0706j
Antenna 20: Signal = 22.5678 + -11.3219j
Antenna 21: Signal = 21.395 + -12.5397j
Antenna 22: Signal = 20.2562 + -13.6508j
Antenna 23: Signal = 19.1632 + -14.7427j
Antenna 24: Signal = 18.1171 + -15.9309j
Antenna 25: Signal = 16.9879 + -16.9879j
Antenna 26: Signal = 15.9308 + -18.117j
Antenna 27: Signal = 14.7427 + -19.1631j
Antenna 28: Signal = 13.6509 + -20.2562j
Antenna 29: Signal = 12.5397 + -21.3951j
Antenna 30: Signal = 11.3217 + -22.5678j
Antenna 31: Signal = 10.0707 + -23.7726j
Antenna 32: Signal = 8.79451 + -25.0286j
Antenna 33: Signal = 7.39686 + -26.3874j
Antenna 34: Signal = 5.87788 + -27.8384j
Antenna 35: Signal = 4.27935 + -29.3486j
Antenna 36: Signal = 2.52987 + -31.0364j
Antenna 37: Signal = 0.485244 + -33.0864j
Antenna 38: Signal = -1.79746 + -35.0529j
Antenna 39: Signal = -4.37616 + -37.4695j
Antenna 40: Signal = -7.42345 + -40.1717j
Antenna 41: Signal = -11.0189 + -43.4561j
Antenna 42: Signal = -15.4566 + -47.4425j
Antenna 43: Signal = -21.1436 + -52.3643j
Antenna 44: Signal = -28.6385 + -58.7366j
Antenna 45: Signal = -39.1056 + -67.3929j
Antenna 46: Signal = -55.0355 + -79.8881j
Antenna 47: Signal = -82.282 + -99.6895j
Antenna 48: Signal = -140.063 + -136.171j
Antenna 49: Signal = -347.697 + -224.357j
Antenna 50: Signal = 4409.24 + 590.76j
Antenna 51: Signal = 224.357 + 347.697j
Antenna 52: Signal = 136.171 + 140.063j
Antenna 53: Signal = 99.6895 + 82.282j
Antenna 54: Signal = 79.8881 + 55.0356j
Antenna 55: Signal = 67.3929 + 39.1056j
Antenna 56: Signal = 58.7366 + 28.6385j
Antenna 57: Signal = 52.3644 + 21.1435j
Antenna 58: Signal = 47.4425 + 15.4567j
Antenna 59: Signal = 43.456 + 11.0189j
Antenna 60: Signal = 40.1718 + 7.42349j
Antenna 61: Signal = 37.4696 + 4.37613j
Antenna 62: Signal = 35.0529 + 1.79747j
Antenna 63: Signal = 33.0863 + -0.485279j
Antenna 64: Signal = 31.0363 + -2.52983j
Antenna 65: Signal = 29.3486 + -4.27932j
Antenna 66: Signal = 27.8383 + -5.8778j
Antenna 67: Signal = 26.3874 + -7.39684j
Antenna 68: Signal = 25.0286 + -8.79448j
Antenna 69: Signal = 23.7726 + -10.0706j
Antenna 70: Signal = 22.5678 + -11.322j
Antenna 71: Signal = 21.395 + -12.5397j
Antenna 72: Signal = 20.2562 + -13.6508j
Antenna 73: Signal = 19.1632 + -14.7427j
Antenna 74: Signal = 18.1171 + -15.9309j
Antenna 75: Signal = 16.9879 + -16.9879j
Antenna 76: Signal = 15.9308 + -18.117j
Antenna 77: Signal = 14.7427 + -19.1631j
Antenna 78: Signal = 13.6509 + -20.2562j
Antenna 79: Signal = 12.5397 + -21.3951j
Antenna 80: Signal = 11.3217 + -22.5679j
Antenna 81: Signal = 10.0707 + -23.7727j
Antenna 82: Signal = 8.79451 + -25.0286j
Antenna 83: Signal = 7.39686 + -26.3874j
Antenna 84: Signal = 5.87788 + -27.8384j
Antenna 85: Signal = 4.27935 + -29.3486j
Antenna 86: Signal = 2.52987 + -31.0364j
Antenna 87: Signal = 0.485244 + -33.0864j
Antenna 88: Signal = -1.79746 + -35.0529j
Antenna 89: Signal = -4.37616 + -37.4695j
Antenna 90: Signal = -7.42345 + -40.1717j
Antenna 91: Signal = -11.0189 + -43.4561j
Antenna 92: Signal = -15.4566 + -47.4425j
Antenna 93: Signal = -21.1436 + -52.3643j
Antenna 94: Signal = -28.6385 + -58.7366j
Antenna 95: Signal = -39.1056 + -67.3929j
Antenna 96: Signal = -55.0355 + -79.8881j
Antenna 97: Signal = -82.282 + -99.6895j
Antenna 98: Signal = -140.063 + -136.171j
Antenna 99: Signal = -347.697 + -224.357j
```

```
Antenna 86: Signal = 2.52987 + -31.0364j
Antenna 87: Signal = 0.485244 + -33.0864j
Antenna 88: Signal = -1.79746 + -35.0529j
Antenna 89: Signal = -4.37616 + -37.4695j
Antenna 90: Signal = -7.42345 + -40.1717j
Antenna 91: Signal = -11.0189 + -43.4561j
Antenna 92: Signal = -15.4566 + -47.4425j
Antenna 93: Signal = -21.1436 + -52.3643j
Antenna 94: Signal = -28.6385 + -58.7366j
Antenna 95: Signal = -39.1056 + -67.3929j
Antenna 96: Signal = -55.0355 + -79.8881j
Antenna 97: Signal = -82.282 + -99.6895j
Antenna 98: Signal = -140.063 + -136.171j
Antenna 99: Signal = -347.697 + -224.357j
```

Time domain beamforming execu\on \me 0.025 ms

Time domain output for 100 antennas where the antenna array is given a complex inputs from the sensors

```
[tgarimel@gpu017 Project612]$ ./beamformer 100 1e9 2e9 0.1
Time-domain beamforming took 0.024255 ms.
Antenna 0: Signal = 0.000208073 + -0.000827454j
Antenna 1: Signal = -0.000370091 + -0.000573792j
Antenna 2: Signal = -0.000765235 + -0.000198823j
Antenna 3: Signal = -0.000780827 + 3.19136e-06j
Antenna 4: Signal = 8.97942e-05 + 0.000157517j
Antenna 5: Signal = -0.000475923 + 0.000165434j
Antenna 6: Signal = -8.66534e-05 + 8.07101e-06j
Antenna 7: Signal = -0.000372007 + 6.86059e-05j
Antenna 8: Signal = -0.000831803 + 0.000153702j
Antenna 9: Signal = -0.000234154 + 0.00065011j
Antenna 10: Signal = -3174.05 + 0.00179839j
Antenna 11: Signal = 0.000299438 + 0.000274633j
Antenna 12: Signal = 0.00107066 + 0.000325599j
Antenna 13: Signal = 0.000494223 + -0.000289404j
Antenna 14: Signal = 0.000142338 + 0.000115948j
Antenna 15: Signal = 0.000574704 + 8.95428e-05j
Antenna 16: Signal = 0.000247114 + -0.000127178j
Antenna 17: Signal = 0.000201396 + -0.000566023j
Antenna 18: Signal = 5.47453e-05 + 2.89905e-05j
Antenna 19: Signal = 0.000513222 + -0.000523336j
Antenna 20: Signal = -0.000405086 + -0.000716375j
Antenna 21: Signal = 0.000166588 + 0.000180637j
Antenna 22: Signal = -0.000580467 + -0.000483131j
Antenna 23: Signal = -0.000532937 + 0.000365682j
Antenna 24: Signal = -0.000143966 + -1.43957e-05j
Antenna 25: Signal = 0.000171661 + 0.000795743j
Antenna 26: Signal = 0.000467686 + -0.000244027j
Antenna 27: Signal = 0.000103883 + 0.000193853j
Antenna 28: Signal = 5.47966e-05 + -0.000417587j
Antenna 29: Signal = 8.69045e-05 + 0.000652653j
Antenna 30: Signal = 0.000487202 + 0.000409542j
Antenna 31: Signal = -0.000293893 + -0.000266212j
Antenna 32: Signal = 0.000528356 + 0.000117178j
Antenna 33: Signal = 0.000241926 + 0.00086597j
Antenna 34: Signal = -0.000575114 + -0.000409401j
Antenna 35: Signal = -0.000298013 + 0.000297349j
Antenna 36: Signal = 0.000357762 + -0.000423163j
Antenna 37: Signal = -0.000590569 + -0.000740616j
Antenna 38: Signal = -0.00109274 + 0.000257602j
Antenna 39: Signal = -7.97746e-05 + 0.000124484j
Antenna 40: Signal = 1557.38 + 0.000664705j
Antenna 41: Signal = -5.68514e-05 + 0.000881604j
Antenna 42: Signal = 0.00063659 + -0.000101319j
Antenna 43: Signal = 0.000338221 + 2.76979e-05j
Antenna 44: Signal = -0.000370232 + -0.000216046j
Antenna 45: Signal = 2.75705e-05 + 0.000494416j
Antenna 46: Signal = -0.000130104 + 0.000192254j
Antenna 47: Signal = 0.000901663 + 0.000278535j
Antenna 48: Signal = 0.000934518 + -0.000370046j
Antenna 49: Signal = -4.44064e-05 + -0.000402812j
Antenna 50: Signal = 0.000158138 + -0.000900014j
Antenna 51: Signal = -0.000370091 + -0.000573792j
Antenna 52: Signal = -0.000765235 + -0.000198823j
Antenna 53: Signal = -0.000780827 + 3.19137e-06j
Antenna 54: Signal = 8.97942e-05 + 0.000157517j
Antenna 55: Signal = -0.000475923 + 0.000165434j
Antenna 56: Signal = -8.66534e-05 + 8.071e-06j
Antenna 57: Signal = -0.000372007 + 6.86059e-05j
Antenna 58: Signal = -0.000831803 + 0.000153702j
Antenna 59: Signal = -0.000234154 + 0.00065011j
Antenna 60: Signal = -3174.05 + 0.00184588j
Antenna 61: Signal = 0.000299438 + 0.000274633j
Antenna 62: Signal = 0.00107066 + 0.000325599j
Antenna 63: Signal = 0.000494223 + -0.000289404j
Antenna 64: Signal = 0.000142338 + -0.000115947j
Antenna 65: Signal = 0.000574704 + 8.95428e-05j
Antenna 66: Signal = 0.000247114 + -0.000127178j
Antenna 67: Signal = 0.000201396 + -0.000566023j
Antenna 68: Signal = 5.47453e-05 + 2.89905e-05j
Antenna 69: Signal = 0.000513222 + -0.000523336j
Antenna 70: Signal = -0.000391047 + -0.000592041j
Antenna 71: Signal = 0.000166588 + 0.000190637j
Antenna 72: Signal = -0.000580467 + -0.000483131j
Antenna 73: Signal = -0.000532937 + 0.000365682j
Antenna 74: Signal = -0.000143966 + -1.43957e-05j
Antenna 75: Signal = 0.000171661 + 0.000795743j
Antenna 76: Signal = 0.000467686 + -0.000244027j
Antenna 77: Signal = 0.000103883 + 0.000193853j
Antenna 78: Signal = 5.47966e-05 + -0.000417587j
Antenna 79: Signal = 8.69045e-05 + 0.000652653j
Antenna 80: Signal = 0.000558462 + -0.000402613j
Antenna 81: Signal = -0.000293893 + -0.000266212j
Antenna 82: Signal = 0.000528356 + 0.000117178j
Antenna 83: Signal = 0.000241926 + -0.00086597j
Antenna 84: Signal = -0.000575114 + -0.000409401j
Antenna 85: Signal = -0.000298013 + 0.000297349j
Antenna 86: Signal = 0.000357762 + -0.000423163j
Antenna 87: Signal = -0.000590569 + -0.000740616j
Antenna 88: Signal = -0.00109274 + 0.000257602j
Antenna 89: Signal = -7.97746e-05 + 0.000124484j
Antenna 90: Signal = 1557.38 + 0.000664705j
Antenna 91: Signal = -5.68514e-05 + 0.000881604j
Antenna 92: Signal = 0.00063659 + -0.000101319j
Antenna 93: Signal = 0.000338221 + 2.76979e-05j
Antenna 94: Signal = -0.000370232 + -0.000216046j
Antenna 95: Signal = 2.75705e-05 + 0.000494416j
Antenna 96: Signal = -0.000130104 + 0.000192254j
Antenna 97: Signal = 0.000901663 + 0.000278535j
Antenna 98: Signal = 0.000934518 + -0.000370046j
Antenna 99: Signal = -4.44064e-05 + -0.000402812j
```

```
Antenna 84: Signal = -0.000575114 + -0.000409401j
Antenna 85: Signal = -0.000298013 + 0.000297349j
Antenna 86: Signal = 0.000357762 + -0.000423163j
Antenna 87: Signal = -0.000590569 + -0.000740616j
Antenna 88: Signal = -0.00109274 + 0.000257602j
Antenna 89: Signal = -7.97746e-05 + 0.000124484j
Antenna 90: Signal = 1557.38 + 0.0006647352j
Antenna 91: Signal = -5.68514e-05 + 0.000881604j
Antenna 92: Signal = 0.00063659 + -0.000101319j
Antenna 93: Signal = 0.000338221 + 2.76979e-05j
Antenna 94: Signal = -0.000370232 + -0.000216046j
Antenna 95: Signal = 2.75705e-05 + 0.000494416j
Antenna 96: Signal = -0.000130104 + 0.000192254j
Antenna 97: Signal = 0.000901663 + 0.000278535j
Antenna 98: Signal = 0.000934518 + -0.000370046j
Antenna 99: Signal = -4.44064e-05 + -0.000402812j
[tgarimel@gpu017 Project612]$ █
```

**Kernel execu\on run \me 0.024 ms**

## GPU UTILIZATION

The screenshot shows a terminal window titled "tgariel [SSH: hopper.orc.gmu.edu]" with the following content:

```

File Edit Selection View Go Run Terminal Help ← → tgariel [SSH: hopper.orc.gmu.edu]
EXPLORER beamformer.cu Makefile
TGARIMEL [SSH: HOPPER... Project612 > beamformer.cu > main(int, char *[])
>.cache >.config >.dotnet >.nv >.ssh >.vscode >.vscode-server > assignment > assignment5 > CUDA > LAB 2 assignment > Lab-3 > Lab6 > present > Project1 > Project2 > Project3 > Astro.bmp > Astronaut.bmp > Astrored.bmp > BirdDetected.bmp > dogL.bmp > dogout.bmp > Makefile > project3 > project3.cu > red.bmp > redbirdL.bmp > Project612 > beamformer > beamformer.cu > Makefile > Sample > workspace > OUTLINE > TIMELINE
beamformer.cu Makefile
Every 1.0s: nvidia-smi
Thu May 2 20:49:41 2024
+-----+
| NVIDIA-SMI 545.23.08 | Driver Version: 545.23.08 | CUDA Version: 12.3 | | |
| GPU Name Persistence-M | Bus-Id Disp.A Volatile Uncorr. ECC |
| Fan Temp Perf Pwr:Usage/Cap | Memory-Usage GPU-Util Compute M. |
| | | | | MIG M. |
+-----+
| 0 NVIDIA A100-SXM4-80GB On 00000000:C1:00.0 Off | On | | | |
| N/A 24C P0 50W / 500W | 87MiB / 81920MiB | N/A Default |
| | | | | |
+-----+
| MIG devices: |
| GPU GI CI MIG | Memory-Usage | Vol. Shared | | |
| ID ID Dev | BAR1-Usage | SM Un. CE ENC DEC OFA JPG |
| | | | | |
+-----+
| 0 10 0 0 | 12MiB / 9728MiB | 14 0 | 1 0 0 0 0 |
| 0MiB / 16383MiB | | | | |
+-----+
| Processes: GPU GI CI PID Type Process name GPU Memory |
| ID ID | | | | Usage |
+-----+

```

## CONCLUSION

The conclusion for this project on implementing digital beamforming using CUDA showcases several key achievements and potential impacts, as well as insights into future developments.

**Efficient Implementation:** The project successfully demonstrates how to leverage CUDA, a powerful parallel computing platform, to implement both time-domain and frequency-domain beamforming. This efficient use of CUDA significantly enhances the processing speed of beamforming algorithms, which are computationally intensive due to the need to manipulate large datasets representing signal data from multiple antennas.

**Integration of Advanced Technologies:** By integrating CUDA with cuFFT, the project allows for the sophisticated manipulation of signal data in the frequency domain, which is critical for applications requiring precise control over signal phases and amplitudes. This integration effectively utilizes GPU resources to accelerate not just the computation of FFT but also the subsequent beamforming operations.

**Scalability and Flexibility:** The designed framework provides a scalable solution to beamforming that can be adjusted based on the number of antennas and the specific requirements of the transmission environment. This flexibility ensures that the solution can be adapted to various practical scenarios in real-world applications.

**Error Handling and Robustness:** The project includes robust error handling mechanisms for both CUDA and cuFFT operations, enhancing the reliability of the application under different operating conditions and hardware configurations.

## APPLICATIONS

**Enhanced Real-time Processing:** The ability to process beamforming operations in real-time opens up numerous possibilities for applications in radar systems, wireless communications, and other areas where directional signal processing is crucial. This capability can lead to more efficient and accurate detection and communication systems, which are vital for both civilian and military applications.

**Potential for Innovation in Signal Processing:** The project lays the groundwork for further innovations in digital beamforming, including adaptive beamforming techniques where the system dynamically adjusts to changing environmental conditions and signal characteristics. This can significantly improve the performance of systems in complex scenarios such as mobile communications and autonomous vehicle navigation.