IMPLEMENTATION OF DIGITAL BEAMFORMER ON GPUs

ECE 612 Real Time Embedded Systems

ROHITH KUMAR KALISETTI G01410445

INTRODUCTION

The field of digital beamforming, par4cularly when applied using General-Purpose Graphics Processing Units (GPGPUs), marks a significant evolu4on in array signal processing. In this domain, GPUs, tradi4onally used for rendering graphics, are repurposed for processing the data received from arrays of sensors. The primary goal of a beamformer is to 4me-align or phase-align signals from a targeted direc4on, enabling coherent signal addi4on. This selec4ve amplifica4on enhances the signal from the desired direc4on while diminishing those from others, thereby increasing the signal's clarity and effec4ve range. This direc4onality is crucial in environments where the poten4al for interference is high, such as in densely populated urban areas or in radar opera4ons where discerning between objects close together is essen4al. By improving the direc4onality of the signal, beamforming significantly enhances the clarity and reach of the communica4on, ensuring that the transmission is both powerful and precise.

The applica4on of CUDA in beamforming processes brings a transforma4onal change in how these computa4ons are performed. GPUs, with their thousands of small, efficient cores capable of handling mul4ple opera4ons simultaneously, are ideally suited to the parallelizable nature of beamforming tasks. This parallel processing capability allows beamforming opera4ons, which involve large-scale computa4ons to determine and adjust the phases and amplitudes of mul4ple signals, to be executed more rapidly and efficiently than ever before.

The applica4on of GPUs in this context is not just about leveraging their computa4onal might but also their architecture, which is inherently suited for parallel data processing. Unlike tradi4onal CPUs, GPUs excel in handling mul4ple opera4ons simultaneously, making them ideal for the computa4onally intensive task of beamforming. This involves processing large volumes of data from numerous sensors and synthesizing this informa4on into a coherent output, oNen in real-4me. The transi4on from using specialized hardware like DSPs or ASICs to more versa4le, programmable plaPorms such as GPUs is driven by the CUDA technology developed by NVIDIA, which has democra4zed access to parallel compu4ng capabili4es, thus broadening the poten4al for innova4ve approaches in digital beamforming.

THEORETICAL BACKGROUND

Beamforming is a signal processing technique employed in antenna arrays for direc4onal signal transmission and recep4on. By manipula4ng the phase and amplitude of the signal at each antenna element, the array can create a paTern of construc4ve and destruc4ve interference in the wavefronts. This interference paTern allows the beam to be "steered" toward specific direc4ons without physically moving the antennas. The core principle behind beamforming is the phase shiN—altering the phase of signals in a way that the signals at the receiver combine construc4vely, thereby enhancing the signal power in the desired direc4on, while signals that arrive out of phase combine destruc4vely, reducing noise and interference from other direc4ons.

The implementa4on of beamforming can be categorized into two approaches: analog and digital. Analog beamforming is performed using hardware like phase shiNers, which adjust the signal phase at each antenna con4nuously. Digital beamforming, however, uses digital signal processing techniques to mul4ply the incoming signals by complex weights that adjust their amplitude and phase. The digital approach offers more flexibility and precision, allowing for mul4ple beams to be formed simultaneously and adjusted dynamically.

The mathema4cal 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 conver4ng signals from 4me domain to frequency domain and vice versa. This conversion is crucial for implemen4ng frequency-domain beamforming, where the array's response is manipulated in the frequency domain to achieve the desired direc4onal paTerns. By using FFT, the computa4onal complexity of performing DFT is drama4cally reduced, making real-4me signal processing feasible even when dealing with large data sets and complex opera4ons typical in modern beamforming applica4ons.

MATHEMATICAL DESIGN FOR BEAMFORMER

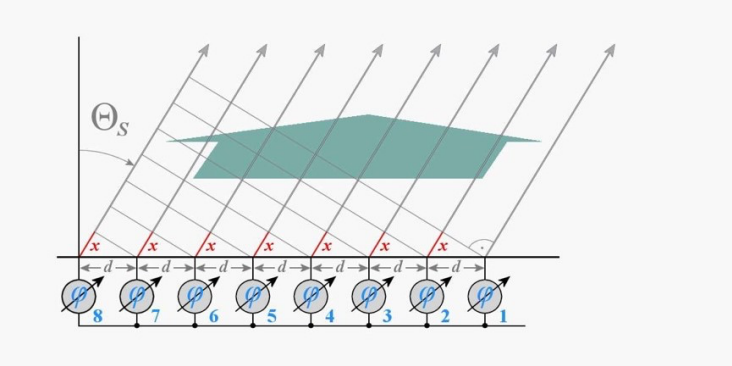


Figure 1 Phased Array Antennas

Phase Delay (θ)

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: θ**=n(antenna)×**D**(delay)×**T**(sinewave)×2**π Where:

- Antenna is presumably the index n of each antenna in the array.
- D(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 fo.

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: ω is the angular frequency corresponding to the carrier frequency fo such that ω = **2**πfo, t is time.
θ 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 $\boldsymbol{N_F}$ **= n x t x** $\boldsymbol{f_0}$**,** where: n: Element index or identifier in the array. t: Time factor or period corresponding to the desired delay. $f_{"}$: The carrier frequency.

By adjusting $\boldsymbol{N_F}$, 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** $\boldsymbol{\pi f_0 t + N_F}$**) Data 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(θ) =** $_D{}^x$ **or sin(θ) =** $_p{}^x$**.**

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.

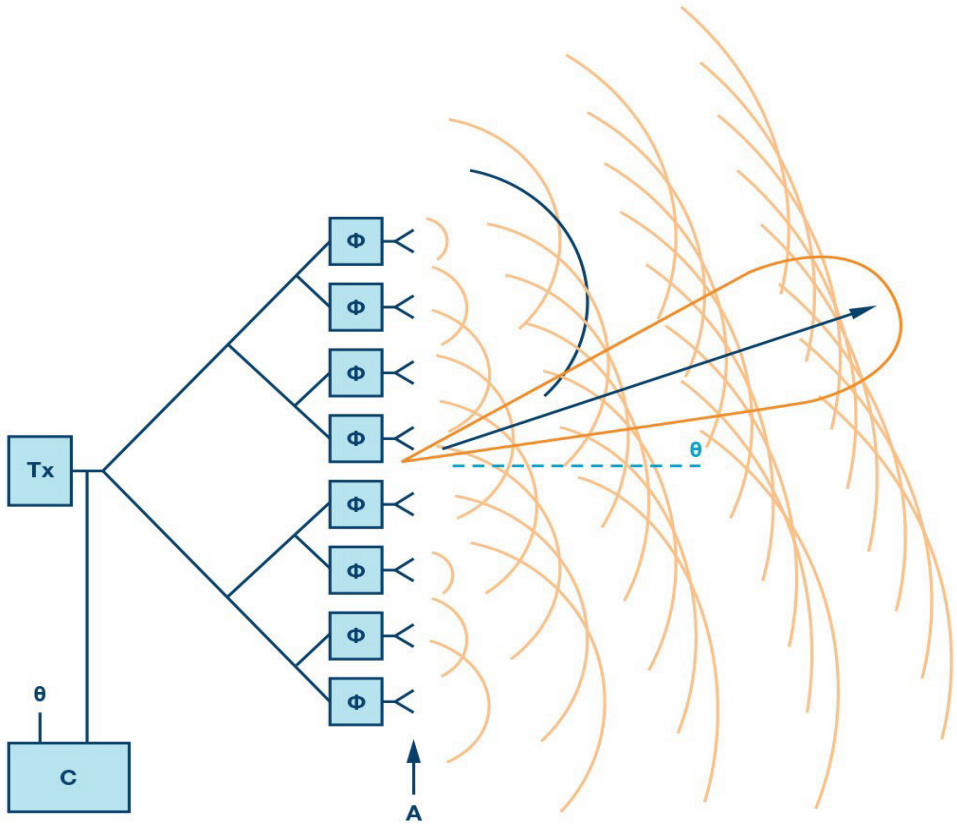


Figure 2 Phase Array Antennas where Beam is Steered in One Direction

CUDA AND GPU COMPUTING

The applica4on of GPUs in signal processing has transformed the capabili4es of many systems, from radar signal processing to audio processing and beyond. GPUs excel in handling the vast number of computa4ons required for signal processing algorithms due to their parallel architecture.

In signal processing, many opera4ons, such as Fourier transforms, convolu4ons, and matrix mul4plica4ons, 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 opera4ons per clock cycle. This makes GPUs par4cularly useful for real-4me signal processing, where large volumes of data must be processed quickly to derive ac4onable insights or responses.

Furthermore, CUDA's support for fast Fourier transforms (cuFFT), which is an op4mized implementa4on of the FFT algorithm, is crucial for many signal processing tasks. cuFFT allows for efficient transforma4on of signals between 4me and frequency domains, a process that is central to many forms of digital signal processing and is cri4cal for implemen4ng frequency-domain beamforming.

By combining the theore4cal aspects of beamforming with the technical capabili4es 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 capabili4es of tradi4onal beamforming applica4ons but also

opens up new possibili4es 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: SeSng 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 installa4on, 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 Integra4on op4on. 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.yourins4tu4on.edu' with your actual username and the cluster's address.

Step 2: Load CUDA Module

- Most clusters use module-based soNware management. To load CUDA, use **Module load cuda**
- Check the loaded modules to confirm CUDA is ac4ve: **Module listmodule**

Part 3: Write and Compile the Code Step1: 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 compila4on 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

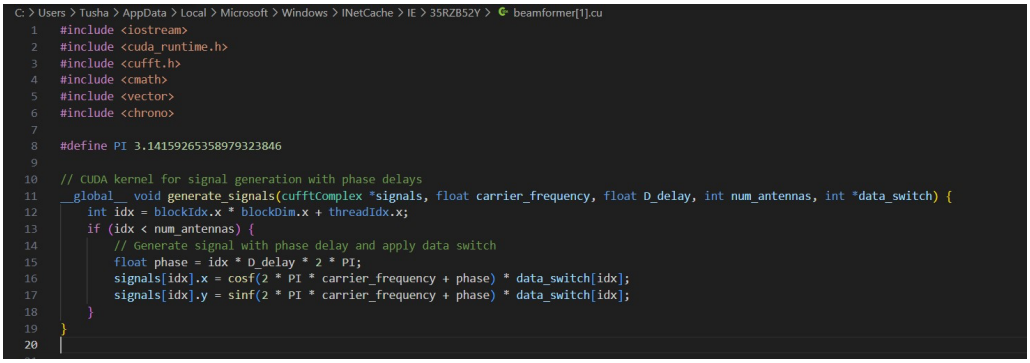
- *Cat Makefile*
- *Make Beamformer.cu* Compila4on and Execu4on:
- *./Beamformer <num_antennas> <carrier_frequency> <@me> <D_delay>*
- *nvcc -o beamformer beamformer.cu*

- *./beamformer 1024 2.4e9 0.001 0.000001*

CODE OVERVIEW

This CUDA program is designed for signal processing in an antenna array, par4cularly for beamforming tasks which combine signals from mul4ple antennas to direct a beam towards a target direc4on.

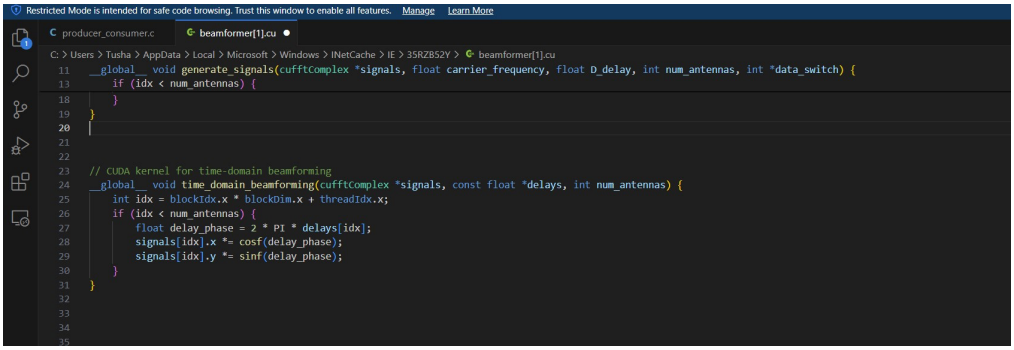
Program Structure: Detailed breakdown of the main components of the program, including the purpose of each part.
Signal Genera\on: signals are generated with specific phase delays and data modula4on using CUDA kernels. Here is the code snippet for this.
Include Necessary Libraries: Standard input-output opera4ons, CUDA run4me, cuFFT library, mathema4cal func4ons, and 4ming func4onali4es are included.
Define Constants: PI is defined to handle mathema4cal computa4ons involving circles and periodic func4ons.



CUDA Kernels:

- **generate_signals**: ANer the kernel generates modulated signals with phase delays.
- **\me_domain_beamforming**: kernel applies 4me-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:



```
// CUDA kernel for frequency-domain beamforming
__global__ void frequency_domain_beamforming(cufftComplex *freq_domain_signals, const cufftComplex *beamforming_weights, int num_antennas) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < num_antennas) {
        // Apply the beamforming weight for each antenna in the frequency domain
        freq_domain_signals[idx].x = freq_domain_signals[idx].x * beamforming_weights[idx].x
                                   - freq_domain_signals[idx].y * beamforming_weights[idx].y;
        freq_domain_signals[idx].y = freq_domain_signals[idx].x * beamforming_weights[idx].y
                                   + freq_domain_signals[idx].y * beamforming_weights[idx].x;
    }
}
```

Error Handling: The macros and inline func4ons are defined to handle errors from CUDA and cuFFT opera4ons, 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);
    }
}
```

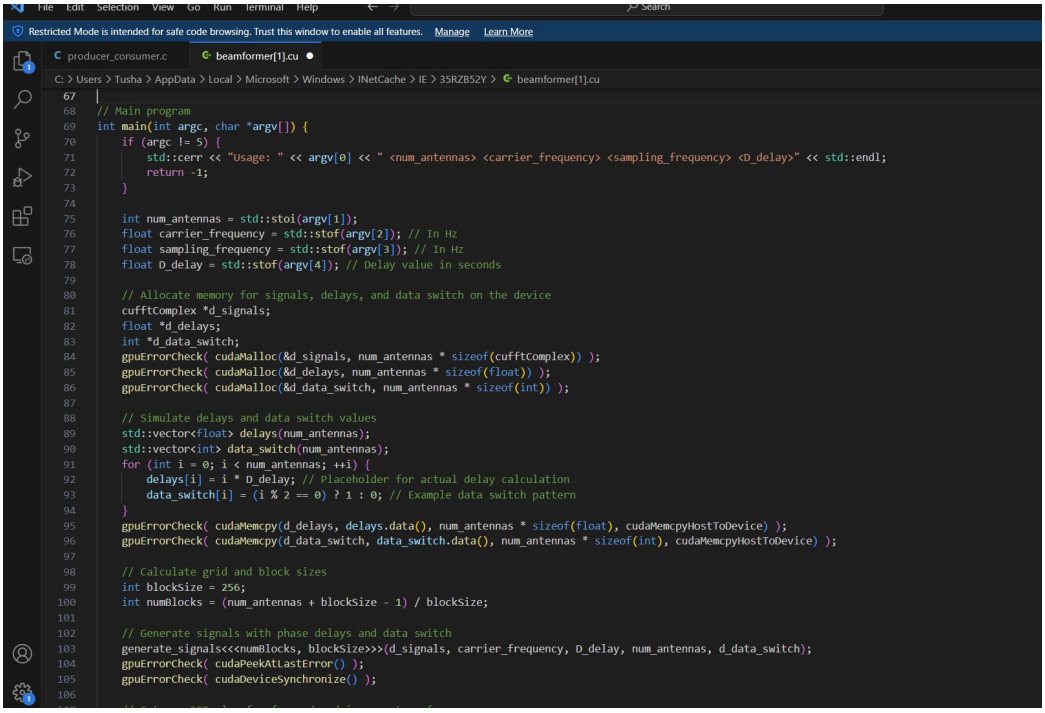
Implementa\on Details

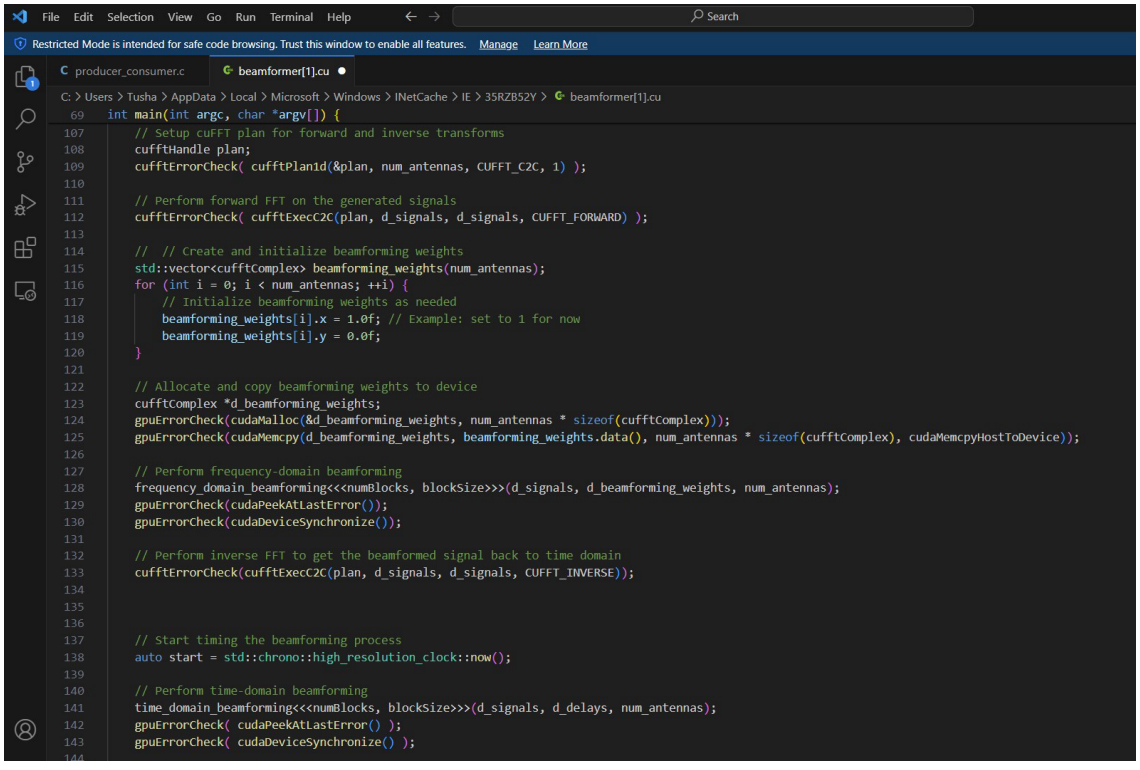
Memory Management: In the main func4on the strategies used for memory alloca4on, transfer between host and device, and dealloca4on.

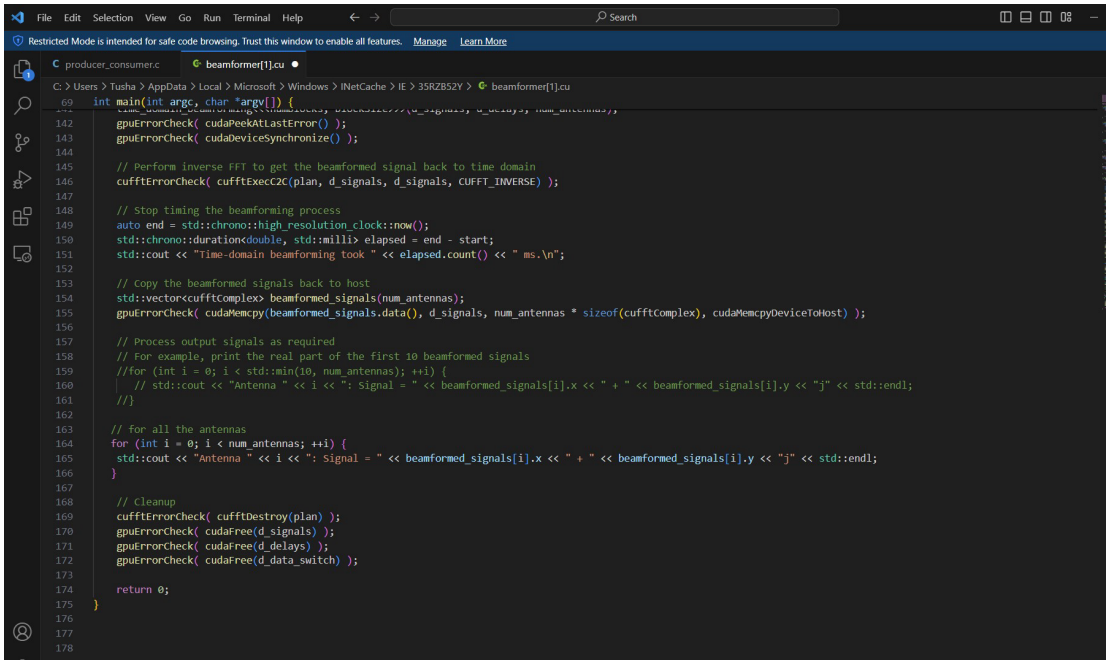
Paralleliza4on Strategy: Analysis of grid and block sizes, and how tasks are divided among threads.

In the main func4on 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.
- ANer alloca4ng memory it ini4alizes data for delays and switching flags.
- Configures CUDA kernel launches with appropriate grid and block sizes.
- Then executes the signal genera4on kernel to modulate ini4al signals.
- Establishes an FFT plan and performs forward and inverse FFT for frequency-domain processing.
- Executes beamforming in both 4me-domain and frequency-domain.
- Now it measures the performance of 4me-domain beamforming opera4ons using high-resolu4on clocks.
- Gives the outputs the processed signals for each antenna and cleans up resources to prevent memory leaks.


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 > 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


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 > 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
137 // Start timing the beamforming process
138 auto start = std::chrono::high_resolution_clock::now();
139
140 // Perform time-domain beamforming
141 time_domain_beamforming<<<numBlocks, blockSize>>>(d_signals, d_delays, num_antennas);
142 gpuErrorCheck(cudaPeekAtLastError());
143 gpuErrorCheck(cudaDeviceSynchronize());



Resource Management and Cleanup

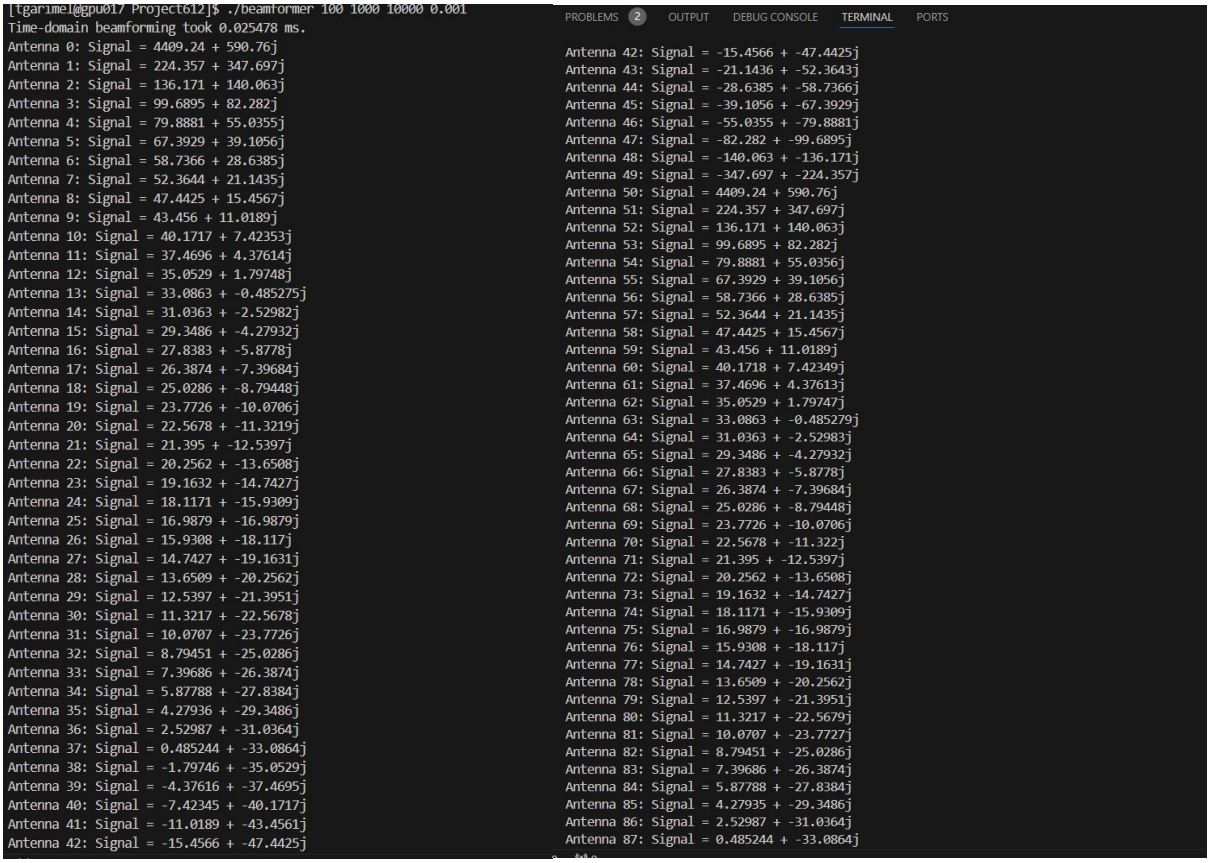
Properly deallocates all dynamically allocated GPU memory and destroys cuFFT plans to ensure there are no resource leaks.

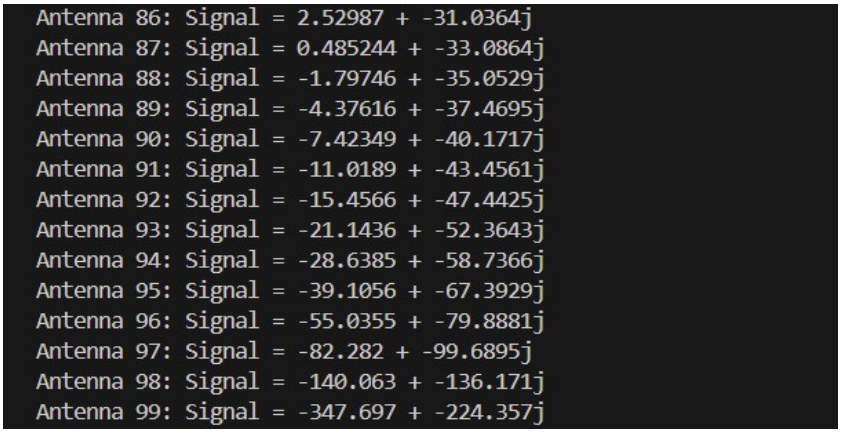
Execu\on Flow

- The program begins by sehng 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 termina4on.

RESULTS

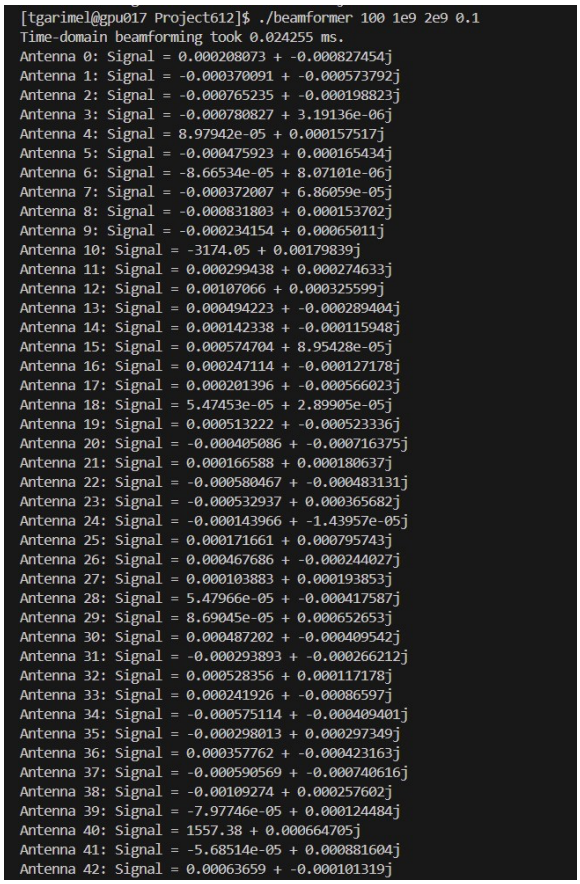
Time domain output for 100 antennas on an array of a beamformer.

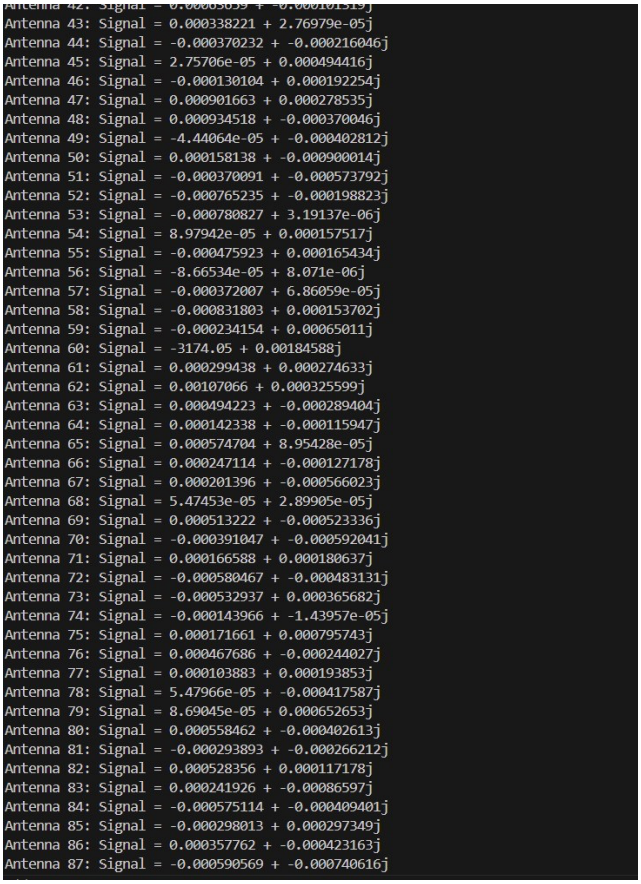


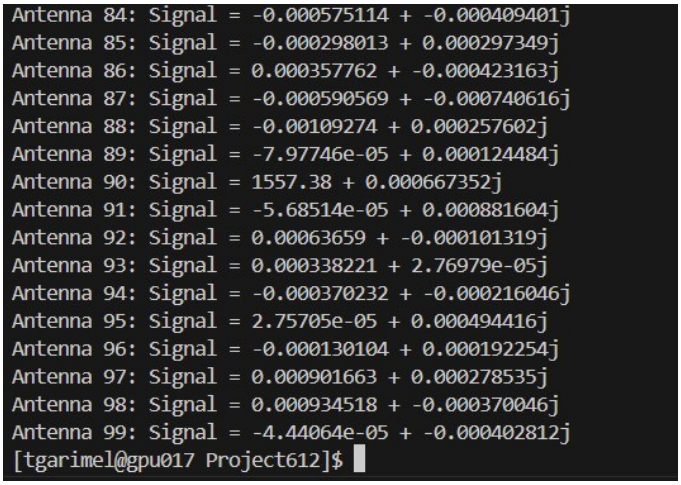


Time domain beamforming execu\on \me 0.025 ms

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

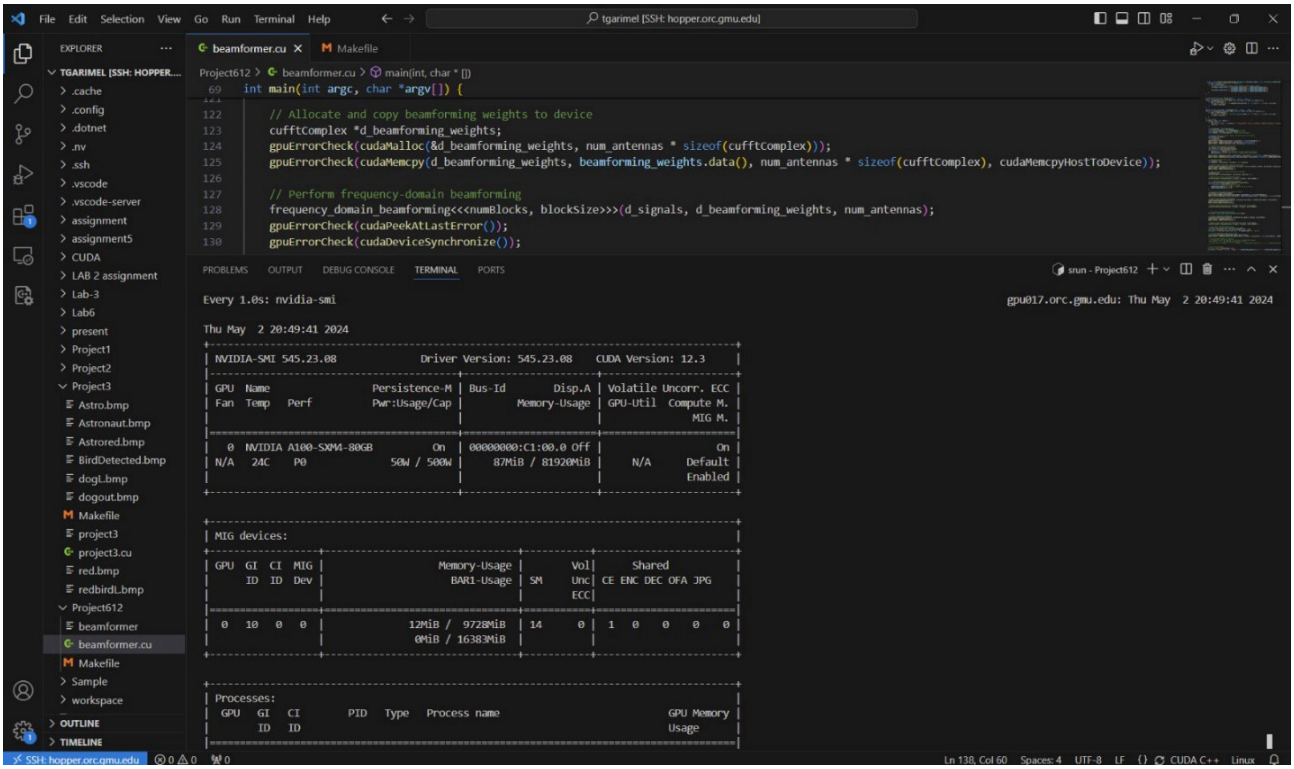






Kernel execu\on run \me 0.024 ms

GPU UTILIZATION



CONCLUSION

The conclusion for this project on implemen4ng digital beamforming using CUDA showcases several key achievements and poten4al impacts, as well as insights into future developments.

Efficient Implementa\on: The project successfully demonstrates how to leverage CUDA, a powerful parallel compu4ng plaPorm, to implement both 4me-domain and frequency-domain beamforming. This efficient use of CUDA significantly enhances the processing speed of beamforming algorithms, which are computa4onally intensive due to the need to manipulate large datasets represen4ng signal data from mul4ple antennas.

Integra\on of Advanced Technologies: By integra4ng CUDA with cuFFT, the project allows for the sophis4cated manipula4on of signal data in the frequency domain, which is cri4cal for applica4ons requiring precise control over signal phases and amplitudes. This integra4on effec4vely u4lizes GPU resources to accelerate not just the computa4on of FFT but also the subsequent beamforming opera4ons.

Scalability and Flexibility: The designed framework provides a scalable solu4on 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 solu4on can be adapted to various prac4cal scenarios in realworld applica4ons.

Error Handling and Robustness: The project includes robust error handling mechanisms for both CUDA and cuFFT opera4ons, enhancing the reliability of the applica4on under different opera4ng condi4ons and hardware configura4ons.

APPLICATIONS

Enhanced Real-\me Processing: The ability to process beamforming opera4ons in real-4me opens up numerous possibili4es for applica4ons in radar systems, wireless communica4ons, and other areas where direc4onal signal processing is crucial. This capability can lead to more efficient and accurate detec4on and communica4on systems, which are vital for both civilian and military applica4ons.

Poten\al for Innova\on in Signal Processing: The project lays the groundwork for further innova4ons in digital beamforming, including adap4ve beamforming techniques where the system dynamically adjusts to changing environmental condi4ons and signal characteris4cs. This can significantly improve the performance of systems in complex scenarios such as mobile communica4ons and autonomous vehicle naviga4on.