Source Code – (PA9.cpp)

```
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <fftw3.h>          // FFTW library for CPU FFT
#include <cufft.h>          // cuFFT library for GPU FFT
#include <cuda_runtime.h>   // CUDA runtime API
#include <chrono>           // For timing

#include "WavFile.h" // Make sure WavFile.h and WavFile.cpp are present

#define BUFF_SIZE   16384 // Processing buffer size in samples
#define MAX_FREQ    48    // KHz (Used if power analysis is added back)
using namespace std;
// Macro for checking CUDA errors
#define CUDA_CHECK(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort = true) {
   // Function to check CUDA API call results and report errors
   if (code != cudaSuccess) {
      fprintf(stderr, "GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
      if (abort)
         exit(code); // Exit if the error is critical
   }
}

// --- Helper Function for Filtering (CPU) ---
// This function modifies the complex frequency data in place.
// It zeros out frequency bins around a target frequency.
void filterFrequencyDomain(fftw_complex* data, int N, double sampleRate, double freqToFilter, int
/*filterWidth*/) {
   // Compute the bin corresponding to freqToFilter
   int target_bin = static_cast<int>(freqToFilter * N / sampleRate);
   // Mirror index for the negative-frequency component
   int mirror_bin = N - target_bin;

   printf("Zeroing bins %d, %d (positive and negative %.2f Hz)\n", target_bin, mirror_bin, freqToFilter);

   // Make sure we can safely zero target_bin, target_bin+1, and mirror_bin-1
   if (target_bin >= 0 && target_bin + 1 < N && mirror_bin - 1 >= 0) {
      // Positive-frequency bins
      data[target_bin][0]     = 0.0;  // real
      data[target_bin][1]     = 0.0;  // imag
      data[target_bin + 1][0] = 0.0;
      data[target_bin + 1][1] = 0.0;

      // Negative-frequency bins (mirror)
      data[mirror_bin][0]       = 0.0;
      data[mirror_bin][1]       = 0.0;
```

```
        data[mirror_bin - 1][0]   = 0.0;
        data[mirror_bin - 1][1]   = 0.0;
    }
}
int main(int argc, char *argv[]) {
    const char *wavfile; // input wav file path
    wavfile = argv[1];
    char *base_name = strdup(wavfile);
    char *dot = strrchr(base_name, '.');
    if (dot && !strcmp(dot, ".wav")) {
        *dot = '\0';
    }

    char *wavfileout_cpu = (char *)malloc(strlen(base_name) + strlen("_cpu_out.wav") + 1);
    char *wavfileout_gpu = (char *)malloc(strlen(base_name) + strlen("_gpu_out.wav") + 1);
    char *logfile = (char *)malloc(strlen(base_name) + strlen("_out.log") + 1);

    if (!wavfileout_cpu || !wavfileout_gpu || !logfile) { // Check allocation success
        fprintf(stderr, "Error allocating memory for filenames.\n");
        free(base_name);
        exit(1);
    }

    sprintf(wavfileout_cpu, "%s_cpu_out.wav", base_name);
    sprintf(wavfileout_gpu, "%s_gpu_out.wav", base_name);
    sprintf(logfile, "%s_out.log", base_name);

    printf("Input WAV file: %s\n", wavfile);
    printf("CPU Output WAV file: %s\n", wavfileout_cpu);
    printf("GPU Output WAV file: %s\n", wavfileout_gpu);
    printf("Log file: %s\n", logfile);

    free(base_name); // Free the duplicated base name now that filenames are constructed

    fftw_complex *h_fft_in, *h_fft_out_cpu, *h_ifft_out_cpu; // Host buffers for CPU path
    h_fft_in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * BUFF_SIZE);
    h_fft_out_cpu = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * BUFF_SIZE);
    h_ifft_out_cpu = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * BUFF_SIZE);

    fftw_complex *h_fft_out_gpu_temp; // Host buffer to hold GPU FFT result for CPU filtering
    h_fft_out_gpu_temp = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * BUFF_SIZE);

    if (!h_fft_in || !h_fft_out_cpu || !h_ifft_out_cpu || !h_fft_out_gpu_temp) {
        fprintf(stderr, "Error allocating FFTW host memory.\n");
        fftw_free(h_fft_in); fftw_free(h_fft_out_cpu); fftw_free(h_ifft_out_cpu);
fftw_free(h_fft_out_gpu_temp);
        free(wavfileout_cpu); free(wavfileout_gpu); free(logfile);
        exit(1);
    }
    fftw_plan plan_forward_cpu = fftw_plan_dft_1d(BUFF_SIZE, h_fft_in, h_fft_out_cpu,
FFTW_FORWARD, FFTW_ESTIMATE);
```

```
    fftw_plan plan_backward_cpu = fftw_plan_dft_1d(BUFF_SIZE, h_fft_out_cpu, h_ifft_out_cpu,
FFTW_BACKWARD, FFTW_ESTIMATE);
    cufftHandle plan_forward_gpu, plan_backward_gpu;
    cufftDoubleComplex *d_fft_data;
    cufftDoubleComplex *h_ifft_out_gpu; // Host buffer for final GPU IFFT result

    int nx = BUFF_SIZE;
    int batch = 1;
    cufftType type = CUFFT_Z2Z;

    CUDA_CHECK(cudaMalloc((void**)&d_fft_data, nx * sizeof(cufftDoubleComplex)));
    h_ifft_out_gpu = (cufftDoubleComplex *)calloc(nx, sizeof(cufftDoubleComplex));
    if (!h_ifft_out_gpu) {
        fprintf(stderr, "Error allocating host memory for cuFFT output.\n");
        cudaFree(d_fft_data);
        fftw_free(h_fft_in); fftw_free(h_fft_out_cpu); fftw_free(h_ifft_out_cpu);
fftw_free(h_fft_out_gpu_temp);
        fftw_destroy_plan(plan_forward_cpu); fftw_destroy_plan(plan_backward_cpu);
        free(wavfileout_cpu); free(wavfileout_gpu); free(logfile);
        exit(1);
    }
    cufftResult status;
    status = cufftPlan1d(&plan_forward_gpu, nx, type, batch);
    if (status != CUFFT_SUCCESS) {
        printf("error: cufftPlan1d (forward GPU) failed.\n");
        cudaFree(d_fft_data); free(h_ifft_out_gpu);
        fftw_free(h_fft_in); fftw_free(h_fft_out_cpu); fftw_free(h_ifft_out_cpu);
fftw_free(h_fft_out_gpu_temp);
        fftw_destroy_plan(plan_forward_cpu); fftw_destroy_plan(plan_backward_cpu);
        free(wavfileout_cpu); free(wavfileout_gpu); free(logfile);
        exit(1);
    }
    status = cufftPlan1d(&plan_backward_gpu, nx, type, batch);
    if (status != CUFFT_SUCCESS) { /* ... error handling & cleanup ... */
        printf("error: cufftPlan1d (backward GPU) failed.\n");
        cufftDestroy(plan_forward_gpu);
        cudaFree(d_fft_data); free(h_ifft_out_gpu);
        fftw_free(h_fft_in); fftw_free(h_fft_out_cpu); fftw_free(h_ifft_out_cpu);
fftw_free(h_fft_out_gpu_temp);
        fftw_destroy_plan(plan_forward_cpu); fftw_destroy_plan(plan_backward_cpu);
        free(wavfileout_cpu); free(wavfileout_gpu); free(logfile);
        exit(1);
    }
    // --- Audio File Handling ---
    short sampleBuffer[BUFF_SIZE];      // Buffer to read samples from input WAV
    short outputBufferCpu[BUFF_SIZE];   // Output buffer for CPU path results
    short outputBufferGpu[BUFF_SIZE];   // Output buffer for GPU path results

    WavInFile inFile(wavfile);
    printf("--- Input WAV File Info ---\n");
    printf("SampleRate: %d Hz\n", inFile.getSampleRate());
```

```
    printf("BitsPerSample: %d\n", inFile.getNumBits());
    printf("NumChannels: %d\n", inFile.getNumChannels());
    printf("NumSamples: %u\n", inFile.getNumSamples());
    printf("---------------------------\n");

    if (inFile.getNumChannels() != 1) { /* ... error handling & cleanup ... */
        fprintf(stderr, "Error: Input file must be mono.\n");
        cufftDestroy(plan_forward_gpu); cufftDestroy(plan_backward_gpu);
        cudaFree(d_fft_data); free(h_ifft_out_gpu);
        fftw_free(h_fft_in); fftw_free(h_fft_out_cpu); fftw_free(h_ifft_out_cpu);
fftw_free(h_fft_out_gpu_temp);
        fftw_destroy_plan(plan_forward_cpu); fftw_destroy_plan(plan_backward_cpu);
        free(wavfileout_cpu); free(wavfileout_gpu); free(logfile);
        exit(1);
    }
    if (inFile.getNumBits() != 16) {
        fprintf(stderr, "Warning: Input file is not 16-bit. Output will be 16-bit.\n");
    }

    // Create output WAV file objects for both CPU and GPU paths
    WavOutFile outFileCpu(wavfileout_cpu, inFile.getSampleRate(), 16, 1);
    WavOutFile outFileGpu(wavfileout_gpu, inFile.getSampleRate(), 16, 1);

    FILE *log_fp;
    if ((log_fp = fopen(logfile, "w")) == NULL) { /* ... error handling & cleanup ... */
        fprintf(stderr, "can't open %s for writing\n", logfile);
        cufftDestroy(plan_forward_gpu); cufftDestroy(plan_backward_gpu);
        cudaFree(d_fft_data); free(h_ifft_out_gpu);
        fftw_free(h_fft_in); fftw_free(h_fft_out_cpu); fftw_free(h_ifft_out_cpu);
fftw_free(h_fft_out_gpu_temp);
        fftw_destroy_plan(plan_forward_cpu); fftw_destroy_plan(plan_backward_cpu);
        free(wavfileout_cpu); free(wavfileout_gpu); free(logfile);
        // Note: WavOutFile destructors will handle closing if objects were created
        exit(1);
    }

    // --- Timing Variables ---
    long long total_cpu_path_duration_us = 0;
    long long total_gpu_path_duration_us = 0;
    int chunk_count = 0;
    double sampleRate = inFile.getSampleRate();
    double freqToFilter = 10000.0;
    int filterWidth = 2;
    printf("\nStarting audio processing...\n");
    while (!inFile.eof()) {
        size_t samplesRead = inFile.read(sampleBuffer, BUFF_SIZE);
        if (samplesRead == 0) break;

        chunk_count++;
        for (size_t i = 0; i < BUFF_SIZE; ++i) {
            if (i < samplesRead) {
```

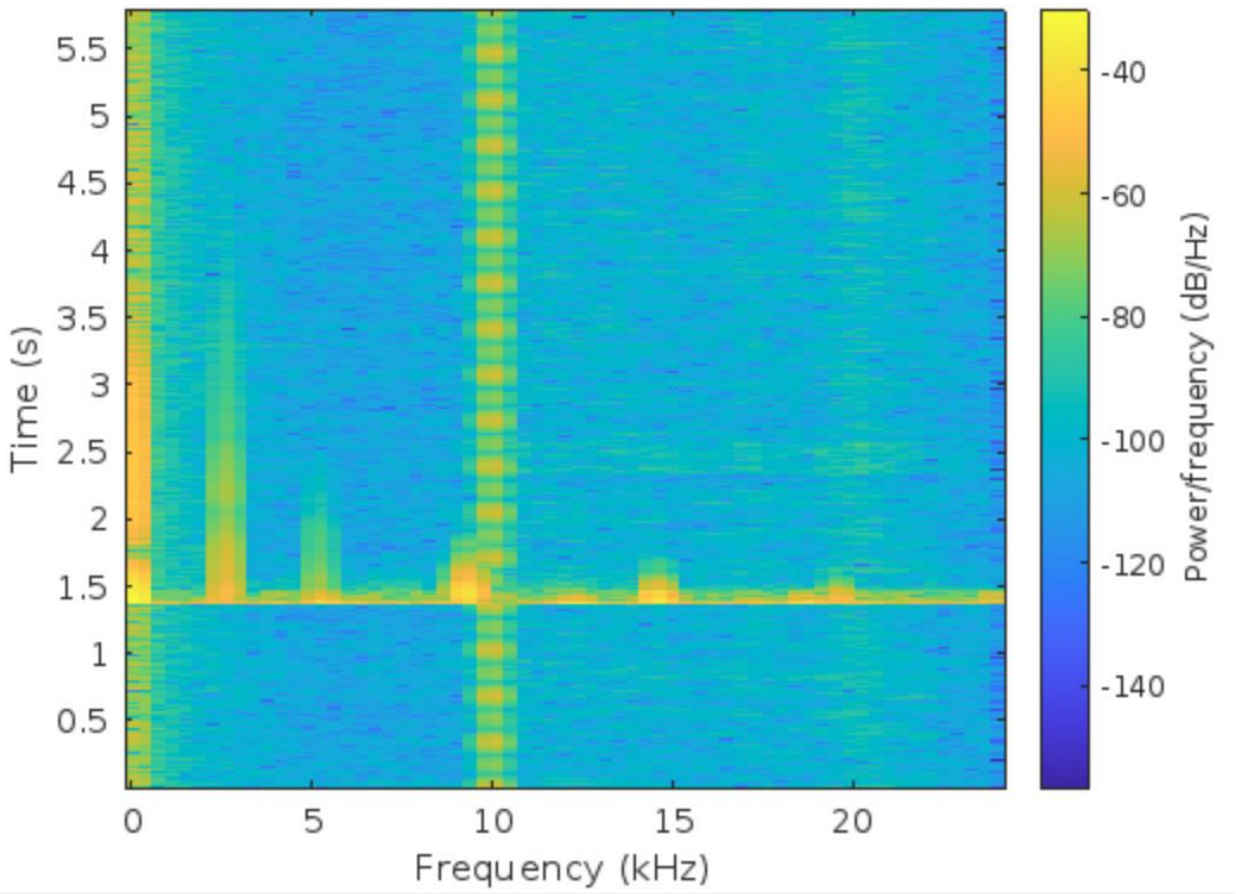
```
            h_fft_in[i][0] = (double)sampleBuffer[i];
            h_fft_in[i][1] = 0.0;
        } else {
            h_fft_in[i][0] = 0.0;
            h_fft_in[i][1] = 0.0;
        }
    }


    auto start_cpu_path = chrono::high_resolution_clock::now();

    // 1. FFTW Forward FFT (CPU)
    fftw_execute(plan_forward_cpu);

    // 2. Filter on CPU

    filterFrequencyDomain(h_fft_out_cpu, BUFF_SIZE, sampleRate, freqToFilter, filterWidth);

    // 3. FFTW Inverse FFT (CPU)
    fftw_execute(plan_backward_cpu);
      auto stop_cpu_path = chrono::high_resolution_clock::now();
    // 4. Prepare CPU Output Buffer
    for (size_t i = 0; i < samplesRead; ++i) {
        // Normalize FFTW IFFT output by dividing by N (BUFF_SIZE)
        double real_part = h_ifft_out_cpu[i][0] / BUFF_SIZE;
        // Clamp values to the 16-bit signed integer range
        if (real_part > 32767.0) real_part = 32767.0;
        else if (real_part < -32768.0) real_part = -32768.0;
        // Cast to short for output
        outputBufferCpu[i] = (short)real_part;
    }
    total_cpu_path_duration_us += chrono::duration_cast<std::chrono::microseconds>(stop_cpu_path -
start_cpu_path).count();

    // 1. Copy input data Host -> Device
    CUDA_CHECK(cudaMemcpy(d_fft_data, (cufftDoubleComplex*)h_fft_in, nx *
sizeof(cufftDoubleComplex), cudaMemcpyHostToDevice));

    // 2. cuFFT Forward FFT (GPU)
      auto start_gpu_path = chrono::high_resolution_clock::now();
    status = cufftExecZ2Z(plan_forward_gpu, d_fft_data, d_fft_data, CUFFT_FORWARD);
    if (status != CUFFT_SUCCESS) { /* ... error handling & cleanup ... */
        printf("error: cufftExecZ2Z (forward GPU) failed.\n");
        cudaFree(d_fft_data); free(h_ifft_out_gpu); cufftDestroy(plan_forward_gpu);
cufftDestroy(plan_backward_gpu);
        fftw_free(h_fft_in); fftw_free(h_fft_out_cpu); fftw_free(h_ifft_out_cpu);
fftw_free(h_fft_out_gpu_temp);
        fftw_destroy_plan(plan_forward_cpu); fftw_destroy_plan(plan_backward_cpu); fclose(log_fp);
        free(wavfileout_cpu); free(wavfileout_gpu); free(logfile);
        exit(1);
    }
```

```
        // 3. Copy FFT result Device -> Host (Temp buffer)
        CUDA_CHECK(cudaMemcpy((cufftDoubleComplex*)h_fft_out_gpu_temp, d_fft_data, nx *
sizeof(cufftDoubleComplex), cudaMemcpyDeviceToHost));

        // 4. Filter on CPU (using the data copied back from GPU)
        filterFrequencyDomain(h_fft_out_gpu_temp, BUFF_SIZE, sampleRate, freqToFilter, filterWidth);

        // 5. Copy Filtered data Host (Temp buffer) -> Device
        CUDA_CHECK(cudaMemcpy(d_fft_data, (cufftDoubleComplex*)h_fft_out_gpu_temp, nx *
sizeof(cufftDoubleComplex), cudaMemcpyHostToDevice));

        // 6. cuFFT Inverse FFT (GPU)
        status = cufftExecZ2Z(plan_backward_gpu, d_fft_data, d_fft_data, CUFFT_INVERSE);
        if (status != CUFFT_SUCCESS) { /* ... error handling & cleanup ... */
            printf("error: cufftExecZ2Z (inverse GPU) failed.\n");
            cudaFree(d_fft_data); free(h_ifft_out_gpu); cufftDestroy(plan_forward_gpu);
cufftDestroy(plan_backward_gpu);
            fftw_free(h_fft_in); fftw_free(h_fft_out_cpu); fftw_free(h_ifft_out_cpu);
fftw_free(h_fft_out_gpu_temp);
            fftw_destroy_plan(plan_forward_cpu); fftw_destroy_plan(plan_backward_cpu); fclose(log_fp);
            free(wavfileout_cpu); free(wavfileout_gpu); free(logfile);
            exit(1);
        }
          auto stop_gpu_path = chrono::high_resolution_clock::now();
        // 7. Copy final result Device -> Host
        CUDA_CHECK(cudaMemcpy(h_ifft_out_gpu, d_fft_data, nx * sizeof(cufftDoubleComplex),
cudaMemcpyDeviceToHost));

        // 8. Synchronize before stopping timer
        CUDA_CHECK(cudaDeviceSynchronize());
        total_gpu_path_duration_us += chrono::duration_cast<std::chrono::microseconds>(stop_gpu_path -
start_gpu_path).count();


        // --- Prepare GPU Output Buffer ---
        for (size_t i = 0; i < samplesRead; ++i) {
            // Normalize cuFFT IFFT output
            double real_part = h_ifft_out_gpu[i].x / nx;
            // Clamp values
            if (real_part > 32767.0) real_part = 32767.0;
            else if (real_part < -32768.0) real_part = -32768.0;
            // Cast to short
            outputBufferGpu[i] = (short)real_part;
        }

        // --- Write Outputs ---
        outFileCpu.write(outputBufferCpu, samplesRead); // Write CPU path result
        outFileGpu.write(outputBufferGpu, samplesRead); // Write GPU path result

    } // End while loop
```

```
    printf("\nProcessing finished. Processed %d chunks.\n", chunk_count);

    // --- Timing Results ---
    if (chunk_count > 0) {
        double avg_cpu_ms = (double)total_cpu_path_duration_us / chunk_count / 1000.0;
        double avg_gpu_ms = (double)total_gpu_path_duration_us / chunk_count / 1000.0;

        printf("\n--- Timing Comparison (Average per %d-sample chunk) ---\n", BUFF_SIZE);
        printf("CPU Path (FFTW Fwd + CPU Filter + FFTW Inv + Prepare Output): %.4f ms\n",
avg_cpu_ms);
        printf("GPU Path (H->D + cuFFT Fwd + D->H + CPU Filter + H->D + cuFFT Inv + D->H + Sync +
Prepare Output): %.4f ms\n", avg_gpu_ms);
        printf("-----------------------------------------------------\n");

        fprintf(log_fp, "Processed %d chunks of size %d.\n", chunk_count, BUFF_SIZE);
        fprintf(log_fp, "Filter Target: %.1f Hz, Width: %d bins\n", freqToFilter, filterWidth);
        fprintf(log_fp, "Average CPU Path time per chunk: %.4f ms\n", avg_cpu_ms);
        fprintf(log_fp, "Average GPU Path time per chunk: %.4f ms\n", avg_gpu_ms);
        fprintf(log_fp, "CPU output saved to: %s\n", wavfileout_cpu);
        fprintf(log_fp, "GPU output saved to: %s\n", wavfileout_gpu);
    } else {
        printf("\nNo data processed, skipping timing results.\n");
        fprintf(log_fp, "No data processed.\n");
    }
    fftw_destroy_plan(plan_forward_cpu);
    fftw_destroy_plan(plan_backward_cpu);
    fftw_free(h_fft_in);
    fftw_free(h_fft_out_cpu);
    fftw_free(h_ifft_out_cpu);
    fftw_free(h_fft_out_gpu_temp);
    cufftDestroy(plan_forward_gpu);
    cufftDestroy(plan_backward_gpu);
    cudaFree(d_fft_data);
    free(h_ifft_out_gpu);
    fclose(log_fp);
    free(wavfileout_cpu);
    free(wavfileout_gpu);
    free(logfile);
    return 0;
                                                    }
```

I used majority of the source provided by Professor with additional benchmarking and attenuation code. Additionally, I dislike using std:: so used namespace std. Apart from that source code is similar to the one provided.

SPECTOGRAM



As seen in spectrogram there is a vertical blue line at 10Khz indicating that the required frequency was attenuated. Neighboring frequencies are still visible (9.5khz, 10.5 Khz) because we are using 2 bins and each bin spans 2.93 Hz (Fs/N per bin).

Timings

Average CPU Path time per chunk: 0.2391 ms
Average GPU Path time per chunk: 0.1696 ms

Discussion

Initially, I was getting higher timing for GPU but I realized that's because I was calculating data transfer (Device -> Host and back to Device). After reading prompt carefully, I used benchmarking only for the forward and inverse Fourier transform calculations resulting in more consistent results.