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);

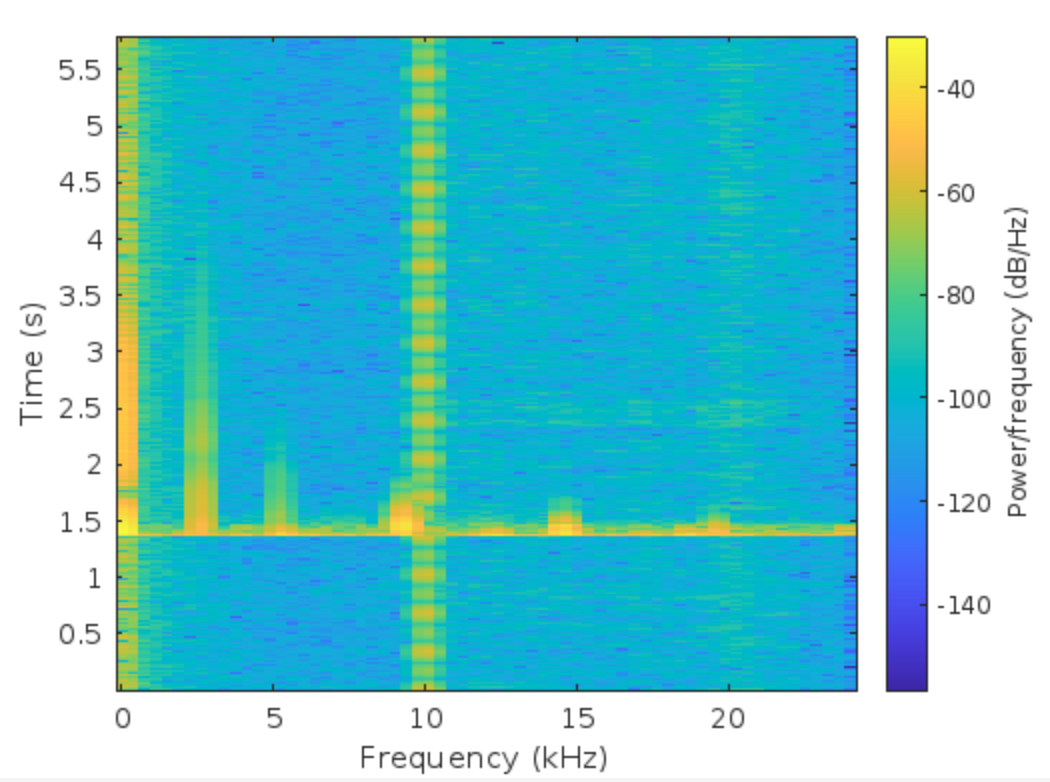
free(wavfileout\_cpu);

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.