# Dynamic Audio Normalization Parallel Implementation (DANPI)

Jeremy Carleton

June 7, 2022

CS/EE 147 GPU Computing and Programming

Final Project Report

# 1. Introduction

The name DANPI stands for Dynamic Audio Normalization Parallel Implementation. DANPI implements parts of the algorithm used in the Dynamic Audio Normalizer library by LoRd_MuldeR in a way that uses GPUs to accelerate processing. The goal of this algorithm is to "'even out' the volume of quiet and loud sections" of an audio file [1]. It does so by grouping the audio samples into frames and determining the gain for each frame that will cause the audio in that frame to achieve the desired volume.

In the Dynamic Audio Normalizer library, the user is given the option to specify a target peak amplitude or a target root-mean-square (RMS) amplitude for audio samples. In DANPI, the target is always given as an RMS amplitude. The RMS amplitude is a better indicator of how loud an audio signal will sound than the peak amplitude since it takes into account all samples rather than just the sample with the greatest magnitude. However, as with the Dynamic Audio Normalizer library, an audio sample is not permitted to exceed a certain upper limit (0.95 by default).

The first three stages of the algorithm used in DANPI directly match the first three stages of the algorithm used in the Dynamic Audio Normalizer library. First, each frame is analyzed in isolation to determine an initial gain value that satisfies the target RMS amplitude. Next, a minimum filter is applied, forcing each frame to take the smallest gain within a certain neighborhood. After that, a gaussian filter spanning three standard deviations on either side is applied to the gain factors. The purpose of these filters is to avoid distortion and smooth the transitions between low and high gain.

In the fourth stage of the algorithm used in DANPI, the gain factors for each frame determined in the first three stages are directly applied to the samples in the frame. This differs from the Dynamic Audio Normalizer library, which uses linear interpolation to determine the per-sample gain from the frame gain. This does not usually have a significant impact on the results since in DANPI, the frame length can be kept small without increasing the runtime by an unacceptable amount.

A major difference in implementation between DANPI and the Dynamic Audio Normalizer library is that the Dynamic Audio Normalizer library applies the aforementioned algorithm on each frame one at a time, whereas in DANPI each stage is applied to the entire set of frames at once. The Dynamic Audio Normalizer library is designed so that it can process an incoming stream of audio, which makes processing one frame at a time necessary. However, DANPI is designed to operate on existing audio files so that it can take advantage of parallelization by performing each operation on as much data as possible.

# 2. Methods for Accelerating the Application

Each of the four stages in the normalization process are performed on the GPU through four kernels. This section details the general strategies used to allow operations to be parallelized and improve efficiency.

### 2.1 Determining the Initial Gain Factors

The first stage of normalization is performed by the kernel called Analyze_Kernel. The procedure used to determine the initial gain values for each frame closely follows the reduction pattern. For each frame, two values are calculated: the RMS amplitude of the existing samples and the maximum magnitude of the existing samples. In order to calculate the RMS amplitude with an operation that is suitable for reduction, the values of the samples must first be squared. Then, the sum of all the samples can be calculated using reduction and the average can be taken at the end.

To obtain both the maximum magnitude and the RMS amplitude within a single reduction kernel, the operations are performed in parallel with each other, where each thread performs the same step in both reduction trees on each iteration. Running two reduction trees in parallel requires two separate arrays, so the shared memory allocated for the kernel is partitioned appropriately.

To reduce the amount of shared memory that must be allocated for each block, during the retrieval stage each thread takes two samples from global memory and immediately performs the first reduction step on those samples. The results are then saved to the arrays in shared memory in the location corresponding to the thread's index. This halves the amount of shared memory used for the reduction trees.

The reduction operations themselves are optimized to minimize warp divergence. At each step, the operation is performed by threads with adjacent indices and the results are compacted towards the front of the corresponding array in shared memory. Finally, accesses to the array of samples in global memory are coalesced to take advantage of DRAM bursting.

### 2.2 Applying Filters to the Gain Factors

The second and third stages of normalization which filter the gain factors are performed by the kernels called Min_Filter_Kernel and Gaussian_Filter_Kernel. The methods used to accelerate the filtering process are identical between the minimum filter and the gaussian filter. With the exception of accesses to the coefficients for the gaussian filter, all accesses to global memory are coalesced. All elements needed for each thread to apply the filter to a single element are cached in shared memory before processing begins. The thread blocks for the filter kernels are sized such that there are at least enough threads to retrieve one side of the filter window in one pass. When the filter width is large, this leads to significant data reuse. For each element to be filtered, all of the operations required are performed by a single thread, and all threads in a block are able to operate in parallel.

### 2.3 Applying the Final Gain Factors

The fourth stage of normalization is performed by the kernel called Gain_Kernel. The thread blocks used for applying the final gain factors to the audio samples are sized based on the length of a frame so that all threads in a block apply the same gain. Each thread is mapped to a single audio sample and operates on that sample.

# 3. Implementation Details

## 3.1 Reading and Writing Audio Files

The AudioFile library by Adam Stark (see [2]) is used to load and save audio files in the .wav format, which is the only format supported by DANPI. DANPI can only process audio files with a single channel. Files are loaded from and saved to two separate directories which must be located within the directory from which the program is run (see section 4.3 for more details).

## 3.2 Configurable Parameters

Several parameters can be modified to change the behavior or performance of DANPI. Most of these parameters are also found in the Dynamic Audio Normalizer library. The first parameter is the target RMS amplitude. This is the RMS amplitude that the algorithm attempts to achieve at each frame. Due to the minimum and gaussian filters which are applied after the gain factor needed to achieve this RMS amplitude is determined, the actual RMS amplitude for a normalized audio file is usually lower than the specified value.

The second parameter is the frame length, or how many audio samples are included in each frame. This affects both the audio quality of the normalized audio file and the execution time of the algorithm. A smaller frame length generally causes the volume to be more uniform since the parts of the audio file which require a higher gain to achieve the desired RMS amplitude are less likely to be held back by the parts which require a lower gain. For an analysis of how the frame length affects the execution time of the algorithm, see section 5.2. The possible values for the frame length are restricted due to how the kernels are implemented and the limit of 1024 threads per thread block on compute capability 7.5 GPUs. Section 4.4 indicates what values are valid for the frame length.

The third parameter is the number of frames on either side of the current frame to consider when applying the minimum filter. Because the minimum filter brings the gain factor for a frame down to the lowest initial gain factor calculated for a frame within the neighborhood covered by the filter, increasing the width of the filter will cause the audio file to have a lower volume overall. Furthermore, regions of high volume between regions of low volume will be reduced in length. For an analysis of how filter width affects the execution time of the algorithm, see section 5.3. If this parameter is set to zero, then the minimum filter will have no effect.

The fourth parameter is the number of frames on either side of the current frame to consider when applying the gaussian filter. Increasing the width of this filter will create smoother transitions between regions with low gain and regions with high gain. As with the minimum filter, setting this parameter to zero will cause the gaussian filter to have no effect.

The fifth and sixth parameters are the upper and lower limits on what gain can be chosen as the initial gain for a frame. The final gain that is applied to the samples will always be lower than the upper limit. As a result of the filtering process, the final gain may be lower than the lower limit.

The seventh parameter is the maximum allowable value for an audio sample, which puts an upper bound on the magnitude that a sample can have in the normalized audio file. This takes

precedence over the lower limit established for the gain, so if a lower gain is needed to keep the frame below the maximum allowable value, it will be used. If a sample in the input file already exceeds this value, then it is possible for the sample to exceed this value in the output file if the filtering process causes the gain factor for its frame to increase.

The eighth parameter is the block size used for both the min filter kernel and the gaussian filter kernel. The program will automatically adjust this value if it is greater than the number of frames or less than the half-width of the filter.

### 3.3 Analyze_Kernel Implementation Details

In the setup for Analyze_Kernel, enough memory is allocated in the GPU's global memory to store all of the audio samples and store a gain factor for each frame. The audio samples are copied to the device right before the kernel is launched. Each block covers a full frame, so the number of thread blocks in the one-dimensional grid equals the number of frames. Each block only uses a number of threads equal to half the frame length because the reduction method used has each thread retrieve two samples. As explained in section 2.1, shared memory is allocated to each block to store the reduction trees. This memory is allocated dynamically since the size of the reduction tree depends on the length of the frame.

Analyze_Kernel is divided into three stages. In the first stage, each thread retrieves one or two of the corresponding frame's samples. Because the frame length is a power of two, the last frame is the only frame for which a thread may need to retrieve only one sample (i.e., when the number of samples is not divisible by the frame length). The first element for each thread is in the first half of the frame's audio samples, and the second element is in the second half. If a thread retrieved two elements, it calculates the maximum magnitude between the elements as well as the sum of their squares.

The second stage is the reduction stage which calculates the maximum magnitude of all samples in the frame as well as the sum of all samples squared. The number of reduction stages required is the base 2 logarithm of half the frame length. In the final stage, a single thread in the thread block determines the RMS amplitude of the frame by dividing the sum of squares by the number of samples and taking the square root. The gain factors which would bring the frame to the maximum allowable value and the target RMS amplitude are also calculated, and the initial gain factor is determined based on these factors and the limits imposed by the user through the configurable parameters (see section 3.2).

When the initial frame gain is calculated at the end of the third stage, it is stored in the array allocated in global memory before launching the kernel. Each block stores the value which it calculated in the index corresponding to its index in the grid. This array of gain factors is kept in global memory to be operated upon by the filter kernels. The audio samples are kept in global memory as well, since they are used again in Gain_Kernel.

### 3.4 Min_Filter_Kernel Implementation Details

In preparation for Min_Filter_Kernel, an array is allocated in global memory to store gain factors which have been filtered by the minimum filter. Each thread block for the kernel always contains at least enough threads to match the number of elements covered by one side of the minimum filter. The thread block size can be larger depending on the size specified by the user.

The one-dimensional grid of block is sized based on the block size so that all gain factors are filtered. Shared memory is allocated for the required number of gain factors to be cached while the threads in a block use them to apply the filter. The total number of gain factors used is equal to the block size plus twice the filter width on one side of the center of the filter.

Min_Filter_Kernel is divided into two stages. In the first stage, gain factors are retrieved from global memory and stored in shared memory in three steps. First, the gain factors which will be filtered are loaded. For all but the last thread block, each thread retrieves one gain factor in this step. In the second and third steps, the additional gain factors that need to be considered when filtering gain factors near the edges are retrieved. For these steps, there will be threads which do not need to retrieve any gain factors when the filter window extends beyond the first or last gain factors in the global memory array.

In the second stage, the filter is applied by finding the minimum value in the window around each gain factor that is being filtered. In cases where there are enough gain factors on either side of the those being filtered, determining the first index and last index for the loop is straightforward since the gain factor that is being filtered is centered within the filter window. In cases where there are not enough gain factors on one side to create the entire window, the first and last indices to include are calculated by considering how far away the gain factor being filtered is from the overall first and last gain factors.

### 3.5 Gaussian_Filter_Kernel Implementation Details

Before launching Gaussian_Filter_Kernel, the program running on the CPU calculates the coefficients for the gaussian filter, which are the weights applied to the gain factors included in the calculation. The array of coefficients is copied to global memory before launching the kernel. The grid size, thread block size, and amount of shared memory per block are determined in the same way that they are for Min_Filter_Kernel.

The process applied within the kernel is identical to the process used in Min_Filter_Kernel except for the calculations performed on the gain factors in the filter window. Each gain factor is multiplied by the coefficient corresponding to its position in the filter window, and the sum of the results is taken as the new gain factor. The results from the gaussian filter are stored in the array which originally held the initial gain factors determined in Analyze_Kernel.

### 3.6 Gain_Kernel Implementation Details

Since the audio samples and the filtered gain factors are both in global memory when Gaussian_Filter_Kernel completes its execution, no additional memory needs to be copied before executing Gain_Kernel. Gain_Kernel uses a grid size equal to the number of frames and a thread block size equal to the frame length. Within the kernel, each thread applies the gain corresponding to its block to the sample in global memory corresponding to its index in the overall grid. After the kernel executes, the samples are copied from global memory directly into the AudioFile object that they were initially retrieved from.

### 3.7 Testing and Logging Execution Time

The chrono utility library is used to measure the time elapsed between points in the program. The macros which control the timers are located in the header file utility_macros.h. Five durations are recorded each time an audio file is processed. The first tracks the time from before the audio file is loaded to after the normalized audio file is saved. The second tracks the time from before the audio samples are copied to global memory to when Analyze_Kernel finishes. The third tracks the time from then until Min_Filter_Kernel finishes. The fourth tracks the time from then until Gaussian_Filter_Kernel finishes. The fifth tracks the time from then until Gain_Kernel finishes. In all cases, the time elapsed is measured in microseconds. Between the end of one timer and the beginning of the next timer, the duration that was recorded is logged to a comma-separated values (CSV) file. This is done using a CSV file generator from Vladimir Shestakov (see [3]) which is contained in the header file csvfile.h. The code used to write to the CSV file is located in the header file log_csv.h.

It is common for the time required to setup and run Analyze_Kernel to be significantly longer for the first audio file processed than for subsequent audio files. One way to prevent this from distorting results when measuring execution time is to process an extra audio file before the ones for which you want to measure the execution time. Since there is normally not a clearly defined order in which audio files are processed, a check is included which forces an audio file named “Dummy.wav” to be processed before any other file. Including an audio file with that name in the list of audio files to be normalized ensures that the execution time bug will not affect the other files.

## 4. Building and Running DANPI

### 4.1 Software and System Requirements

DANPI is intended to be run in a Linux environment. A minimum CMake version of 3.18 is enforced when building DANPI. The compiler used must be able to compile with the C++17 standard. The target GPU architecture level is 7.5, which matches the NVIDIA RTX 2070 GPUs on the University of California, Riverside’s Bender server.

### 4.2 Building the Program

After cloning the DANPI GitHub repository, use the following steps to build the code and generate the executable:

1. Initialize the AudioFile submodule (ie. `git submodule update --init --recursive`)
2. If desired, create a build directory and navigate to it so that the build products are placed in that directory after the next step
3. Run cmake targeting src/ as the source directory. The build files will appear in your current directory.
4. Run make to compile the program and generate the executable. The executable will be named “DANPI”.

### 4.3 Running the Program

Before running the executable, create two directories: “AudioSamples” and “Normalized”. Place all .wav files to be processed in the directory named “AudioSamples”. After executing the program, a .wav file will appear in the directory named “Normalized” for each file processed. These files will have the names of the input files with a “_n” appended to them. To perform normalization using the default parameters, run the executable named “DANPI” with no arguments.

### 4.4 Modifying Parameters Through the Command Line

Table 4.1 lists the parameters that can be modified using command line arguments. These parameters are described in section 3.2. Arguments must be given in the order in which they appear in the table, so in order to specify the frame length an argument must be given for target RMS value as well. To keep the default value for a parameter while passing in an argument, use the letter ‘d’.

Table 4.1 List of parameters that the user can modify

| Parameter | Valid Inputs | Default |
|---|---|---|
| Target RMS Value | Between 0 and 1, inclusive | 0.06 |
| Frame Length | Any power of 2 between 2 and 1024, inclusive | 1024 |
| Minimum Filter Width One Side | Any positive integer less than or equal to 1024 | 15 |
| Gaussian Filter Width One Side | Any positive integer less than or equal to 1024 | 15 |
| Gain Upper Limit | Any positive real number | 10 |
| Gain Lower Limit | Any positive real number | 0.1 |
| Maximum Allowable Value | Greater than 0 and less than or equal to 1 | 0.95 |
| Filter Kernel Block Size | Greater than 0 and less than or equal to 1024 | 30 |

### 4.5 Testing Performance

By default, a comma-separated values (CSV) file named Log.csv is written to the directory that DANPI is run from. The file includes a separate entry for each execution of the algorithm. Included in the entry is the file number indicating in what order files were processed, the number of samples in the audio file, a summary of the parameters which affect performance,

and the execution times for the four steps of the algorithm along with the overall execution time. All times are given in units of microseconds. To disable the measurement of execution time and the generation of this log file, comment out the line in main.cu which defines TEST_RUNTIME. Figure 4.1 shows an example of what Log.csv looks like after processing a single audio file.

| File # | # Samples | Frame Length | Filter Width | Analyze Time | Min Filter Time | Gauss Filter Time | Gain Time | Full Duration | Filter Block Size |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 893423 | 128 | 7 | 118964 | 19 | 41 | 1287 | 285451 | 30 |

Figure 4.1

# 5. Performance Evaluation

## 5.1 Comparison Between Serial and Parallel Implementations

This section compares the performance of DANPI against the performance of a second "serial" algorithm which performs the same steps without using a GPU. The source code for this program can be found in the GitHub repository inside the folder called "Serial Implementation". Both programs were tested on the Bender server. The main goal in comparing the serial and parallel implementations was to determine how the performance improvement of the parallel implementation over the serial implementation would scale with audio files of increasing size. To test this, the execution times for both programs were measured (see section 3.7) for five trials each, where in each trial a set of eleven audio files were normalized (this does not include the dummy audio file used for the parallel implementation). The lengths used were 10 to 50 seconds on 10-second increments and 1 to 6 minutes on 1-minute increments. All user-selectable parameters were kept at their default values.

The average acceleration factor, or the ratio of the execution time of the serial implementation to the execution time of the parallel implementation when normalizing the same audio file, is plotted as a function of the number of samples in the audio file in figure 5.1. This measurement of execution time takes into account the full process, including the time required to load and store the audio samples using the AudioFile library. For the file with 440687 samples (10 second duration), the serial implementation was about 25% faster. The acceleration factor rapidly increases as the number of samples increases, achieving an acceleration of 13% at 1331180 samples (30 second duration). However, acceleration factor does not increase much beyond this, as it quickly flattens out. As the rest of this section will show, this results from the time required to perform tasks that were not parallelized (namely loading and storing audio samples and outputting text to the console) overshadowing the time required to perform normalization rather than a failure of the parallel algorithm to improve performance.

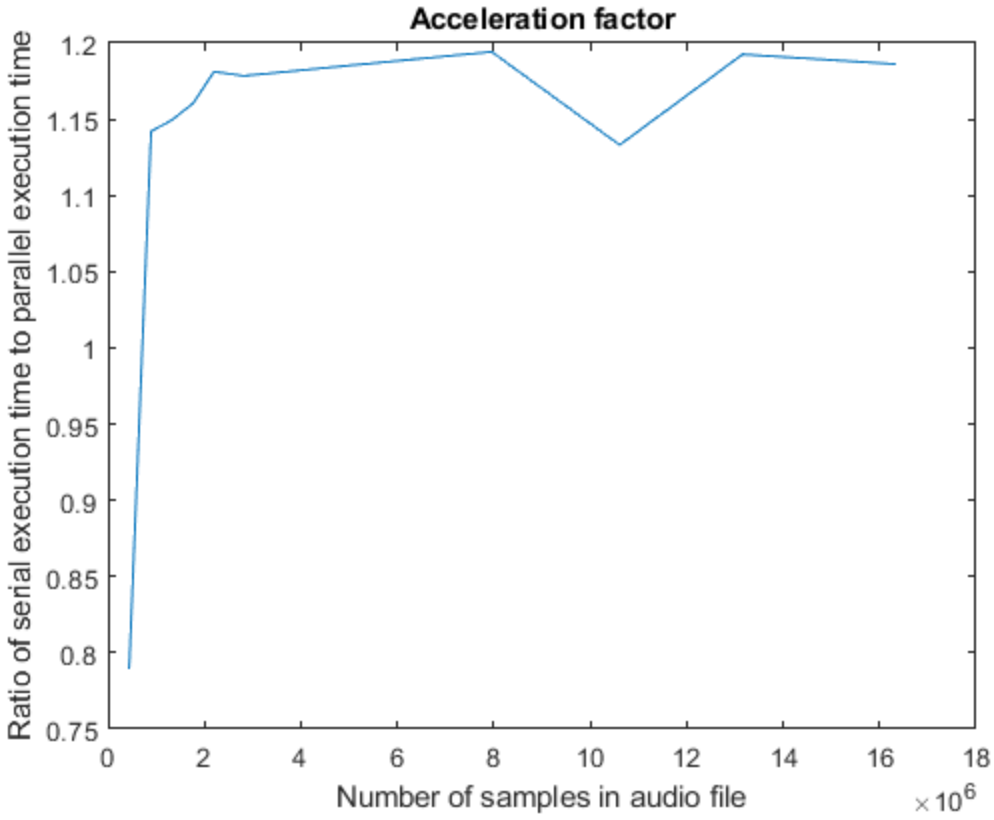


Figure 5.1

On average, the normalization steps (finding initial gain factors for each frame, applying the minimum and gaussian filters to the gain factors, and applying the final gain to the samples) account for 20% of the time required for the serial implementation to process an audio file. Of this, more than two-thirds are spent on determining the initial gain factors. In contrast, the normalization steps account for less than 3% of the time required for the parallel implementation to process an audio file. This was consistent across all audio files tested. Thus, the vast majority of the execution time is spent on processes which the program does not attempt to parallelize, and variations in the execution time for these processes can overshadow the execution time for the normalization steps. Consequently, the rest of this comparison will focus on the execution time measured for only the normalization steps.

The acceleration factor calculated based on the execution time spent finding the initial gain factors is plotted in figure 5.2. For the 10-second audio file, the parallel implementation is already almost 12 times faster than the serial implementation. This increases to 16 times for the 50-second audio file. The highest acceleration factor is achieved with the 4-minute audio file with over 10 million samples. Afterwards, the acceleration factor decreases, suggesting that a limit has been reached.

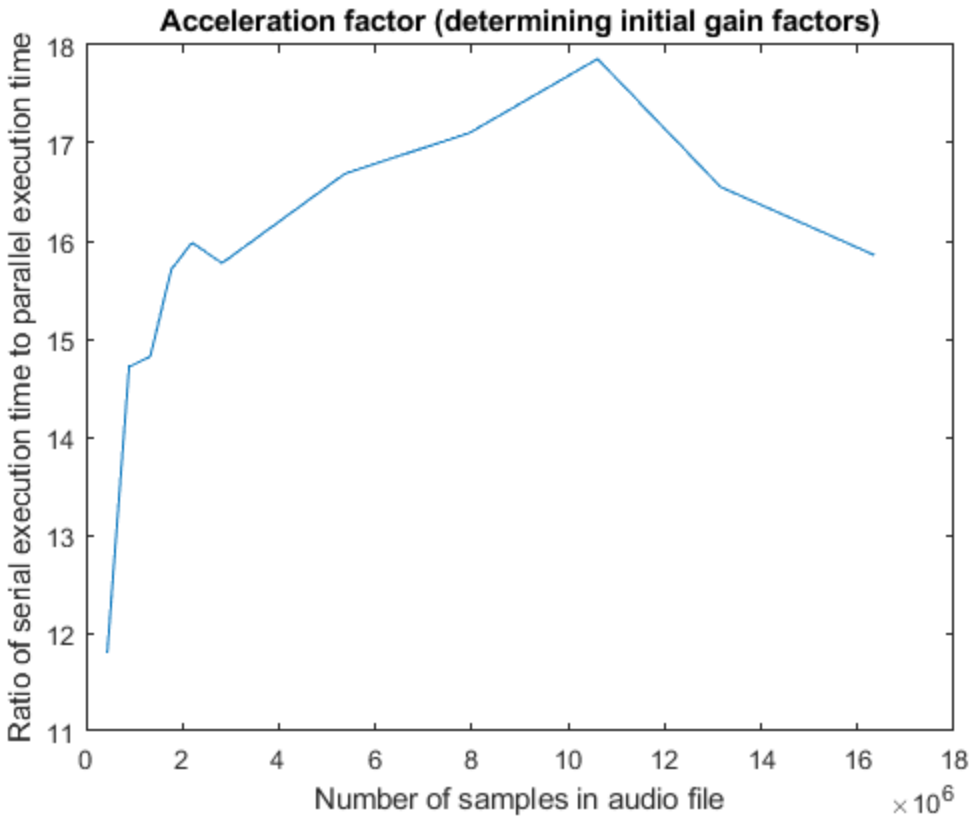


Figure 5.2

The acceleration factor calculated based on the execution time spent applying the minimum and gaussian filters combined is plotted in figure 5.3. Similar to figure 5.2, the acceleration factor increases until 10 million samples is reached. In this case, however, the growth is more linear. The greatest acceleration factor achieved is also much higher at just over 73 times faster.

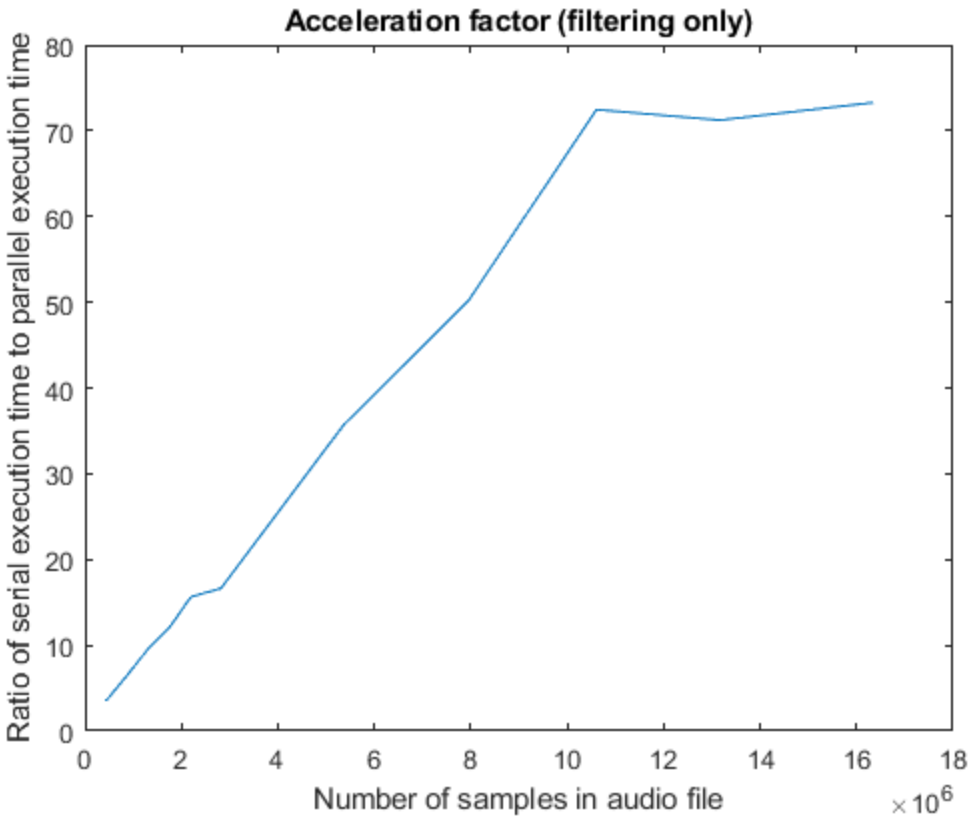


Figure 5.3

The acceleration factor calculated based on the execution time spent applying the final gain is plotted in figure 5.4. Here, the acceleration factor fluctuates across of the files tested. However, it remains above 3.8, showing that the parallelization does provide an increase in speed.

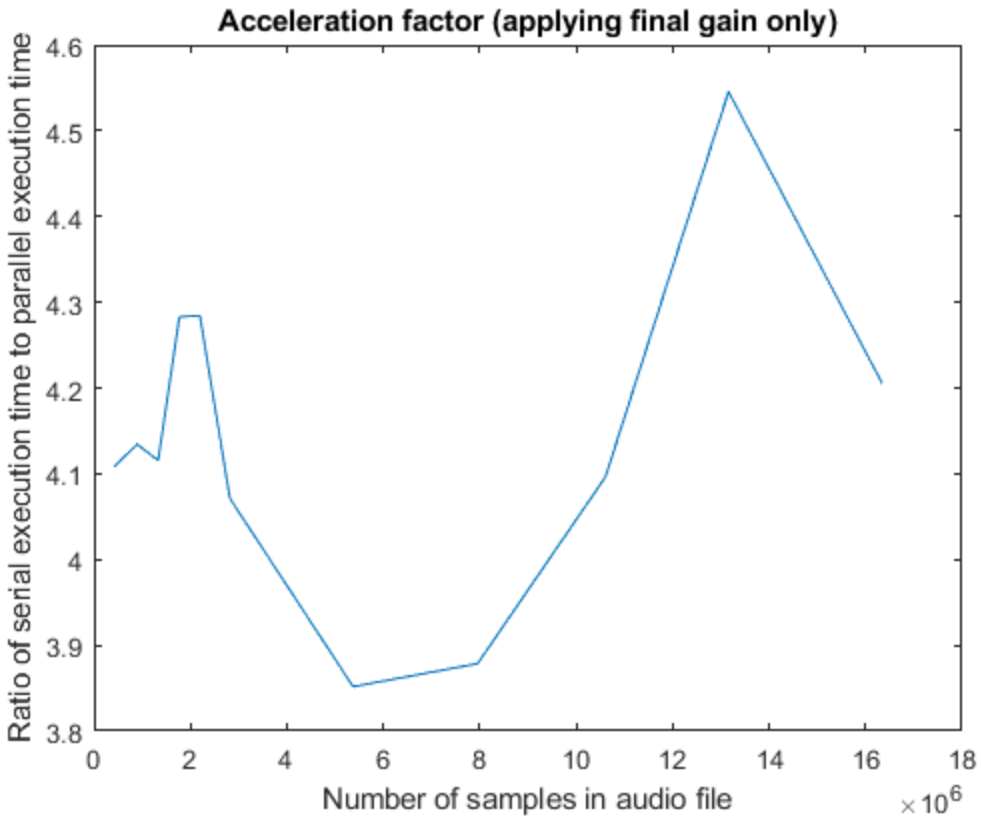


Figure 5.4

Finally, calculating the acceleration factor based on the four steps of the normalization process combined gives the plot in figure 5.5. Compared to figure 5.1, the greatest acceleration factor is still achieved with the 4-minute audio file, but the parallel implementation is actually 12 times faster at performing normalization than the serial implementation for files of that size. On the other hand, for short audio files the parallel implementation is still at least nine times faster.

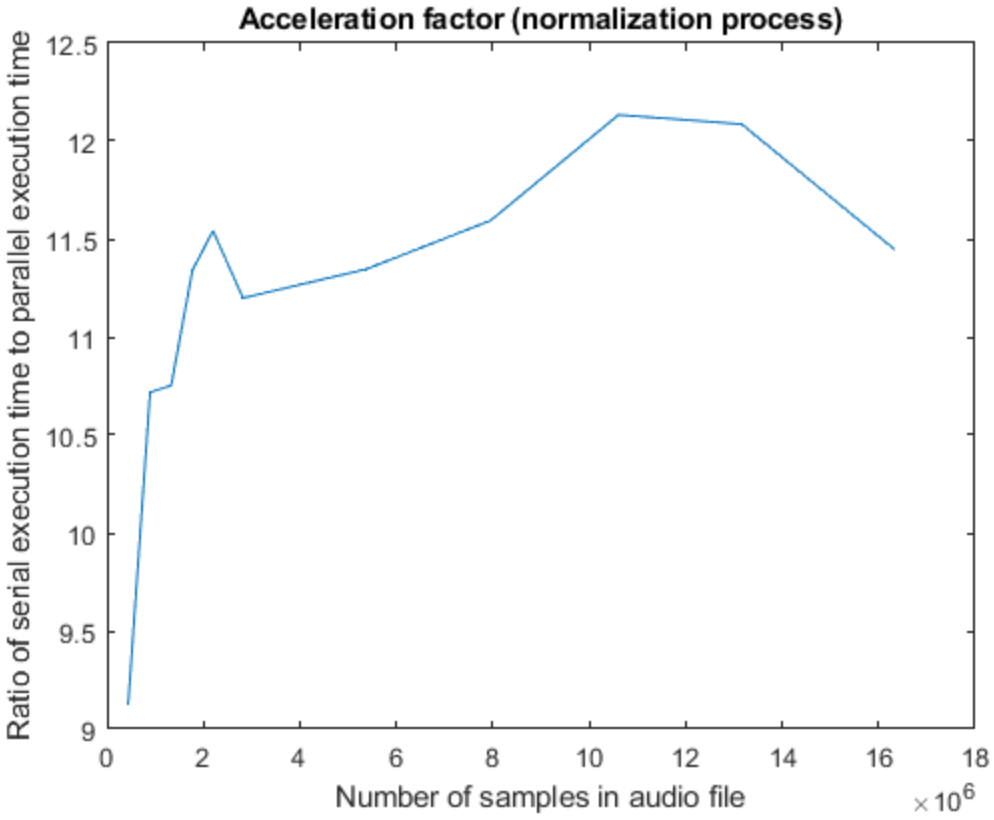


Figure 5.5

**5.2 Effect of Varying Frame Length on Execution Time**

The frame length used in DANPI can range from as low as two samples per frame to as high as 1024 samples per frame. Because the number of frames, and by extension the frame length, determines how many separate reductions need to take place in Analyze_Kernel, it is expected that the execution time for that kernel will depend on the frame length. Since the number of frames also determines how many gain factors there are to be filtered, it is expected that the execution time for the filtering kernels will depend on the frame length. It is less clear whether the execution time of Gain_Kernel should be influenced by the frame length. In this section, the results of five trials of normalizing a 3-minute audio file with each possible frame length (while keeping the other parameters at their default values) are analyzed. The analyses in sections 5.3 and 5.4 will use a similar method.

Figure 5.6 shows the execution time in microseconds of each step of the normalization process. The first three steps all show a similar inverse relationship between frame length and execution time. When applying the final gain factors, there is no clear relationship. These results make intuitive sense since a larger frame length means that the input samples are divided into fewer, larger reduction trees in Analyze_Kernel and that there are fewer gain factors to filter, but in Gain_Kernel the same number of operations needs to be performed by the same number of threads regardless of the flame length. It should be noted, however, that the dip in the execution time for applying the final gain occurs at a frame length of 32, and for that frame length each

block in Gain_Kernel has exactly one warp. This suggests that running Gain_Kernel in this specific configuration may serve to reduce the execution time. The relationship between the frame length and the overall time required to process a file (including loading and saving the audio file) is plotted in figure 5.7. In this plot, the individual data points for each trial are shown, indicating how much deviation there is from the average. Based on this plot, if the goal is to achieve the fastest possible execution time, then a frame length of at least 64 should be used. The fastest execution time that was achieved for determining the initial gain factors is 14 milliseconds. The fastest execution times achieved for the minimum and gaussian filters were 29 and 38 microseconds, respectively.

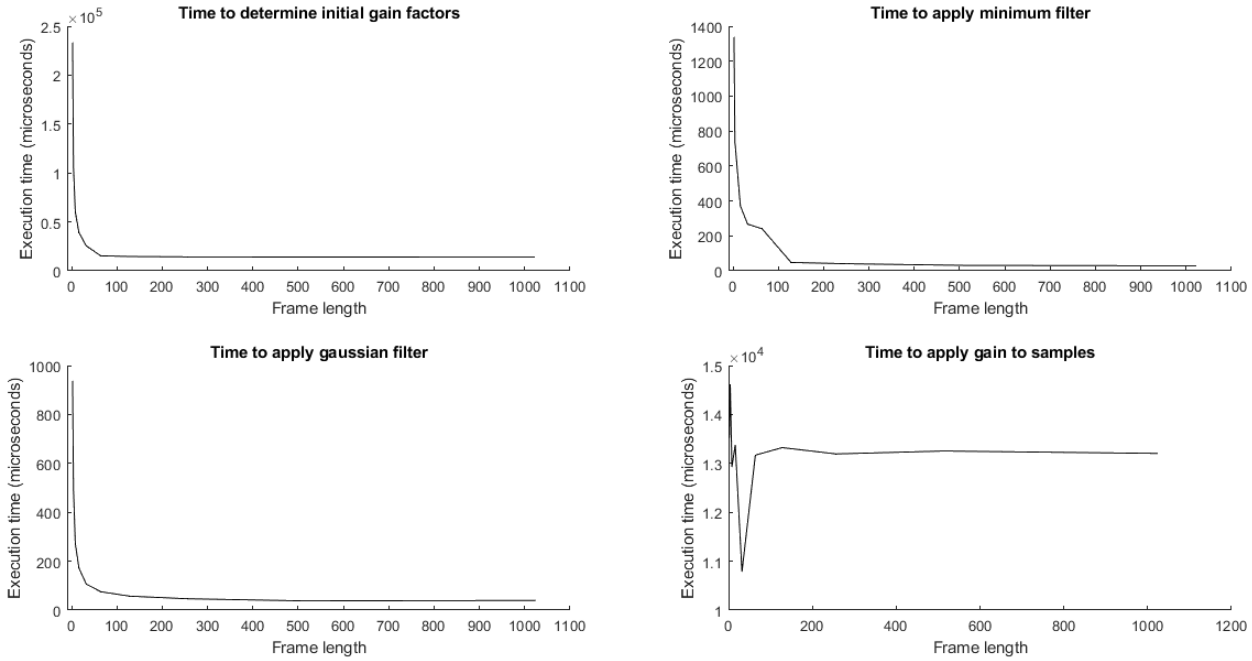


Figure 5.6

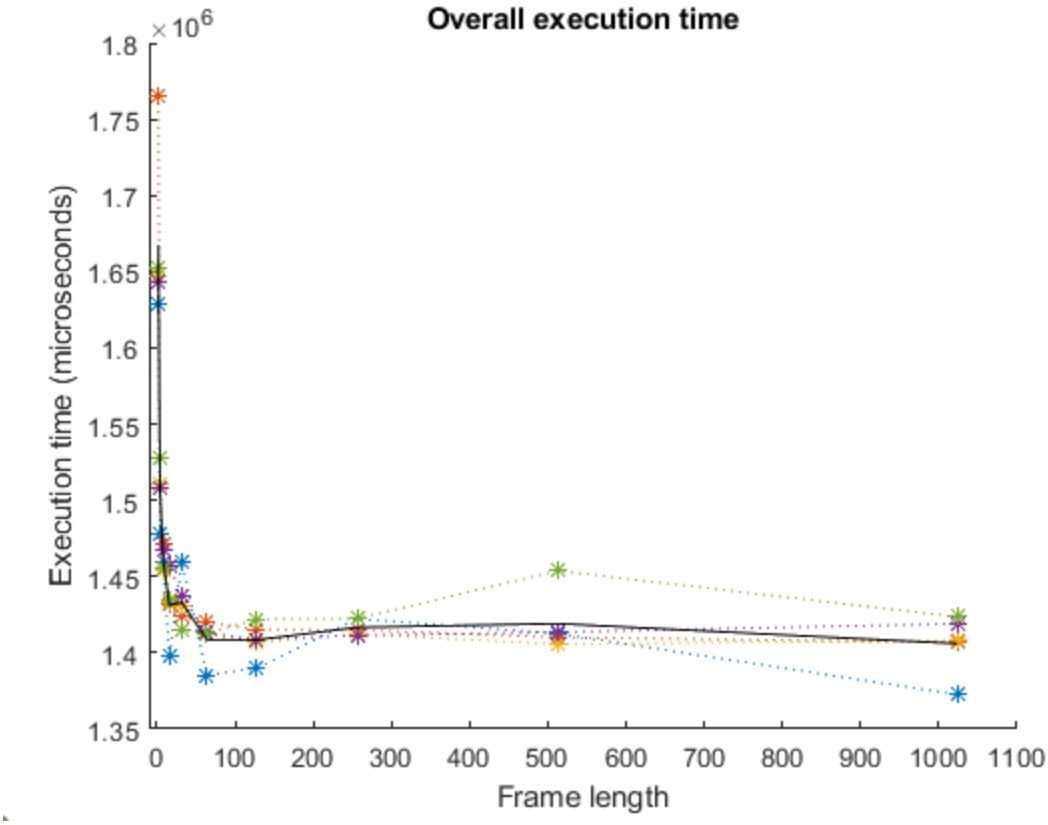


Figure 5.7

### 5.3 Effect of Varying Filter Width on Execution Time

DANPI allows for a wide range of filter widths to be used when applying the minimum and gaussian filters to the gain factors, with the maximum overall width being 2049. The main purpose of changing the filter width is to achieve better sound quality, but it has an impact on the execution time of the filtering kernels. Figure 5.8 shows the relationship between the overall filter width and execution time required to apply both of the filters, again using the average of five trials. It can be seen that increasing the filter width increases the execution time. This is expected because each time the filter width is increased by one, all threads in Min_Filter_Kernel and Gaussian_Filter_Kernel need to perform one or two additional computations depending on their thread index.

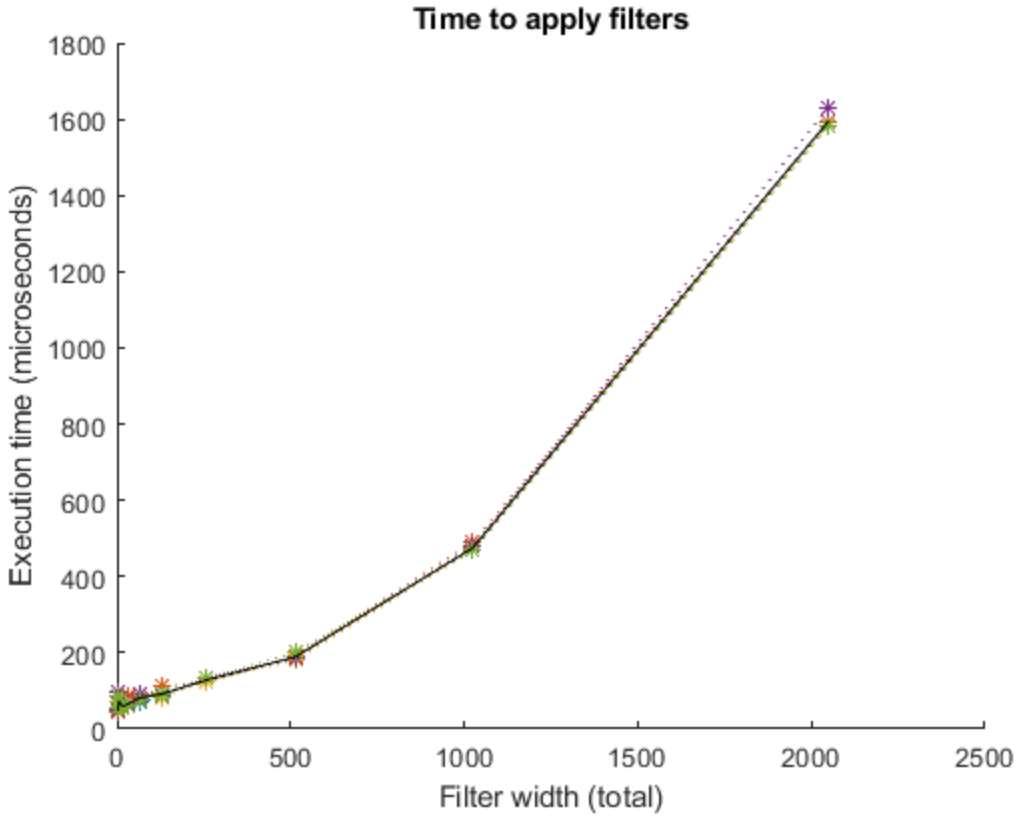


Figure 5.8

**5.4 Effect of Varying Filter Kernel Block Size on Execution Time**

Increasing the block size used in the filter kernels causes a greater number of samples to be filtered by each block in Min_Filter_Kernel and Gaussian_Filter_Kernel while reducing the total number of blocks used. I expected that increasing the filter block size would decrease the time required to apply the filters by enabling more data reuse. However, the test results shown in figure 5.9 indicate the opposite relationship. For the largest filter block sizes, it is possible that the number of idle threads in the last block on the grid impacted performance, but this does not fully explain the phenomenon.

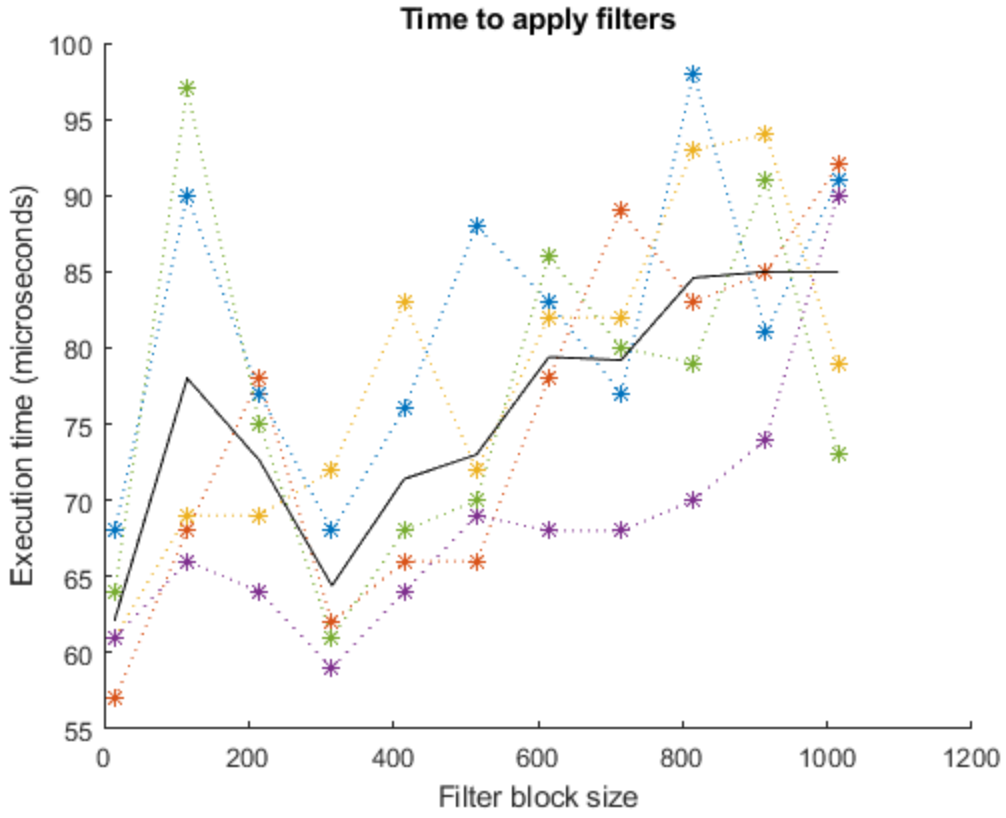


Figure 5.9

## 6. Challenges faced

Designing Analyze_Kernel to efficiently process the data in a frame required a large portion of the time spent creating DANPI. I decided early on to use reduction for processing the data, but extending the reduction algorithm to perform two reductions in parallel was a challenge. One potential improvement to Analyze_Kernel could be to better distribute the computations between threads during the reduction process. Currently, each thread must compute two operations on each iteration, and the number of threads performing operations is halved on each iteration. After the first iteration, one of the two computations could be given to the threads that would otherwise be idle, allowing twice as many threads to perform half as many computations. This would allow each block to finish faster, although it may fail to increase performance if it limits the number of thread blocks which are active.

When I was first designing Min_Filter_Kernel, I looked for a way to allow the filtered gain factors to be stored back in the same array that the initial gain factors were taken from. This would have eliminated the need for a temporary array of gain factors to be created. To allow the gain factors to be overwritten, the thread blocks would need to be synchronized so that no block attempts to overwrite data until that data has been retrieved by all thread blocks which need it. I experimented with using cooperative groups, which allow for all threads in a grid to be

synchronized [4], but could not achieve the desired effect. Ultimately, I used the workaround of creating the temporary array instead.

## 7. Link to Video Presentation

https://youtu.be/OZYipEa7qgM

## 8. References

[1] "DynamicAudioNormalizer," Oct. 28, 2019. GitHub Repository. [Online]. Available: https://github.com/lordmulder/DynamicAudioNormalizer (Accessed: Apr. 22 2022).

[2] A. Stark, "AudioFile," Jan. 15, 2022. GitHub Repository. [Online]. Available: https://github.com/adamstark/AudioFile (Accessed: May 18 2022).

[3] V. Shestakov, "CSV File Generator," Jan. 9, 2018. GitHub Gist. [Online]. Available: https://gist.github.com/rudolfovich/f250900f1a833e715260a66c87369d15 (Accessed: Jun. 3 2022).

[4] NVIDIA, Santa Clara, CA, USA. "CUDA Toolkit Documentation," [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#grid-synchronization-cg (Accessed: May 26 2022).

## 9. Breakdown of Contributions

| Task | Breakdown |
|---|---|
| Serial Implementation | Jeremy Carleton – 100% |
| Analyze_Kernel | Jeremy Carleton – 100% |
| Min_Filter_Kernel | Jeremy Carleton – 100% |
| Gaussian_Filter_Kernel | Jeremy Carleton – 100% |
| Gain_Kernel | Jeremy Carleton – 100% |
| Report & Presentation | Jeremy Carleton – 100% |