EECE4634 Final Report

“Chord Finder” - Identifying chords in guitar recordings

Tom Doyle, David Boullie

Project Description

The purpose of this project is to analyze the frequency content of a .wav file to determine its 3 most prominent frequency components. By obtaining these 3 frequencies, we can determine which notes are being played in the file, and thus which chord these notes represent. Since we identify only the top 3 frequencies, this project is best suited to recordings of simple chords known as “triads”, which are chords made up of 3 tones[1].

The project uses an algorithm known as the Fast Fourier Transform (FFT) to obtain frequency data from a given “.wav” file. This frequency data is then processed using a custom IP created in Vivado HLS to determine the top three frequencies.

Background

The Fourier Transform is a mathematical operation that transforms a time-domain signal into the frequency domain by decomposing it into a series of complex exponentials[2]. In other words, it can show how the intensity of a signal varies over frequency rather than over time (Fig 1). Since we are analyzing .wav files, we are transforming discrete samples. The particular implementation of the discrete-time Fourier Transform (DFT) we explored is known as the Fast Fourier Transform (FFT).

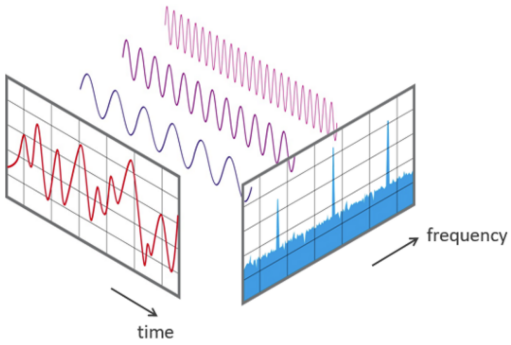

Figure 1 - Decomposition of time-varying signal into its frequency components using the DFT

The FFT is a divide-and-conquer algorithm that implements the Fourier transform for discrete-time signals. There are many different variations of the FFT, but the one most commonly used is the Cooley-Tukey algorithm[3], which we focused on in our project. The Cooley-Tukey makes use of the periodic nature of complex exponentials to break a DFT of size *N* into two DFTs, each of size *N/2*. The results of the two DFTs are then summed to obtain the full frequency spectrum of the original signal. A visualization of this method is shown below (Fig 2). This divide-and-conquer approach can be scaled to a sample size of an arbitrary power of two and applied recursively, which offers a high degree of parallelism and lends the algorithm well to a hardware-software exploration.

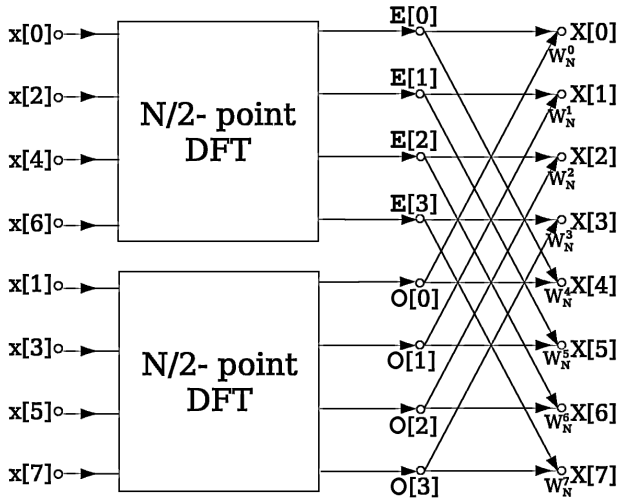

Figure 2 - Divide-and-conquer approach of the Cooley-Tukey algorithm

Code Description

The work for this project was done separated into two parts and developed in parallel. The first part involved decomposing the .wav file into audio samples with an appropriate size and sampling frequency, and performing an FFT on those samples in both hardware and software. This portion of the project was completed by Tom Doyle.

The second part of the project involved identifying the most prominent frequencies in a stream of incoming frequency data, and determining the resultant chord from the individual tones identified. This portion of the project was completed by David Boullie. The logical flow of the project is shown below (Fig 3).

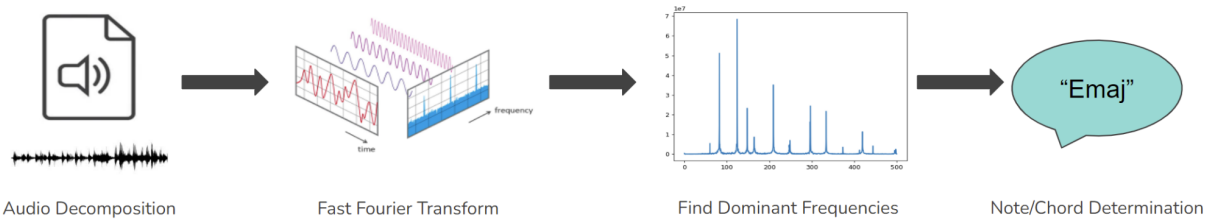

Figure 3 - Overall structure of project

Sampling / FFT

The first portion of this project involved obtaining discrete audio samples from .wav files, and performing an FFT on those samples. The .wav file audio samples were obtained using the SciPy.io Python library[4]. The audio files had a sampling rate of 44.1 kHz and were several seconds long, so they contained far too many samples to process in hardware. We decided we would sample the first second of audio to ensure our sample would contain all desired frequencies. The sample size was determined by the transform size of the FFT IP.

For example, our first hardware FFT operated on 1024 samples. Taking 1024 consecutive samples from the file would yield a frequency resolution of 44100/1024 = 43 Hz/sample, meaning we would only be able to distinguish between frequencies more than 43 Hz apart. Since we wanted to be able to distinguish between adjacent notes, this resolution was unacceptable. To be safe, we designed our resolution to be 1 Hz/sample by adjusting our sampling frequency to be equal to our sample size. For simplicity, we sampled the first second of audio for each overlay, with a sample size corresponding to the transform size of the overlay.

The FFT was performed using an overlay created in Vivado that streams the discrete time-domain audio samples over DMA to the Xilinx LogiCORE IP Fast Fourier Transform[5]. A block diagram is shown below (Fig 4).

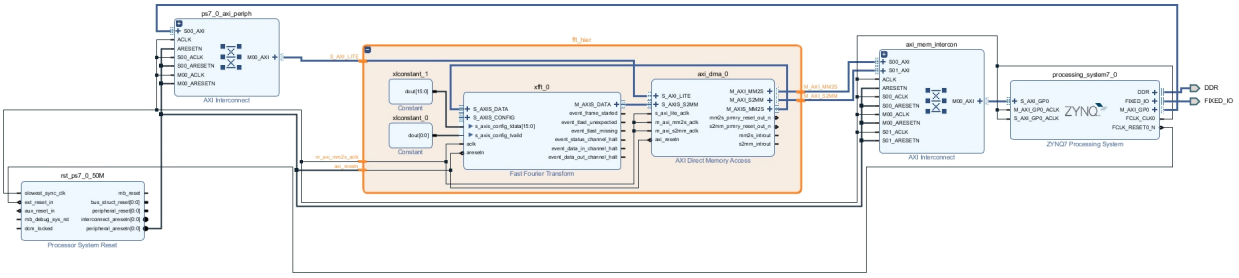

Figure 4 - Block diagram of FFT overlay

In total, 5 different overlays were created using this design, each with a different transform size. The 5 overlays operate on 1024, 2048, 4096, 8192, and 16384 samples, respectively. The FFT IP supports transform sizes up to 65536 samples, but we were unable to generate designs with transform sizes greater than 16384 samples due to memory constraints on the Zynq. No HLS

code was generated, so testing was done in a Jupyter notebook using the Pynq overlays from each Vivado project.

In order to compare tradeoffs between hardware and software implementations, the FFT was also performed using the NumPy FFT function. The NumPy FFT was also performed on the same 5 sample sizes to create apt comparisons. The results of the hardware and software implementations can be seen in the Results section.

Loudest Chord Algorithm

The second portion of this assignment involves creating a custom IP to extract the three most prominent frequencies detected in the sound file, also known as the triad. This IP takes in an AXI_STREAM stream of amplitudes coming from the FFT, which correspond to the “loudness” of recorded frequencies. Because the stream of inputs would be evaluated one sample at a time, the frequency of a given sample was to be determined by the index and an inputted scaling factor held in an IP register. Static variables (variables that don’t reset between function calls) were used to hold the running maximum amplitude and the corresponding frequency value. The output of the IP would be three of the loudest frequencies values using IP registers, which would be put through a “note-finder” function in software. While the algorithm to determine the three loudest frequencies was written from scratch, the Python code for finding a note from a given frequency was found through online resources[6].

The process of creating the “loudest chord” algorithm was to create an accurate implementation in software written in Python, then adapt it to C++ in Vivado HLS and compare using a testbench file before turning it into an IP. Testing of this stage occurred in parallel with the testing of the FFT IP block, so samples of FFT amplitude arrays were created in software to serve as a way to verify algorithm functionality. Sample audio files, which can be found in the “audio_files” folder on the GitHub page[9], were used in testing, and the loudest chords were confirmed by observing the FFT plot of the audio file and matching the values to the output of test functions. The scipy.io library was used for decoding the wav files, the numpy library was used for performing the software FFT, and the matplot.pyplot library was used to plot the soundbites and FFT data.

Early implementations first tested outputting a single maximum value of a given stream of inputs. A simple comparison would be made between the static, “running” maximum and the current value, and a swap was made if the new value was greater. Once successful, the algorithm gained complexity with the addition of a second and third maximum. If a new first maximum was found, the old first maximum became the new second maximum, and the old second highest became the new third highest. The logic of this decision resulted in three separate if statements, where the highest maximum was checked first, then the second, then the third; if the “comparing value” became the new maximum value, then the old maximum became the “comparing value” where it would be compared with the next highest maximum.

Placeholder values, or “temporary” values, were used to hold the swapping numbers in memory such that they weren’t lost.

Initial testing of this algorithm proved logical success, but an application-specific problem arose. When using examples of software-created FFT amplitude arrays, the Python function would pull frequency values that were considered too close together. This is because musical instruments don’t play at precise frequencies, and adjacent frequencies were being considered as the second and third loudest even though they represented the note found in the first loudest. In order to ignore frequencies that were too close, bounds of notes had to be defined and checked to see if the comparing amplitude’s frequency lies between them.

It seemed most simple to calculate the bounds of a note by taking the midpoint between two note’s frequency values. However, as seen in Fig. 5, the bounds of a note grow linearly as the note’s frequency increases exponentially over octaves. Linear representations of the upper and lower bounds of a note were found by plotting the data in the Desmos online graphing calculator. The frequencies of notes through nine octaves used to find these formulas were found through online resources[7].

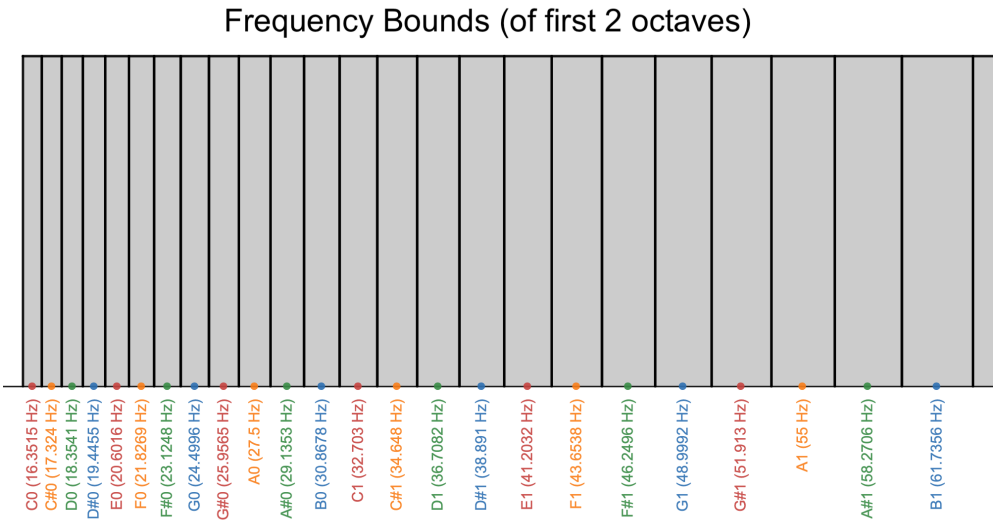

Figure 5 - Frequency bounds of first two octaves

It was later suggested to remove harmonic frequencies which would result in duplicating notes frequency values being outputted by the algorithm. However, music notes are harmonics of each other, and the better solution would be to remove duplicate notes across octaves. The frequencies of notes across octaves are found using equations similar to that found in Fig 6.

$$f_C(oct) = f_{C_0} * 2^{oct}$$

$$oct = octave\ number$$

Figure 6 - Formula for calculating frequency of C in given octave

In order to check all nine major octaves, the function that checked frequency bounds was put into a for-loop, now verifying the comparing frequency does not lie within the bounds of all nine octaves of the current maximum's frequency.

In order for adjacent or octave-adjacent frequencies to be properly handled, the logic of managing the three maximums had to be adjusted. If the comparing amplitude was greater and its frequency "overlapped" with the current maximum's frequency, it would only perform the swap with that maximum; the next if statements would be skipped to prevent the old maximum from becoming the next highest maximum. If the comparing amplitude wasn't greater and its frequency overlapped, then the following if statements would be skipped. This change would allow the highest absolute highest amplitude within a frequency range to be considered the maximum.

Results

Hardware/Software Speedup

The results of the FFT portion of this project can be seen summarized below (Fig. 7). As shown, the performance of the hardware FFT is roughly equivalent to, but slightly slower than, the software implementation at the smallest sample size supported by the hardware, 1024. For the next smallest sample size, 2048, the hardware already outperforms the software implementation. This trend continues, and at our highest sample size, 16384, the speedup of the hardware implementation over the corresponding software implementation is significant – 3.93 ms for the overlay as compared to 6.75 ms for the software implementation, corresponding to a 1.72x speedup.

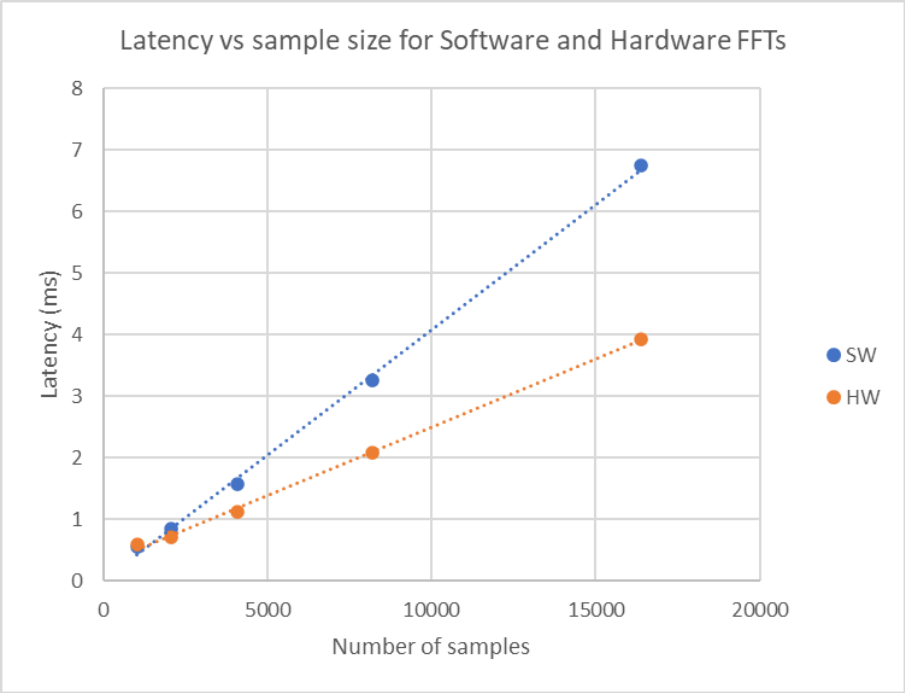

Figure 7 - Comparison between hardware and software implementations of wav data FFT for different sample sizes

We were unable to test larger sample sizes in hardware due to memory constraints, but this trend. However, as shown by the trendlines, the relationship between latency and sample size seems to be almost perfectly linear for both software and hardware, so we suspect this trend will hold consistent for sample sizes up to 65536.

The results of the algorithm portion of this project can be seen in Fig. 8. A dataset the size of 1024 was used for testing the speedup between hardware and software.The significant speedup observed in hardware aligns with the expected result that our algorithm runs faster on the FPGA.

Method	Latency (ms)	Speedup vs. Software
Software	205	1x
Hardware	8.52	24.06x

Figure 8 - Table of algorithm timing differences between hardware and software

Algorithm Optimizations

The algorithm was tested to see if there were further optimizations available to decrease latency and area used on the FPGA. Because the crux of the algorithm was an if statement that did not include any arrays, the only segment of code that could be optimized was the “overlap” function described earlier. The results of different directives can be seen in Fig. 9 and Fig. 10.

#	Implementation Name	Latency (us)	BRAMs #	DSPs #	FFs #	LUTs #	Area	Symbol
1	Baseline	9.11	15	46	11236	13846	19946	●
2	1 + Optimized Data-types	9.21	17	46	8293	13404	19704	■
3	2 + Loop flatten (Overlap)	9.21	17	46	8293	13404	19704	▲
4	2 + Pipelining (Overlap)	9.01	17	46	8291	13432	19732	★
5	4 + Loop unroll (Overlap) fact=unspecified	2.13	0	31	4075	8153	11253	◆
6	4 + Loop unroll (Overlap) fact=1	9.01	17	46	8291	13432	19732	◆
7	4 + Loop unroll (Overlap) fact=2	10.57	17	46	8360	13760	20060	◆
8	4 + Loop unroll (Overlap) fact=4	13.81	17	46	8452	14336	20636	◆
9	4 + Loop unroll (Overlap) fact=8	20.83	17	46	8613	15501	21801	◆
10	5 + Pipeline (main)	2.14	12	423	52696	57500	101000	N/A
11	5 + Loop unroll (main) fact=unspecified	2.13	0	31	4075	8153	11253	◆

Figure 9 - Table of Optimized Implementations of Loudest Triad Algorithm

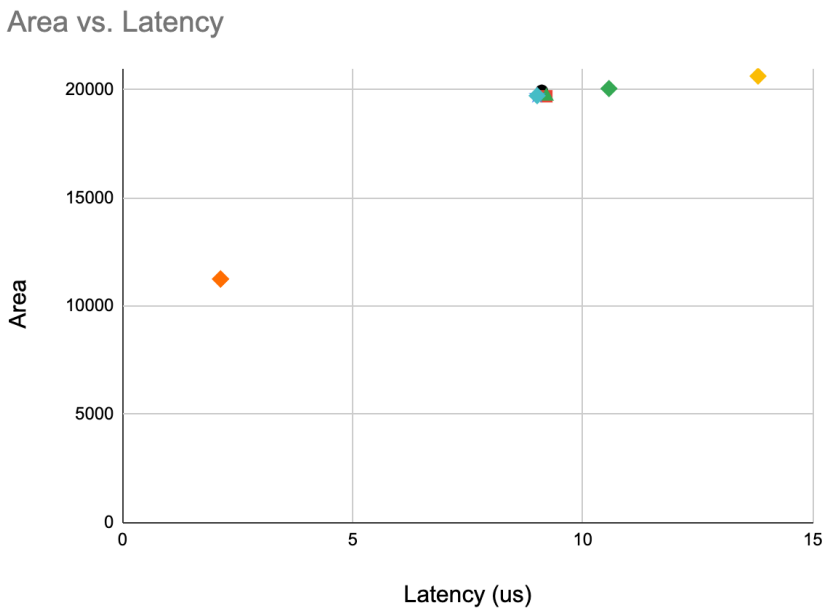

Figure 10 - Graph of Optimized Implementations of Loudest Triad Algorithm

As stated previously, the main algorithm couldn't be optimized significantly; the only speedup was seen by using integers instead of floating points to hold frequency values. Pipelining and unrolling the for-loop in the overlap function, implementation 5, resulted in the best hardware speedup by far. What was interesting was how a full unroll worked so well, while unrolling with

various factors only worsened area use and latency. This could be explained by the factors chosen for testing. The for-loop will always run 9 times, meaning that even numbered factors may have caused the unroll to be more costly for the FPGA to manage. If factors of 3 or 9 were used, a speedup would have been most likely observed. Another observation made was how pipelining the main function caused the hardware to overuse its lookup-tables and DSPs. This would make sense as any pipelining of a non-for-loop function would rely on segmenting the code between function calls, resulting in storing parts of the code inefficiently.

Debugging

Vivado FFT IP

The FFT IP did not originally function as intended because we either sent too few or too many samples to the FFT. We also had problems with the FFT IP not asserting TREADY and TLAST signals to the DMA. Since we initially followed Vivado User Guide 871[8] for working with the FFT IP, we had problems adapting the tutorial to run on the board rather than only in simulation.

Another big problem was figuring out our true sampling rate and how that affected our FFT data. Since our .wav files were sampled at 44.1 kHz, there was the issue of re-sampling the data to be able to process in hardware. With larger sample sizes, it became important to keep track of the exact time range that the sample size spanned in order to calculate the effective sampling frequency. The frequency resolution is a function of the sample size and sampling frequency, so an inaccurate sampling frequency yields unintelligible frequency data with intensities that do not correspond to the correct frequencies. Using the NumPy fft function was a useful benchmark to determine expected output so that we could isolate problems between hardware bugs and incorrect sampling technique.

Algorithm in HLS

In the loudest frequency portion of the assignment, the initial attempt used classes to hold the frequency and amplitude values of each maximum, along with class-defined functions for calculating for overlap. This implementation would have a static array of these “Max” classes, one for each 1st, 2nd, and 3rd loudest notes. It was the fastest version of the software in Python, though adapting the algorithm to C++ in HLS was an issue. For what we believe to be issues with how Vivado HLS handles classes, the algorithm would work in testbenches but fail when put into an overlay. This method was investigated thoroughly, and unfortunately took up a majority of the time spent on the algorithm; no implementation was ever successful. We decided to go to the method that currently works, with a static float for each of the amplitude and frequency for each maximum, along with separate functions for overlap.

Integration

After working on the two sections in parallel, the Vivado FFT IP was combined with the custom algorithm IP in the PYNQ board's datapath, seen in Fig. 11. After fixing minor integration-related issues with the algorithm, we generated a bitstream and created the overlay in order to test the functionality of both IPs working in sequence. The test of this overlay can be seen in the Jupyter notebook in the GitHub repository[9].

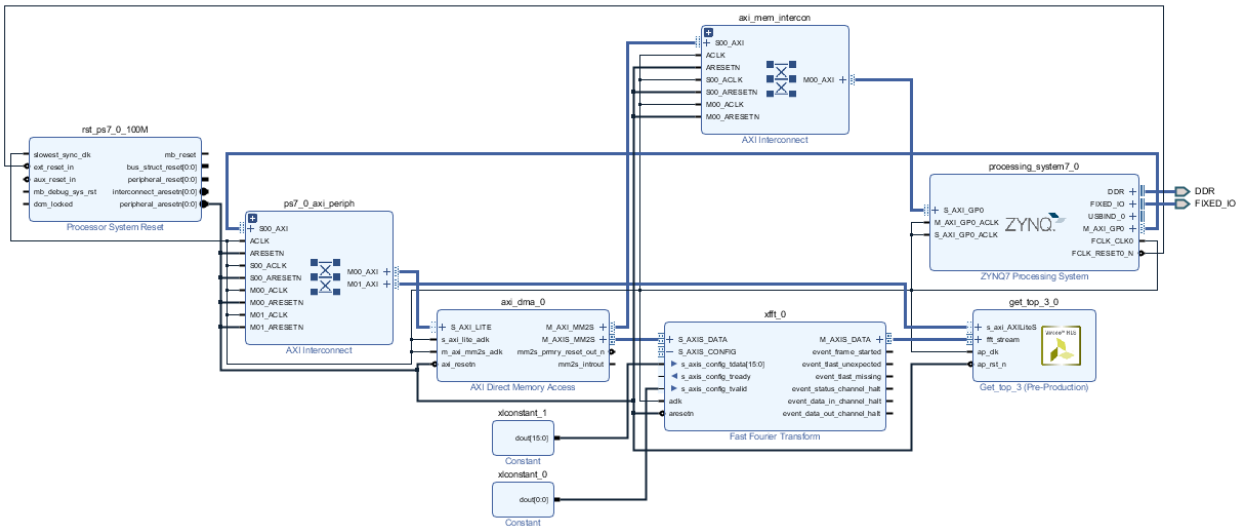

Figure 11 - PYNQ datapath containing full integration of FFT and algorithm IP blocks.

Testing proved unsuccessful, as the numbers sent back to software did not match the previously acquired numbers. However, there was partial accuracy in the results; the frequencies received were of different octaves than what was to be expected. This discrepancy was not explored further due to time constraints of this project.

Unlike previous tests, timing was an issue to calculate with this new integration. The stream buffer would be sent to the FFT IP, but there was no way for the software to know when the FFT was finished processing and when the algorithm IP was done executing. This meant that checking the IP registers for the frequency values would return inaccurate results if checked immediately after sending the input buffer to the FFT IP. Because of this, we weren't able to acquire a speedup calculation as there was no accurate way to determine when the values would be correct.

Future work

The next steps for this project are to improve the FFT Vivado project to support a larger sample size than 16384. The processing stage of the FFT IP naturally produces a "bit-reversed" output, flipping the bits of each output sample. There is an option to produce output in "natural order", so the endianness of the output matches that of the input. We opted for the "natural order"

option for simplicity, but this incurs a memory penalty on the Zynq, because the FFT has to perform an extra reversal. Future work for this project could include implementing a “bit-reversed” design, and then adding a custom reversal IP to explore tradeoffs between “natural ordered” and “bit-reversed” FFT implementations.

Once the FFT IP is successfully utilized with the peak implementation, the next steps for this project would be integrating the two IPs into a successful implementation. This would require fixing the problem with the wrong octaves within the algorithm portion, and to find a way to poll the loudest frequency IP registers so that the software knows when the program is complete and the values are accurate.

References

[1] https://musictheory.pugetsound.edu/mt21c/TriadsIntroduction.html
[2] https://mathworld.wolfram.com/FourierTransform.html
[3] https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/
[4] https://docs.scipy.org/doc/scipy/reference/io.html
[5] https://docs.xilinx.com/v/u/en-US/pg109-xfft
[6] https://www.chciken.com/digital/signal/processing/2020/05/13/guitar-tuner.html
[7] https://pages.mtu.edu/~suits/notefreqs.html
[8] https://docs.xilinx.com/v/u/2014.2-English/ug871-vivado-high-level-synthesis-tutorial
[9] https://github.com/db00lean/triad_finder