Audio analysis using parallelized Fourier transforms

ME766 Final Project
Team Members: Amit Meena,
Ashutosh Sharma, Satdhruti Paul

Goals of this project:

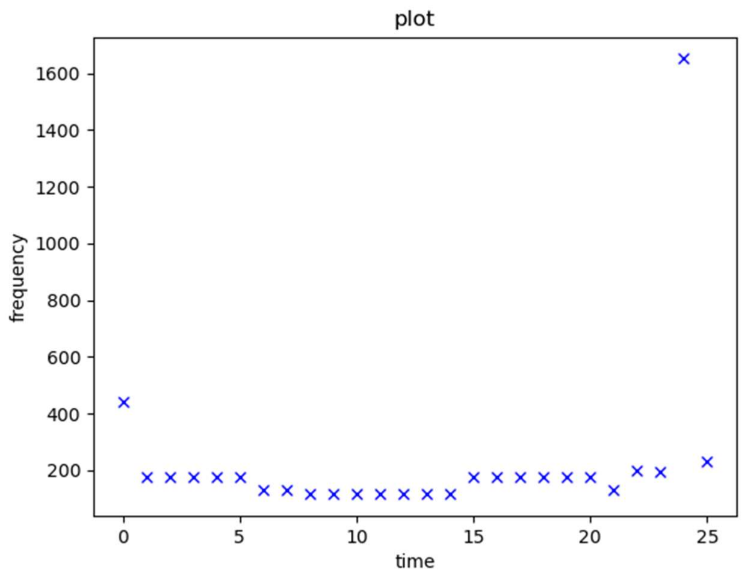
A beginner musician, while listening to some music, might want to play along with it on his instrument but finding the treble/bass notes being played might be a difficult task. We propose a solution to this by using Fourier transforms to identify the most commonly occurring frequencies in an interval. With this we can find the note being played in the 4th second or the 3+1/8th second by varying the interval size.

A Fourier transform, as defined by Wikipedia, *"is a mathematical transform that decomposes functions depending on space or time into functions depending on spatial or temporal frequency, such as the expression of a musical chord in terms of the volumes and frequencies of its constituent notes".*

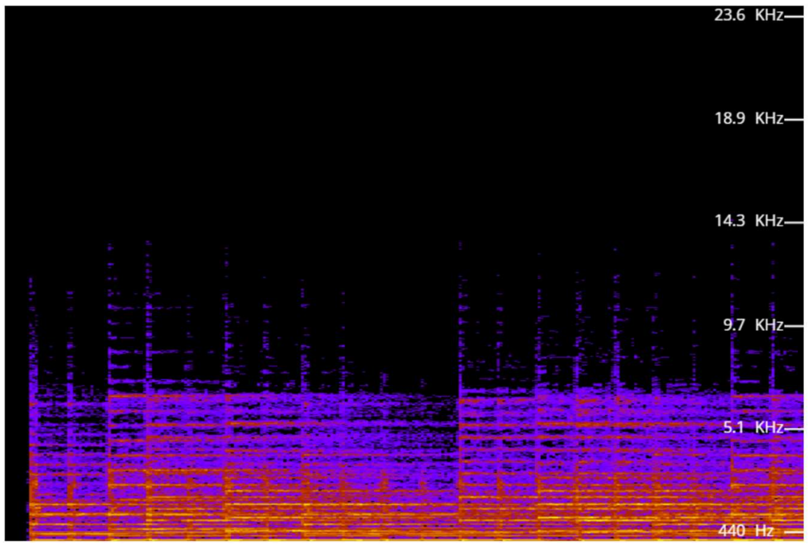
This is exactly what we want, to convert an audio file which is a sample amplitude vs time graph into a magnitude vs frequency graph, from which we can find the note being played.

Sample rate is the amount of input values in one second of the audio file, a file with a 44 kHz sample rate will have 44000 values in 1 second of the 1D audio vector. This is important because when performing Fourier transforms we get frequency readings upto (sample rate)/2.

Results/Graphs of some basic analysis



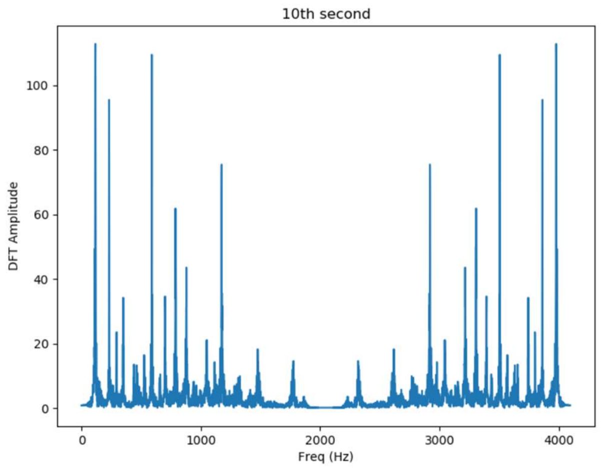
This is the result over intervals of 1 second, and since bass notes are held longer than the treble, high pitched, notes we're getting low frequency readings.



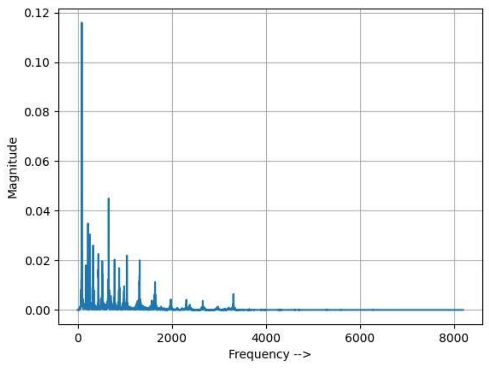
This is the corresponding spectrogram of the audio file which over loads us with too much data. There are bass frequencies, treble frequencies and the corresponding resonant frequencies as well. It is comparatively harder to understand such a plot.

Fixing the problem of too many bass readings

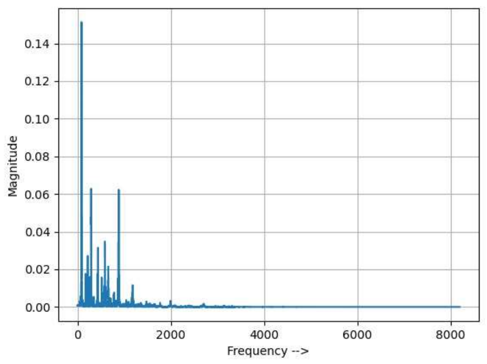
To take care of this we have to notice that in a single second,
i) There will be quarter notes and half notes so we can reduce the interval size.
ii) We can also take 2 or more high amplitude readings from the plot to get the treble frequencies.



Low frequencies around 100 Hz and high frequencies around 500 Hz, 1200 Hz are also visible.



1-1.5s

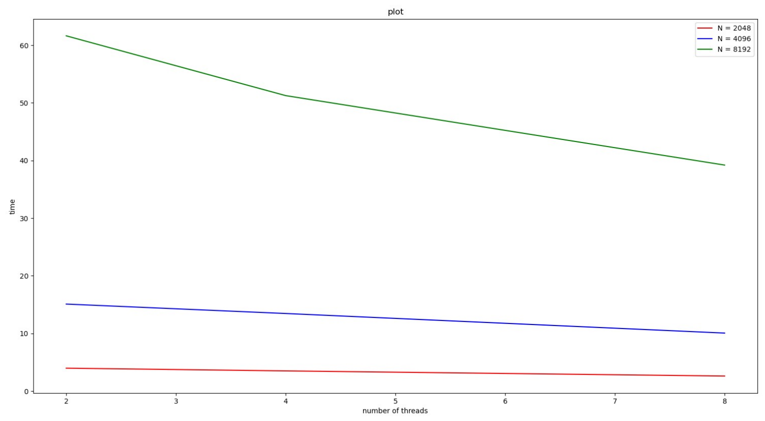


1.5-2s

Timing studies and final results

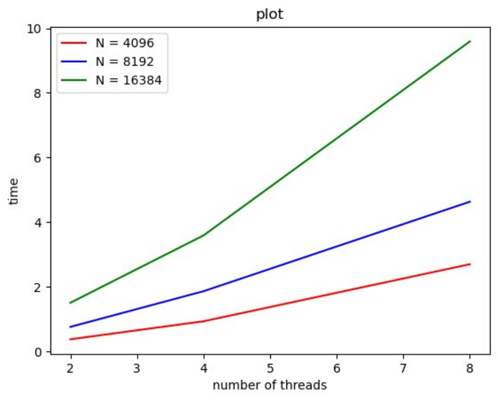
Now for the most important part of the project we have to see the effect of OpenMP and CUDA on the DFT and FFT codes. DFT on average takes around 40 seconds. (40.5 + 41.3 + 41.8 + 40.6 + 41.2)/5 = 41.1 seconds. FFT on average takes around 0.71 seconds. (0.666 + 0.736 + 0.711 + 0.761 + 0.669)/5 = 0.71 seconds.

On using OpenMP on DFT, for high sample rates we see time increases and with increase in number of processors, the time decreases. Fig below



On using OpenMP on the FFT algorithm, we see a rise in time from 0.7 seconds to 2 seconds on average with 2 threads. This is at first an unusual result but on considering the fact that we are using a recursive approach and as the array size decreases, at the lowest height of the recursive tree, each node will take 2 or more processors to perform the same task again and again thereby increasing time taken.

Similarly on using CUDA on the FFT algorithm the time to perform increases drastically to upto 20 seconds. This is a combination of the above reason and the fact that at each level we are initializing pointers to memory locations and freeing them again and again thereby wasting a lot of time. Fig below



This could be fixed if we had used a dynamic programming approach to FFT instead of a recursive function. But as we can see in DFT parallelized code using multiple processors does decrease time to run the code if it is used correctly.