Algorithms for Short Time Fourier Transforms

1st Matthew Molter
Cockrell School of Engineering
University of Texas at Austin
Austin, TX
mmolter@utexas.edu

2nd Justin Roznik
Cockrell School of Engineering
University of Texas at Austin
Austin, TX
roznikj@utexas.edu

***Abstract*—This paper recaps the basic algorithm for the Fast Fourier Transform (FFT), its implementation, its extension to Short Time Fourier Transforms (STFT), and a parallel implementation for dealing with STFTs. The run times of these implementations will also be discussed and compared.**

I. Introduction

Fourier transforms were first introduced in the early 19th century by Joseph Fourier, extending from his work on Fourier series, which allowed for a periodic function to be broken up into its constituent parts. These constituent parts are trigonometric functions. Initially used as a solution for the heat equation, a partial differential equation which had no exact solution prior, it has found use in numerous applications over the past 200 years, the most relevant of which to this paper is digital signal processing.

When applied to signals, it allows us to convert a signal from the time domain to the frequency domain (its constituent parts). This representation allows for convenient filtering, modification, etc., as well as feature generation for machine learning models. In particular, this paper will focus on the implementation of the discrete Fourier transform since we will be dealing with digital signals. First we will introduce, briefly, the naive method with a $O(n^2)$ run time, and follow up with the Fast Fourier Transform (FFT), first published by Cooley and Tukey in [1], with a $O(n \log n)$ run time, which we will utilize and extend further.

Once the FFT has been introduced, we will show how it can be used on a signal for the Short Time Fourier Transform (STFT), which is more useful for feature generation for modern machine learning algorithms. We will then show a parallel version of this algorithm, and compare performance.

Short Time Fourier Transforms are an extension to Discrete Fourier Transforms, and introduces time dependency to the transform. The STFT has the ability to show how spectral components of a signal change with time.

The STFT has a fixed resolution which cannot be modified without any external algorithms. A wider time window will allow greater frequency resolution at the cost of time resolution; while a small time window will give a greater time resolution, however the frequency resolution will decrease. This requires the user to determine an appropriate time window and frequency resolution. If these requirements aren't satisfied, faster sampling or image interpolation may be required.

Additionally, there are concerns with how much overlap to use between two windows. Shorter overlap times can lead to Nyquist frequency artifacts that lead to false signals in the spectral output. Longer times negate the benefits of using an STFT.

For simplicity and clarity, we decided that the prior two issues are outside the scope of this class. Instead, we focused on the implementation and performance comparisons of serial/parallel STFTs with no overlap between windows, and various window sizes. Window sizes were chosen purely for illustrative purposes, with no performance related design considerations.

II. Fast Fourier Transform

The following serves as a brief introduction to the naive Discrete Fourier Transform (DFT) algorithm and FFT algorithm. The formula for the DFT on a series of N samples is as below, and results in an output with N number of frequency bins.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \quad k = 0, ..., N-1$$

Naively using this formula to generate an algorithm results in a time complexity of $O(N^2)$. This can be seen on inspection. For every bin k, N samples are evaluated, and since k is of size N, it is trivial that the naive calculation results in the $O(N^2)$ time complexity.

As in any situation where a complexity such as this arises, it is appropriate to ask if a better algorithm exists. A better algorithm, the FFT, was discovered by Cooley and Tukey, and published in [1]. We will recap the fundamentals of what is referred to as the radix-2 decimation-in-time (DIT) case below.

At its core, the radix-2 DIT algorithm is a recursive divide and conquer algorithm. At each step, it breaks a DFT of size N into two DFTs of size $N/2$ and combines the results into a single DFT, hence the name radix-2. There are two main options for the decomposition strategy; taking the odd and even points and splitting the input by index, or taking the first/second half of the input signal as the splits. The former is referred to as decimation-in-time (DIT), while the latter is referred to as decimation-in-frequency (DIF). We use the former for this paper as we found it simpler to deal with.

Other radices, or even mixed radices, are possible, however for simplicity we did not compare different radices. There are

disadvantages to the radix-2 approach. The most obvious is that the input signal must be of a size that is a power of 2 (which in the real world can be compensated for by padding, under-sampling, etc.) in order for the algorithm to reach a base case of two samples. Another disadvantage is the number of floating point operations, which can be improved upon using a radix-4 or mixed radix approach [3].

Our implementation is an iterative version of this algorithm, based on a Java implementation from Columbia university [2]. The iterative version was used for two reasons. First, it is more stack efficient and we will be dealing with relatively large signal sizes, and second, it is a more straightforwardly parallelizable algorithm.

In the execution, the algorithm has two phases. There is the time domain decomposition, which consists of a bit reversal on the input. This allows us to easily calculate the frequency spectrum of each sample, as the frequency spectrum of a single sample is simply the value of that single sample. A signal of size N becomes N signals of size 1. The runtime of this portion is on the order of $O(NlogN)$.

Once the time domain signal has been decomposed into its single sample parts, we can begin the combination portion of the algorithm. This consists of an outer loop of size $log(N)$ for each stage of the FFT. Inside of this loop are two inner loops. The outer of these two is all of the data points at each level in the FFT, which corresponds to each of the frequency spectra at each level. The inner most loop is what is referred to in most literature as the 'butterfly'. The reason for this name can be seen in figure 1 below as provided by [4].

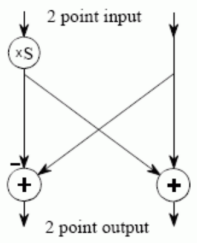

Figure 1

The inner most loop takes each pair of FFTs to be combined, multiplies one in the pair by a sine wave (shifts to the right), then subtracts/adds this to the other one in the pair. This results in a combination of the two FFTs sized $N/2$ into a single FFT of size N. The run time of the inner two loops is not immediately apparent, but on observation can be seen to be $O(N)$. The reason for this is that the outer loop is on the order of $O(N)$. On the first iteration, it will run 1 time, on the second 2 times, etc. until the outer loop reaches $logN$, at which point the inner loop will run $N/2$ times. The innermost loop is the opposite. On the first iteration of the outer loop, it will run $N/2$ times, then $N/4$ and on until the final loop where it will run once. Multiplying all of these out gives us a complexity of $O(NlogN)$. Since this is on the same order as the decomposition, we can conclude that the entire algorithm runs on the order of $O(NlogN)$, a significant improvement over the naive complexity of $O(N^2)$.

III. Short Time Fourier Transform

One disadvantage to the Fourier Transform for longer sample sizes is that frequency content of a signal is not always time independent. By taking the transformation of the entire signal the majority of information for time dependent frequency distributions is lost. For some applications this may not be a concern, however for most deep learning applications this data can be crucial. Take for example the sonograms of deep diving equipment. The sonogram will create time-independent noise that has a frequency model to it, the sonogram can also pick up noises such as whales and other marine life. As the noise of the sonogram is always present, an FFT of the entire sample will drown out any details of marine life. However by splitting the sample into windows, the noise characteristics of marine life can become more pronounced.

By definition, the Short Time Fourier Transform (STFT) is defined by the equation below. One can note that it is very similar to the Fourier Transform. By adding a window convolution function term, we can create a time dependent transform.

$$X(t,f) = \int_{-\infty}^{\infty} w(t-\tau)x(\tau)e^{-i2\pi f\tau}\,d\tau$$

Let's extend this definition to it's discrete form. The first steps is to make the variables t, f and p discrete.

$$t = n\Delta t, \quad f = m\Delta f, \quad \tau = p\Delta t \tag{1}$$

Now suppose our window function w(t) is zero outside our bandwidth B such that,

$$w(t) \cong 0, \quad for|t| > B, \quad B/\Delta t = Q$$

We can now write the workable discrete STFT as follows.

$$X(n\Delta t, m\Delta f) = \sum_{p=n-Q}^{n+Q} w((n-p)\Delta t)x(p\Delta t)e^{-i2\pi pm\Delta t\Delta f}$$

This definition allows us to implement different methods to create the STFT for various signals. There are three common methods to implement the STFT: FFT based, Recursive based, and Chirp-Z method. Each method has it's own drawbacks and limitations.

The method in this paper is a more simplified FFT version. We took our signal and divided the samples by a non sliding window. In a sound file, we partition the file into T windows, the number of windows need to be equal to a power 2 to allow computation from our FFT method. Each window is then transformed using our FFT window to create a pseudo-STFT of the signal. In comparison to the full STFT, we do not use a sliding window or any type of discrete convolution. A drawback of this is less detail compared to a full STFT method.

As an example, we generated multiple waveforms in Java and found their STFT using the above method. We then exported the results into a CSV file. The results were then uploaded to a Python notebook to showcase its results.

The below image shows a sine wave of frequency, F = 7 Hz.

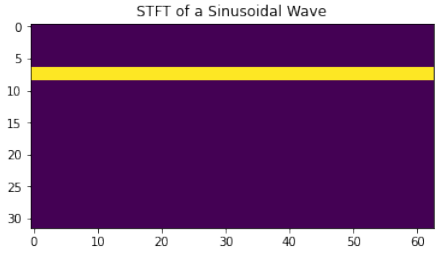

Figure 2

Below is an image showing the STFT of multiple time dependent sine waves.

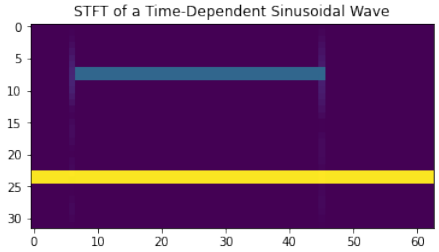

Figure 3

Our STFT algorithm is now able to show the frequency spectrum's of time dependent signals.

A. Serial STFT

The simplest way to implement an STFT is to simply break up the data into chunks and perform an FFT on each individual chunk in serial. The overall time complexity for this is similar to the original FFT algorithm. If we were to split our input N into c chunks, the time complexity on each individual chunk can be seen to be $O(\frac{N}{c}\log(\frac{N}{c}))$. Since this is done c times, we have an overall time complexity of $O(c * \frac{N}{c}\log(\frac{N}{c}))$, which simplifies to $O(nlog(\frac{N}{c}))$, similar to the base FFT case.

We reused our base FFT implementation when implementing our serial STFT. This resulted in what is essentially a wrapper class with some overhead. The method begins by calculating window size based on number of windows. This was done for one reason in particular over specifying window size ourselves. Our base FFT method requires input of size that is a power of 2, and rather than relying on us to specify the correct window size, the wrapper will get it for us. This could trivially be changed, but for the purposes of this paper is sufficient.

Once the window size has been calculated, we make copies of each window of the array and pass them to our FFT algorithm. The returns from these are then combined into an ArrayList of an ArrayList of double arrays, which we can address and use in whatever manner we may choose in the future. The real part is the first index of the outer ArrayList, while the imaginary part is the second index. Each window is a $double$ array contained within the inner ArrayList. This is the return value of the STFT function.

B. Parallel STFT

We will show in the results below that an STFT of a similar size to an FFT scales in roughly the same manner. However, unlike a standard FFT, there is an immediately apparent, simple way to parallelize it. This consists of breaking each window up and sending it to its own thread to run an FFT, and return the results to a centralized thread which can combine the results into the ArrayList we expected from above.

We implemented this by extending $Callable$, and having it return a List of Lists of double arrays. This class is referred to as $ParallelSTFT$. The $call()$ method is simply running the base FFT algorithm and returning the results as an ArrayList of double arrays. Then there is the method $parallelSTFT$ which is the interface for our $main()$ method. This method breaks the input up into windows, creates a ThreadExecutorPool and combines the individual results into an ArrayList of ArrayList of $double$ arrays.

The complexity for this, with P number of cores then becomes $O(c*\frac{N}{cP}\log(\frac{N}{c}))$. There are of course changes to the smaller components of the complexity as well, so the speed up is not perfect, but we will see a semi-linear speed up based on number of cores used.

IV. Results

All testing was performed on a 2019 MacBook Pro with a 6-core, 2.2GHz Intel i7 processor. Each algorithm at each input size and processor count was run for 10 iterations, and the data averaged over those 10 iterations for our final data as shown in the tables below.

All initial testing was performed on a simple 1Hz sine wave signal to ensure that there was no influence of signal type on performance, or any aliasing effects from the smaller sample sizes. Various signal sizes were tested, from 2^10 samples up to 2^24 samples.

For baseline performance, our first results presented in table 1 and figure 4 below are run times in milliseconds of our simple FFT algorithm. It scales exactly as expected vs the $O(nlogn)$ baseline.

Input Size	Execution Time
1024	0.6775
16384	5.834
4194304	1535.1
8388608	3496.9
16777216	7333.4

Table 1

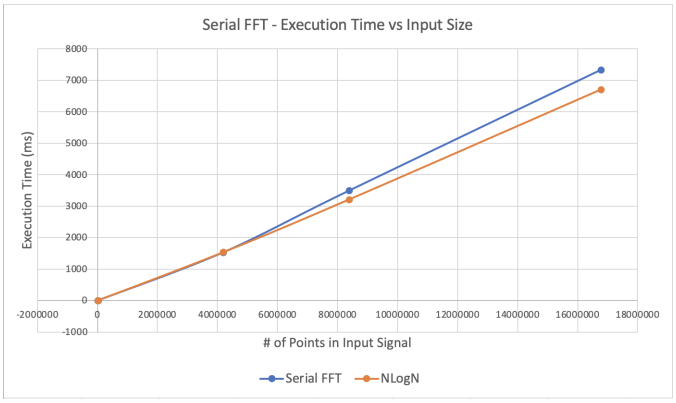

Figure 4

Our initial STFT algorithm, the serial version described in section IIIa., was run with 16 windows over the same signal sizes. As can be seen in table 2 and figure 6, the algorithm is actually faster despite additional overhead for a given number of sample sizes. On initial observation, this was counterintuitive. However, upon examination, there are two reasons for this. First, the cos/sin look up tables are calculated only once, and are of a size $\frac{1}{16}$ of the original look up tables, leading to an amortization of the $O(N)$ portion of the algorithm. Secondly, The combination of the individual STFTs is of $O(1)$ complexity, and the logarithmic portion of the smaller STFTs is $O(log(\frac{N}{c}))$

Input Size	Execution Time (ms)	
	FFT	Serial STFT (16 window)
1024	0.6775	1.1689
16384	5.834	8.3422
4194304	1535.1	1084.4
8388608	3496.9	2393.8
16777216	7333.4	5900.1

Table 2

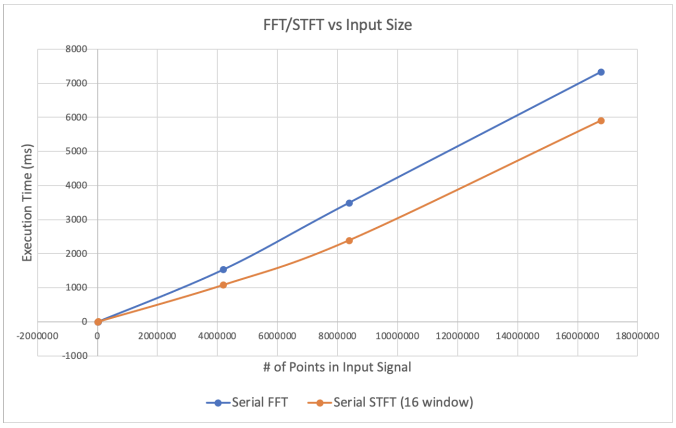

Figure 5

Parallel STFT results can be seen below in table 3 and figure 6. Same test conditions were used on this version as in the prior serial algorithm. Scaling is as expected up until 6 cores, which has a marginal if any improvement on on smaller sizes and a slight decrease in performance at large sizes. There are three potential reasons for this. First, overhead costs could be coming to dominate run time. Second, after running tests, we realized that our implementation of the parallel STFT does not reuse look up tables as in the serial version. Unfortunately, we did not have time to go back and fix this. Finally, there is a possibility that there are oversubscription issues on our test system.

Input Size	Execution Times (ms)				
	Serial	1-core	2-core	4-core	6-core
1024	1.1689	5.3519	5.5634	5.3676	5.1769
16384	8.3422	14.2099	11.3023	10.5787	8.9825
4194304	1084.4	1114.5	634.2	560.2	558.3
8388608	2393.8	2497.7	1651.8	1339.9	1296.5
16777216	5900.1	6142.4	3822.1	2855.1	2911.1

Table 3

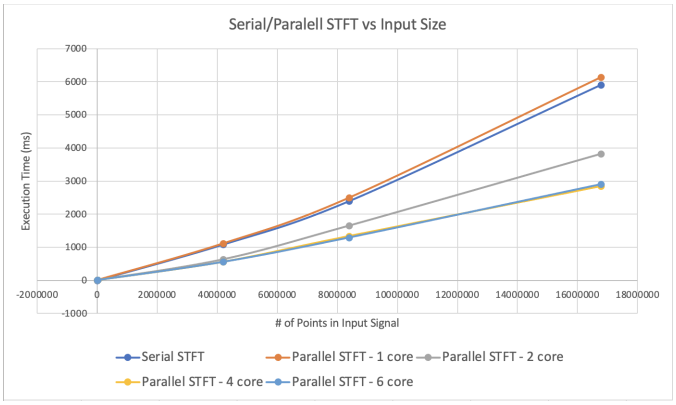

Figure 6

There is no simple way to test overhead costs, so we began by removing the oversubscription. The results show that at the very least, oversubscription is definitely a contributor, if not the main contributor to non improvement with increasing cores. To test, we ran an identical test to above, but with an input of 6 times various powers of 2. This allowed us to use 6 windows and run the parallel model, the results of which can be seen below in table 4 and figure 7.

Input Size	Execution Times (ms)	
	4-core	6-core
6144	6.8066	6.6107
393216	83.2	74.3
6291456	1348.1	1226.9
25165824	5771.8	5253.4

Table 4

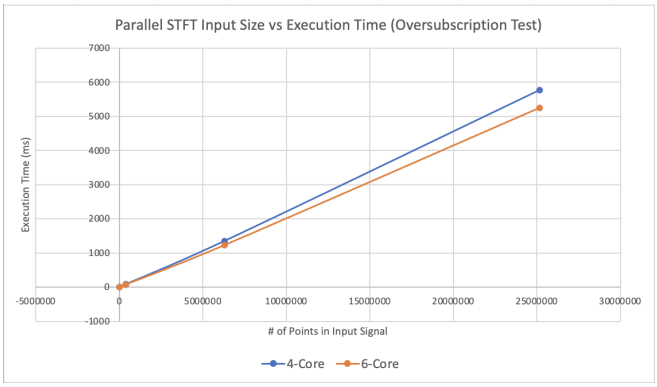

Figure 7

V. Conclusion

We have presented the original FFT algorithm discovered by Cooley and Tukey, its extension to STFT, and a parallel version of the STFT algorithm. The serial STFT is quicker than the base FFT for similar size inputs due to windowing, which we believe was not obvious on initial application. The parallel version shows the expected speed ups, with the caveat that you can not oversubscribe the processor. We believe that the parallel version would show more impressive speed ups should we do the FFT in place rather than splitting and recombining results.

As for future works, there are a handful of obvious avenues to explore. First and foremost, we will implement synchronized access to the input signal so that the parallel version can operate in place. This will improve efficiency by removing unnecessary array copying and overhead. Secondly, a simple padding function could be added so that we can perform the FFT on input signals that are not of size 2^n. Third, it would be beneficial to test on a large scale parallel system to see what the ultimate limits of improvement would be. Finally, we would be interested in adding overlapping window functionality and utilizing properties of this to determine if we can further speed up calculations.

References

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," Mathematics of Computation, vol. 19, no. 90, pp. 297–301, Mar. 1965.

[2] R. Weiss and D. Ellis, "MEAPSoft Documentation," Meapsoft.org, 25-Nov-2008. [Online]. Available: https://www.ee.columbia.edu/ ronw/code/MEAPsoft/doc/html/main.html. [Accessed: 12-Nov-2021].

[3] M. Soni and P. Kunthe, "A General Comparison of FFT Algorithms," Cypress.com. [Online]. Available: https://www.cypress.com/file/55401/download.

[4] S. W. Smith, "The Scientist and Engineer's Guide to Digital Signal Processing," 2011. [Online]. Available: https://www.dspguide.com/ch12/2.htm.