Efficient Simultaneous Multi-Scale Computation of FFTs

Dave Cohen, Camille Goudeseune, Mark Hasegawa-Johnson
University of Illinois at Urbana-Champaign

Abstract

An efficient algorithm for simultaneously computing FFTs at multiple window sizes is introduced. It requires 42% fewer complex multiplies for an 8192-sample window. It is motivated by STFT visualization of non-speech audio events, which can vary widely in scale and resolution of both time and frequency. A reference implementation in C agrees with MATLAB's implementation, for numerous test inputs.

Introduction

The Fast Fourier Transform (FFT) is the most common method for computing a Discrete Fourier Transform (DFT). There are two common types of FFTs: Decimation in Time (DIT) and Decimation in Frequency (DIF). A Short Time Fourier Transform (STFT) slides a fixed-length window in time and computes an FFT at each position. The effect is a sequence of windows containing frequency information for a specific snapshot of the input signal. Concatenating this sequence then plots frequency vs. time.

We introduce a modication to the standard FFT, namely efficient simultaneous multi-scale computation of FFTs at multiple window sizes. When calculating an FFT of a given size (say 2^{13}), this multi-scale FFT computes all smaller FFTs as intermediate steps (sizes 2^{12}, 2^{11}, ... , 2^{1}). This generates a multi-scale FFT representation usable for separate STFT representations. While the algorithm is more computationally complex than the traditional FFT, it offers savings when compared to using the traditional FFT to compute all intermediate sizes. The standard number of operations per second for an FFT is $RN\log_2(N)$, where R is the number of FFTs computed per second, and N is the FFT length. (All our logarithms are implicitly base 2.) If FFTs are computed of sizes N_0, $N_1=2^1N_0$, ..., $N_K=2^KN_0$ at rates of Rk, then the total computational complexity is $RN\cdot(\log(N_0) + ... + \log(N_K))$. Our algorithm achieves the same multi-scale FFT representation of the signal, but at much lower computational cost.

For any STFT, a long window gives accurate frequency measurements but blurs temporal measurements, while a short window does the opposite. Speech analysis works well with a known fixed window size, but non-speech audio events have no *a priori* spectral or temporal characteristics. Here, no fixed window size suffices to classify audio events. For example, a door slam is sharply localized in time with a large frequency spread, while the hum of mechanical equipment has the opposite attributes. Traditionally, working around the uncertainty principle has been computationally expensive. One could either recompute the same STFT at multiple window sizes, or use expensive long windows with large overlap.

Algorithm

A traditional FFT splits the input signal $x[n]$ into its first half, $x_0[n]$, and its second half, $x_1[n]$. It computes one $N/2$-point FFT, then two $N/4$-point FFTs, and so on to $N/2$ 2-point FFTs. The output is the N-point DFT of the input signal with bit-reversed indexes, so $x[6 = 110_2]$ becomes $X[3 = 011_2]$. The equation for an N-point DFT is:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{\frac{-j2\pi kn}{N}} \quad (1)$$

If we split the signal as before into $x_0[n]$ and $x_1[n]$, then we can also calculate $X_0[k]$ and $X_1[k]$:

$$X_0[k] = \sum_{n=0}^{N/2-1} x_0[n] \cdot e^{\frac{-j4\pi kn}{N}} \quad (2)$$

$$X_1[k] = \sum_{n=N/2}^{N/2-1} x[n] \cdot e^{\frac{-j4\pi kn}{N}} \quad (3a)$$

We simplify Eq. 3a by substituting $m = n - N/2$, and by noting that $x[m + N/2] = x_1[m]$.

$$X_1[k] = \sum_{m=0}^{N/2-1} x_1[m] \cdot e^{\frac{-j4\pi k(m+N/2)}{N}} \quad (3b)$$

Because

$$e^{\frac{-j4\pi k(m+N/2)}{N}} = e^{\frac{-j4\pi km}{N}} e^{\frac{-j4\pi kN/2}{N}}$$
$$= e^{\frac{-j4\pi km}{N}} e^{-j2\pi k} = e^{\frac{-j4\pi km}{N}} \cdot 1$$

we resimplify:

$$X_1[k] = \sum_{m=0}^{N/2-1} x_1[m] \cdot e^{\frac{-j4\pi km}{N}} \quad (3c)$$

By considering Eq. 1 for even k,

$$X[2k] = \sum_{n=0}^{N-1} x[n] \cdot e^{\frac{-j4\pi kn}{N}} \quad (4)$$

we can add Eqs. 2 and 3c:

$$X[2k] = X_0[k] + X_1[k] \quad (5)$$

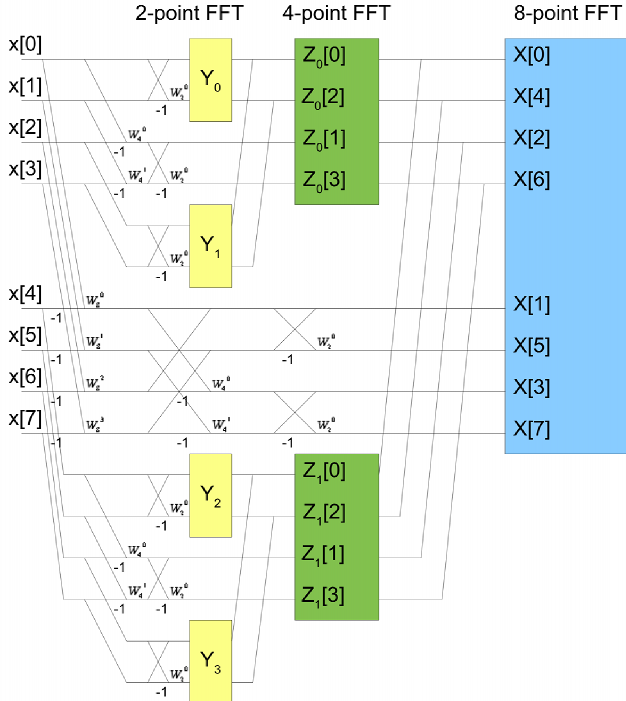

Figure 1. 8-point multi-scale FFT.

In other words, the even values of an N-point FFT can be constructed by adding the values of the two $N/2$-point FFTs. The traditional FFT does "butterfly" adding/subtracting of $x_0[n]$ and $x_1[n]$ at the *input*. The multi-scale FFT similarly subtracts and twiddle-multiplies $(x_0[n] - x_1[n]) \cdot W_N^n$ at the input. But first it computes $X_0[k]$ and $X_1[k]$, and then butterfly-adds $X_0[k]$ and $X_1[k]$ at the *output*. Fig. 1 shows the butterfly diagram for an 8-point multi-scale FFT.

Each level uses the second half of its input twice. Fig. 1 shows these double uses as forks that are not part of a butterfly 'X'. The first use computes the *odd* values of FFTs at that level (these are called middle branches, the horizontal prong of a fork). The second use, the fork's lower prong, computes the second-half FFTs. These are eventually added to the first-half FFTs to create the *even* values of FFTs at that level.

In Fig. 1, the 2-point FFTs Y_0 through Y_3 are calculated directly from the input samples. $Y_0 + Y_1$ yields the even samples of one 4-point FFT, Z_0. $Y_2 + Y_3$ yields the even samples of the other, Z_1. These two FFTs' odd samples are then calculated from two sets of middle branches. At the next level, Z_0 and Z_1 sum to form the even samples of $X[k]$, the 8-point FFT. The odd samples come from the largest middle branch.

The computational complexity of a traditional N-point FFT is $N \log N$. Because the multi-scale FFT requires one extra $N/2$-point FFT calculation, two extra $N/4$-point FFT calculations, etc., its total complexity is $N \log N + (N/2) \log(N/2) + 2(N/4) \log(N/4) + 4(N/8) \log(N/8) + \ldots + (N/4)(2) \log(2)$. This is more than the traditional N-point FFT. But to compute the extra FFTs to match the nested method, tradition needs two more $N/2$-point calculations, four more $N/4$-point calculations, etc. Thus the total complexity for the traditional method is $N \log N + 2(N/2)\log(N/2) + 4(N/4)\log(N/4) + \ldots + (N/2)(2)\log(2)$. The nested method's savings over the traditional method, then, is

$$
\begin{aligned}
& (N/2) \log(N/2) + 2(N/4) \log(N/4) + \ldots + (N/4)(2) \log(2) \\
&= (N/2) (\log(N/2) + \log(N/4) + \ldots + 1) \\
&= (N/4) (\log(N/2)+1) \log(N/2) \\
&= (N/4) \log(N) \log(N/2) \qquad (6)
\end{aligned}
$$

When $N = 8192$ ($\log N = 13$), the nesting eliminates about 319,000 complex multiplies, or 42%.

Experimental Methods

We have implemented the multi-scale FFT algorithm in the C programming language. An array $X[l][m][n]$ stores all the FFTs. Dimension l indicates an FFT's size: 2-point FFTs are stored in entries with $l = 0$, 4-point FFTs have $l = 1$, and n-point FFTs have $l = \log(n) - 1$. Dimension m corresponds to the FFT number. For example, when $N = 8$, there are four 2-point FFTs, so m goes from 0 to 3. Dimension n indexes into a particular FFT result. The calculated values are saved in .csv format for analysis and graphing.

If an STFT is being calculated, the multi-scale FFT is then run on the next chunk of samples in the input file, overwriting the previous values stored in the array X, and then appended to the .csv file.

We have written several test cases to verify the algorithm's correctness and accuracy. These consist of common inputs for which outputs are well known:

- All zeros
- Constant, nonzero
- Samples alternating between 1 and –1
- Impulse at sample 0
- Shifted impulse
- Dual impulse
- Sine wave
- Dual sine wave
- Band-limited noise (as a sum of sines).

Results and Discussion

All test cases produced the expected output. Tests were also run using the widely accepted FFT function in MATLAB; outputs for both methods were identical in nearly every case.

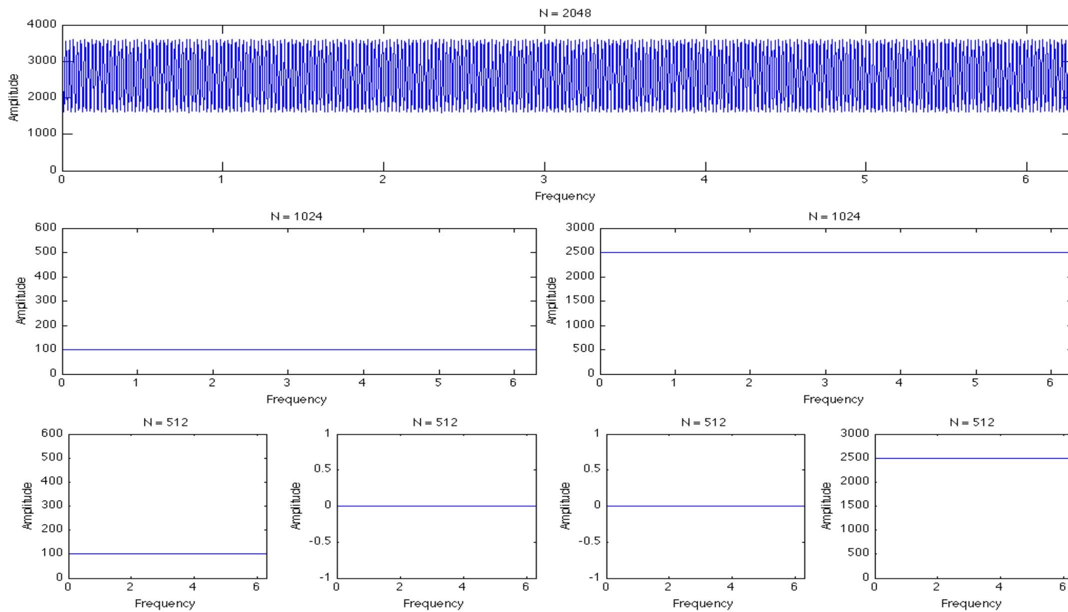

Figure 2. Multi-scale FFT output for a dual-impulse input.

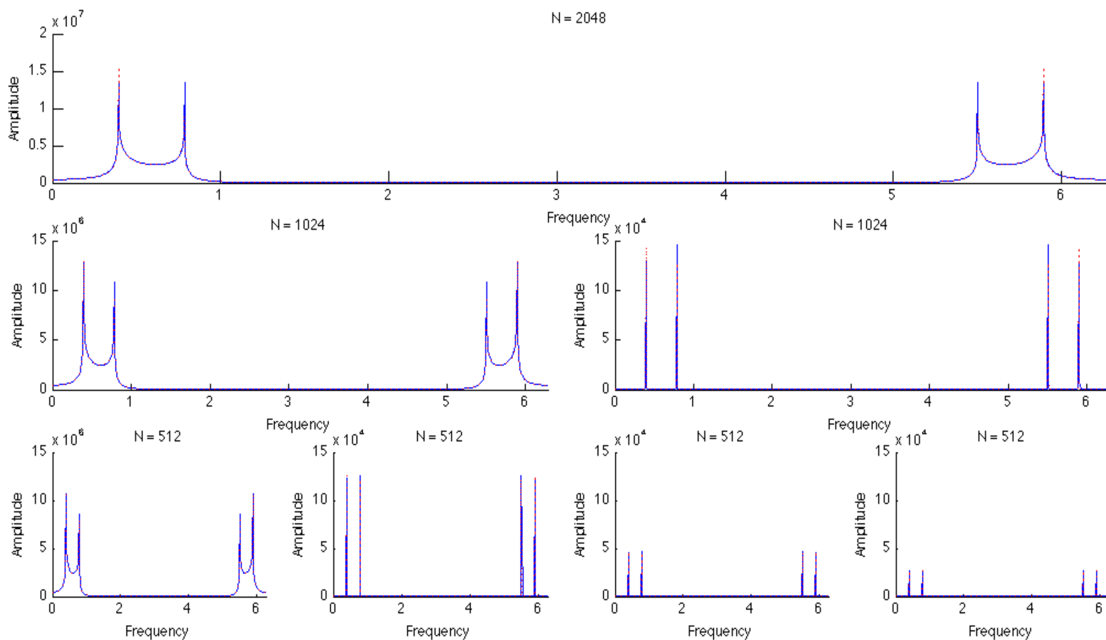

Figure 3. Outputs for a band-limited input: multi-scale FFT (blue) and MATLAB (dashed red).

Fig. 2 shows, for a dual-impulse input, the output from the three largest window sizes, $N = 2048$, 1024, and 512. The impulse at sample 0 has magnitude 10; the one at sample 1780 has magnitude 50. The output shows the power spectral density. The largest window, the only one containing both impulses, shows a sinusoid centered at 2500 with a peak-to-peak amplitude of 2000. The smaller windows contain at most one impulse, so their output is a constant, the square of the magnitude of any contained impulse.

The only input whose output disagreed with MATLAB's was band-limited noise (Fig. 3). This slight discrepancy may be an artifact caused by differences in how MATLAB and our C code generate band-limited inputs.

Conclusion

Our reference implementation is available at http://mickey.ifp.uiuc.edu/wiki/Software. The code compiles with the Gnu C compiler, and runs in Windows/MinGW and Linux. It is not optimized for low memory usage or exploiting particular hardware architectures. Readability suffers under such optimizations.