Cooley-Tukey FFT Algorithms

Amente Bekele

***Abstract*—The objective of this work is to discuss a class of efficient algorithms for computing the Discrete Fourier Transform (DFT). The direct way of computing the DFT problem of size N takes O(N^2) operations, where each operation consists of multiplication and addition of complex values. When using the so called Cooley-Tukey FFT Algorithms, the computation time can be reduced to O($Nlog(N)$). In this report a special case of such algorithm when N is a power of 2 is presented. The case when N is a highly composite number will also be discussed.**

I. Introduction

FOURIER Transformation is the decomposition of a function into sums of simpler trigonometric functions. Fourier Transform is the output of such decomposition. It has wide applications in science, engineering and mathematics.[1]

When the input to a fourier transform is uniformly-spaced samples of a continuous function, the transformation is called Discrete-time Fourier Transform (DTFT). The input is discrete samples and the output DTFT is a continuous function. If samples of the DTFT output that are equal in length are taken, then the transformation is called Discrete Fourier Transform (DFT).

Definition 1: *The DFT of an array of N complex input values A, is an array of N complex output values Y, such that:*

$$Y[j] = \sum_{k=0}^{N-1} A[k]W_N^{jk}$$

Where, $W_N = e^{-2\pi i/N}$ is the Nth complex root of unity. Positive integers $j \geq 0$, are indexes of Y, and positive integer $k \geq 0$ are indexes of A.

II. Direct Approach for calculating the DFT

Directly calculating the DFT requires computing the sum from $k = 0$ to $N - 1$ for each respective values of j . When the complete output is computed, there will be a total of N^2 operations where each operation is a multiplication followed by an addition. Therefore, the total time is O(N^2).

III. Cooley-Tukey FFT Algorithm

James W. Cooley and John W. Tukey in their 1965 paper [4] discussed an algorithm for computing the DFT using a divide and conquer approach. Prior to them a similar technique was discussed in various formats. Their work was different since it focused on the choice of N. They showed how special advantage is gained when choosing N to be a power of two, $N = 2^m$.

IV. FFT Algorithm when N is a power of two

Theorem 2: *The DFT where the input array A has a size $N = 2^m$ for integer $m \geq 0$ can be calculated in O($Nlog(N)$) time with the following algorithm: [3]*

procedure FFT(A)
Input: An array of complex values which has a size of 2^m for $m \geq 0$.
Output: An array of complex values which is the DFT of the input
N := A.length
if N = 1 **then return** A
else
$W_N := e^{2\pi i/N}$
W := 1
A_{even} := $(A_0, A_2, ..., A_{N-2})$
A_{odd} := $(A_1, A_3, ..., A_{N-1})$
Y_{even} := FFT(A_{even})
Y_{odd} := FFT(A_{odd})
for j:=0 to $N/2 - 1$ **do**
$Y[j]$ = $Y_{even}[j]$ + W * $Y_{odd}[j]$
$Y[j + N/2]$ = $Y_{even}[j]$ - W * $Y_{odd}[j]$
W := W * W_N
return Y

Running time:
The above algorithm divides the input into two parts each having a size of $N/2$. The dividing operation and updating of the result takes O(N). From this the following recurrence relation can be derived:

$$T(N) = 2T(N/2) + O(N)$$

The recurrence shows the total running time is O($Nlog(N)$).

In order to show that the above recursive algorithm does indeed compute the correct DFT, some concepts will be introduced in the subsequent sections. Most of the proof is presented based on ideas from CLRS book [3].

Since the DFT computation is a simple formula of sums and multiplications, one can suspect that the gain in running time compared to the direct approach comes from some special properties of the Nth complex root of unity, W_N.

Lemma 3: $(W_p^k)^d$ = (W_{pd}^{kd}) = (W_p^k)

Proof:

$W_{pd} = e^{-2\pi i/pd}$ by definition. Therefore (W_{pd}^{kd}) = $(e^{-2\pi i/pd})^{kd}$ = $(e^{-2\pi i/p})^k$ = (W_p^k)

Lemma 4: *Given arrays:*

$$S = [(W_N^0)^2, (W_N^1)^2, (W_N^2)^2 ... (W_N^{N-1})^2]$$

and

$$H = [(W_{N/2}^0), (W_{N/2}^1), (W_{N/2}^2) ... (W_{N/2}^{N/2-1})]$$

then, S contains H twice. That is for $k = 0$ to $N/2 - 1$ $S[k] = H[k]$ and for $k = N/2$ to $N - 1$, $S[k] = H[k]$.

Proof:

The first half of S can be generated with $(W_N^k)^2$ for k = 0 to $N/2 - 1$.

Note that $(W_N^k)^2$ = $(W_{N/2}^k)$ by Lemma 3 with $d = 2$ and $p = N/2$. Therefore the first half of S is the same as H.

The second half of S can be generated with $(W_N^{k+N/2})^2$ for k = 0 to $N/2 - 1$. Note that

$$(W_N^{k+N/2})^2 = (W_N^{2k+N}) = (W_N^{2k})(W_N^N) = (W_N^{2k})$$

Let $N = 2p$, then (W_N^{2k}) = (W_{2p}^{2k}) = (W_p^k) by Lemma 3 with $d = 2$. Therefore:

$$(W_N^{k+N/2})^2 = (W_N^{2k}) = (W_p^k) = (W_{N/2}^k)$$

Corollary 5: *The DFT of an array A is equivalent to evaluating the polynomial of degree $N - 1$ $P(x)$:*

$$P(x) = \sum_{k=0}^{N-1} A[k]x^k$$

at $x = W_N^j$ for $j = 0$ to $N - 1$.

Lemma 6: *Given a polynomial:*

$$P(x) = \sum_{k=0}^{N-1} A[k]x^k$$

it can be separated into polynomials:

$$P_{even}(x) = \sum_{j=0}^{N/2-1} A[2j]x^j$$

and,

$$P_{odd}(x) = \sum_{j=0}^{N/2-1} A[2j+1]x^j$$

then:

$$P(x) = P_{even}(x^2) + xP_{odd}(x^2)$$

Proof:

$$P_{even}(x^2) = \sum_{j=0}^{N/2-1} A[2j]x^{2j}$$

and

$$xP_{odd}(x^2) = x \sum_{k=0}^{N/2-1} A[2j+1]x^{2j}$$

When the sum is evaluated, the even part becomes:

$$A[0] + A[2]x^2 + ..A[N-2]x^{N-2}$$

and the odd part will be similar to the even part, but with both the coefficients and the powers of x increased by one:

$$A[1] + A[3]x^3 + ..A[N-1]x^{N-1}$$

Therefore it can be observed that the sum of the two parts is equivalent to $P(x)$.

Lemma 7: $(W_N^{k+N/2})$ = -(W_N^k)

Proof:

$$(W_N^{k+N/2}) = (W_N^k)(W_N^{N/2})$$

by Lemma 3, with $p = 1$, $d = N$ and $k = 1$ we have

$$(W_N^k)(W_N^{N/2}) = (W_N^k)(W_N^{2N}) = (W_N^k)(W_1^2) = -(W_N^k)$$

Proof for Correctness:

Using the above concepts the proof for the correctness of the FFT algorithm from Theorem 2 is as follows:

The DFT of an array with a single value is the array itself. Since both j and k are zero:

$$Y[0] = A[0]W_N^0 = A[0]$$

This is the base case for the recursion.

Note that the recursive cases of the algorithm divide the input in a similar way to what is stated in Corollary 5 and Lemma 6.

Also note that the algorithm keeps a running value for W_N^j instead of recalculating it for every iteration of j.

The for loop from $j = 0$ to $N/2 - 1$ combines the results as stated in Lemma 6. Note that in each recursive case the polynomial is evaluated at values $W_{N/2}^k$ which by Lemma 4 is the same as $(W_N^k)^2$.

The second assignment $Y[j + N/2] = Y_{even}[j] - WY_{odd}[j]$ is correct since by Lemma 7, $(W_N^{j+N/2})$ = -(W_N^j).

V. FFT Algorithm when N is highly composite

A highly composite number N is a positive integer which has more divisors than any other positive integer $N' < N$. [5]

Theorem 8: *The DFT where the input array A has a size $N = r_1.r_2$ can be calculated in running time $T = N(r_1 + r_2)$.[4]*

Proof:

The goal is to calculate the DFT as per Definition 1. The index for the output and input arrays can be re-expressed as follows:

$$j = j_1 r_1 + j_0$$

for $j_0 = 0, 1, ..., r_1 - 1$, $j_1 = 0, 1, ..., r_2 - 1$ and,

$$k = k_1 r_2 + k_0$$

for $k_0 = 0, 1, ..., r_2 - 1$, $k_1 = 0, 1, ..., r_1 - 1$

Now the DFT can be re-defined as:

$$Y[(j_1, j_0)] = \sum_{k_0=0}^{r_2-1} \sum_{k_1=0}^{r_1-1} A[(k_1, k_0)] W_N^{(j_1 r_1 + j_0).(k_1 r_2 + k_0)}$$

If the multiplication for the powers of W_N is expanded:

$$Y[(j_1, j_0)] =$$

$$\sum_{k_0=0}^{r_2-1} \sum_{k_1=0}^{r_1-1} A[(k_1, k_0)] W_N^{(j_1 r_1 + j_0).(k_1 r_2)} W_N^{(j_1 r_1 + j_0)(k_0)}$$

From the inner sum over k_1:

$$W_N^{(j_1 r_1 + j_0).(k_1 r_2)} = W_N^{r_1 r_2 j_1 k_1} W_N^{j_0.k_1 r_2} = W_N^{N j_1 k_1} W_N^{j_0.k_1 r_2}$$
$$= W_N^{j_0.k_1 r_2}$$

Since the sum over k_1 is dependent only on j_0, a new array A_1 can be defined as follows:

$$A_1[(j_0, k_0)] = \sum_{k_1=0}^{r_1-1} A[(k_1, k_0)] W_N^{j_0.k_1 r_2}$$

Therefore:

$$Y[(j_1, j_0)] = \sum_{k_0=0}^{r_2-1} A_1[(j_0, k_0)] W_N^{(j_1 r_1 + j_0)(k_0)}$$

It can be observed that there are N elements in A_1 each requiring r_1 operations to calculate, and given A_1, Y can be calculated in r_2 operations, resulting in a total time:

$$T = N(r_1 + r_2)$$

If Theorem 8 is recursively applied to $N = r_1.r_2. \ ... \ .r_m$ the following can be stated:

Corollary 9: *The DFT where the input array A has a size $N = r_1.r_2. \ ... \ .r_m$ can be calculated in running time $T = N(r_1 + r_2 + ... + r_m)$. [4]*

Note that Corollary 9 shows any gain obtained in computing the DFT using the Cooley-Tuckey type FFT algorithms instead of direct approach comes from the bound for $r_1 + r_2 + ... + r_m$.

If $r_1 = r_2 = \ ... \ = r_m = 2$, i.e $N = 2^m$, then this is similar to the special case that is discussed in IV, where $r_1 + r_2 + \ ... \ + r_m = 2m$ and $m = log_2(N)$. The running time becomes:

$$T = N(2log_2(N)) = O(Nlog(N))$$

If $N = r^m.s^n.t^p \ ...$, then it follows:

$$T = N(m.r + n.s + p.t + ...)$$

,and we have:

$$\frac{T}{N} = m.r + n.s + p.t + ...$$

Since:

$$log(N) = mlog(r) + nlog(s) + tlog(p) + \ ...$$

, the following can be drived:

$$\frac{T}{Nlog(N)} = \frac{m.r + n.s + p.t + ...}{mlog(r) + nlog(s) + tlog(p) + \ ...}$$

When N is highly composite, the term on the right side will be bounded by some 'constant'.

VI. Applications of FFT Algorithms

FFT algorithms are widely used is several applications. The IEEE journal of Computing in Science and Engineering named FFT as one of the Top 10 Algorithms with the greatest influence on the development and practice of science and engineering in the 20th century. [6]

A numeric application of FFT for polynomial multiplication has been discussed in [3]. To calculate $P_3 = P_1.P_2$. Where P_1 and P_2 are polynomials of degree bound N with array of coefficients A_1 and A_2 respectively. First the FFT for A_1 and A_2 is computed to obtain FFT(A_1) and FFT(A_2). Then:

$$FFT(A_3) = FFT(A_1) + FFT(A_2)$$

Therefore A_3 is obtained by computing the inverse FFT. The total time taken for the computation is O($Nlog(N)$).

VII. Conclusion

FFT algorithms compute the DFT of an array of size N in O($Nlog(N)$) time. This is a significant gain compared to direct computation which is O(N^2). A recursive implementation of the FFT algorithm for the special case when N is a power of 2 was discussed. For the case where N is a highly composite number, it was shown that the running time depends on the weighted sum for the factors of N.

REFERENCES

[1] Oppenheim, Alan V.; Schafer, Ronald W. (1999). *Discrete-Time Signal Processing (2nd ed.).* Prentice Hall Signal Processing Series

[2] Anil Maheshwari *Topics in Algorithm Design - COMP5703 Course notes*, School of Computer Science, Carleton University, December 2015

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliord Stein. *Introduction to Algorithms*, Third Edition. The MIT Press, 3rd edition, 2009.

[4] Cooley, James W.; Tukey, John W. (1965) *An algorithm for the machine calculation of complex Fourier series* Mathematics of Computation. 19 (90): 297301.

[5] Ramanujan, S. (1915) *Highly composite numbers* Proc. London Math. Soc. (2). 14: 347–409.

[6] Dongarra, J. Sullivan, F. (January 2000). *Guest Editors Introduction to the top 10 algorithms* Computing in Science Engineering. 2 (1): 22–23.