332:573 – Data Structures & Algorithms

Project Report

Improvements to Polynomial Multiplication Through Fast Fourier Transforms

By David Lambropoulos, Demetrios Lambropoulos

Professor Shantenu Jha
May 6th, 2018

Rutgers University
School of Engineering

Contents

1 Introduction/Motivation

Polynomials are expressions built up of a combination of constants c and symbols called variables. Polynomials with a single indeterminate x can always be written in the form

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_2 x^2 + a_1 x + a_0$$

where $a_0, ..., a_n$ are constants and x is the indeterminate variable. Polynomials can also be expressed in the following canonical form

$$\sum_{k=0}^{n} a_k x^k$$

Each term in a polynomial is product of some constant called the coefficient of the term and indeterminate raised to a nonnegative integer power n (i.e.$n \in \mathbb{Z}^+$). Polynomials are used in many fields to represent problems or model behavior such as Chemistry, Physics, Economics, Social Scientists, Calculus, Numerical Analysis, etc.

Addition and subtraction of polynomials is easy to implement costing only $O(N)$ time. However, the way polynomials are multiplied is through a method called First, Outer, Inner, Last (FOIL) as can be seen below in Figure 1. The issue with the FOIL method is that it requires too many operations to perform multiplications, especially for large polynomials. Usually real world problems will be modeled with polynomials of 10000+ terms.

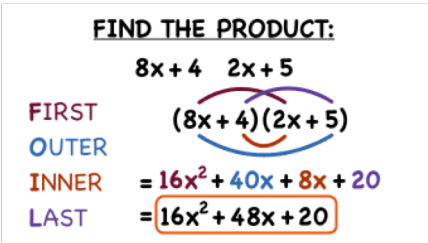

Figure 1: Demonstration of FOIL method

The main scientific motivation behind improving the speed in which we multiple polynomials is that solving large polynomials by hand is tedious and error prone. As discussed previous solving large polynomial multiplications through quadratic algorithms (i.e. FOIL) would cost more time than any person might have. A way that can fix this costly calculation is to implement recursion which uses the divide and conquer method.

2 Algorithms/Theory

Given a polynomial $A(x)$ of length m and a secondary polynomial $B(x)$ of length n then the resultant product would have a length of $m + n - 1$. As mentioned before, a polynomial

can be represented as a sum of terms consisting of constant coefficients and indeterminate variables as follows:

$$A(x) = \sum_{k=0}^{n} a_k x^k = a_n x^n + a_{n-1} x^{n-2} + ... + a_1 x + a_0$$

When representing a polynomial in code, we can represent as an array of coefficients in reverse order ($a = [a_0, a_1, ..., a_{n-1}]$). The main advantage to representing this as such allows us to reference coefficients via array indexes.

We can start analyzing the FOIL method more formally as follows. Given a polynomial:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

and a polynomial

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

The polynomial multiplication in brute force would be given by

$$C(x) = \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} a_j b_k x^{j+k}$$

resulting in a runtime of $O(n^2)$

The Fast Fourier Transform (FFT) is an improvement on the Discrete Fourier Transform (DFT) algorithm and has been recognized as on the best algorithms of the 20^{th} century[1]. The DFT is represented as:

$$X(n) = \sum_{k=0}^{N-1} x[k] e^{-j(2\pi kn/N)}$$

The DFT algorithm can easily be seen that there are N additions by N multiplications resulting in a runtime of $O(N^2)$. The FFT factors the DFT matrix into sparse factors resulting in a runtime of $O(NlogN)$.

The Cooley-Tukey FFT [2] [3] is one of the most used versions of the FFT toda and uses the idea of divide and conquer to solve the problem also with a running time of $O(NlogN)$.

3 Experimental Setup

The datasets that were used in the program were randomly generated with *generateDatasets.py* file. This generated datasets of sizes 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, and 4096. The experiment was run on Windows (Processor Intel (R) Core (TM) i7-3630QM CPU @ 2.40GHz, 2401 Mhz, 4 Cores(s), 8 Logical Processor, 8GB RAM) running Windows 10. The program was implemented with Python 3.6.

4 Results and Analysis

Too better see the competition of the Brute Force algorithm and the FFT algorithm we analyzed the results for polynomials less than degree 40. The output can be seen in Figures 5 and 3. The zoomed in FFT has less assignments and comparisons in the long run. However, there exists a small period where for small polynomials brute force outperforms the FFT. Next, looking at polynomials up to 8192 terms shows that the FFT almost disappears in comparison to the brute force algorithm. This can be seen in Figures 4 and 2.

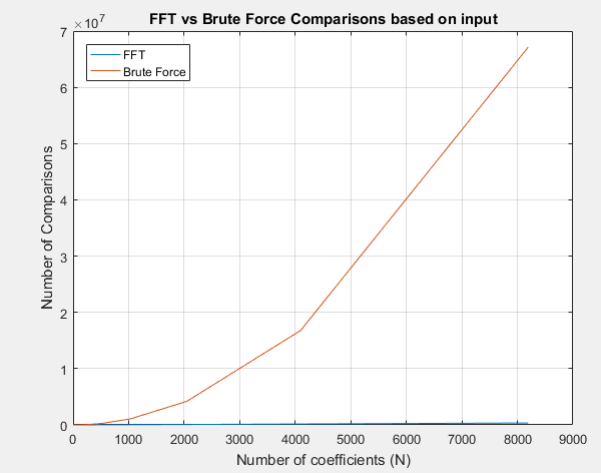

Figure 2: Number of Comparisons. FFT vs Brute Force.

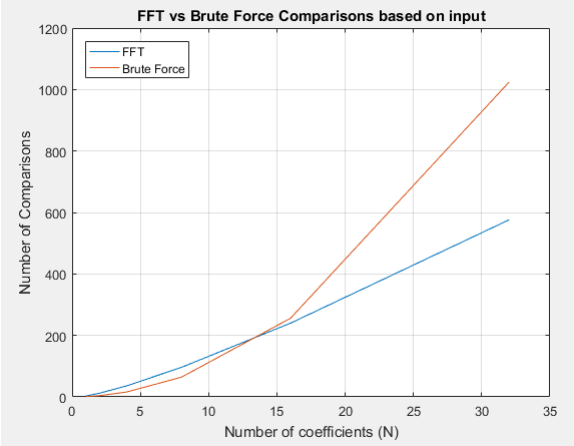

Figure 3: Number of Comparisons. FFT vs Brute Force.

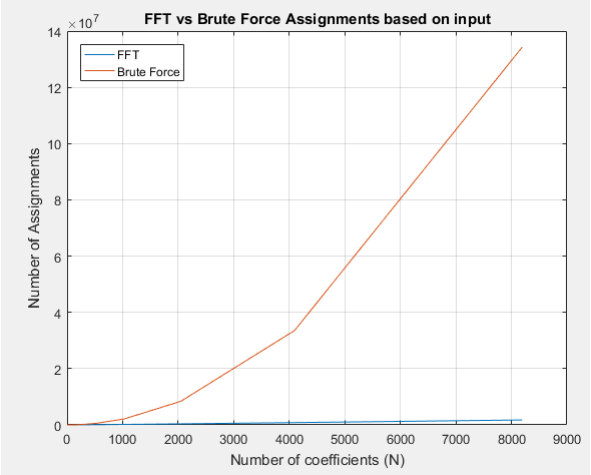

Figure 4: Number of Assignments. FFT vs Brute Force.

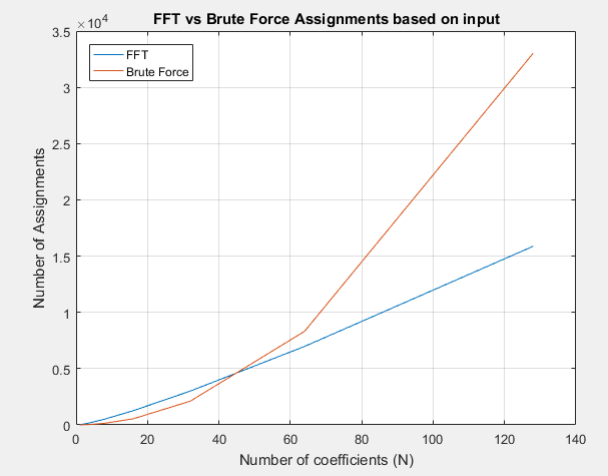

Figure 5: Number of Assignments. FFT vs Brute Force.

5 Discussion

Brute Force performs better for smaller degree polynomials (i.e. < 45 terms). Fast Fourier transforms have an overhead, however it quickly overcomes its quadratic counterpart.

References

[1] J. Dongarra and F. Sullivan, "Guest editors? introduction: The top 10 algorithms," *Computing in Science & Engineering*, vol. 2, no. 1, pp. 22–23, 2000.

[2] "Cooley?tukey fft algorithm - wikipedia." `https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm`. (Accessed on 05/06/2018).

[3] W. T. Cochran, J. W. Cooley, D. L. Favin, H. D. Helms, R. A. Kaenel, W. W. Lang, G. C. Maling, D. E. Nelson, C. M. Rader, and P. D. Welch, "What is the fast fourier transform?," *Proceedings of the IEEE*, vol. 55, no. 10, pp. 1664–1674, 1967.