Large Number Multiplication using Fast Fourier Transforms

Rio Carasco[1] e-mail: py19rc@leeds.ac.uk

***Abstract* – The ability to perform multiplication on large numbers efficiently is essential to many areas of Mathematics and Computer Science, including FHE Cryptography, Machine Learning and primality testing to name a few. In particular, The Fast Fourier Transform (FFT) Algorithm is perhaps, one of the most important and influential algorithms developed in the last century. Throughout this project, I will be researching the diverse range of algorithms constructed with the incentive of multiplying large numbers efficiently. Furthermore, I will be exploring the different variations of the Fast Fourier Transform algorithm each with their own positive and negative attributes. Moreover, it is necessary to examine the differentiations in efficiency between all methods and algorithms and how this can subsequently be applied to different fields of study. I will be analysing data originating from self-developed code and explaining the full method of my attempt at Large Number Multiplication using FFT.**

Introduction

The traditional approach to the multiplication of two numbers requires an operation on every digit of the first number with every other digit of the second number. If both numbers are 'n' digits long, this generates a complexity of $O(n^2)$, where time correlates with complexity and number of operations. This is irrelevant if 'n' is small. However, when 'n' is 'large', the time inefficiency starts becoming significant. Considering the practical applications of LNM such as prime testing, where the main variable being altered is the time it takes to complete, time efficiency becomes an important property in an algorithm. This is where the Fast Fourier Transform algorithm plays its role; it has been shown that using the FFT to multiply numbers reduces the complexity to $O(nlogn)$, vastly decreasing the number of operations especially for 'large' numbers. The FFT is not a stand-alone algorithm, algorithms like the Super FFT [10] and [18] develop the efficiency of the FFT even further. The DFT, the simplistic version of the FFT also has its uses in parallel computing as seen in [16], [25] and [30]. Alternate versions of the DFT also exist extending this method further [29]. Going beyond, we have the Quantum Fourier Transform, which is the modern equivalent of the original algorithms as seen in [24]. Developing this further we have Optalysys, a company specialising in Optical Fourier Transformation chips. I have partnered with them for the duration of this project to collect data and gain insights into their cutting-edge process of finding Fourier Transformations and their applications.

The development of methods for efficient Large Number Multiplication are key to a number of different applications. Cryptography and particularly FHE encryption, both require the use of fast efficient large number multiplication [17]. Optalysys have also had their own venture in cryptography with their lattice based encryption derived from their optical computing endeavours [4]. With regard to machine learning and AI, large matrix multiplication is a core part of these fields. FFT multiplication proves very useful here [20]. To add to this, Optalysys is currently developing their Convolutional neural networks (CNNs). CNNs evaluate image input using Fourier Transformations and are key to a diverse range of applications including, autonomous vehicles and robotic surgery [2].

As a side note, the number of scientific literature pertaining to Fast Fourier Transformations and Large Number Multiplication is much fewer than many other fields. With more than 10 times less scientific articles/journals than Quantum computing for example. Therefore finding supporting and related articles are hard to come by.

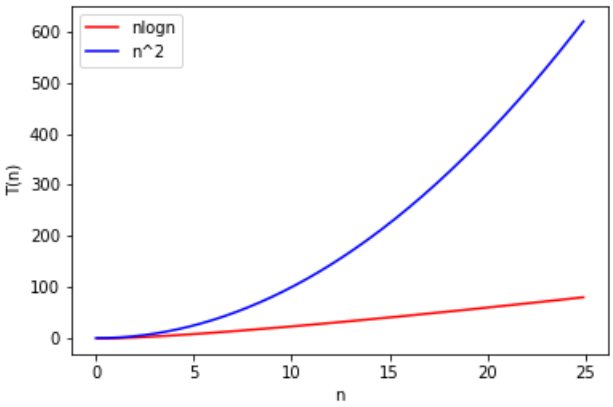

Fig. 1. illustrates the difference in computation time between *nlogn* and n^2 for values of n between 0 and 25

A Different Method to Multiplying Large Numbers

In order to multiply numbers using the FFT, there are a few prerequisites to cover. Throughout this method we will be using and manipulating polynomials, allowing us to take advantage of the properties that polynomials possess. To exploit this, we'll need to represent the integers we are multiplying as polynomials.

For example:

$$5327 \equiv 7\times 10^0 + 2\times 10^1 + 3\times 10^2 + 5\times 10^3$$

By taking $x = 10$, this becomes:

$$7x^0 + 2x^1 + 3x^2 + 5x^3$$

The value of x also indicates what base you are working in.

To make this more formal:
For an integer of length n, and base $x = 10$

$$p(x) = \sum_{i=0}^{n-1} a_i x^i \tag{1}$$

Our two integers are now of the form:

$$a(x) = a_0 + a_1 x + a_2 x^2 + ... + a_{n-1} x^{n-1} \tag{2}$$
$$b(x) = b_0 + b_1 x + b_2 x^2 + ... + b_{n-1} x^{n-1} \tag{3}$$

We will then represent this as a coefficient vector, in order to easily implement it into code,

$$\mathbf{a} = [a_0, a_1, a_2, ..., a_{n-1}]. \tag{4}$$
$$\mathbf{b} = [b_0, b_1, a_2, ..., b_{n-1}] \tag{5}$$

Conveniently, this allows us to express the two numbers as lists, comprised of the digits of the two numbers being multiplied.

```
def Num2Arr(Number):
    a = []
    for i in str(Number):
        a.append(int(i))
    a = np.array(a)
    return a
```

Above, you can see that the function "Num2Arr" converts an inputted number into a list of its digits.

We now have our integers a and b, which are now represented as coefficient vectors **a** and **b**. If we multiply the polynomials at this stage, in their "coefficient representation", it results in:

$$\begin{aligned} c(x) &= a(x)b(x) \\ &= a_0 b_0 + (a_0 b_1 + a_1 b_0)x + (a_0 b_2 + a_1 b_1 + a_2 b_0)x^2 + ... \\ &... + a_{n-1} b_{n-1} x^{2n-2} \end{aligned} \tag{6}$$

Using the distributive property to simplify the expression, we can expect a complexity of O(n^2). This is the same complexity as traditional multiplication. Evidently, there is a need to expand our method to reduce complexity.

An alternative way to identify a polynomial is through its "value representation", which involves the coordinates of which they are comprised. Rather than multiplying the equations in their "coefficient representation", we can alternately multiply a number of coordinates from both polynomials and multiply them in their "value representation". The question now is, how many coordinates are needed? [11] Furthermore, once we have multiplied the coordinates, how do we do the inverse and find the resulting polynomial?

The Interpolation Theorem

Given a set of n points in the plane, $S = (x_0, y_0), (x_1, y_1), (x_2, y_2), ..., (x_{n-1}, y_{n-1})$, such that the x_i's are all distinct, there is a unique $n-1$ polynomial $p(x)$ with $p(x_i) = y_i$, for $i = 0, 1, ..., n-1$.

Revisiting equations (4) and (5) of length "n" and degree "$n-1$", we have established that through multiplying them together we obtain equation (6) of degree "$2n-2$". Expanding on this, the Interpolation Theorem states that we will need "$2n-1$" distinct points in order to uniquely specify a polynomial of degree "$2n-2$".

To illustrate what I have explained so far we'll go through an example:

Example

When given two numbers a and b, in this case 5237 and 4619 respectfully, we'll find their "value representation" then multiply them to find the resulting polynomial.

$$a(x) = 5 + 3x + 2x^2 + 1x^3 \qquad b(x) = 4 + 6x + x^2 + 9x^3$$

As both polynomials are both of length $n = 4$, the resulting polynomial will be of degree 6. Accordingly, we are going to acquire 7 distinct points from each polynomial.
For the sake of simplicity, we'll evaluate each polynomial at $x \in \{-3, -2, -1, 0, 1, 2, 3\}$.

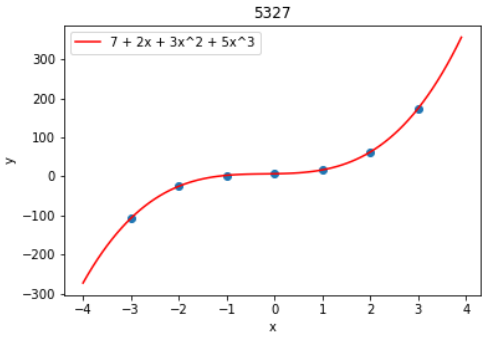

Fig. 2

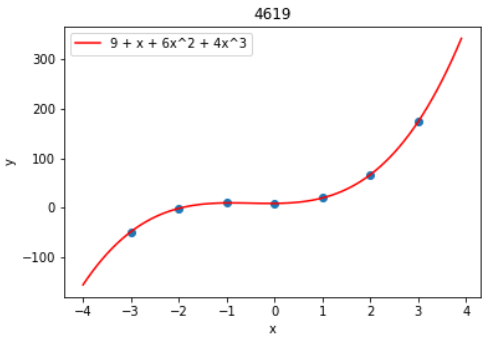

Fig. 3

The blue dots on the polynomials a(x) and b(x) are the 7 coordinates that will be multiplied together to create 7 distinct points that belong to c(x). Following this, I used a python function to evaluate the right polynomial for the coordinates presented. Once the polynomial has been determined we can then substitute x for 10 and then simplify to provide us with the result of the multiplication.

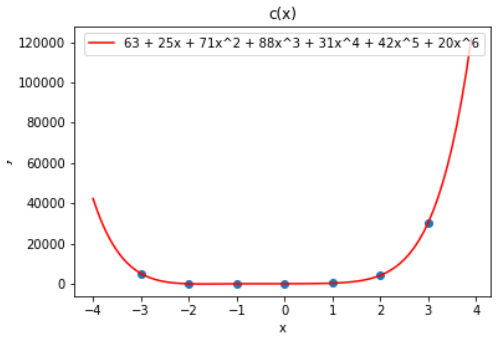

Fig. 4

Analysis of current method

At present, we have developed a non-traditional approach to multiplication. Multiplying in the "value representation" vastly reduces the number of operation to only $O(d)$ operations. However, delving into the method further reveals a few issues that need resolving.

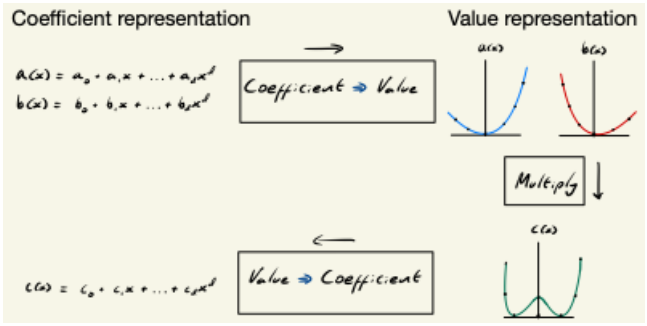

Fig. 5

1) The problem with the current evaluation: a(x) and b(x) are currently being evaluated at 7 different x values, each of which were chosen without much thought. Consider a polynomial of degree d:

$$p(x) = p_0 + p_1x + p_2x^2 + ... + p_dx^d.$$

For a polynomial of degree d, the interpolation theorem states that we need to evaluate at $n \geqslant d+1$ points. This requires finding, $(1,P(1)),(2,P(2)),...,(n,P(n))$. Evaluation time is demonstrated nicely using Horner's rule:

$$p(x) = p_0 + x(p_1 + x(p_2 + ...x(p_d)...)).$$

Clearly, for a single evaluation it will take linear time to evaluate $O(n)$. For n evaluations we are looking at a complexity of $O(nd)$ or $O(d^2)$. In the interest of minimising complexity, we will need to find a more efficient form of evaluation.

2) Switching between polynomial representations: An essential part of our method is transforming between representations. We convert from coefficient to value representation by evaluating at different values of x. We have found that our form of evaluation takes $O(d^2)$. To further develop this, I used a function to return the "coefficient representation". The traditional method would be finding the inverse of a $(n \times d+1)$ matrix to solve n different simultaneous equations.

	"coefficient representation"	"value representation"
Evaluation	$O(n)$	$O(n^2)$
Multiplication	$O(n^2)$	$O(n)$

This table demonstrates that no representation is perfect; nonetheless, we can always switch between representations as a means to reduce complexity. The obvious method would be to evaluate in the "coefficient representation", switch to the "value representation", and conduct the multiplication there. Here, we discover that the FFT algorithm is the solution to our problems. It is an extremely efficient method of evaluating polynomials, and therefore converting between representations.

The Fourier Transform

The fundamental idea behind the Fourier Transform is analysing repeating patterns by decomposing functions into its constituent sinusoids. When given some input data such as an image, machine-learning data-set, or audio track, the Fourier Transform can isolate harmonic patterns (sines and cosines). One especially useful aspect of the FT is the ability to convert between representations. In the case of audio processing, the audio input is in its amplitude domain. Whereas the output, after applying the Fourier Transform, is in its frequency domain. The FT is a key function for the entirety of data and information processing. A few of the many practical applications include decomposition of signals and file compressing.

The continuous Fourier Transform that we are familiar with is defined as:

$$F(t) = \int_{-\infty}^{\infty} f(t)e^{-2\pi ikt}\,dt.$$

In the interest of signal decomposition, we want to find a system that can treat signals with certain frequencies differently to those with other frequencies.

The most significant part of the Fourier Transform is the analysis element of the equation:

$$e^{-2\pi ikt} \tag{7}$$

Equation (7) represents a line of length 1 originating at the origin, rotating 2π radians around the axis. The "t" in the exponent allows the line to rotate around the circle. The negative represents the direction of the rotation. 2π serves as the number of rotations per time interval. The frequency k adjusts how fast the line rotates around the origin. Finally, by multiplying the function to equation (7), the line now adjusts in size depending on the value of the function. Equation (7) is also sometimes called the "twiddle factors" of the Fourirer Transform. The Fourier Transform essentially analyses the given function by "wrapping" it around the origin in the complex plane and investigating its overall movement as the winding frequency is adjusted. Judging the overall movement allows us to identify the different frequencies that make it up. To investigate the general movement, we must find the average

value of the points wrapped around the origin. This means sampling a number of points from the original signal and evaluating their complex number values when in the complex plane. Dividing by the total number of points will the give the average position of the points for a given winding frequency. Ideally, we take the integral of all the points to collate all the complex values. This enables us to then divide by the time interval to obtain the average. However, we usually leave the integral by itself in order to exaggerate the spikes depending on how prominent each frequency is.

The Discrete Fourier Transform

The Fourier Transform is very powerful but has little use with most practical applications and on algorithms. This is mainly due to the fact that computers and algorithms work with discrete values and finite sequences of data. The Discrete Fourier Transform is an altered version of the original continuous Fourier Transform.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i f n}{N}}$$

$$\frac{f}{N} \equiv k \qquad n \equiv t$$

Rather than summing over all points on a continuous function, we can use a sum over all known points. The equivalences show the discrepancy between the discrete and continuous transformations, where "f" represents the different frequencies of the sample data and "N" is the number of samples.

The DFT is responsible for the signal decompositions we see in computer programs. We can input a list of amplitudes related to a sound wave into the DFT, which will output the strength of each frequency in the sample data.

Although the DFT is a neat algorithm, it evaluates polynomials naively at only $O(n^2)$. In light of these shortcomings, our search continues for a more efficient method.

The advantage of the DFT is that the algorithm can be processed in parallel. This means that each "twiddle factor" of the DFT can be calculated separately, resulting in faster processing speeds.

Adding to this. New algorithms are being developed for the DFT. In particular a $q \times 2^m$DFT. This algorithm achieves a reduction in complexity over related algorithms. [23] [29]

The Fast Fourier Transform and the missing piece

The results of the FFT are identical to the DFT whilst achieving a complexity of only $O(nlogn)$. This is accomplished by using more advanced techniques to reduce computation and increase efficiency. [15]

There are a variety of FFT algorithms to choose from. We will be looking at the most common FFT algorithm: the Cooley-Tukey algorithm which uses a Divide and Conquer strategy to reduce computations. The disadvantage of this algorithm is that polynomials need to be of the size 2^n to keep the complexity at a minimum, because it can only acquire 2^n points for evaluation. For polynomials of size not equal to 2^n, the number of points being evaluated will be rounded up to the nearest 2^n. In reality this only increases the computation time by a factor of 2. There are other algorithms that exist that can have polynomial inputs of any size, though they are very complex and not as elegant as the Cooley-Tukey algorithm.

Divide and Conquer

The goal of this algorithm is to be able to evaluate a polynomial $P(x)$ for $x \in X$, where X denotes the set of evaluation points.

A. Divide

The first step is to divide the given problem into sub-problems. Then we will recursively repeat this further dividing into more sub-problems.

For this particular scenario, our original problem is our polynomial $P(x)$. We will split $P(x)$ into its odd and even coefficients, to create our sub-problems.

$$\begin{aligned} P(x) &= p_0 + p_1 x + p_2 x^2 + ... + p_{n-1} x^{n-1} \\ &= p_0 + p_2 x^2 + p_4 x^4 ... \\ &+ p_1 x + p_3 x^3 + p_5 x^5 ... \\ &= p_0 + p_2 x^2 + p_4 x^4 ... \\ &+ x(p_1 + p_3 x^2 + p_5 x^4 ...) \end{aligned}$$

We have now separated the odd and even coefficients. In the interest of defining our odd and even coefficient lists, we need to have them in orthodox form.
We will define:

$$P_{even}(x) = \sum_{k=0}^{n/2} p_{2k} x^k = \left[p_0, p_2, p_4, ...\right]$$

$$P_{odd}(x) = \sum_{k=0}^{n/2} p_{2k+1} x^k = \left[p_1, p_3, p_5, ...\right]$$

Now the two polynomials can be written in vector form, combining them to create $P(x)$:

$$\begin{aligned} P(x) &= P_{even}(x^2) + x P_{odd}(x^2) \\ &= p_0 + p_2 x^2 + p_4 x^4 ... \\ &+ x(p_1 + p_3 x^2 + p_5 x^4 ...) \end{aligned} \tag{8}$$

B. Conquer

We now have our sub-problems: $P(x)$ is split into $P_{even}(x^2)$ and $P_{odd}(x^2)$. Evaluating these sub-problems gives us $P(x)$. The key to the Divide and Conquer algorithm is to then recursively compute $P_{even}(y)$ and $P_{odd}(y)$ for $y \in X^2$, where $X^2 = \{x^2 \mid x \in X\}$, keeping in mind $|X| = |X^2|$

C. Combine

After computing $P_{even}(x^2)$ and $P_{odd}(x^2)$, we now want to use equation (8) to find $P(x)\ \forall\ x \in X$. Which was the objective of our algorithm.

Calculating Complexity

With regard to calculating the complexity of the algorithm, we need to first compute the complexity of its constituent parts.

Firstly, we created our two sub-problems each with size $n/2$: $2T(n/2,|X|)$. Here we introduce the Divide and Combine step. Dividing the polynomial into even and odd coefficients takes linear time $O(n)$. Combining $P_{even}(x^2)$ and $P_{odd}(x^2)$ using equation (8) involves a constant number of arithmetic operations for each x, which is also conducted in linear time. [19] Combining these elements results in:

$$\begin{aligned} T(n,|X|) &= 2T(n/2,|X|)+O(n+|X|) \\ &= O(n^2) \end{aligned}$$

We are on the right track, but we are still missing a key piece of the method that reduces the complexity to $O(nlogn)$.

The bulk of the efficiency will come from the recursive step of the algorithm. In total, there are $logn$ levels to the recursion and it terminates at the point where there is only one coefficient to evaluate.

The principal issue with the current algorithm is the set of evaluation points is always $|X|$ at all levels of the recursion. If our aim is to get our current $O(n^2)$ algorithm to $O(nlogn)$, we will need to reduce the size of the evaluation points as the recursion continues.

For instance, reducing the set of evaluation points by a factor of 2 for each recursions yields:

$$\begin{aligned} T(n,|X|) &= 2T(n/2,|X|/2)+O(n+|X|) \\ T(n) &= 2T(n/2)+O(n) \\ &= O(nlogn) \end{aligned}$$

We have finally found the part of our algorithm we need to modify to achieve a complexity of $O(nlogn)$. Thus, when converting our set X to X^2 we want to reduce the size of the set X by a factor of 2.

Collapsing sets

Ideally we want to achieve $|X^2| = \frac{|X|}{2}$ recursively. This is by definition a *Collapsing Set*. If $|X^2| = \frac{|X|}{2}$ and recursively X^2 is collapsing, **or** $|X| = 1$.

Positive-Negative pairs

When considering the points we want to obtain for evaluation, are there a set of points where knowing the value of one point immediately implies the value of another? This is the feature that positive-negative points possess.

For any polynomial, if we have a set of positive values, the negative counterparts can be calculated with no cost to

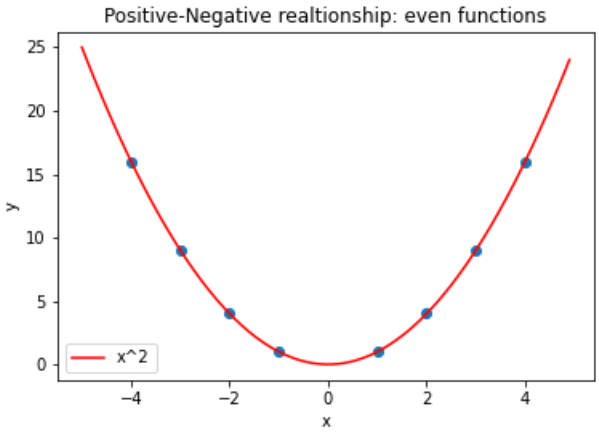

Fig. 6

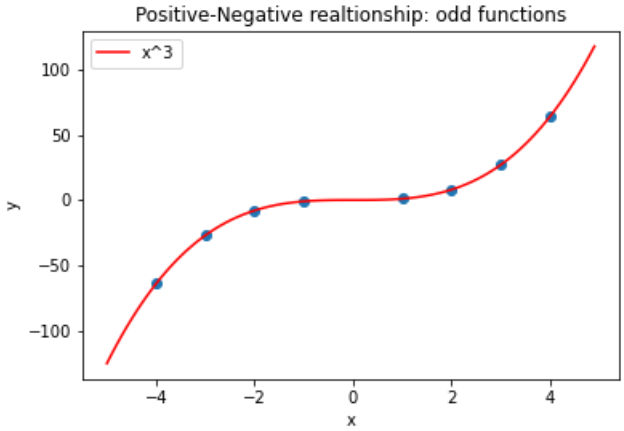

Fig. 7

complexity and vice versa. For even functions:

$$P(-x) = P(x)$$

This also works in a similar way for odd functions:

$$P(-x) = -P(x)$$

Consider our polynomial equation, $P(x) = P_{even}(x^2) + xP_{odd}(x^2)$, for a value x_i where $x \in X$:

$$P(x_i) = P_{even}(x_i^2) + x_i P_{odd}(x_i^2) \tag{9}$$

$$P(-x_i) = P_{even}(x_i^2) - x_i P_{odd}(x_i^2) \tag{10}$$

The aim now is to find the ideal approach to creating our set of evaluation points, including the features of collapsing sets and positive-negative pairs. We want to start with a set of values that collapses after we square the contents. In addition, after squaring, we want this set to be comprised of positive-negative pairs. In the interest of simplicity, we are going to invert our method. Thus, we are looking for a method to double the number of values after taking the square root of all values in the set, while also creating positive-negative pairs. [11]

Starting with the base case and also our final evaluation

value:

$$X = \{1\} \qquad |X| = 1, \tag{11}$$

$$X = \{1, -1\} \qquad |X| = 2 \tag{12}$$

After taking the square root of our initial value, it is clear this has achieved our goal. A positive-negative pair is produced in conjunction with the set size doubling.
Conducting this process twice more, yields:

$$X = \{1, -1, i, -i\} \qquad |X| = 4 \tag{13}$$

$$X = \{1, -1, i, -i, \pm\frac{\sqrt{2}}{2}(1+i), \pm\frac{\sqrt{2}}{2}(1-i)\} \qquad |X| = 8 \tag{14}$$

Here we are acquainted with complex values, which are the result of $\sqrt{-1}$. These are additionally the results of the roots of unity.

Roots of Unity

The roots of unity are fundamentally the solutions to $z^n = 1$, where $z \in \mathbb{C}$. When found, the roots hold a number of different properties which prove useful for our algorithm.

First we'll convert z into polar form. This representation allows us to easily find our roots. Since z is a complex number:

$$z = a + bi$$
$$z = cos(\theta) + isin(\theta)$$

Going back to our roots of unity equation, we need $z = 1$. Substituting $\theta = 2\pi$ yields:

$$cos(2\pi) + isin(2\pi) = 1$$

Expanding on this, we know that the *cos* and *sin* functions are periodic. Any multiple of 2π will give the same result:

$$cos(2k\pi) + isin(2k\pi) = 1 \qquad k = 0, 1, 2, ...$$

Now that we have found z, we will broaden this to z^n. [12]

De Moivre's Theorem

$$(cos(\theta) + isin(\theta))^n = cos(n\theta) + isin(n\theta)$$

If $1 = cos(2k\pi) + isin(2k\pi)$ for $k = 0, 1, 2, ...$, then applying De Moivre's Theorem $z^n = 1$ is equivalent to:

$$z^n = cos(2k\pi) + isin(2k\pi)$$
$$z = (cos(2k\pi) + isin(2k\pi))^{\frac{1}{n}}$$
$$z = \frac{cos(2k\pi)}{n} + i\frac{sin(2k\pi)}{n} \qquad k = 0, 1, 2, ...n-1 \tag{15}$$

Where k denotes the root number and n denotes the number of roots. To illustrate this, let us find the 4th roots of unity, $n = 4$:

$$n = 4 \qquad z = \frac{cos(2k\pi)}{4} + i\frac{sin(2k\pi)}{4}$$
$$k = 1 \qquad z_1 = cos(\frac{\pi}{2}) + isin(\frac{\pi}{2}) \qquad z_1 = i$$
$$k = 2 \qquad z_2 = cos(\pi) + isin(\pi) \qquad z_2 = -1$$
$$k = 3 \qquad z_3 = cos(\frac{3\pi}{2}) + isin(\frac{3\pi}{2}) \qquad z_3 = -i$$
$$k = 4 \qquad z_4 = cos(2\pi) + isin(2\pi) \qquad z_4 = 1$$

The primitive roots of unity denotes the complex value z for $k = 1$. By employing De Moivre's Theorem on our primitive root of unity we can obtain the rest of the roots by taking z_1^k.

The results of the 4th roots of unity are the same as the evaluation values for $|X| = 4$. It turns out the ideal $n = 2^j$, where $j \in \mathbb{N}$, evaluation values for our algorithm correspond to the n^{th} roots of unity. These values create collapsing sets: $|X^2| = \frac{|X|}{2}$.
Our algorithm splits the input data into two groups, of odd and even coefficients. The DFT is then calculated of them both before combining them to generate the full DFT. The FFT continues to use this idea by recursively performing this action.
Putting this all together, we have created the Cooley-Tukey FFT, which operates in *nlogn* time. [27] [14] [22]
There are certain caveats to this algorithm. The number of digits allowed to be inputted into the system in its basic form is only 2^n. Additionally, when analysing efficiency, there is a point in the algorithm where it is more efficient to use DFT than the FFT. This is dependant on several factors but a combination of the two would allow us to find a more optimal algorithm. This is not to take away from efficiency we have achieved - for "large" numbers we have vastly reduced the computation time. [13]

The Inverse FT/DFT/FFT

The inverse FT converts data from the frequency domain back into the spacial domain. With respect to the FFT, we use the inverse FFT to gain back our coefficients for our polynomial (Interpolation). The Inverse Fourier Transformation is as follows:

$$f(t) = \int_{-\infty}^{\infty} F(t) e^{2\pi i k t}\, dt.$$

. The Inverse Fourier Transform is very similar to the Fourier Transform, this is also true for the DFT and FFT. Once the DFT/FFT algorithms are developed, creating the corresponding inverse is very straight forward. [1] [6]

Errors

While conducting the Fourier Transformation and converting between representations, round-off errors are inevitable. Taking the standard deviation between the input signal and the calculated output gives a single value for this error. The round-off error of the FFT is much lower than of the DFT, due to the fewer number of calculations required [5], [28].

The Optical method

The methods we have previously discussed are algorithms which are ultimately based on conventional computer processing on a silicon chip. However, recently this "conventional computing" is approaching its physical limits. As the size of transistors continue to decrease in size, the increase in energy and thermal requirements outweighs the improvements in performance. In order to maintain progression we need to explore alternative processing solutions. Optical computing, where calculations are conducted using light, is an example of an unorthodox processing method.

Information processing through optical means has been proposed for over 50 years. The vastly lower power levels and extreme processing speeds compared to conventional silicon chips, make it a very attractive alternative. Information is encoded and recovered through silicon photonics, allowing for a free-space optical Fourier Transform operation. This is the technique employed by Optalysys. [9]

1) Optical Computing and Fourier Optics: Light can be thought of as a duality of a particle and a wave. The wave-like properties that light possesses allows it to interfere with itself producing interference patterns. This forms the foundation of silicon photonics. Light propagates in a sinusoidal fashion, thus the information required for the DFT can be represented as beams of light by adjusting properties of the light. For example, passing the beams through a medium to slow down the light modifies its phase. This form of calculation can be performed in absence of any transistors. [3]

We can focus the interference patterns created by a light source by directing the light through a lens at the appropriate focal length. The Fourier Transform of the light source is then found at the focal plane.

This technique is extremely energy efficient and the complexity of this process is of the order $O(1)$! The beauty of this process is that not only is the result returned at the speed of light, but this processing speed stays consistent no matter how much information is inputted.

Optical systems use a similar structure to FFT's. Considering an FFT algorithm, the smallest Fourier Transform the algorithm performs is known as the radix. The equivalent for an Optical system is the max resolution that can be inputted. Finding the Fourier Transform of a data-set, through optical means, requires the data be split into frames, where each frame is equivalent to the radix of the optical system. Akin to the FFT, the FT of each frame is calculated individually, using the optical device. These are then combined to produced the full FT of the data-set.

The outcome is a much smaller complexity than the FFT's *nlogn*

Results

I've collated results from different sources to compare efficiency. For these results I have tested my own Large Number Multiplication algorithm with Optalys's Optical Fourier Transformation simulator.

The results show that the Optical chip has a clear speed advantage compared to my algorithm. Additionally, the time the Optical chip takes scales up slower compared the LNM algorithm. As the numbers get larger the Optical chip is better suited for the calculation.

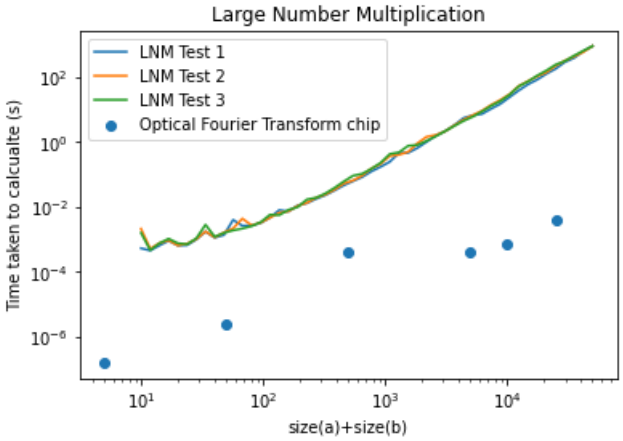

Fig. 8. Illustrates the difference in computation time between LNM algorithm and the Optalysys Optical system

When comparing results of two algorithms, hardware must be taken into account. The device I am running my algorithm on must be compared to the simulator hardware. With this many variables the comparison between efficiency is hard to be made. Therefore I have decided to focus my efforts, in this report, on formulating the algorithms and researching them instead of collecting results. More information on this is in the appendix.

Conclusion

While large number multiplication is a prominent computational problem in current times, there does not exist an algorithm superior to all others. Different algorithms focus on different aspects of efficiency. With some focusing on parallel processing adaptations [25], for example, while others focus on efficient padding [16]. We have covered the issues that arise when developing algorithms and methods for multiplying large numbers and how to overcome them.

It is clear that the using FFT's, and variations of it, to multiply large numbers is a significantly more efficient process than traditional multiplication. However, exploring new advances in this field suggests that Quantum Fourier Transforms and Optical systems show far more superior efficiency and energy management. While also making use of past algorithms to speed them up further. New developments are continuously arising and more specific programs are being developed to cater for specific advantages.

Looking in the direction of future research. Optalysys and their current Optical Fourier Transform system, supersedes previous systems in nearly all aspects. The various advantages include, faster processing speeds, more reliable computation and energy efficiency to name a few. The focus should be on expanding the Quantum systems while developing algorithms to be able to utilise them to their full potential [24].

Appendix

2) DFT Code: To demonstrate how the DFT would work in practical applications, I have developed the following code in

python:

```
import numpy as np
 def DFT(sampleYValues):
1    N = len(sampleYValues)
2    DFTarray = np.array([])
3    for k in range(N):
4        X_k = np.array([])
5        for n in range(N):
6            b = (-2*np.pi*k*n)/(N)
7            complex = np.cos(b)+1j*np.sin(b)
8            X_i = sampleYValues[n]*(complex)
9            X_k = np.append(X_k, X_i)
10        X_k = np.sum(X_k)
11       DFTarray = np.append(DFTarray, X_k)
12   return DFTarray
```

Analysing the code, I use two different "for" loops to iterate through the sample list. Firstly, a given frequency is chosen through the "for" loop on line 3. Then, for this frequency, the "for" loop on line 5 finds the corresponding complex value for each sample "n". When all the samples have been evaluated, the sum is taken of all these values and is appended to a list "DFTarray". This denotes a result for a given frequency, which is then repeated for the rest of the frequencies. The function then ends by returning this list.

3) FFT Code: I have also created a python implementation to demonstrate the FFT:

```
import numpy as np
def EvenOdd(List):
1   ListShape = np.reshape(List, (int(len(List)/2), 2))
2   Even = ListShape[:, 0]
3   Odd = ListShape[:, 1]
4   return Even, Odd

def FFT(P):
5   n = int(len(P))
6   if n == 1:
7       return P
8   w = np.cos(2*np.pi/n) + np.sin(2*np.pi/n)*1j
9   P_e, P_o = EvenOdd(P)
10  y_e, y_o = FFT(P_e), FFT(P_o)
11  y = np.zeros(n, dtype = 'complex_')
12  for i in range(int(n/2)):
13      y[i] = y_e[i] + w**i*y_o[i]
14      y[int(i + n/2)] = y_e[i] - w**i*y_o[i]
15  return y
```

I'll go through an example to illustrate what is happening in the code.

Example

In this instance we'll be finding the FFT of 5321. Running the main FFT function from lines 5 to 9, yields:

$$FFT([5,3,2,1])$$

$P(x) = 5+3x+2x^2+x^3$ $\qquad$ $n = 4$

$\omega = e^{\frac{2\pi i}{4}} = i$

$P_e(x^2) = 5+2x \rightarrow [5,2]$ $\qquad$ $P_o(x^2) = 3+x \rightarrow [3,1]$

So far, we have found the value of ω and n. On line 9 $P(x)$ is split into it's even and odd coefficients using the function "EvenOdd". As we move to line 10 of the code, the recursive component of the algorithm is now applied.

$$FFT([5,2])$$

$P(x) = 5+2x$ $\qquad$ $n = 2$

$\omega = e^{\frac{2\pi i}{2}} = -1$

$P_e(x^2) = 5 \rightarrow [5]$ $\qquad$ $P_o(x^2) = 3 \rightarrow [3]$

$FFT([5]) \rightarrow [5]$ $\qquad$ $FFT([2]) \rightarrow [2]$

We have now acquired our y_e and y_o values for $FFT([5,2])$, therefore we can now move onto creating our array "y". Taking advantage of equation (9) and (10), we can find elements of "y" by taking the positive negative pairs of the roots of unity. This is the first stage of evaluation.

$$[\omega^0, \omega^1, \omega^2, \omega^3,] = [1, i, -1, -i]$$

$$y = [0,0]$$

$y[0] = y_e[0] + \omega^0 y_o[0] = 7$ $\qquad$ $y[1] = y_e[0] - \omega^0 y_o[0] = 3$

$$FFT([5,2]) \rightarrow [7,3]$$

$$FFT([3,1])$$

$P(x) = 3+x$ $\qquad$ $n = 2$

$\omega = e^{\frac{2\pi i}{2}} = -1$

$P_e(x^2) = 3 \rightarrow [3]$ $\qquad$ $P_o(x^2) = 1 \rightarrow [1]$

$FFT([3]) \rightarrow [3]$ $\qquad$ $FFT([1]) \rightarrow [1]$

$y_e = 3$ $\qquad$ $y_o = 1$

$$y = [0,0]$$

$y[0] = y_e[0] + \omega^0 y_o[0] = 4$ $\qquad$ $y[1] = y_e[0] - \omega^0 y_o[0] = 2$

$$FFT([3,1]) \rightarrow [4,2]$$

In this instance we have found the evaluations of all the recursive stages. Now we need to substitute the values back into the initial FFT computation.

$FFT([5,3,2,1])$, $\quad$ continued...

$$y = [0,0,0,0]$$

$$y[0] = y_e[0] + \omega^0 y_o[0] = 11$$
$$y[2] = y_e[0] - \omega^0 y_o[0] = 3$$

$$y[1] = y_e[1] + \omega^1 y_o[1] = 3 + 2i$$
$$y[3] = y_e[1] - \omega^1 y_o[1] = 3 - 2i$$

$$FFT([5,3,2,1]) \rightarrow [11, 3+2i, 3, 3-2i]$$

4) Large Number Multiplication FFT Code:

```
import numpy as np
from scipy.fft import fft, ifft
from random import randint

# Generates random digits of specific and
equal length
(e.g between 10^9 = 1000000000 and (10^10 - 1) =
9999999999)
def randomDigits(n):
    range_start = 10**(n-1)
    range_end = (10**n)-1
    return randint(range_start, range_end)

# Checks if a number is a power of 2; if not,
log2 will return a non-integer value
def Pwr2(Number):
    if np.log2(Number).is_integer():
        return True
    else:
        return False

# Finds a value of L such that L+M-1 is a
power of 2 precursor for acyclic -> cyclic/
negacyclic convolution method
def Info(A, B):
    M = len(B)
    L = 1
    while not Pwr2(L+M-1):
        L+=1
    OL = len(A) + len(B) - 1
    N = L + M - 1
    return M, L, N, OL

#Converts a single number to a string and
then separates the string into individual
integer values placed in an array
def Num2Arr(Number):
    a = []
    for i in str(Number):
        a.append(int(i))
    a = np.array(a)
    return a

# Iteratively slices an array of polynomial
values into sub-arrays of specific length
# Takes in L from a prior instance of Info()
as a testing metric (intuition; Galois group
for cyclotomic extension. Calculating a splitting
field here?)
def Split(Arr, L):
if L == 1:
    return np.array_split(Arr, len(Arr))
else:
    R = len(Arr)%L
    # Length of array modulo L; tells us
    how "long" the array is relative to a
    pertinent value. Relates to calculations
    in power 2, given the earlier role of L.
    NoR = len(Arr) - R
    # Subtracts the excess length of the
    array (returning an "ideal" array
    length for splitting to power 2?
    NewArr = Arr[:NoR]
    # Selects the region of the array
    up to the length of NoR
    if len(NewArr) == L:
    # Check to see if the length of the
    new array is equivalent to L; if so:
     # np.empty returns a new array of a
     given shape without filling with
     zeros. Passing 2 to np.empty
     presumably initiates an empty array
     of size 2
        FinalArr = np.empty(2, object)
        FinalArr[0] = NewArr    # Stores
        the cut-down region
        of a passed array in the
        first entry of the new empty array
        if R == 0:  # Check to see if there
        is a remainder left over (i.e is the
        length of Arr identified with 0
        in modulus L)
            pass
        else:
            RArr = np.array(Arr[-R:])
            # If not, stores the cut-off
            section of the original array
            in the second entry of FinalArr
            FinalArr[1] = RArr
        return FinalArr
    NewArrSplit = np.array_split(NewArr,
    len(NewArr)/L)
    if R == 0:
        pass
    else:
        RArr = np.array(Arr[-R:])
    if R == 0:
        R_Value = 0  # this function splits
        an array into sub-arrays of equal
        power-of-two lengths
    else:
        R_Value = 1
    FinalArr = np.empty(len(NewArrSplit) +
    R_Value, object)
    for i in range(len(NewArrSplit)):
        FinalArr[i] = NewArrSplit[i]
    if R == 0:
        pass
    else:
```

```
            FinalArr[-1] = RArr
        return FinalArr

#pad to ensure power-2 calculations for
efficient FT
def Padding(Arr, N):
    return np.pad(Arr, (0, N-len(Arr)),
    'constant')


A = #insert first number
B = #insert second number
# or use randomDigits(N) where N represents
the number of digits you want your random
number to have

print(A)
print(B)

A = Num2Arr(A)    # Convert these numbers to
individual elements in an array
B = Num2Arr(B)

M, L, N, OL = Info(A, B)   # Extracts pertinent
info from A,B for decision making on rearrangement
efficient calculation


ASplit = Split(A, L)   # Split A into tractable
sub-arrays, power of two length

#Arrays returned from Split are combined
into OAList
OAList = np.empty((len(ASplit), N))
# Empty array the length of A, each
sub-array N long
for i in range(len(ASplit)):
# Enters array of N padding zeros into each
entry of A that isn't totally filled; ensures
parity in length with B.
    OAList[i] = Padding(ASplit[i], N)

# Pads B to power of two length defined by N;
B is *not* split
h = Padding(B, N)

#This is the evaluation and multiplication stage.
FOAList contains arrays of the result
FOAList = []
for i in OAList:
    #FFT component
    FOAList.append(ifft(fft(i) * fft(h)))
# Classic convolution theorem; Schonhage-Strassen.
Multiplying each FTd sub-array with the FT of
B and then taking the inversion (bit-shifting)
FOAList = np.array(FOAList)
FOAList = np.reshape(FOAList,
(len(OAList), N))
FOAList = np.round(np.real(FOAList))
k = round(len(OAList))
height = k
width = (((k - 1) * L) + N)
m = np.zeros((height, width))

for j in range(np.shape(FOAList)[0]):
    m[j,j*L:N+(j*L)] = FOAList[j]

#Final contains the Arrays ready for
Overlap Add
Final = []
for i in range(np.shape(m)[1]):
    Final.append(np.sum(m[:,i]))
Final = np.array(Final)
Final = Final[:OL]
Final = np.flip(Final, 0).astype(int)

digitArray = []
for i in Final:
    digits = [int(x) for x in str(i)]
    digitArray.append(digits)

dA = len(digitArray)

#Determines the dimensions of the Overlap
Add array
Len = 0
for i in range(dA):
    Len_i = len(digitArray[i]) + i
    if Len < Len_i:
        Len = Len_i
    else:
        pass

FinalSum = np.zeros((dA, Len))

#Inputs the arrays from digitArray into the
Overlapp Add array
for i in range(dA):
    FinalSum[dA-i-1, (Len-1-i)-
    (len(digitArray[i]))+1
    : Len-i] = digitArray[i]

#Adds up the individual digits, carrying
remainders when needed
Carry = 0
Result = []
for i in range(np.shape(FinalSum)[1]):
    ResultCol = sum(FinalSum[:,-(i+1)]) + Carry
    if ResultCol >= 10:
        Sum = [int(x) for x in str
        (int(ResultCol))]
        Carry = Sum[0]
        Result.append(int(Sum[1]))
    else:
        Result.append(int(ResultCol))
        Carry = 0

Result = np.flipud(Result)
```

```
Result = ''.join(str(i) for i in Result)
print(Result)
```

The objective of this code is to use the Fast Fourier Transform to multiply "large" numbers. The overlap add method allows me to determine the convolution of large sequences.

This piece of code took around 6 months to develop and refine. The Split function had to be re-written several times to account for every different outcome possible. One of the biggest issues arised from the overlap add method. In early versions of my code I collected the coefficients for the final polynomial to create the outputted result. For each digit a_i I attached $\times 10^i$. This gave the right answer but was very inefficient. This primarily because the computer had to calculate 10 to the power of large indices. The solution was the final adding array. This allowed the digits to be added individually without the need of large calculations. [8] [7]

As a result of the time and effort put into this code, I have licensed it and Optalysys have requested permission to use the code to develop their state of the art Optical Fourier Transform chip. This is mainly because of the practical applications of my code and because the Fast Fourier Transform input vector is continuously changing in size.

Simulator results

Using OPT SIMULATOR 2; 1x4 in-plane optical FT core @ 1GHz (Scalable) [21] [26]

5x5 digit number: ————————

Total number of optical frames: 1536 Runtime: 1.536e-07s Energy cost: 3.072e-07J

50x50 digit number: ————————

Total number of optical frames: 24576 Runtime: 2.4576e-06s Energy cost: 4.9152e-06J

500x500 digit number: ————————

Total number of optical frames: 3,833,856 Runtime: 0.0003833856s Energy cost: 0.0007667712J

5000x5000 digit number: ————————

Total number of optical frames: 4,194,304 Runtime: 0.0004194304s Energy cost: 0.0008388608J

10,000x10,000 digit number: ————————

Total number of optical frames: 7,340,032 Runtime: 0.00073400320000000001s Energy cost: 0.00146800640000000001J

25,000x25,000 digit number: ————————

Total number of optical frames: 37,748,736 Runtime: 0.0037748736s Energy cost: 0.0075497472J

Interpretation of results:

The simulator applied here simulates a single 1x4 optical FT process that can be constructed as an in-plane silicon-photonic component and scaled to multiple such sub-units across a SiPh die. Each such 1x4 optical core is designed to operate at a modulation rate of 1 GHz with a native 10 bit (signed complex 4 bit real/4 bit imaginary) precision. Optical outputs are reconstructed into larger FTs of greater size and precision. The runtime can be further reduced by scaling operations across multiple such optical sub-units. Associated electronic transposition and memory transfer energy costs are not included in the calculation, only the core photonic circuit components (laser power consumption, ADC/DAC, TIA and photodiodes).

Reported frame values indicate the number of 1x4 10-bit "frames" (synchronous modulator operations encoding individual bit-slices of input data) required to execute all sub-computations required for the task. Runtime and energy cost are cumulative and based on designed operating parameters.

REFERENCES

[1] Fast fourier transform. how to implement the fast fourier... | by cory maklin | towards data science.

[2] Model uncertainty in cnns — optalysys.

[3] The multiply and fourier transform unit: A micro-scale optical processor.

[4] Optical computing for cryptography: Lattice-based cryptography — optalysys.

[5] Speed and precision comparisons.

[6] Understanding fast fourier transform from scratch — to solve polynomial multiplication. | by aiswarya prakasan | medium.

[7] Understanding the fft algorithm | pythonic perambulations.

[8] Using fourier transforms to multiply numbers - interactive examples.

[9] What we do (and why we do it) — optalysys.

[10] Sos S. Agaian and Okan Caglayan. Super-fast Fourier transform. 6064:127 – 138, 2006.

[11] A. A. Aleksashkina, A. N. Kostromin, and Yu V. Nesterenko. On a fast algorithm for computing the fourier transform. *Moscow University Mathematics Bulletin*, 76:123–128, 5 2021.

[12] G. D. Bergland. A guided tour of the fast fourier transform. *IEEE Spectrum*, 6:41–52, 1969.

[13] E. O. Brigham and R. E. Morrow. The fast fourier transform. *IEEE Spectrum*, 4:63–70, 12 1967.

[14] Patrick Chiu. Transforms, finite fields, and fast multiplication. *Mathematics Magazine*, 63:330, 12 1990.

[15] James W. Cooley, Peter A.W. Lewis, and Peter D. Welch. The fast fourier transform and its applications. *IEEE Transactions on Education*, 12:27–34, 1969.

[16] Richard Crandall and Barry Fagin. Discrete weighted transforms and large-integer arithmetic. *Mathematics of Computation*, 62:305, 1 1994.

[17] Jean Pierre David, Kassem Kalach, and Nicolas Tittley. Hardware complexity of modular multiplication and exponentiation. *IEEE Transactions on Computers*, 56:1308–1319, 2007.

[18] Sergey Dolgov, Boris Khoromskij, and Dmitry Savostyanov. Superfast fourier transform using qtt approximation. *Journal of Fourier Analysis and Applications*, 18:915–953, 10 2012.

[19] Ando C Emerencia. Multiplying huge integers using fourier transforms. 2009.

[20] W. Morven Gentleman. Matrix multiplication and fast fourier transforms. *Bell System Technical Journal*, 47:1099–1103, 1968.

[21] Alan H. Karp. Bit reversal on uniprocessors. *http://dx.doi.org/10.1137/1038001*, 38:1–26, 7 2006.

[22] HwaJoon Kim and Somchai Lekcharoen. A cooley-tukey modified algorithm in fast fourier transform. *Korean Journal of Mathematics*, 19:243–253, 9 2011.

[23] Kenli Li, Weihua Zheng, and Keqin Li. A fast algorithm with less operations for length-n=q× 2m dfts. *IEEE Transactions on Signal Processing*, 63:673–683, 2 2015.

[24] Joseph L. Pachuau, Arnab Roy, and Anish Kumar Saha. Integer numeric multiplication using quantum fourier transform. *Quantum Studies: Mathematics and Foundations*, 9:155–164, 2 2022.

[25] Marshall C. Pease. An adaptation of the fast fourier transform for parallel processing. *Journal of the ACM (JACM)*, 15:252–264, 4 1968.

[26] Z. Qian, M. An, R. Tolimieri, and C. Lu. Self-sorting in-place fft algorithm with minimum working space. *IEEE Transactions on Signal Processing*, 42:2835–2836, 1994.

[27] CHARLES M. RADER and GEORGE C. MALING. What is the fast fourier transform? *Proceedings of the IEEE*, 55:1664–1674, 1967.

[28] James C. Schatzman and James C. Schatzman. Accuracy of the discrete fourier transform and the fast fourier transform. *JOURNAL OF SCIENTIFIC COMPUTING*, pages 1150–1166, 1996.

[29] S. Winograd. On computing the discrete fourier transform. *Mathematics of Computation*, 32(141):175–199, 1978.

[30] Gang Xie and Yang Chun Li. Parallel computing for the radix-2 fast fourier transform. *Proceedings - 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science, DCABES 2014*, pages 133–137, 12 2014.