Fast Fourier Transform

Danny August Ramaputra
Muhamad Ilman Nafian
Wahyu Ananda Duli Tokan

December 2019

1 Introduction

1.1 What is Fourier Transform?

Fourier Transform forms the decomposition of a function into some simpler functions, in a sense that the simpler functions we produce are part of the generating function from the function we decomposed. This transformation was first proposed by Joseph Fourier in 1822, defined by the following equation.

$$F(\omega) = \int_{-\infty}^{\infty} f(x) \cdot e^{-2\pi i x \omega} dx \tag{1}$$

It is often being related to the decomposition of waves into the smaller sinusoidal waves. This decomposition is usually called the transformation from time domain into frequency domain by sampling the function $f(x)$ on the given time steps ω, we can analogize this as finding the 'ingredients' which composes a given function by filtering it.[1] Such signal transformation is especially useful in signal processing, where we can analyze a given signal, and be able to break it down into the sinusoidal signals which adds up to it.

The fact that the function involves the use of imaginary numbers, the transform will convert a real number space into a complex number space. And using the inverse of this Fourier Transform, we can compose back this complex number space of 'ingredients' back to the original function. The inverse of the transform is not much different, it simply has the periodic part inverted and scaled back to 2π

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) \cdot e^{2\pi i x \omega} d\omega \tag{2}$$

1. Kalid Azad, "An Interactive Guide To The Fourier Transform," Accessed 2019-12-4, 2017, `https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/`.

1.2 Discrete Fourier Transform

Discrete Fourier Transform (DFT) is a computational implementation of Fourier Transform. This is done by reformulating the original transform 1 which uses integrals into a computable function which does not need symbolic algebra manipulations, this is done by using Riemann Sum. It was proposed by Gauss, as discovered by Heideman and Johnson.[2]

DFT will compute a series of real number samples x_n from the function $f(x)$ into a series of complex number sample k of $F(\omega)$. Using Riemann Sum, we can define the member of k as the Fourier transform of $f(x)$ with evenly-spaced time steps $\omega = n/N$ where N is the total number of samples.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{2\pi i}{N} nk} \tag{3}$$

Since we need to compute an $O(n)$ complexity equation for every member of k, we end up with an $O(n^2)$ complexity.

1.3 Fast Fourier Transform

Fast Fourier Transform (FFT) is a collection of algorithms that tries computes DFT in a faster manner. There are some properties of the periodic part on DFT, $e^{-\frac{2\pi i}{N} nk}$, that can be exploited as an advantage for computing a faster Fourier Transform. This part has a periodic property due to its nature of being complex numbers. With Euler's formula $e^{ix} = \cos(x) + i\sin(x)$ we can rewrite it as follows.

$$X_k = \sum_{n=0}^{N-1} x_n \cdot (\cos \frac{2\pi}{N} nk - \sin \frac{2\pi i}{N} nk)$$

The essence is now to harvest this feature, together with breaking down this problem into smaller ones. A strategy is to divide this sum into two by splitting them based on their index. We will attempt to divide them by their even and odd indices. The following equations are derived from Cooley and Tukey's equations[3] for FFT with radix splitting. Vladimir Stojanovic clarifies these expansions in his lecture notes.[4]

2. Michael Heideman, Don Johnson, and Charles Burrus, "Gauss and the history of the fast Fourier transform," *IEEE ASSP Magazine* 1, no. 4 (1984): 14–21.

3. James W. Cooley and John W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation* 19, no. 90 (1965): 197–301.

4. Vladimir Stojanovic, "Course materials for 6.973 Communication System Design. MIT OpenCourseWare (http://ocw.mit.edu/), Massachusetts Institute of Technology," Accessed 2019-12-18, 2006, `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-973-communication-system-design-spring-2006/lecture-notes/lecture_8.pdf`.

$$
\begin{aligned}
X_k &= \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i \frac{nk}{N}} \\
&= \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{-2\pi i \frac{(2n)k}{N}} + \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{-2\pi i \frac{(2n+1)k}{N}} \\
&= \sum_{n=0}^{N/2-1} x_{2n} \cdot e^{-2\pi i \frac{nk}{N/2}} + \sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{-2\pi i \frac{nk}{N/2}} \cdot e^{-2\pi i \frac{k}{N}} \\
&= \left[\sum_{n=0}^{N/2-1} x_{2n} \cdot e^{-2\pi i \frac{nk}{N/2}} \right] + e^{-2\pi i \frac{k}{N}} \cdot \left[\sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{-2\pi i \frac{nk}{N/2}} \right]
\end{aligned}
$$

By utilizing the nature of the composite number's periodicity, we can obtain the other half of the combination stage by calculating the periodic factor half many times, hence deduce the following.

$$
\begin{aligned}
X_{k+N/2} &= \left[\sum_{n=0}^{N/2-1} x_{2n} \cdot e^{-2\pi i \frac{n(k+N/2)}{N/2}} \right] + e^{-2\pi i \frac{(k+N/2)}{N}} \cdot \left[\sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{-2\pi i \frac{n(k+N/2)}{N/2}} \right] \\
&= \left[\sum_{n=0}^{N/2-1} x_{2n} \cdot e^{-2\pi i \frac{nk}{N/2}} \cdot e^{-2\pi i n} \right] + e^{-2\pi i \frac{k}{N}} \cdot e^{-\pi i} \cdot \left[\sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{-2\pi i \frac{nk}{N/2}} \cdot e^{-2\pi i n} \right] \\
&= \left[\sum_{n=0}^{N/2-1} x_{2n} \cdot e^{-2\pi i \frac{nk}{N/2}} \cdot 1 \right] + e^{-2\pi i \frac{k}{N}} \cdot -1 \cdot \left[\sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{-2\pi i \frac{nk}{N/2}} \cdot 1 \right] \\
&= \left[\sum_{n=0}^{N/2-1} x_{2n} \cdot e^{-2\pi i \frac{nk}{N/2}} \right] - e^{-2\pi i \frac{k}{N}} \cdot \left[\sum_{n=0}^{N/2-1} x_{2n+1} \cdot e^{-2\pi i \frac{nk}{N/2}} \right]
\end{aligned}
$$

This divide and conquer approach is the method proposed by Cooley and Tukey in their paper.[5] The approach is echoed by Jeff in his book.[6] It is elaborated that the nature of the periodicity of complex numbers is the core of this FFT algorithm. The $n^{"th"}$ root of unity allows us to reduce half of the additional computation needed during the combination stage. This root of unity gives rise to the symmetrical property of the values for ω that we pick.

We have split the problem into 2 sub-problems by separating them by even and odd indices, this is the Decimation in Time (DIT) variant with radix 2. Another variant is Decimation in Frequency (DIF), here we are splitting the problem into 2 by the first half and the last half.

5. Cooley and Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series."
6. Jeff Erickson, *Algorithms* (2019).

1.4 Parallel Fast Fourier Transform

Another approach to executing this FFT algorithm is by doing it in a parallel fashion, as described by Chu.[7] A similar approach was echoes by Velez and Texeira,[8] where they are applying this parallelized FFT for use in a clustered system. Our implementation has two recursive call to get the even and odd result. This two code are independent of each other. When we are trying to get the even result, the main function are idling, waiting to get result of even. We can eliminate this idling by executing both lines asynchronously. This way, there will be less time wasted by the function which will improve the overall execution time.

1.5 Hybrid Fast Fourier Transform

Mathematically speaking, DFT will perform worse than FFT asymptotically, since $O(n^2)$ is greater than $O(n \log n)$. In programs however, this is not always true. Since Cooley-Tukey algorithm is using the divide and conquer approach which divide the problem into sub-problems, we implement the algorithm with a recursive inside it. This recursive call will create many function call, even at lower input size. These calls created some overhead that makes the Cooley-Tukey implementation run slower than naive DFT at lower input size.

We propose a solution to this problem by adding a simple logic on the recursive case check. If the input size is lower than a set threshold, we can, instead of doing more recursive FFT calls, run the Naive DFT implementation which will result in marginally better execution time.

Note that we do not claim to invent or own this strategy as ours. There might be some other existing implementation of this hybrid algorithm. This proposed solution is simply a manifestation of what we observed and analyzed from Cooley-Tukey's implementation.

We hypothesize with a threshold of problem size 4. An empirical test will be performed to evaluate this analysis.

7. Eleanor Chu and Alan George, *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms* (Florida: CRC Press, November 1999), ISBN: 0849302706.

8. William Velez and Marvi Teixeira, "Parallel DFT Implementation and Benchmarking in a Cluster Architecture," May 2018, doi:10.13140/RG.2.2.11740.69765.

2 Implementation

2.1 Naive DFT

The following is a naive DFT implementation in Python. We implement the equation 3 above in Python. Here we can see the two nested loops that makes the $O(n^2)$ complexity. For every item in the array, we need to compute the value, and each of them will iterate every item in the array again, hence the overall $O(n^2)$ complexity.

```
1 def dft(vector, N=None):
2     if N is None:
3         N = len(vector)
4     fourier = []
5     for k in range(N):
6         fourier.append(complex(0))
7         for n, x in enumerate(vector):
8             theta = math.tau * k * n / N
9             fourier[k] += x * complex(math.cos(theta), math.sin(theta))
10    return fourier
```

2.2 Cooley-Tukey FFT

Below is the FFT implementation in Python using divide and conquer approach. This method applies the aforementioned strategy by Cooley and Tukey,[9] hence is being named after them, the Cooley-Tukey algorithm.

```
1 def fft(vector, N=None, w=None):
2     if N == 1:
3         return vector
4     else:
5         if N is None:
6             N = len(vector)
7         if w is None:
8             w = complex(math.cos(math.tau / N), math.sin(math.tau / N))
9         vector = padding(vector, nearest_power(N))
10        fourier_even = fft(vector[0::2], nearest_power(N) // 2, w ** 2)
11        fourier_odd = fft(vector[1::2], nearest_power(N) // 2, w ** 2)
12        fourier = [0] * N
```

9. Cooley and Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series."

```
13          x = 1
14          for i in range(N // 2):
15              fourier[i] = fourier_even[i] + x * fourier_odd[i]
16              fourier[i + N // 2] = fourier_even[i] - x * fourier_odd[i]
17              x *= w
18          return fourier
19
20 def nearest_power(number):
21      return 1 << (number - 1).bit_length()
22
23 def padding(vector, pad):
24      return np.append(vector, np.zeros(pad - len(vector)))
```

We first divide the problem into two sets based on their index. The base case for the recursive is when the input is only one vector, return said vector. On each recursive call, after we get the value of the odd and even number, we iteratively do the 'butterfly' operation to get the result of the transform.

2.3 Parallel FFT

Parallel FFT is implemented very similarly to Cooley-Tukey FFT. With the only difference being the recursion calls are done in parallel. Multi-threading or multi-processing can be used in Python's implementation. Due to Python language limitation, a threaded solution is not feasible, where threads are executed serially. On the other hand, processes are run in parallel, but is still not feasible due to the limited computational power. Hence, we suggest a solution to spawn new processes only until a certain depth of the recursion tree, e.g. only the first 2 layers of the tree, hence 6 processes.

2.4 Hybrid FFT

There is not much of a difference in comparison to Cooley-Tukey FFT. An if statement is added as one of the base case of the recursion. This statement will catch any problem size less or equals to 4, and perform a Naive DFT on this sub-problem.

3 Analysis

3.1 Naive DFT

Refer back to DFT algorithm in the previous section. In DFT, we have two nested loops that compute and then iterate through every item in the array. In general, it will create $O(n^2)$

3.2 Cooley-Tukey FFT

Below is the complexity analysis of Cooley-Tukey FFT algorithm. Beginning with a recursion tree of the algorithm, we have the following figure.

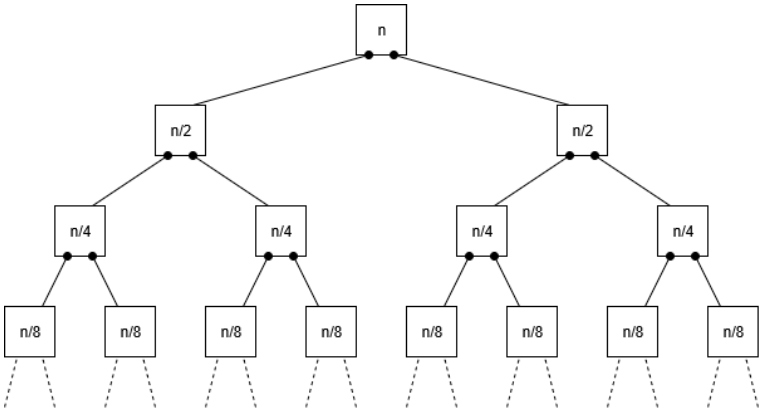


Figure 1: Recurrence Tree

Total cost at depth $i = 2^i c(\frac{n}{2^i})$

$$T(n) = cn + \frac{1}{2}cn + \left(\frac{1}{2}\right)^2 cn + ... + \left(\frac{1}{2}\right)^{\log_2 n - 1} + \Theta\left(\frac{n}{2}\right)$$

Since the Tree is equal in all levels, then $T(n) = O(n \log_2(n))$
Using Master Theorem, we can have

$$T(n) = 2T\left(\frac{n}{2}\right) + O\left(\frac{n}{2}\right)$$
$$a = 2, b = 2, f(n) = n$$
$$n^{(\log_b a)} = n^{(\log_2 2)} = n^1 = n$$
$$f(n) = \Theta(n^{(\log_2 2)}) = \Theta\left(\frac{n}{2}\right)$$

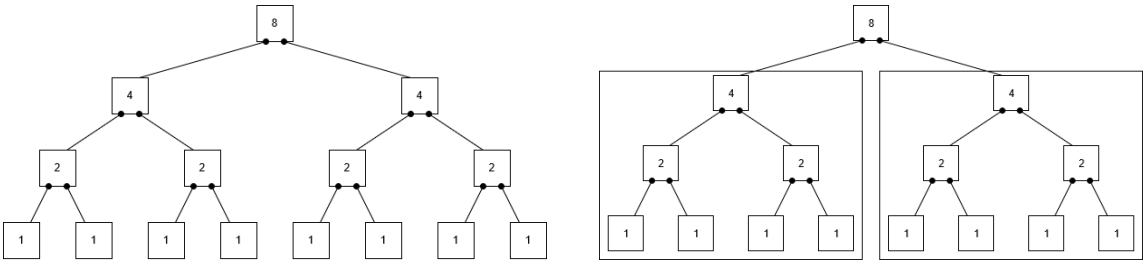
Using case 2, we can conclude that the solution is $T(n) = \Theta(n \log n)$

3.3 Parallel FFT

The Parallel FFT algorithm produced more of the same as Cooley-Tukey's. The use of parallelization does not change the time growth function, only the idle times are reduced. This leads to the same complexity of $O(n \log_2(n))$, but with a smaller constant.

3.4 Hybrid FFT

The Hybrid FFT on the other side, has slightly different problem Tree interpretation from the original FFT.



(a) Cooley-Tukey FFT (b) Hybrid FFT

Figure 2: Recurrence Tree

In the figure above, we illustrate how we combine the DFT with the FFT. So when the problem size has reached 4, we will stop using FFT and implement the DFT.

$$T(n) = cn + \frac{1}{2}cn + \left(\frac{1}{2}\right)^2 cn + ... + \left(\frac{1}{2}\right)^{\log_2 n - 4} + \Theta\left(\frac{n}{2}\right)$$

Which still results in $O(n \log n)$

4 Application

4.1 Polynomial Convolution

The convolution of a polynomial is what we usually refer as the multiplication of polynomials. When we are multiplying two polynomials, we have to keep in mind the general structure of the polynomial itself, and the common method in multiplying them.

A commonly used method for polynomial multiplication is the Long Multiplication method, or better known as the Horner's method. This method will yield a complexity of $O(n^2)$

There is a phenomenon where if we pairwise multiply samples, or points, from the two input polynomials, we will get the value of the point which lies on the convoluted polynomial. The point-to-point multiplication phenomenon is what we will exploit to achieve $O(n \log n)$ polynomial multiplication.

Using Fourier Transform, we are able to transform the coefficients of these polynomials into point values, which then we can pairwise multiply them, and reverting it back to coefficients using inverse Fourier Transform. Converting it to and back from the point values has a complexity of $O(n \log n)$, and the pairwise multiplication has the complexity of $O(n)$. The following diagram[10] describes the stages for using FFT for convoluting polynomials.

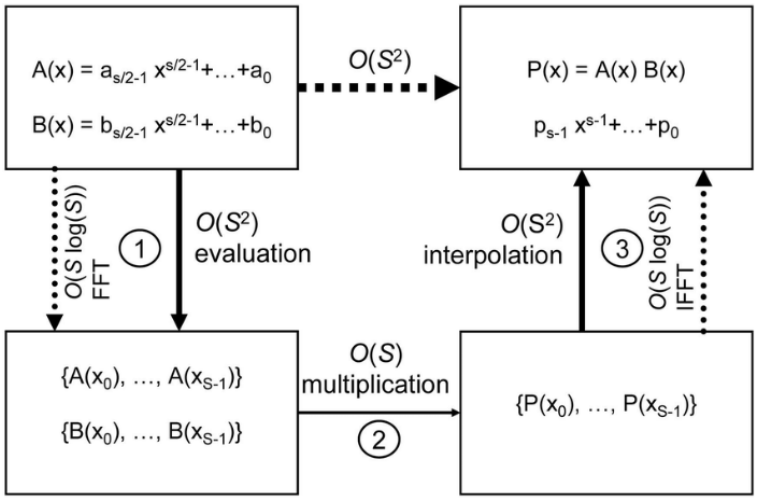


Figure 3: Polynomial convolution steps[10]

10. Jean David, Kassem Kalach, and Nicolas Tittley, "Hardware Complexity of Modular Multiplication and Exponentiation," *Computers, IEEE Transactions on* 56 (November 2007): 1308–1319, doi:10.1109/TC.2007.1084.

4.2 Image Compression

Most of our activity on our phone now require an internet access. From chatting with friends and family, watching videos, to working on assignment. More and more data are transferred through the internet at once, even with out current average internet speed, there is still a demand for faster overall load times. The usual culprits are the digital media such as videos and pictures. In order to both save spaces and sped up load times, we can compress the image.

DFT or in our case, FFT can be used to compress an image. By transforming the image data into the frequency space, we can sample the important frequency while ignoring frequency that are too small or don't have much impact. The resulting image will only contain data that has enough magnitude to affect the actual image. This is called a lossy compression where some detail will be missing in the final form.

In implementing an image compressor, we will be using a 2-dimensional DFT/FFT. We can store the image data in a 2-dimensional matrix where every matrix item represent a single pixel. In our example, we will be using a grayscale image where every pixel can be represented as one value which is the brightness. The brightness value goes from 0 or black to 255 or white.

A common method of 2D FFT using 2D FFT is doing it row-by-row, then column-by-column on the previously transformed vector. Chu described it very clearly in Chapter 23 of his book[11] by showing the following diagram.

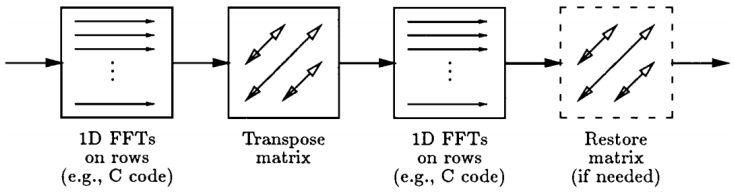


Figure 4: 2D FFT as visualized by Chu[11]

With this in mind, a 2D FFT would be implemented as follows. Not only it would work for FFT, our other implementation of the transforms can use this method.

```
1 def fft2(matrix):
2     image = np.zeros(matrix.shape, dtype=complex)
3     for row in range(image.shape[0]):
4         image[row, :] = fft(matrix[row, :])
5     for col in range(image.shape[1]):
6         image[:, col] = fft(image[:, col])
7     return image
```

11. Chu and George, *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms.*

With this 2D Fourier Transform, we are converting the image into a frequency spectrum. This spectrum represents the whole image losslessly, and there are some portions of this spectrum that we can discard which leads too little to no noticeable change in our original image. Removing some of these pixels is what is known as compressing the image, as the total number of pixels being stored is reduced. Removing a ratio of this spectrum, means the image is lossly compressed by that ratio.

With 2D Inverse Fourier Transform, we will be retrieving the image back from this spectrum space, definitely with a reduction of fidelity with the fact that some of the components which composes the full image has been removed. To reduce the impact of image quality upon reduction, the parts of the spectrum space to be removed has to be picked selectively, by choosing the spectrum with the lower intensity means we are removing the less significant 'building blocks' of the image.

5 Empirical Testing

With the various Fourier Transform implementation that we have discussed, an empirical testing for the analyzed time complexity had to be done. The test that is done is an actual use case of compressing an image. The infamous image of Lenna will be used as the image to be compressed. This image is obtained from a database of standard test images from ImageProcessingPlace.com.[12]

Figure 5: Lenna

When we were implementing FFT using parallelism, we notice that running the code in Windows OS takes longer than normal. We then tested the code on a server running Linux with 20 core CPU, the results are way better, in a sense that the process initialization overhead is much smaller. We then run the test on a smaller Linux machine CPU which still resulted in faster execution time than Windows. All the time comparison below was run on a Linux system.

The system that we run these empirical tests on has the following specifications.

- Intel Core i7-7700HQ @ 2.8 GHz 4 Core 8 Thread CPU
- 8GB DDR4-24006 RAM

The algorithm implementations that we will be testing against are the following.

- NumPy FFT
- Naive DFT
- Cooley-Tukey FFT
- Hybrid FFT
- Parallel FFT

12. ImageProcessingPlace.com, "Standard Test Images," Accessed 2019-12-5, `http://www.imageprocessingplace.com/root_files_V3/image_databases.htm`.

5.1 Full Comparison

We are including NumPy's FFT, where NumPy is a well renowned library for numerical computations in Python, into the test as a control of the actual known to be working FFT implementation. We will be using various resolutions of Lenna, they are 2×2, 4×4, 8×8, 16×16, 32×32, 64×64, 128×128, 256×256, 512×512, 1024×1024. These images are generated by up- or down- scaling the original image of 512×512. Scaling the image is done by using 'scikit-image' Python library with anti-aliasing enabled.

To keep it simple and consistent, we will be using a grayscale version of Lenna with a compression rate of 95% (i.e. 95% of the pixels in spectrum space will be zeroed). This compression rate should not impact the Fourier Transformations, this number is picked just to have a visible change in the image compression result.

The execution time of only the Fourier Transforms are recorded during the image compression process. Note that the transforms will occur twice per compression since we also need to do the inverse FFT to be able to see the result of the compression. The test yields the elapsed time as described in Table 1.

Size	Naive DFT	Numpy FFT	Cooley-Tukey FFT	Parallel FFT	Hybrid FFT
2	0.000192	0.000142	0.000552	0.376954	0.000342
4	0.000526	0.000138	0.000944	0.402597	0.000606
8	0.003881	0.000189	0.003712	0.414056	0.003521
16	0.032410	0.000227	0.015648	0.462044	0.012350
32	0.211289	0.000214	0.066745	0.396816	0.058013
64	2.240428	0.000274	0.245048	0.445946	0.243763
128	15.445040	0.001081	1.284378	0.640790	1.204814
256	110.013697	0.004331	4.271058	1.365311	4.241263
512	987.753255	0.016011	18.247052	4.931996	17.431362
1024	7261.018879	0.126694	72.955286	19.111572	70.630778

Table 1: Pixel Count against Transformation Time

These data are visualized in Figure 6 and Figure 7. The crosses are the actual data that we retrieve from our test with the resolutions stated above. Spline interpolation with no smoothing is applied to each of the result set to yield a smooth approximation of the growth rates.

Naive DFT is seen to be having a very incremental growth compared to all the other algorithms. This growth rate is expected for a time complexity of $O(n^2)$, compared to the other's of $O(n \log n)$. We did not expect that this asymptotic growth to be very noticeable on a rather small data set. We need to keep in mind that the actual number pixels being processed are the squares of the pixel

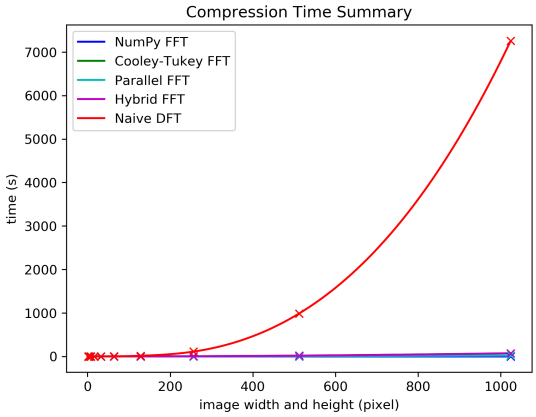


Figure 6: Pixel count against time

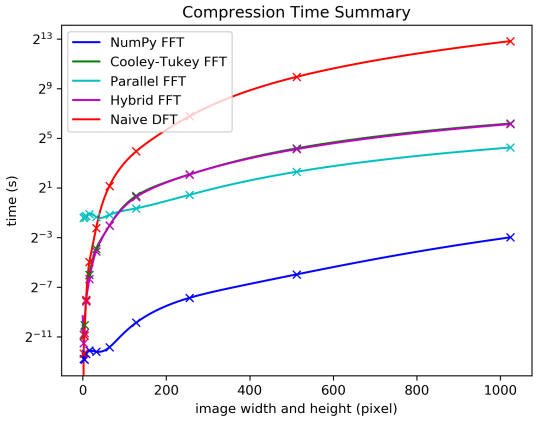


Figure 7: Pixel count against log-scaled time

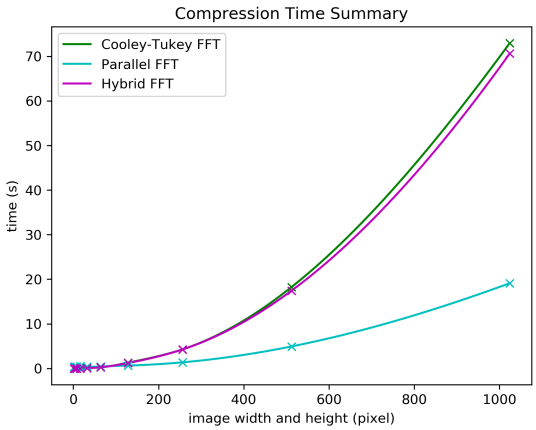


Figure 8: Pixel count against time

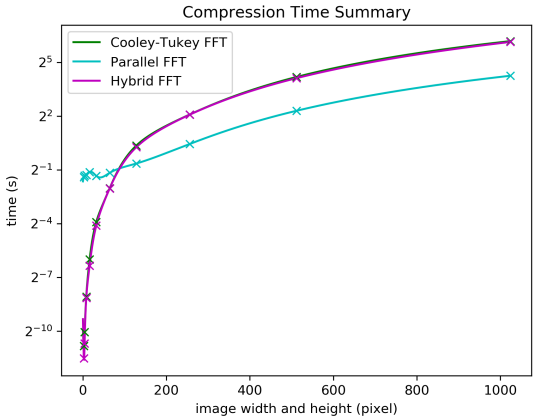


Figure 9: Pixel count against log-scaled time

count in the x-axis, due to the fact that we are using 2D Fourier Transforms on the 2D image.

Using a log-scaled time axis, as shown in Figure, we can analyze the overall growth more clearly. We are seeing how NumPy FFT, Cooley-Tukey FFT, Parallel FFT, and Hybrid FFT are on the same growth slope. Comparing that on the increasing growth of Naive DFT. This shows that all the FFT algorithms that we are testing grows on the same asymptotic $O(n \log n)$, but with different constants.

NumPy's implementation seems to have a very small complexity constant, making it the most efficient from the rest. After a breakdown of NumPy's implementation of FFT,[13] apparently they are using an optimised binary in CPython to allow a faster vector multiplication during the process.

13. NumPy, "numpy/fft/_pocketfft.c," Accessed 2019-12-8, 2019, `https://github.com/numpy/numpy/blob/master/numpy/fft/_pocketfft.c`.

Eliminating the extremes of NumPy FFT and Naive DFT, a closer look on the remaining algorithms is revealed, shown in Figure 8 and Figure 9. Hybrid FFT, seems to perform very similarly to Cooley-Tukey's FFT. This pattern remains true until a larger sized data set is introduced - Hybrid FFT performs better by a slight amount.

5.2 Hybrid FFT Tresholds

To verify the validity of our hypothesis of $threshold = 4$, we tested the Hybrid FFT algorithms with various other threshold values $\in [2, 4, 8, 16, 32, 64]$. We used Lenna test image of sizes up to 2048 pixels. This yields an interesting result as shown in Figure 10 and Figure 11.

Unfortunately, our test did not produce much meaningful data. The graph shows very similar running time for each constant. We have tried run the test several times, and each run produce similar yet inconsistent result. Compared to Cooley-Tukey FFT, only Hybrid FFT with a constant of 4 constantly run marginally faster. The rest of the constant produce inconsistent and sporadic result, some are faster in a certain size, then on the next run it ran worse at that same size. We speculate this behavior was caused by an uncontrolled external factor of the CPU usage and scheduling.

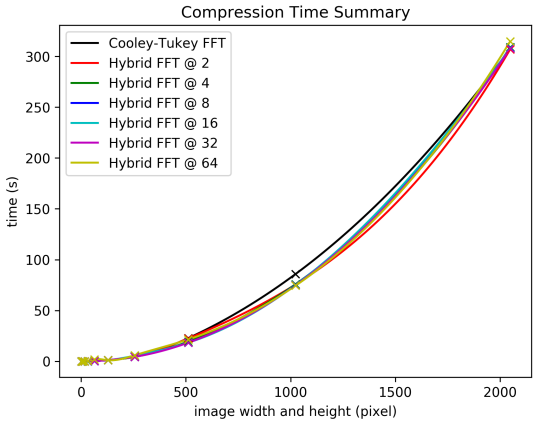


Figure 10: Compression times with various threshold values

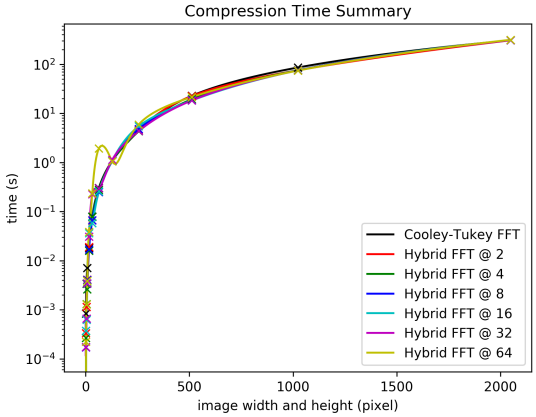


Figure 11: Log-scaled compression times with various threshold values

5.3 Parallel FFT CPU Cores

Parallel FFT does not seem to perform very well on the smaller data sets. This is likely caused by the large overhead that Python's multiprocessing package induces when a new process is being initialized. This overhead seems to be a limitation on the specific implementation, using other

languages will reduce this overhead count, hence smaller time constants. Despite so, the overall growth will not be any better than $O(n \log n)$. Given a very large data set, it seems that this parallelized FFT will perform best with it's distributed computation resources.

Figure 12 clearly visualizes the impact of multiprocessing to our algorithm. Our test simply runs Parallel FFT with core count $\in [1, 2, 3, 4, 6, 8]$ with a larger test image up-scaled up to 2048×2048 pixels. Each run produce faster and faster result, up until core count 6. After that point, we can see in Figure 13 that the speedup growth is decreasing in core count 8. This is most likely caused by the CPU bring fully utilized, bottle-necked by the FFT computation in each CPU cores.

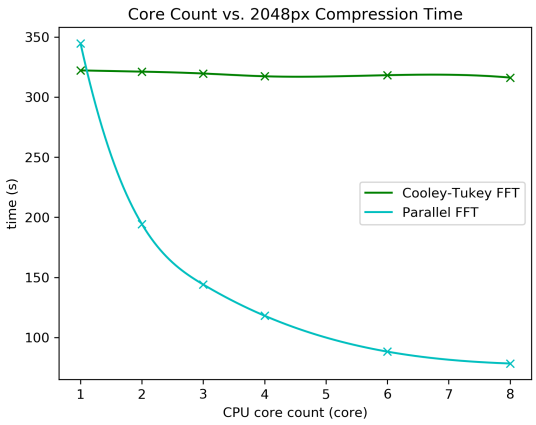


Figure 12: CPU core count against time

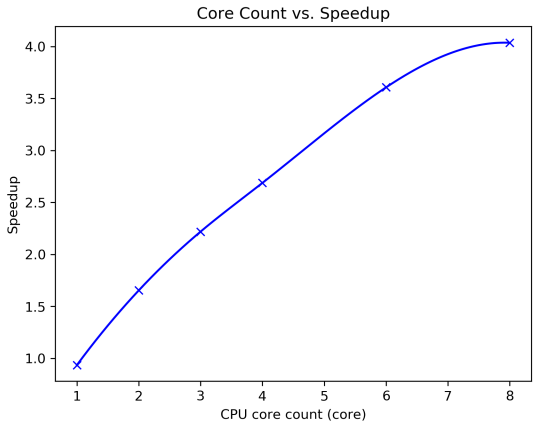
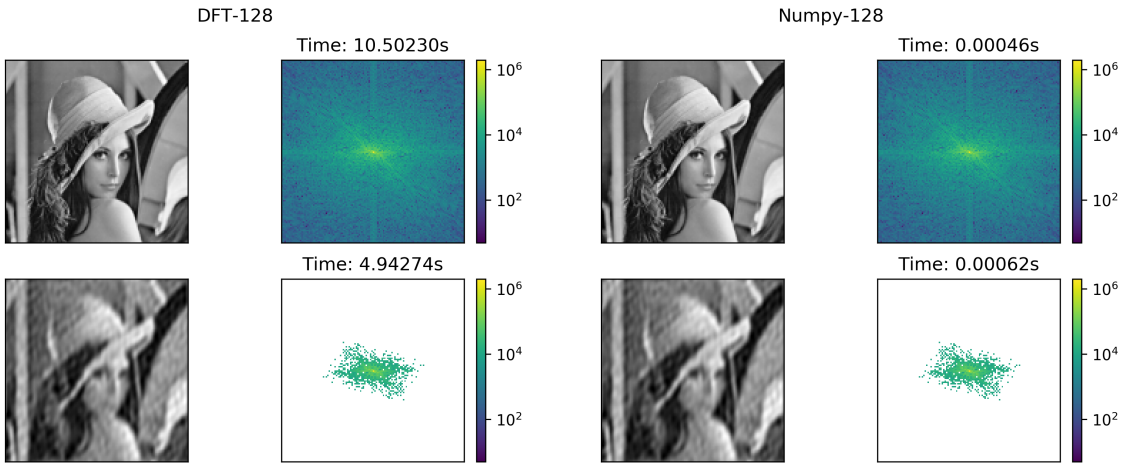


Figure 13: CPU core count against speedup

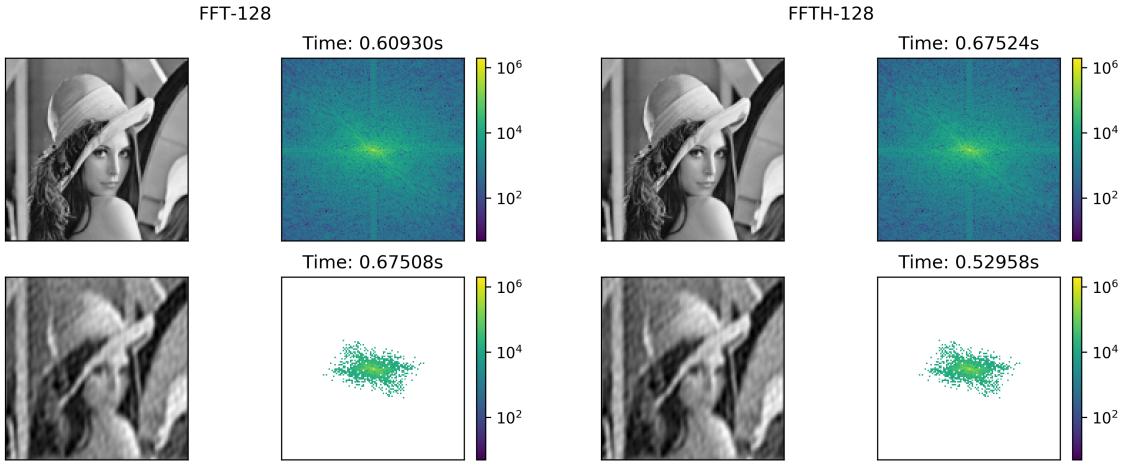
5.4 Additional Results

Here are the actual results of the compression. Note that we did not include the result of Parallel FFT since it is algorithmically the same as the ordinary FFT, only the way it is executed is different. The images on the left are the actual image, the right ones are the left images in spectrum space. The spectrum is reduced from the one above to below. This is then inverse transformed back to the compressed image on the bottom left. These are shown in Figure 14 and Figure 15. This set of compression results using different Fourier Transformation methods shows that changing the transformation method will not make any difference to the compression results.



(a) Naive DFT

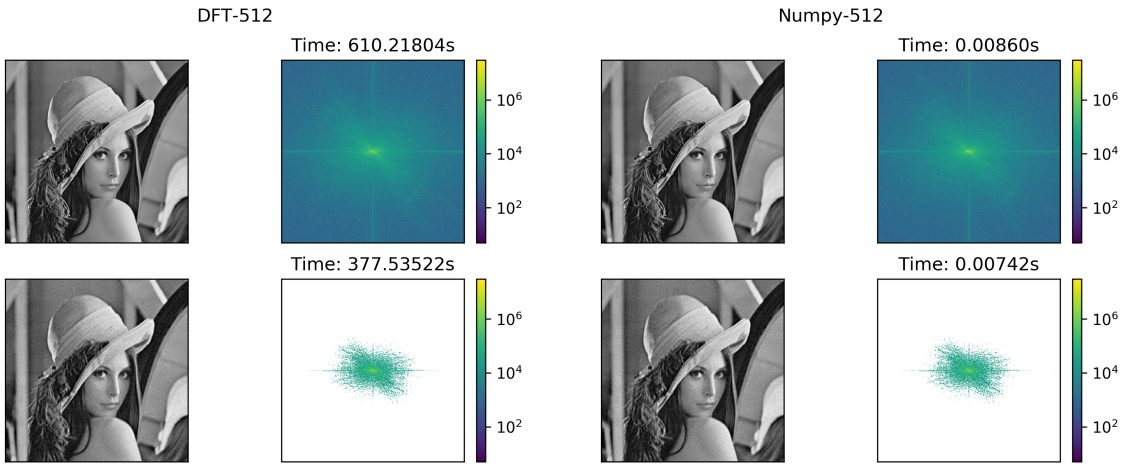
(b) NumPy FFT



(c) Cooley-Tukey FFT

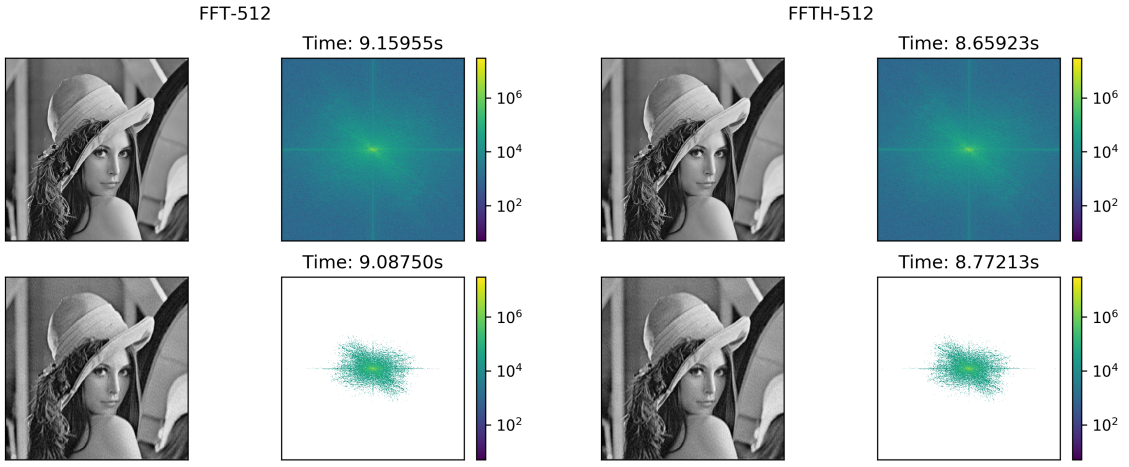
(d) Hybrid FFT

Figure 14: 128×128 Compression Results



(a) Naive DFT

(b) NumPy FFT



(c) Cooley-Tukey FFT

(d) Hybrid FFT

Figure 15: 512×512 Compression Results

6 References

Azad, Kalid. "An Interactive Guide To The Fourier Transform." Accessed 2019-12-4. 2017. https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/.

Chu, Eleanor, and Alan George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. Florida: CRC Press, November 1999. ISBN: 0849302706.

Cooley, James W., and John W. Tukey. "An Algorithm for the Machine Calculation of Complex Fourier Series." *Mathematics of Computation* 19, no. 90 (1965): 197–301.

David, Jean, Kassem Kalach, and Nicolas Tittley. "Hardware Complexity of Modular Multiplication and Exponentiation." *Computers, IEEE Transactions on* 56 (November 2007): 1308–1319. doi:10.1109/TC.2007.1084.

Erickson, Jeff. *Algorithms*. 2019.

Heideman, Michael, Don Johnson, and Charles Burrus. "Gauss and the history of the fast Fourier transform." *IEEE ASSP Magazine* 1, no. 4 (1984): 14–21.

ImageProcessingPlace.com. "Standard Test Images." Accessed 2019-12-5. http://www.imageprocessingplace.com/root_files_V3/image_databases.htm.

NumPy. "numpy/fft/_pocketfft.c." Accessed 2019-12-8. 2019. https://github.com/numpy/numpy/blob/master/numpy/fft/_pocketfft.c.

Stojanovic, Vladimir. "Course materials for 6.973 Communication System Design. MIT OpenCourseWare (http://ocw.mit.edu/), Massachusetts Institute of Technology." Accessed 2019-12-18. 2006. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-973-communication-system-design-spring-2006/lecture-notes/lecture_8.pdf.

Velez, William, and Marvi Teixeira. "Parallel DFT Implementation and Benchmarking in a Cluster Architecture," May 2018. doi:10.13140/RG.2.2.11740.69765.