Fast Fourier transforms (FFTs): a brief overview

Manas Upadhyay

Assistant Professor

LMS, CNRS, Ecole Polytechnique, France

Discrete Fourier Transforms (DFT)

- To understand the working principle of FFTs, one must be familiar with DFTs.
- Here it is assumed that the reader is already familiar with the DFT which transforms a sequence of N complex numbers $\{\boldsymbol{x_n}\} \coloneqq x_0, x_1, x_2, \dots, x_{N-1}$ into another sequence of N complex numbers $\{\boldsymbol{X_k}\} \coloneqq X_0, X_1, \dots, X_{N-1}$ as follows:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i\frac{2\pi kn}{N}}, k \in \{0,1, \dots, N-1\}$$

- The DFT is an invertible linear transformation whose inverse is knows as the inverse DFT and it is computed as

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{i\frac{2\pi kn}{N}}, n \in \{0,1, \dots, N-1\}$$

But what are the computational requirements?

- Prior to that we need to answer the question, how do we evaluate computational complexity?
  - By counting the number of real multiplications, divisions, additions and subtractions (equivalent to additions) are required
    - Note that by real we mean either fixed-point or floading point operations depending on the specific hardware
- Other operations such as loading from memory, storing in memory, loop counting, indexing, etc. are not counted (depends on implementation/architecture)
  - These are considered as **overheads** here

DFT as a vector operation

- Reconsider our sequence of sampled values x_n and their transformed values X_k given by

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i\frac{2\pi kn}{N}}, k \in \{0,1,\dots,N-1\}$$

- Let us denote

$$W = e^{-i\frac{2\pi}{N}}$$

- Then

$$X_k = \sum_{n=0}^{N-1} x_n W^{nk}, k \in \{0,1,\dots,N-1\}$$

Matrix interpretation of DFT

$$X_k = \sum_{n=0}^{N-1} x_n W^{nk}, k \in \{0,1,\dots,N-1\}$$

Complex numbers which can be computed once and stored!

- can be rewritten in matrix form as

$$\begin{bmatrix} X_0 \\ X_1 \\ \dots \\ X_{N-1} \end{bmatrix} = \underbrace{\begin{bmatrix} W^0 & W^0 & W^0 & \dots & W^0 \\ W^0 & W^1 & \boxed{W^2} & \dots & W^{N-1} \\ \dots & \dots & \dots & \dots & \dots \\ W^0 & W^{n-1} & W^{2(n-1)} & \dots & W^{(n-1)^2} \end{bmatrix}}_{\boldsymbol{N \times N} \textbf{ matrix}} \begin{bmatrix} x_0 \\ x_1 \\ \dots \\ x_{N-1} \end{bmatrix}$$

DFT evaluation – operation count

- Multiplication of a complex $N \times N$ matrix by a complex N −dimensional vector
  - No attempts to save operations gives
    - N^2 complex multiplications ("**x**"s)
    - $N(N-1)$ complex additions ("**+**"s)
- However, first row and first column are equal to, which
  - Saves $2N-1$ "**x**"s, resulting in $(N-1)^2$ complex "**x**"s & $N(N-1)$ "**+**"s
- Complex "**x**": **4** real "**x**"s and 2 real "**+**"s;

  Complex "**+**": 2 real "**+**"s
- Total: $4(N-1)^2$ real "**x**" & $4(N-0.5)(N-1)$ real "**+**"s;

DFT computational complexity

- Thus for the input sequence of length N, the number of arithmetic operations in direct computation of DFT is proportional to N^2.
- For $N = 1000$, about a million operations are needed!
- In 1960’s, such a number was considered prohibitive in most applications
- This led to the (re)discovery of FFT by Cooley and Tukey in 196; Gauss had already discovered the principle of FFT in 1806 (even before Fourier). Although the Cooley - Tukey algorithm was originally meant for military applications, their paper was nevertheless published in a public domain which went on to become one of the most important FFT algorithms for varied applications.

The Fast Fourier Transform (FFT)

- The FFT is a highly elegant and efficient algorithm which is still one of the most used algorithm in speech processing, communications, frequency estimation, etc. – one of the most highly developed area of digital signal processing
- There are many different types and variations of FFTs
  - Cooley – Tukey FFT algorithm
  - Prime Factor (Good Thomas) FFT algorithm
  - Bruun's FFT algorithm
  - Bluestein's FFT algorithm
  - Goertzel FFT algorithm
  - Etc…
- FFTs are smarter computational schemes for computing the DFT; they are not new transforms !
- Here we consider Cooley-Tukey's most basic radix-2 algorithm which requires N to be a power of 2, and restrict ourselves to the case of one-dimensional FFTs; the transition to multi-dimensional FFTs is fairly straightforward and will be vey briefly discussed towards the end of the presentation

FFT Derivation 1

- Lets take the basic DFT equation:

$$X_k = \sum_{n=0}^{N-1} x_n W^{nk}\,, k \in \{0,1,\dots,N-1\}$$

And split into two parts: one for even n and one for odd n

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi(2n)k}{N}} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-i\frac{2\pi(2n+1)k}{N}}$$

$$= \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi nk}{N/2}} + \boxed{e^{-i\frac{2\pi k}{N}}} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-i\frac{2\pi nk}{N/2}} = A_k + W^k B_k$$

Where

$$A_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi nk}{N/2}}$$

$$B_k = \sum_{n=0}^{N/2-1} x_{2n+1} e^{-i\frac{2\pi nk}{N/2}}$$

$$\mathrm{W}^{\mathrm{k}} = e^{-i\frac{2\pi}{N}}$$

FFT Derivation 2

$$A_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi nk}{N/2}}$$

$$B_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi nk}{N/2}}$$

- Note that A_k and B_k are themselves DFTs each of length $N/2$
  - A_k is the DFT of a sequence $\{x_{2n}\} = \{x_0, x_2, x_4, \dots, x_{N-4}, x_{N-2}\}$
  - B_k is the DFT of a sequence $\{x_{2n+1}\} = \{x_1, x_3, x_5, \dots, x_{N-3}, x_{N-1}\}$

- We know, however that the DFT is periodic in the frequency domain (in this case with a period $N/2$). This leads to further simplifications, as follows:

FFT Derivation 3

- We take the following equation again

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi(2n)k}{N}} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-i\frac{2\pi(2n+1)k}{N}}$$

And evaluate at frequencies $k + N/2$

$$X_{k+N/2} = \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi n(k+N/2)}{N/2}} + e^{-i\frac{2\pi(k+N/2)}{N}} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-i\frac{2\pi n(k+N/2)}{N/2}}$$

Now simplify the terms as follows

$$e^{-i\frac{2\pi n(k+N/2)}{N/2}} = e^{-i\frac{2\pi nk}{N/2}} \text{ and } e^{-i\frac{2\pi(k+N/2)}{N}} = -e^{-i\frac{2\pi k}{N}}$$

FFT Derivation 4

- Therefore,

$$X_{k+N/2} = \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi nk}{N/2}} - e^{-i\frac{2\pi k}{N}} \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} e^{-i\frac{2\pi nk}{\frac{N}{2}}}$$

$$= A_k + W^k B_k$$

With A_k, W^k and B_k defined as before

$$A_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi nk}{N/2}}$$

$$B_k = \sum_{n=0}^{N/2-1} x_{2n+1} e^{-i\frac{2\pi nk}{N/2}}$$

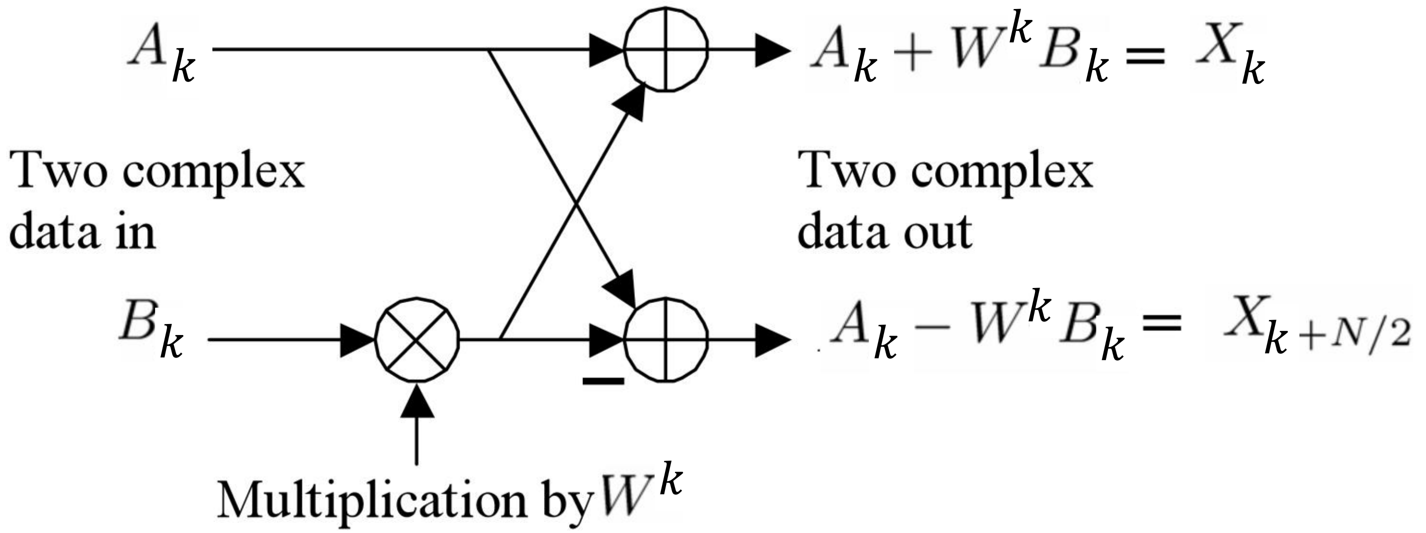
$$\mathrm{W}^{\mathrm{k}} = e^{-i\frac{2\pi}{N}}$$

FFT Derivation 5

- Now compare the two equations

$$X_k = A_k + W^k B_k, \qquad X_{k+N/2} = A_k - W^k B_k$$

This defines the FFT butterfly structure

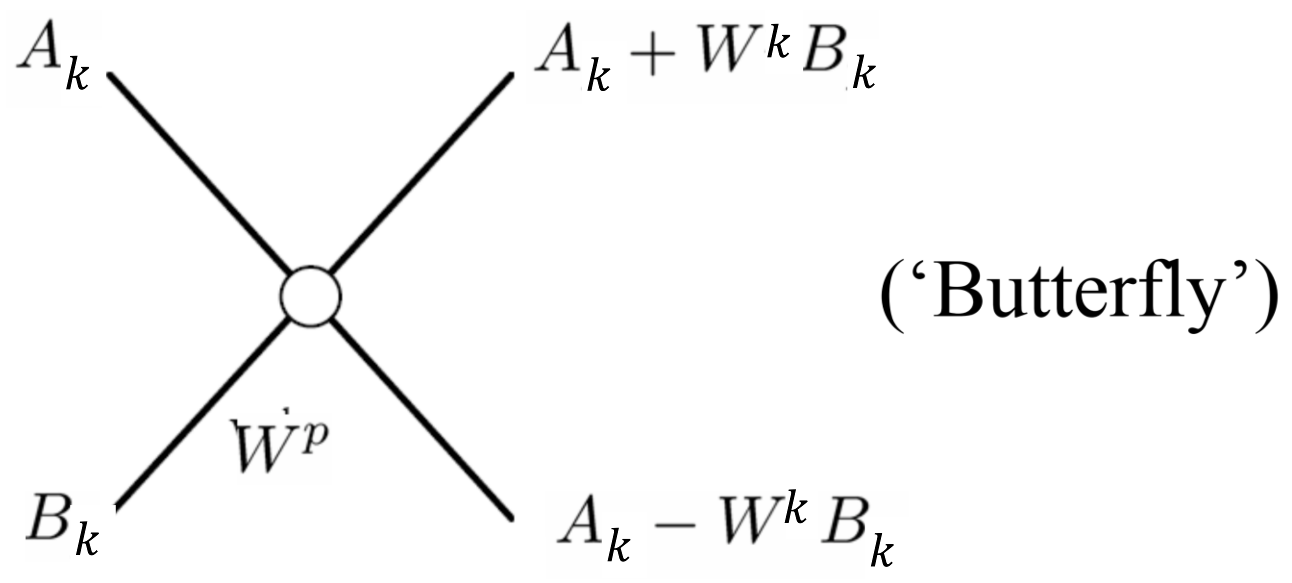


FFT Derivation 6

- More compactly

$$X_k = A_k + W^k B_k, \qquad X_{k+N/2} = A_k - W^k B_k$$

This defines the FFT butterfly structure



FFT Derivation redundancy

- Now, W^k can be assumed precomputed and stored, and we worry only about the following two terms

$$A_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi nk}{N/2}}, \qquad B_k = \sum_{n=0}^{N/2-1} x_{2n+1} e^{-i\frac{2\pi nk}{N/2}}$$

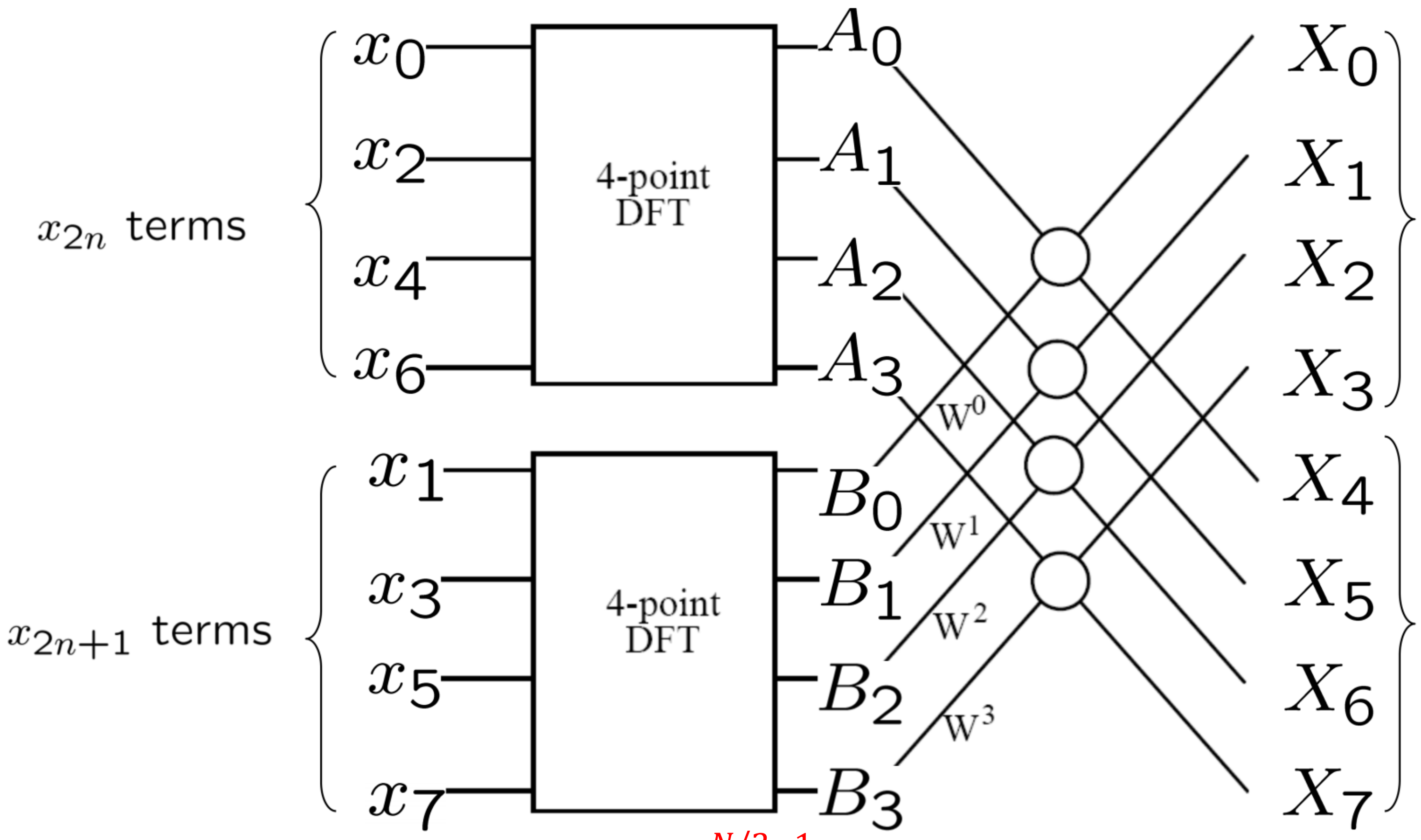
- The above terms need only be computed for $p = 0,2, \dots, \frac{N}{2} - 1$, since $X_{k+N/2}$ has been expressed in terms of these terms, hence we have uncovered a **redundancy** in the DFT computation.
- We can calculate A_k and B_k for $p = 0,1, \dots, \frac{N}{2} - 1$ and use them for the calculation of both X_k and $X_{k+N/2}$

FFT Derivation – Computational load

- The number of complex multiplications and additions are:
  - A_k and B_k each require $N/2$ complex multiplications and additions. The total for all $p = 0,1,\dots,N/2-1$ is then $2(N/2)^2$ multiplications and additions for the calculation of all A_k and B_k
  - Then, there are $N/2$ multiplications for the computation of $W^k B_k$ for all $k = 0,1,2,\dots,N/2-1$
  - Finally, $N/2 + N/2 = N$ additions for calculations of $A_k + W^k B_k$ and $A_k - W^k B_k$
- Thus the total number of complex additions and multiplications is approximately $N^2/2$ for a large N
- The computation is approximately halved in comparison to direct DFT evaluation

Flow chart for an $N = 8$ DFT

$$A_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-i\frac{2\pi nk}{N/2}}$$



$$X_k = A_k + W^k B_k$$

$$X_{k+\frac{N}{2}} = A_k - W^k B_k$$

$$B_k = \sum_{n=0}^{N/2-1} x_{2n+1} e^{-i\frac{2\pi nk}{N/2}}$$

Further decomposition

- We have $X_k = A_k + W^k B_k$ and $X_{k+N/2} = A_k - W^k B_k$; where $W^k = e^{-i\frac{2\pi}{N}}$
- Now,

  A_0, A_1, A_2, A_3 are N/2 point DFTs

  Using the redundancy just discovered $A_k = \alpha_k + W_{N/2}^k \beta_k$ and $A_{k+N/2} = \alpha_k - W_{N/2}^k \beta_k$

  B_0, B_1, B_2, B_3 are N/2 point DFTs

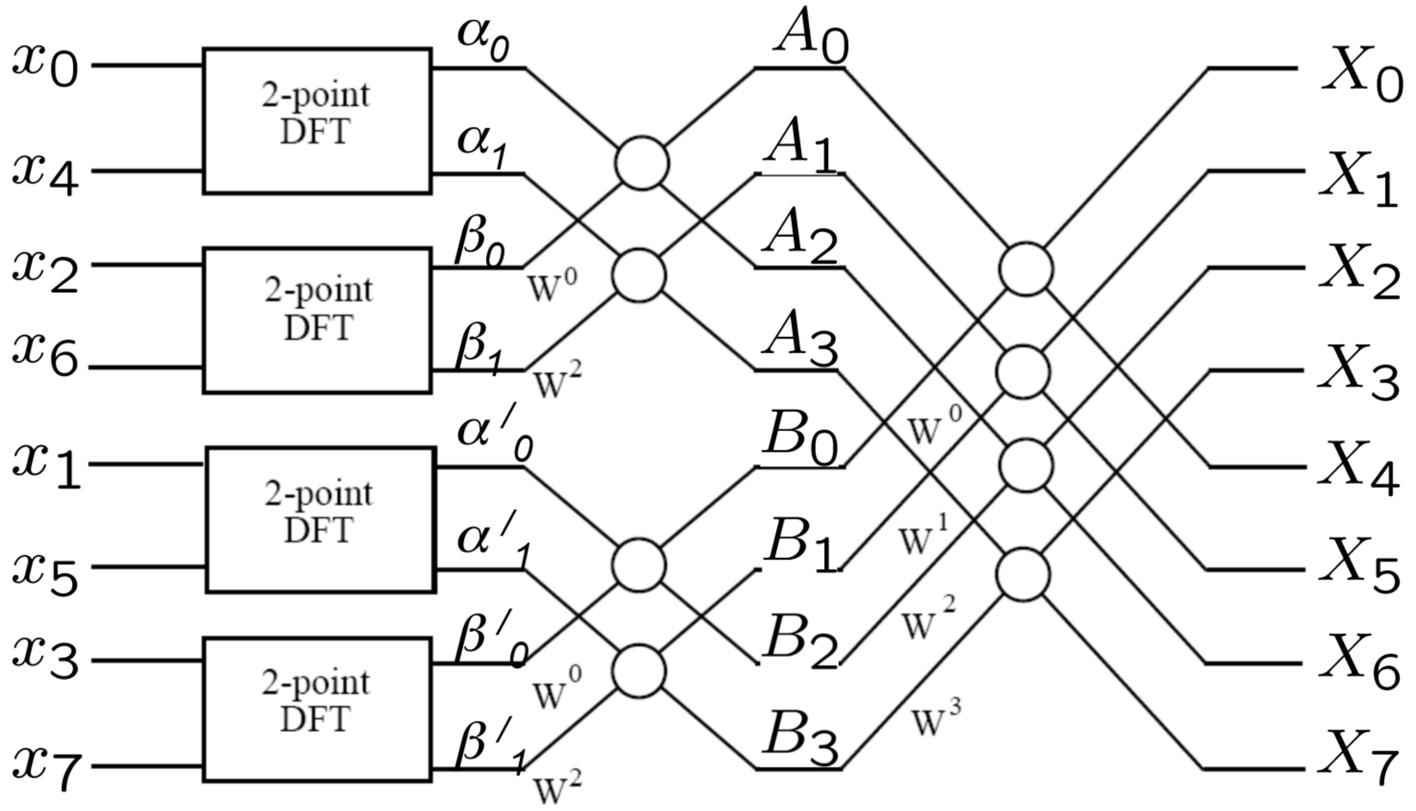
  Using the redundancy just discovered $B_k = \alpha'_k + W_{N/2}^k \beta'_k$ and $B_{k+N/2} = \alpha'_k - W_{N/2}^k \beta'_k$

  Finally,

$$W_{N/2}^k = e^{-i\frac{2\pi k}{N/2}} = e^{-i\frac{2\pi 2k}{N}} = W_N^{2k}$$

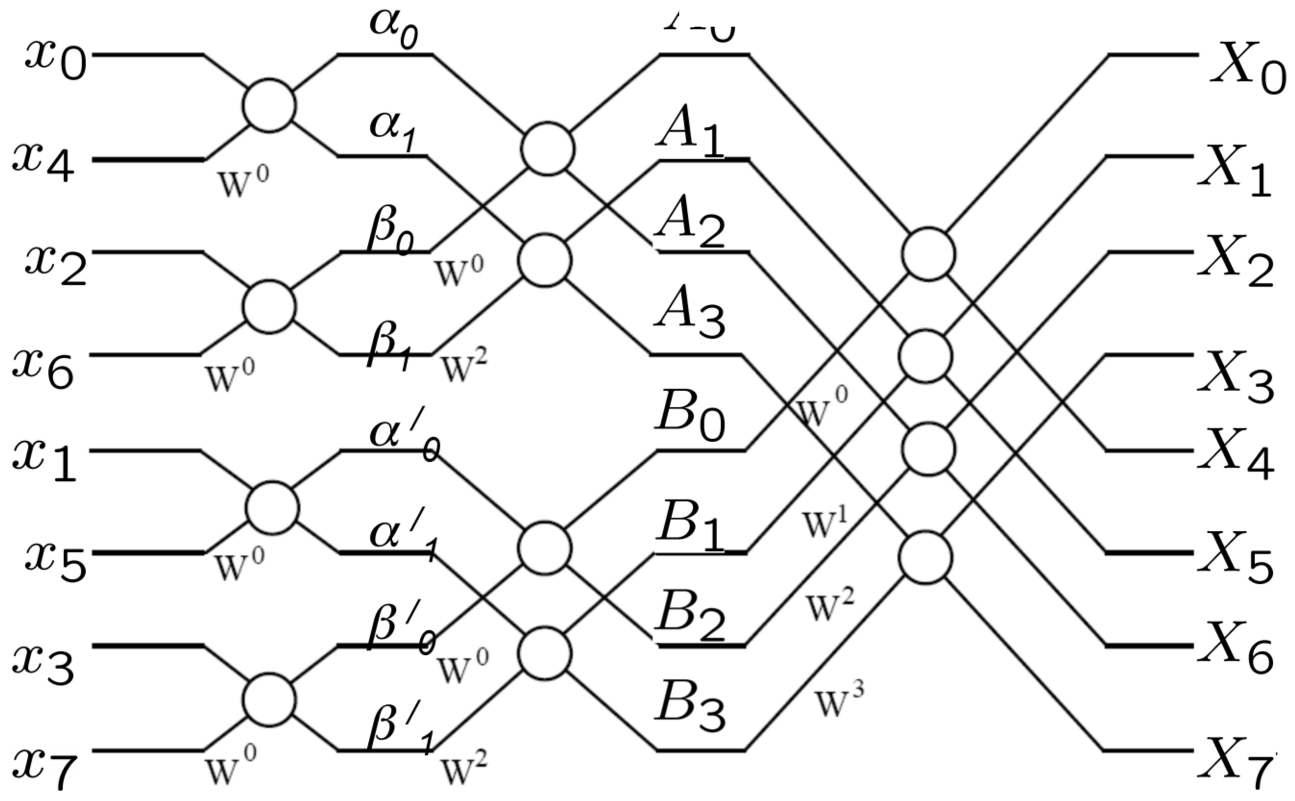
Flow chart for an $N = 8$ DFT - Decomposition

- Assuming that $N/2$ is even, the same process can be carried on each of the $N/2$ point DFTs to further reduce the computation. The diagram for incorporating this extra stage of decomposition into the computation of $N = 8$ point DFT is shown below



Flow chart for an $N = 8$ DFT – Decomposition 2

- If $N = 2^M$ then a further decomposition is possible by repeating the process M times to reduce the computation to that of evaluating N single point DFTs.



Bit reversal

- Examination of the previous chart shows that it is necessary to shuffle the order of the input data. This data shuffle is usually termed *bit-reversal* for reasons that are clear if the indices of the shuffled data are written in binary

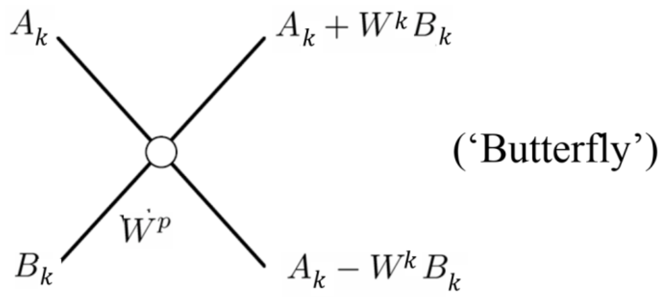
Binary	Bit Reverse	Decimal
000	000	0
001	100	4
010	010	2
011	110	6
100	001	1
101	101	5
110	011	3
111	111	7

FFT derivation summary

- The FFT exploits the redundancies in the calculation of the basic DFT
- A recursive algorithm is derived that repeatedly rearranges the problem into two simpler problems of half the size
- Hence the basic algorithm operates on signals of length that is a power of 2, i.e.

$N = 2^M$ (for some integer M)

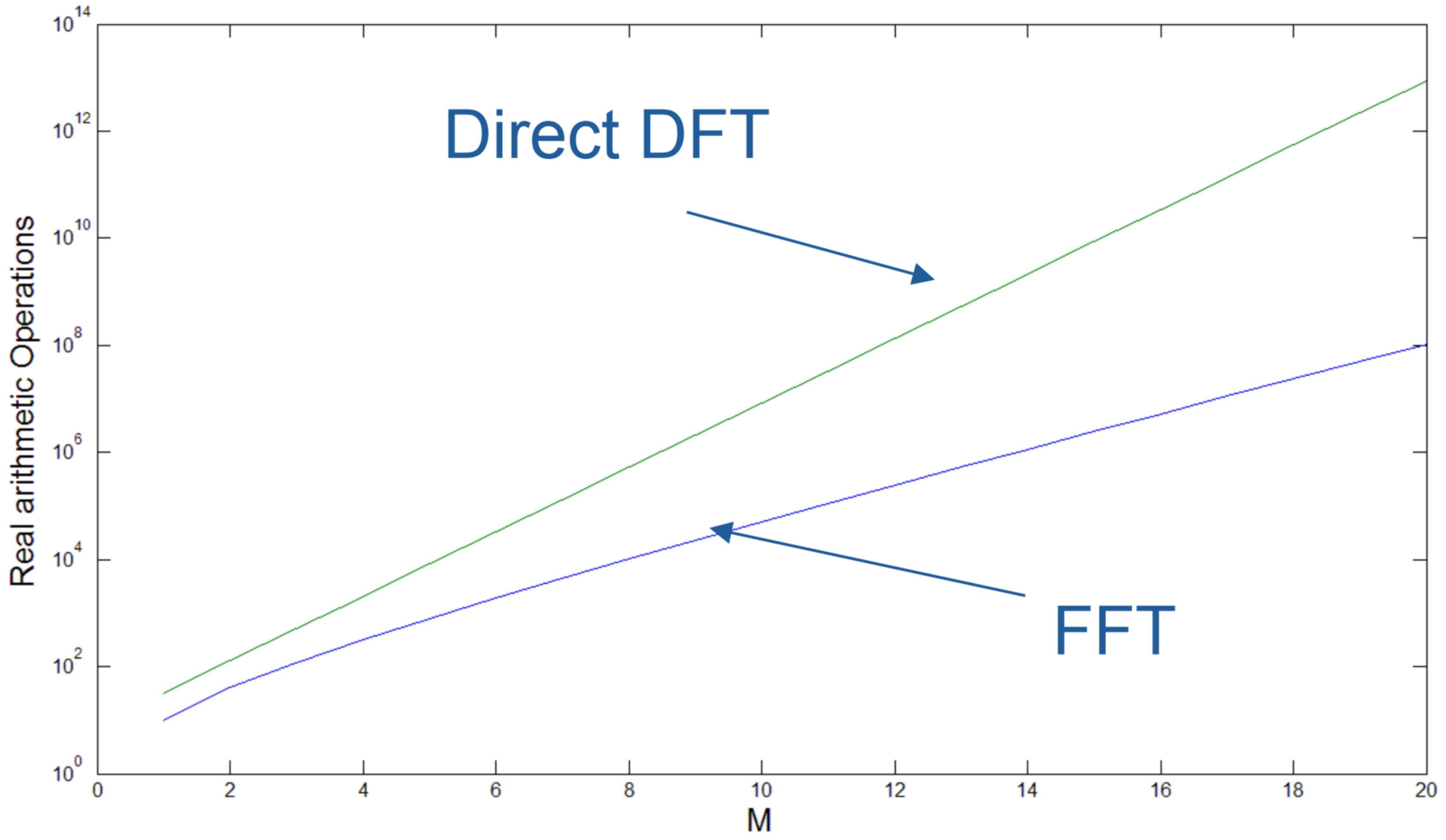
At the bottom of the tree we have the classic FFT butterfly structure



Computational load of the full FFT algorithm

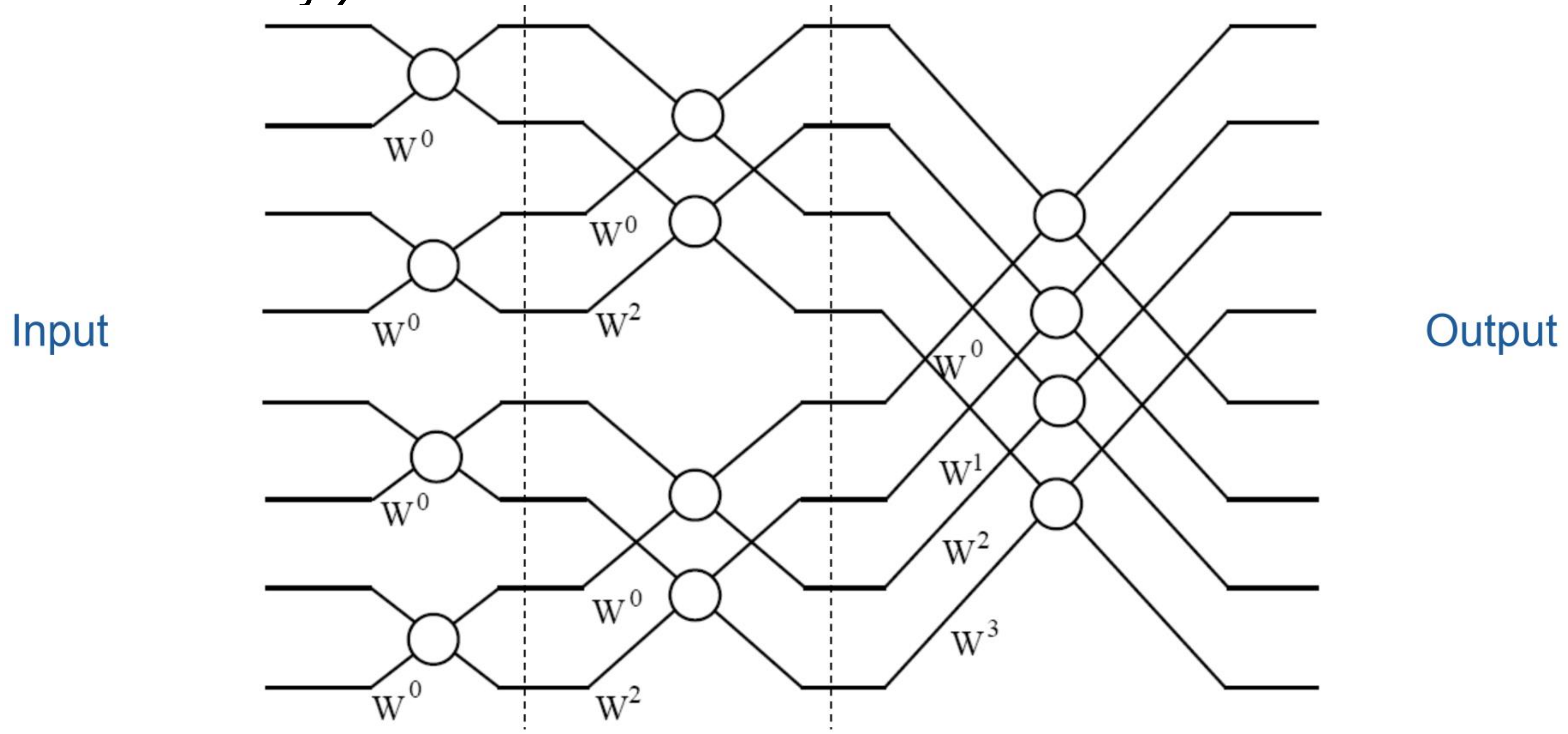
- The type of FFT we have considered, where $N = 2^M$, is called a **radix-2** FFT.
- It has M = $\log_2 N$ stages, each using $N/2$ butterflies
- Complex multiplication requires 4 real multiplications and 2 real additions
- Complex addition/subtraction requires 2 real additions
- Thus, butterfly requires 10 real operations.
- Hence the radix-2 *N*-point FFT requires $10(N/2)\log_2 N$ real operations compared to about $8N^2$ real operations for the DFT
- This is a huge speed-up in typical applications, where N ranges from $2^6 – 2^{20}$ (see comparison with direct DFT in next page)

Computational load of the full FFT algorithm



Further advantage

- The FFT algorithm has a further significant advantage over direct evaluation of the DFT expression in that the computation can be performed *in-place*. This is best illustrated in the final flow chart where it can be seen that after two data values have been processed by the *butterfly* structure, those data are not required again the computation and the memory (in the RAM or cache) associated with them can either be freed or replaced with the values at the output of the *butterfly* structure.



Inverse FFT

- The IDFT is different from the DFT:
  - it uses positive power of instead of negative ones
  - There is an additional division of each output value by *N*
- Any FFT algorithm can be modified to perform the IDFT by
  - using positive powers instead of negatives
  - Multiplying each component of the output by *1 / N*
    Hence the algorithm is the same but computational load
    increases due to *N* extra multiplications

Multi-dimensional FFT – origin from DFT

- A multi-dimensional DFT is defined as

$$X_{\boldsymbol{k}} = \sum_{\boldsymbol{n}=0}^{\boldsymbol{N}-1} x_{\boldsymbol{n}} e^{-i\frac{2\pi \boldsymbol{k}\cdot\boldsymbol{n}}{\boldsymbol{N}}}$$

- It transforms an array $x_{\boldsymbol{n}}$ with a d −dimensional vector on indices $\boldsymbol{n} = (n_1, n_2, \dots, n_d)$ by a set of d nested summations (over $n_j = 0 \dots N_j - 1$ for each j) where the division $\boldsymbol{n}/\boldsymbol{N}$, defined as $\boldsymbol{n}/\boldsymbol{N} = (n_1/N_1, \dots, n_d/N_d)$ is performed element-wise.
- Or Equivalently, it is the composition of a sequence of d sets of one-dimensional DFTs, performed along one dimension at a time (in any order)
  - This provides one of simplest and most common DFT algorithm known as the **row-column** algorithm

Multi-dimensional FFT – row/column algorithm

- One simply perform a sequence of d one-dimensional FFTs using any of the FFT algorithms
  - First transform along the n_1 dimension
  - Then along the n_2 dimension, and so on…
- This method also has the usual $\mathrm{O}(N\log N)$ computational time where $N = N_1 \cdot N_2 \cdot \ldots \cdot N_d$ is the total number of data point transformed.
  - Specifically, there are N/N_1 transform of size N_1, and so on, so the complexity of the sequence of FFTs is:

$$\frac{N}{N_1}\mathrm{O}(N_1\log N_1) + \cdots + \frac{N}{N_d}\mathrm{O}(N_d\log N_d) = \mathrm{O}(N[\log N_1 + \cdots + \log N_d]) = \mathrm{O}(N\log N)$$

- In 2-dimensions, $x_{\boldsymbol{n}}$ can be viewed as an $n_1 \times n_2$ matrix, and this algorithm corresponds to first performing the FFT of all the rows (resp. columns), grouping the resulting transform rows (resp. columns) together as another $n_1 \times n_2$ matrix, and then performing the FFT on each of the columns (resp. rows) of this second matrix, and similarly grouping the results into the final result matrix. Hence, the name **row/column**
- In higher dimensions, it can be advantageous for the cache (as in cache memory in computers) locality to group the dimensions recursively. For example, a three-dimensional FFT might first perform two-dimensional FFTs of each planar “slice” for each fixed n_1,and then perform the one-dimensional FFTs along the n_1 direction.

Where to find FFT codes?

- Unless you are ready to spend a whole lot of time coding, its better to download a pre-programmed FFT code that is benchmarked, freely available, well documented, upgrades available, and user support too.
- Luckily, there is one such code that is widely available, well documented and quite easy to implement. This is the FFTW package found at (www.fftw.org). It was designed by Matteo Frigo and Steven G. Johnson at MIT and they made it available open source.
    - FFTW = Fast Fourier Transform in the West

References

- Many of the slides on FFT slides are adapted from the presentation of Dr. Elena Punskaya from Cambridge university, UK which can be found on https://www.slideshare.net/op205/fast-fourier-transform-presentation
- Another major source has been Wikipedia articles.
- The FFTW website www.fftw.org contains documentation that provides a lot more details on the numerical implementations of the algorithm, as well as, how to use it.