1 研究意义

快速傅里叶变换（FFT）是一种高效计算离散傅里叶变换（DFT）的算法，它在多个领域中都有广泛应用，并且推动了许多科技进步，被称为人类史上最伟大的算法之一。

FFT在现实的生活中无处不在：在数字信号处理（DSP）中，它用于快速分析信号的频谱成分，在音频处理、图像处理、通信系统等方面具有广泛应用；现代通信系统依赖于调制和解调技术，在正交频分复用（OFDM）等调制技术中，FFT是核心算法。它能够快速执行频谱分析和信号调制，提升数据传输速率和通信质量。

以往人们对FFT的探索更多基于算法层面：使用分而治之的思想降低运算复杂度，设计算法减少复数乘法次数..... 在算法复杂度上，FFT完成了从$O(n^2)$到$O(n\log n)$的跨越，但是要在$O(n\log n)$的理论上限基础上继续加速，难度大大增加。

随着现代计算机体系结构和并行计算技术的进步，高性能计算发展如火如荼，FFT也面临着新一步的革新。现代计算机体系结构和并行计算技术的进步，使得FFT算法得以在各种硬件平台上高效实现，从而进一步扩展了其应用范围。现代计算机科学家的努力方向是在不同的平台（通常是GPU、CPU）中部署FFT算法，从而更加高效地利用算力资源。如何尽可能减少运算次数、如何在硬件加速的平台上高效计算、如何针对不同的指令集优化算法，称为当下FFT算法的难题。

2 离散傅里叶变换简介

2.1 DFT定义

对于N点的序列$\{x[n]\}_{0\le n<N}$，假设其只在$0\sim N-1$有值，那么它的离散傅里叶变换（DFT）定义为：

$$X[k]=\sum_{n=0}^{N-1}e^{-j\frac{2\pi}{N}nk}x[n]\quad k=0,1,\dots,N-1$$

如果将$e^{-j\frac{2\pi}{N}}$记作w_N，那么上式可以简写为：

$$X[k]=\sum_{n=0}^{N-1}x[n]w_N^{nk},\quad 0\le k\le n-1.$$

同时，我们给出离散傅里叶逆变换（inverse DFT）的公式：

$$x[n]=\frac{1}{N}\sum_{k=0}^{N-1}y[k]w_N^{-nk},\quad 0\le j\le n-1,$$

2.2 DFT：复杂度分析

在计算N点DFT的过程中，对于每一个频域点$X[k],\ \ k=0,\cdots N-1$，需要经过N次的复数乘法、N-1次的复数加法。

因此要得到完整的频谱，总共的计算时间复杂度为$O(n^2+n(n-1))=O(n^2)$。

2.3 矩阵表示DFT

许多参考书将矩阵乘法引入DFT，如下所示。将信号表示为向量：$X[k]=(y_0,y_1,\cdots,y_{n-1})$，$x[n]=(x_1,x_2,\cdots,x_{n-1})$。则DFT可以写作矩阵乘法的形式：

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

3 DIT Radix-2 Cooley-Tukey 算法

2指数库里-图基算法是最简单的一个FFT算法。利用分治思想，将计算DFT的复杂度降低为$O(n\log n)$。下面推导其算法。

3.1 单位复数根

在正式介绍FFT之前，我们先了解n次单位复数根。这是满足$w^n=1$的复数w。n次单位复数根恰好有n个，对于$k=0,1,\cdots,n-1$，他们分别是$e^{2\pi jk/n}$。这里j是复数单位，$j=\sqrt{-1}$

为了在书写上化繁为简，我们一般用符号w_N来表示$e^{-j\frac{2\pi}{N}}$。称为N次单位根。

$$w_N = e^{-j\frac{2\pi}{N}}$$

可以证明，N 次单位根有如下性质：

1. 周期性：w_N的指数具有周期N，即：$w_N^{k+N}=w_N^k$。

   证明：

   $$w_N^{k+N} = e^{-j\frac{2\pi}{N}(k+N)} = e^{-j2\frac{2\pi k}{N}}$$

2. 对称性：$w_N^{k+N/2} = -w_N^k$

   证明：

   $$w_N^{k+N/2} = e^{-j\frac{2\pi}{N}(k+N/2)} = e^{-j\frac{2\pi k}{N}-j\pi} = -e^{-j\frac{2\pi k}{N}}$$

3. 可消去性：若m是N的约数，则$w_N^{m*kn}=w_{\frac{N}{m}}^{kn}$

   证明：

   $$w_N^{m*kn} = e^{-j\frac{2\pi}{N}*mkn} = e^{-j\frac{2\pi}{N/m}*kn}$$

我们会利用以上性质对DFT进行变形。引入单位根后，DFT就写为：

$$X[k] = \sum_{n=0}^{N-1} x[n] * w_N^{kn}, \quad k=0,\ldots,N-1$$

3.2 分治算法

接下来就是库利-图基算法的核心：分治。我们将求和部分分为**奇下标和偶下标**两部分：

$$\begin{aligned} X[k] &= \sum_{t=0}^{N/2-1} x[2t] w_N^{k*2t} + \sum_{t=0}^{N/2-1} x[2t+1] w_N^{k*(2t+1)} \\ &= \sum_{t=0}^{N/2-1} x[2t] w_{N/2}^{kt} + w_N^k \sum_{t=0}^{N/2-1} x[2t+1] w_{N/2}^{kt} \\ &= F_{even}[k] + w_N^k F_{odd}[k] \end{aligned}$$

为了方便和原公式对照，我们可以看一眼更加Verbose的公式：

$$X[k] = \underbrace{\sum_{t=0}^{N/2-1} x[2t]e^{-\frac{2\pi j}{N/2}tk}}_{x[n]\text{偶下标序列的DFT}} + e^{-\frac{2\pi j}{N}k} \underbrace{\sum_{t=0}^{N/2-1} x[2t+1]e^{-\frac{2\pi j}{N/2}tk}}_{x[n]\text{奇下标序列的DFT}} = F_{even}[k] + e^{-\frac{2\pi j}{N}k}F_{odd}[k]$$

$$\text{for } k = 0, \ldots, \frac{N}{2} - 1.$$

正如上式所写，$F_{even}[k], F_{odd}[k]$正好分别是$x[n]$奇数下标、偶数下标序列的$N/2$点DFT。具体来说：

$$F_{even}[k] = \sum_{t=0}^{N/2-1} x[2t]e^{-\frac{2\pi j}{N/2}tk}$$

$$F_{odd}[k] = \sum_{t=0}^{N/2-1} x[2t+1]e^{-\frac{2\pi j}{N/2}tk}$$

如果将原序列奇下标、偶下标部分分别记作$x_1[n]$，　$x_2[n]$：

$$\begin{cases} x_1[k] = x[2k] \\ x_2[k] = x[2k+1] \end{cases} \qquad \text{for } k = 0, \ldots, \frac{N}{2} - 1$$

那么$F_{even}[k], F_{odd}[k]$就变为DFT的标准形式：

$$F_{even}[k] = \sum_{t=0}^{N/2-1} x_1[t]e^{-\frac{2\pi j}{N/2}tk}$$

$$F_{odd}[k] = \sum_{t=0}^{N/2-1} x_2[t]e^{-\frac{2\pi j}{N/2}tk}$$

因此，我们可以用同样的方法计算$x_1[n], x_2[n]$离散傅里叶变换。这是递归的思想，在不断的递归、合并中，我们就得到了原序列的傅里叶变换。

最后，我们可以利用单位根的**周期性、对称性**来减少运算量。我们可以很快地推导出$k > N/2$时$X[k]$的计算公式

$$X[k + N/2] = F_{even}[k] - w_N^k F_{odd}[k]$$

所以我们可以将DFT的结果写成如下形式：

$$X[k] = F_{even}[k] + w_N^k F_{odd}[k]$$

$$X[k + \frac{N}{2}] = F_{even}[k] - w_N^k F_{odd}[k]$$

$$\text{for } k = 0, \ldots, \frac{N}{2} - 1.$$

综上，我们将一个规模为N的问题分为规模为N/2的两个子问题，可以利用分治递归求解。伪代码如下：

```
RECURSIVE-FFT(x) # x 是一个长度为N的向量，且n是2的指数次幂
n = a.length
if n==1          # 当序列长度为1时，返回自身
    return a
w_n = exp(2 * pi * 1j / n)
w = 1
x_0 = (x[0], x[2], ..., x[n-2]) # 偶数下标向量
x_1 = (x[1], x[3], ..., x[n-1]) # 奇数下标向量
y_0 = RECURSIVE-FFT(y_0)
y_1 = RECURSIVE-FFT(y_1)
for  k = 0 to n / 2 - 1
    y[k] = y_0[k] + w * y_1[k]
    y[k + n / 2] = y_0[k] - w *y_1[k]
    w = w * w_n
```

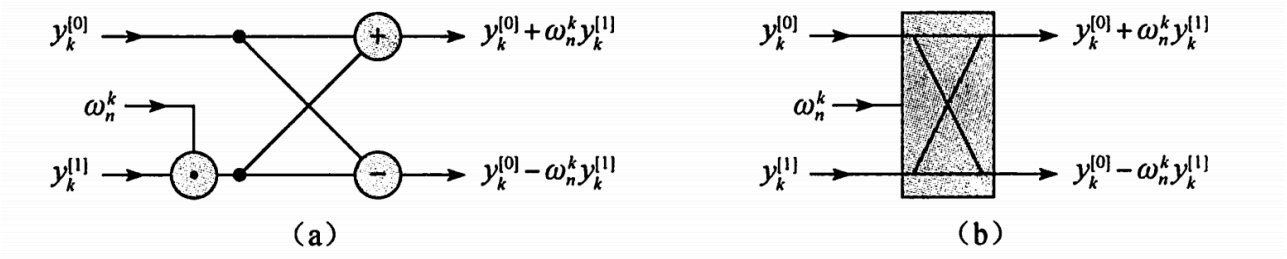
```
15  end
16  return y
```

在第12-14行，我们对w进行了累乘，每次乘上$w_n = e^{-j2\pi/n}$。我们将其称为**旋转因子**。

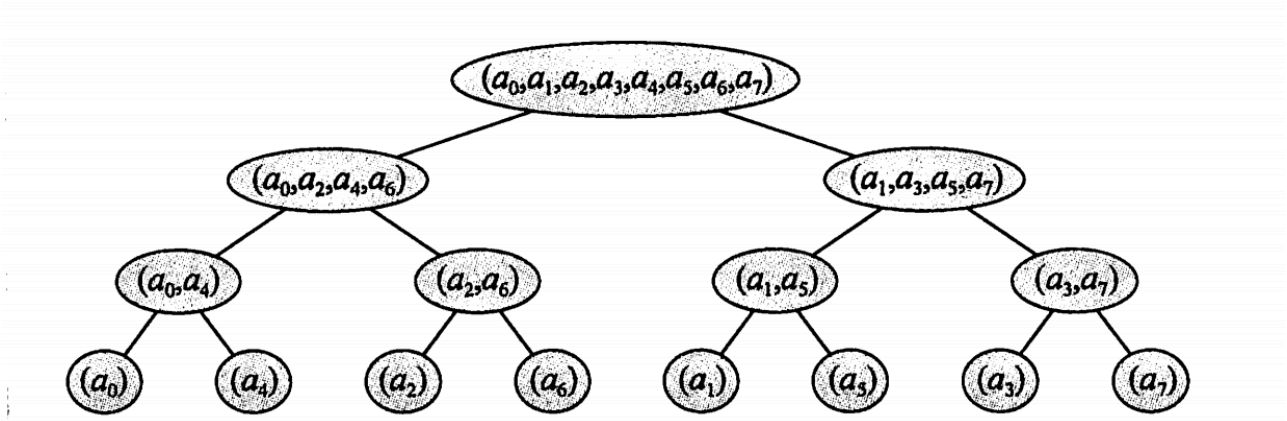
3.3 递推：蝶形网络

使用递归的方式计算显然消耗了很多不必要的堆栈资源。在大多数工程场景下，甚至是不允许出现递归代码的。因此我们用**迭代**替代递归操作。那么如何迭代计算FFT？在递推实现的2-radix C-T的算法中，我们引入**蝶形网络**和**位翻转**方法。

首先我们注意到，在伪代码第12-14行，重复利用了 `w*y_1[k]` 这一公共项。可以将所得的乘积存入一个临时变量 `t` 中，便于重复利用，从而减少了一次复数乘法。下面的计算流程图表示了这一系列操作，我们称之为**蝴蝶操作**



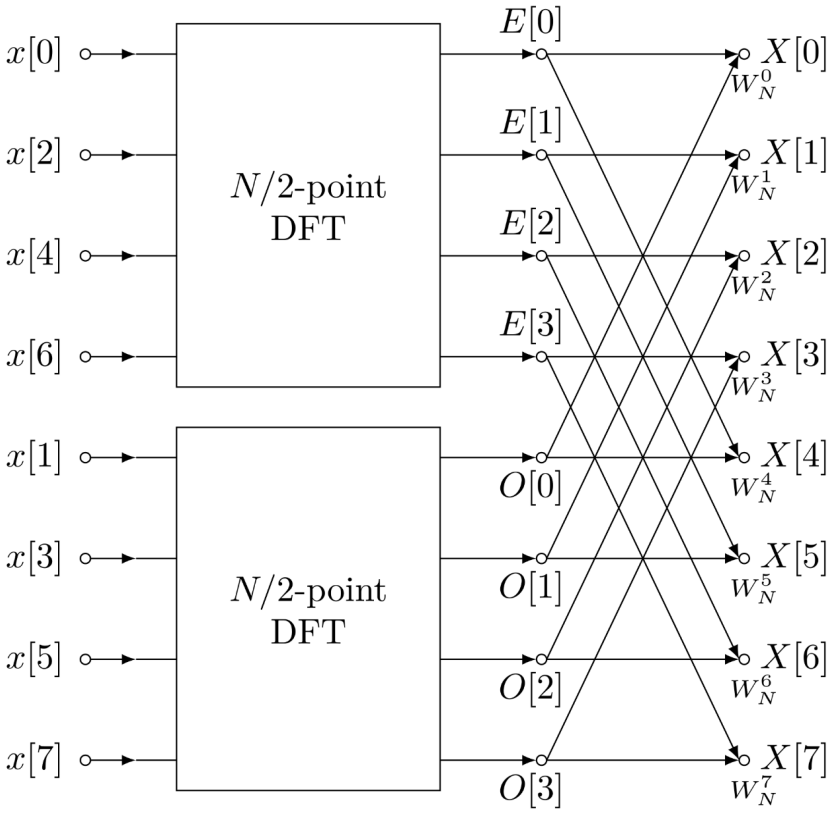
递归算法实现了自顶向下完成计算，其递归调用树如下所示。为了实现递推，我们希望自底向上推导。



加入我们能够将原始序列按照上图叶子节点的顺序排列，并引入一个变量 s 代表计算树的层次，取值范围为从 1（最底层） 到 $\log_2 n$ （最顶层）。在每一层，我们要对两个具有 2^{s-1} 个元素的 DFT 进行组合，以产生最后结果。伪代码应当如下所示：

```
for s = 1 to log2(n)
    for k = 0 to n - 1 by 2^s   # 每次跨越2^s步长
        combine the two 2^{s-1} element DFTs
        from A[k ... k+2^{s-1}-1] and A[k+2^{s-1} ... k+2^s - 1]
        into one 2^s-element DFT in A[k ... k + 2^s - 1]
```

最内层的实现细节与回溯实现的内层循环一致，对于每一个(s,k)组合，需要进行2^s组蝴蝶变换。画出计算流程图如下所示，我们将其称为**蝶形网络**。



- 最左侧是原序列，分为偶数下标$x[0], x[2], x[4], x[6]$和奇数下标$x[1], x[3], x[5], x[7]$两组。对应$x_1[n], x_2[n]$。
- 它们经过$N/2$个点的DFT，得到两组结果：$E[0] \cdots E[3]$，　$O[0] \cdots O[3]$。分别对应$F_{even}[k]$，　$F_{odd}[k]$。

继续将$N/2$点的DFT展开，我们将得到如下数据流向图：

xxxxx

3.4　递推：比特翻转

在递推过程中，我们希望序列按照一定的顺序重排，即输入数据的顺序需要被打乱。这种乱序其实有规律，我们把顺序的序号用二进制数列在下表中的左边，把乱序的序号用二进制数列在下表中的右边。

Normal order of index n	Binary bits Of index n	Reversed bits of index n	Bit-reversed of order index n
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

表1. 基2，8点FFT序列的比特翻转

从表中我们可以看出，乱序序号的二进制码可由顺序序号的二进制码**镜像反转**得到(例如$001 \rightarrow 100$)，这种规律被叫做比特反转。如果我们将原始序列按照上述规则重排，就能按照计算图的规律来编写递推代码，从而实现递推计算FFT。

3.5 补零

假如一个时域信号长度不是2^k，如何计算FFT？可以将序列扩展到大于或等于序列长度的最小的 2 的幂，并用零填充缺失部分，再进行标准的FFT算法处理。

3.6 速度瓶颈

计算机访问缓存和内存的原理涉及到多级缓存体系结构以及数据的存储和读取方式。现代处理器通常采用三级缓存（L1、L2、L3）和主内存（RAM）来提高数据访问速度。CPU需要的数据在缓存中，访问速度快；CPU需要的数据不在缓存中，需要从较慢的内存中加载数据到缓存，然后再访问。因此，一个优秀的程序应当在内存访问上做优化。

我们对C-T算法进行分析：当数据量N很大时，由于比特翻转访问的内存空间的不连续且跨度很大，而缓存容量有限，计算机需要反复从内存中读取数据，造成了大量的时间浪费。

同时，在内层循环中，计算机需要同时访问内存跨度为2^k的两点数据。大多数情况下（k>5左右），这都会要求计算机重新从内存中加载数据到缓存，再进行计算。因而，这种方法的空间效率非常低。

在下面的章节中，会尝试从内存访问的角度提升计算速度。事实上，后人对FFT在算法上有许多创新，但对于速度的提升十分有限。速度的提升更多是基于**底层优化、并行计算**。

3.7 Code Implementation

在此仅展现关键C++代码。完整代码详见附件。

```
1
```

在位翻转操作中，我们通过预处理函数`get_reversed`计算得到了每一个`i`对应的翻转下表`reversed[i]`。

4 DIF C-T 算法

快速傅里叶变换有两种常用的序列拆解方式，一种即上面所提及的时域抽取算法（decimation in time, DIT）。相应还有一个对偶算法：频域抽取（decimation in frequency, DIF）的FFT。DIT将N点序列分为奇下标、偶下标两部分；DIF则将序列分为前N/2个点和后N/2个点两部分。

$$
\begin{aligned}
X\left[\mathrm{k}\right] &= \sum_{n=0}^{N/2-1}\left\{x\left[n\right]\omega_N^{nk} + x(n+N/2)\omega_N^{(n+N/2)k}\right\} \\
&= \sum_{n=0}^{N/2-1}\left\{x\left[n\right] + x(n+N/2)\omega_N^{(N/2)k}\right\}\omega_N^{nk}, \quad 0 \le k \le N-1.
\end{aligned}
$$

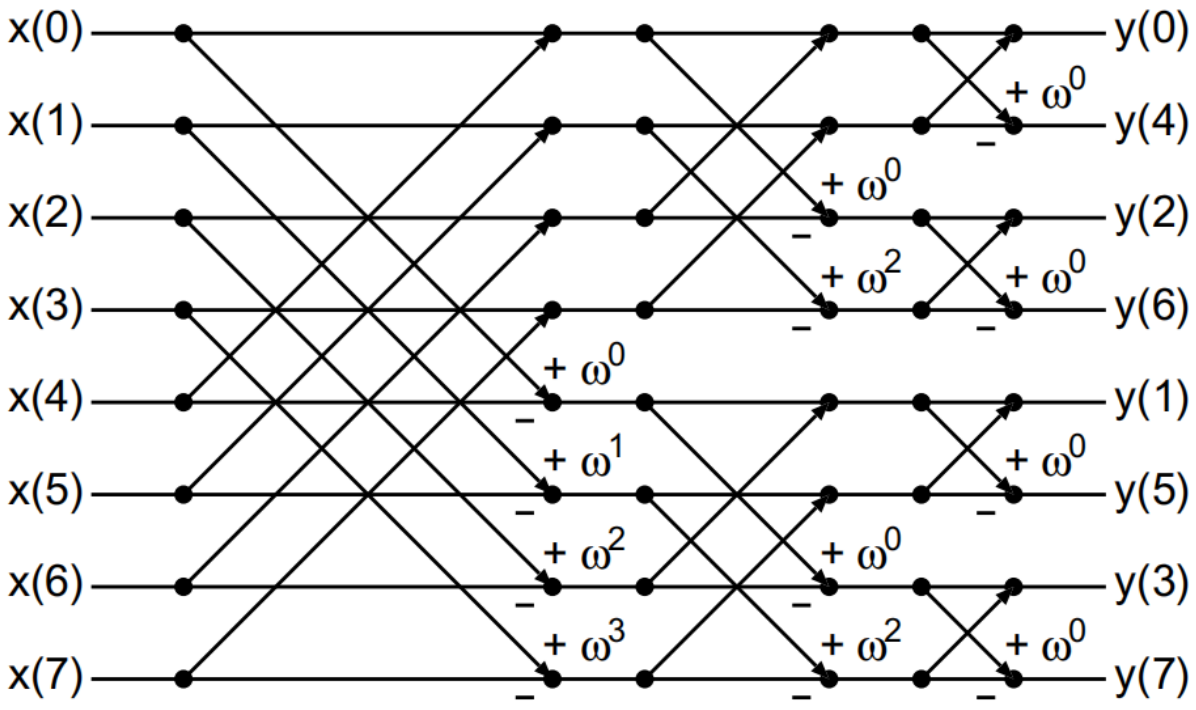
上式中

$$
w_N^{(N/2)k} = e^{-j\pi k} = \begin{cases} 1 & k\text{是偶数} \\ -1 & k\text{是奇数} \end{cases}
$$

因此，n点的DFT可以分为频域奇数下表、频域偶数下标两部分：

$$
\begin{aligned}
X[2k] &= \sum_{n=0}^{N/2-1}\{x[n] + x[n+N/2]\}\omega_N^{n*2k} \\
&+ \sum_{n=0}^{N/2-1}\{x[n] + x[n+N/2]\}\omega_{N/2}^{nk}, \quad 0 \le k \le n/2-1, \\
X[2k+1] &= \sum_{n=0}^{N/2-1}\{x[n] - x[n+N/2]\}\omega_N^{n*(2k+1)} \\
&+ \sum_{n=0}^{N/2-1}\{\{x[n] - x[n+N/2]\}\omega_N^{n}\}\omega_{N/2}^{nk}, \quad 0 \le k \le n/2-1
\end{aligned}
$$

同样的，可将上式看做N/2点DFT的组合。一个8点DFT的数据流向图如下：



和时域抽取DIT不同，DIF方法的蝴蝶操作单元如下：

$$X = x + y$$
$$Y = (x - y)w^j$$

在DIF 的库利-图基算法中，输出数据的顺序被打乱了，因此需要在最后对输出数据进行比特翻转操作。

5 Radix-4 C-T 算法

5.1 算法概述

如果点数N是4的整数次方，$N = 4^k$，那么采用基4FFT算法可以进一步减少运算量。对于时域采样 基-4 C-T 算法，将DFT按照如下方式分为4组 $N/4$ 点DFT：

$$\begin{aligned} X[k] &= \sum_{n=0}^{N-1} e^{-j\frac{2\pi}{N}nk} x[n] \\ &= \sum_{m=0}^{N/4-1} x[4m] w_{N/4}^{mk} + W_N^k \sum_{m=0}^{N/4-1} x[4m+1] w_{N/4}^{mk} \\ &\quad + w_N^{2k} \sum_{m=0}^{N/4-1} x[4m+2] w_{N/4}^{mk} + w_N^{3k} \sum_{m=0}^{N/4-1} x[4m+3] w_{N/4}^{mk} \\ &= F_0[k] + w_N^k F_1[k] + w_N^{2k} F_2[k] + w_N^{3k} F_3[k] \end{aligned}$$

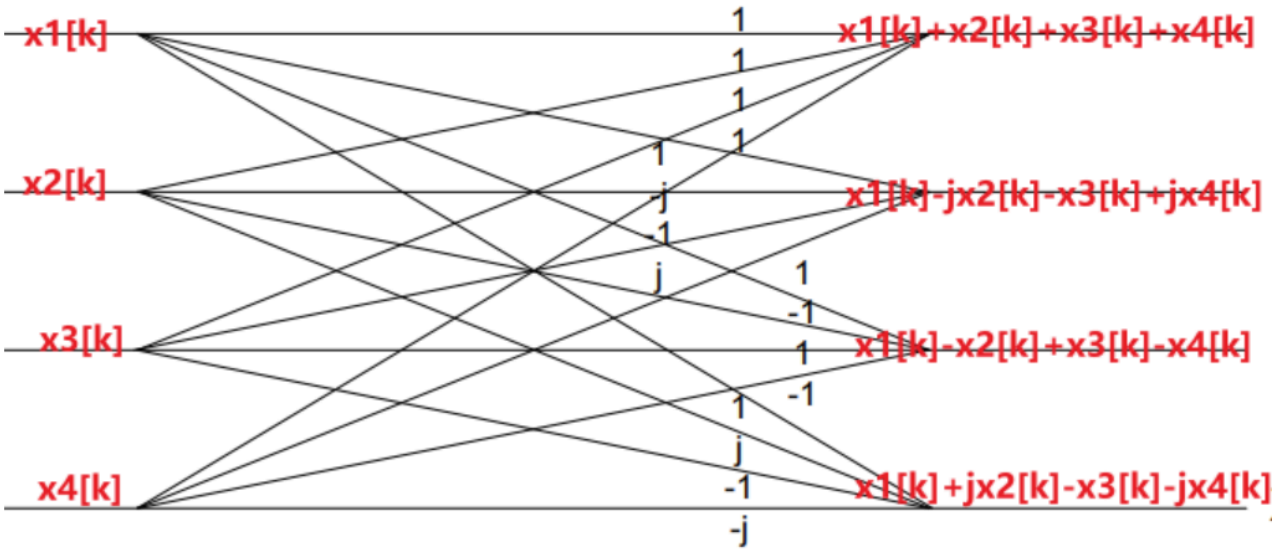
上式是以时域抽取（DIT）为例的。

其中，$F_i[k](i = 0, 1, 2, 3)$分别对应序列$x[4m + i], (m = 0 \cdots \frac{N}{4} - 1)$ DFT的第k项，k满足$k \leq \frac{N}{4} - 1$

进一步利用旋转因子的性质，不难得到$k > \frac{N}{4}$时DFT的值：

$$X[k+\frac{N}{4}] = F_0[k] - j*w_N^k F_1[k] - 1*w_N^{2k}F_2[k] + j*w_N^{3k}F_3[k]$$
$$X[k+\frac{2N}{4}] = F_0[k] - 1*w_N^k F_1[k] + 1*w_N^{2k}F_2[k] - 1*w_N^{3k}F_3[k]$$
$$X[k+\frac{3N}{4}] = F_0[k] + j*w_N^k F_1[k] - 1*w_N^{2k}F_2[k] - j*w_N^{3k}F_3[k]$$

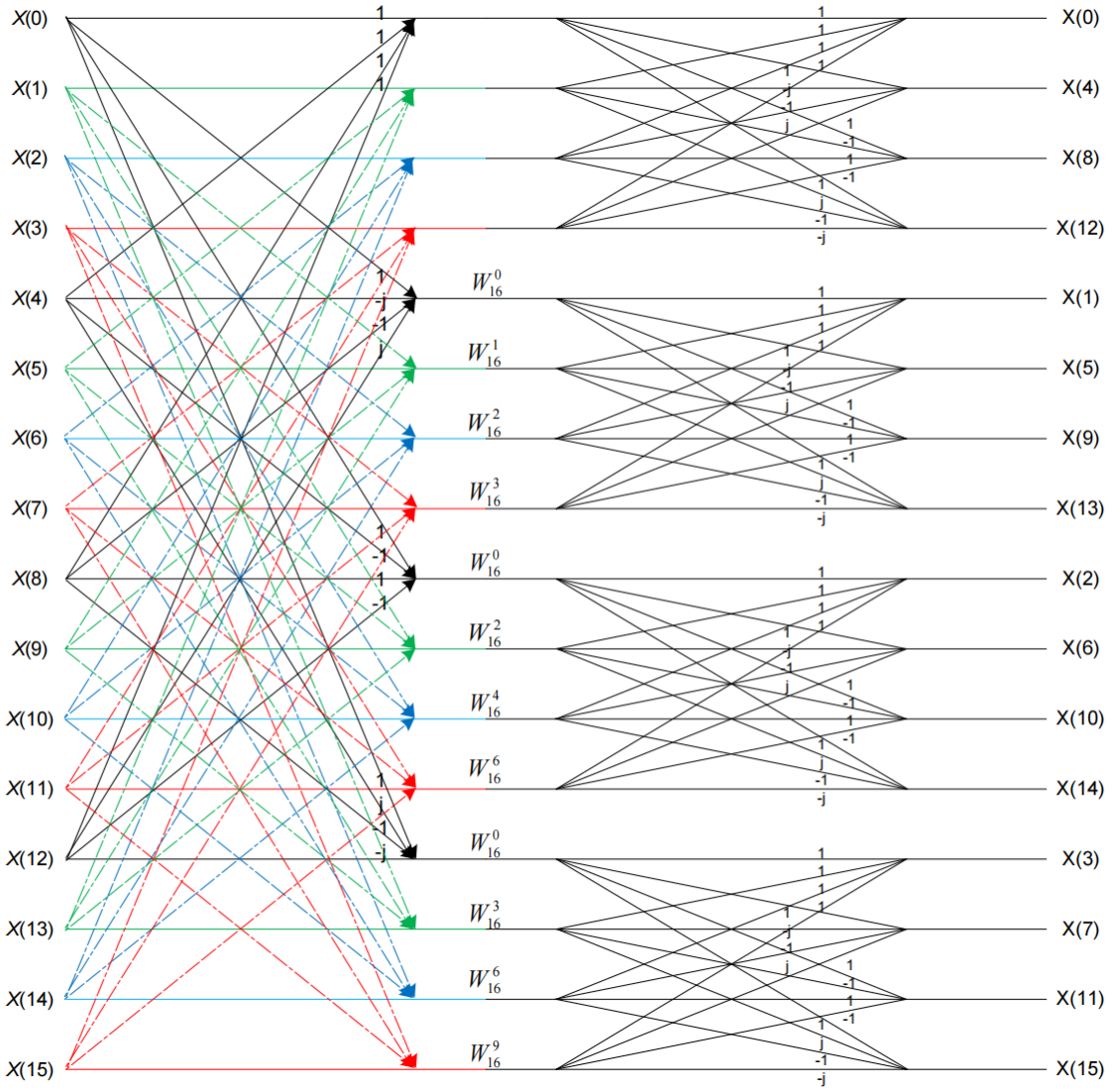
蝴蝶运算单元如下图，与上式是能够一一对应的。



对于逆变换，则有：

$$x[k+\frac{N}{4}] = F_0[k] + j*W_N^k F_1[k] - 1*W_N^{2k}F_2[k] - j*W_N^{3k}F_3[k]$$
$$x[k+\frac{2N}{4}] = F_0[k] - 1*W_N^k F_1[k] + 1*W_N^{2k}F_2[k] - 1*W_n^{3k}F_3[k]$$
$$x[k+\frac{3N}{4}] = F_0[k] - j*W_N^K F_1[k] - 1*W_N^{2k}F_2[k] + j*w_N^{3k}F_3[k]$$

频域采样的基-4 FFT算法类似，在此只给出其数据流向图，不做推导上的过多赘述。



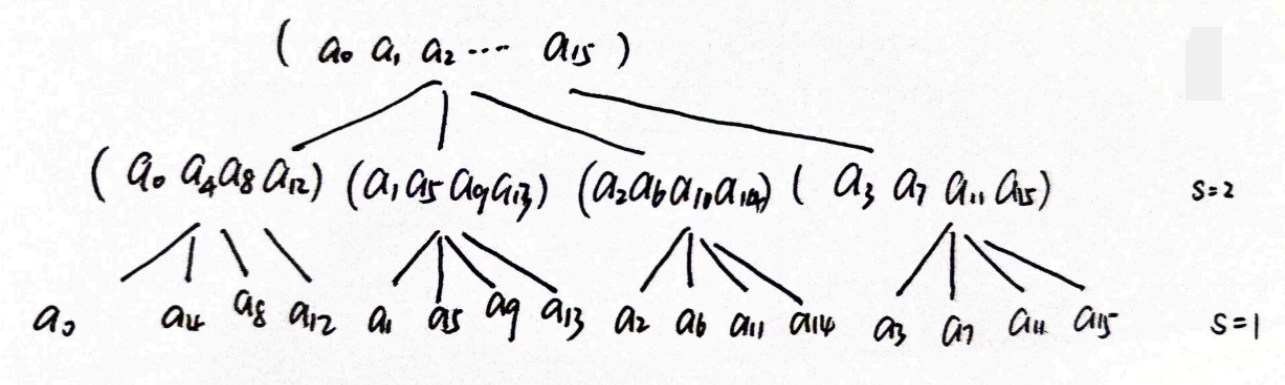
从上图 可以看出，输出数据的顺序也被打乱了。我们把顺序的序号用二进制数列在表 1 中的左边，把乱序的序号用二进制数列在表 1 中的右边。从表中我们可以看出，乱序序号的二进制码可由顺序序号的二进制码以 2 比特为单位反转得到。

Normal order of index n	Binary bits Of index n	Reversed bits of index n	Bit-reversed of order index n
0	00 00	00 00	0
1	00 01	01 00	4
2	00 10	10 00	8
3	00 11	11 00	12
4	01 00	00 01	1
5	01 01	01 01	5
6	01 10	10 01	9
7	01 11	11 01	13
8	10 00	00 10	2
9	10 01	01 10	6
10	10 10	10 10	10
11	10 11	11 10	14
12	11 00	00 11	3
13	11 01	01 11	7
14	11 10	10 11	11
15	11 11	11 11	15

5.2 蝶形网络的优化

如果按照直接按照推导的公式进行计算，蝶形运算单元的伪代码将如下所示：

```
for s = 1 to log4(n) # 树的层数
    size = 4^s;      # 当前DFT的规模，同样也是下一个循环的跨度
    w1 = w2 = w3 = 1; wm = exp(-2j*PI/size); #定义旋转因子wm，w1，w2，w3分别是wm的指数。
    for k = 0 to n - 1 by 4^s
        for q = 0 to size / 4
            tmp0, tmp1, tmp2, tmp3 =
                    a[k + q], a[k + q + size/4], a[k + q + 2*size/4],a[k + q +
3*size/4];
            a[k] = tmp0 + tmp1 + tmp2 + tmp3;
            a[k + size/4] = tmp0 - j * w1 * tmp1 - w2 * tmp2 + j * w3 * tmp3;
            a[k + 2*size/4] = tmp0 - w1 * tmp1 + w2 * tmp2 - w3 * tmp3;
            a[k + 3*size/4] = tmp0 + j * w1 * tmp1 - w2 * tmp2 - j * w3 * tmp3;
            w1 *= wm; w2 *= wm^2; w3 *= wm^3;
```



仔细计算发现，一个蝶形运算包括12次复数加法、12次复数乘法。在不优化的情况下，该程序会比基-2的算法更慢。我们对一个蝶形运算单元进行如下优化：

$$
\begin{aligned}
P_0 &= F_0[k] \\
P_1 &= w_N^k * F_1[k] \\
P_2 &= w_N^{2k} * F_2[k] \\
P_3 &= w_N^{3k} * F_3[k]
\end{aligned}
$$

又令

$$
\begin{aligned}
U_0 &= P_0 + P_2 \\
U_1 &= P_1 + P_3 \\
U_2 &= P_0 - P_2 \\
U_3 &= P_1 - P_3
\end{aligned}
$$

最后，根据公式

$$
\begin{aligned}
X[k] &= U_0 + U_1 \\
X[k + \frac{N}{4}] &= U_2 - j * U3 \\
X[k + \frac{2N}{4}] &= U_0 - U_1 \\
X[k + \frac{3N}{4}] &= U_2 + j * U_3
\end{aligned}
$$

预先计算旋转因子，并通过以上操作，我们将计算次数减少为：8次复数加法，3次复数乘法，能够显著提高程序运行效率。

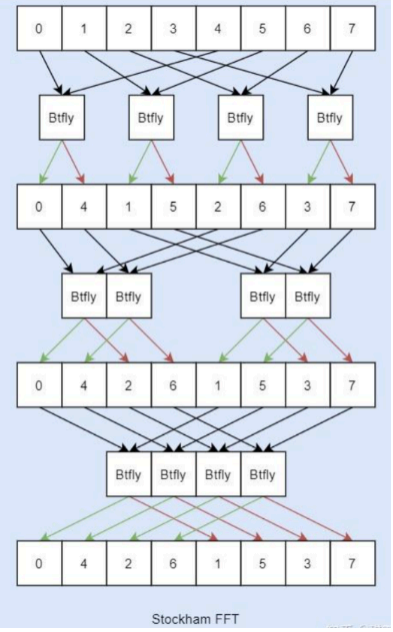
5.3 速度瓶颈

显然，如果数据长度为4^n，基-4FFT回比基-2FFT有更好的效率。但由于在实际运算中需要用到零填充操作，一个$4^n + 1$长度的序列会被扩展为4^{n+1}，造成了大量的时间浪费。因此，在很多数据集上，基-4 FFT算法反而不如基-2 FFT。

6 Stockham FFT：原地自动整序算法

不管是时域采样（DIT）还是频域采样（DIF），库利-图基算法会导致得到的FFT序列顺序改变，或者需要提前通过比特翻转来改变序列的顺序。在比特翻转过程中，不连续的内存空间导致Cache的利用效率变低，这也是人们在努力研究去解决的问题。

Clive Temperton于1991年在《Self-Sorting In-Place Fast Fourier Transforms》一文中给出了适用于混合基数的原地FFT算法，不需要对输入或输出重新排序。通过将计算的中间结果存储到另一片区域，下次变换的时候再存储回来，如此往复，即可省去比特翻转的过程，这也被称为Stockham FFT。



如上图所示，算法的核心难点在于如何对序列原地重排。在代码实现上，Stockham FFT 利用两块存储空间，在蝶形操作时将`x[q + s*(p + 0)], x[q + s*(p + m)]`的相互运算结果存放到`y[q + s*(2*p + 0)], y[q + s*(2*p + 1)]`中，从而合并了蝶形运算和重排序操作。

7 混合基算法

我们假设$N = N_1 * N_2$

$$X[k] = \sum_{j=0}^{N-1} x\,[j]\omega_N^{jk}, \quad 0 \le k \le N-1$$

如果N能被分解为$N_1 * N_2$，那么上式的下标k, j可以表示为：

$$j = j_1 + j_2 N_1, \quad k = k_2 + k_1 N_2$$

把X[k]，x[j]分别用二元组来表示：

$$x[j] = x[j_1, j_2], \quad 0 \le j_1 \le n_1 - 1, \quad 0 \le j_2 \le n_2 - 1$$
$$X[k] = X[k_2, k_1], \quad 0 \le k_1 \le n_1 - 1, \quad 0 \le k_2 \le n_2 - 1$$

例如，$N = 6 = 2 * 3$，那么$X[0] = X[0 + 0 * 3], X[]1$

8 使用FFT快速卷积

8.1 卷积定义

两个n点时域信号，定义卷积运算如下：

$$y[n] = \sum_{i=-\infty}^{\infty} x[i] \cdot h[n-i] = x[n] * h[n]$$

8.2 FFT-Convolve

下面，我们借助FFT与IFFT（傅里叶逆变换）来实现卷积操作。具体流程如下：

1. 计算$f[n]$和$g[n]$的傅里叶变换

$$F[n] = \mathcal{F}\{f[n]\}$$
$$G[n] = \mathcal{F}\{g[n]\}$$

2. 计算$F[n] \times G[n]$的傅里叶逆变换。

$$y[n] = \mathcal{F}^{-1}\{F[n] \times G[n]\}$$

8.3 冗余优化

在上述计算过程中，我们需要调用三次FFT函数：对f, g分别FFT，对$F \times G$做IFFT。对于一般的实数输入，这种做法存在着计算冗余。原因在于，FFT同时对f, g的虚部信息进行了处理，尽管这部分的数据为零。下面一份常见的FFT数据输入部分的伪代码。

```
for (i = 0;i < N; i++) {
   samplesA[i][0] = read(), samplesA[i][1] = 0;
   samplesB[i][0] = read(), samplesB[i][1] = 0;
   // 0为实部，1为虚部。虚部存0，属于冗余信息
}
answerArray = iFFT(FFT(A) * FFT(B));
```

从信息论的角度，我们可以对冗余数据进行无损压缩处理，而不损失其信息量。不妨尝试利用虚部空间，令

$$h[n] = f[n] + g[n]j$$

并假设其DFT为$H[k] = \mathcal{F}\{h[n]\}$。

傅里叶变换保留了原信号的所有信息，只是频域维度信号的再现。因为输入信号$h[n]$包含$f[n]$与$g[n]$的所有信息，我们就一定能从$H[n]$中还原出$F[n], G[n]$。推导如下：

$$\begin{aligned} H[k] &= \mathcal{F}\{h[n]\} \\ &= \mathcal{F}\{f + gj\} = \mathcal{F}\{f\} + j\mathcal{F}\{g\} \qquad \text{线性性} \\ &= F[k] + G[k]j \end{aligned}$$

对$H[k]$作平方：

$$\begin{aligned} H^2[k] &= (\mathcal{F}\{f\} + j\mathcal{F}\{g\})^2 \\ &= (F^2[k] - G^2[k]) + 2F[k] \cdot G[k]j \end{aligned}$$

求逆变换，令：

$$\begin{aligned} z[n] &= \mathcal{F}^{-1}\{H^2[k]\} \\ &= \mathcal{F}^{-1}\{(F^2[k] - G^2[k])\} + 2j \cdot \mathcal{F}^{-1}\{F[k] \cdot G[k]\} \end{aligned}$$

上式中，根据实信号DFT的性质，$F[k] \cdot G[k]$对应的IDFT（离散傅里叶逆变换）一定是实数；$F^2[k], G^2[k]$对应的IDFT也一定是实数。所以$z[n]$的实部与虚部一定分别对应$\mathcal{F}^{-1}\{(F^2[k] - G^2[k])\}$，$\mathcal{F}^{-1}\{F[k] \cdot G[k]\}$。

并且根据DFT的时域卷积性质，我们知道

$$\mathcal{F}^{-1}\{F[k] \cdot G[k]\} = f[n] * g[n]$$

综上，我们只需要对$H^2[k]$做逆变换，取$\frac{1}{2}Im\{z[n] = \mathcal{F}^{-1}(H^2[n])\}$就是原时域信号$f[n], g[n]$的卷积。

我们用很少的改动就能显著提高运行速度。即使不能得知优化该算法的前辈是否了解信息论，但这并不影响我们从信息论的角度理解这种优化。

9 针对硬件、指令集的优化

9.1 FFTW

FFTW是由 Frigo 和 Johnson开发的一个快速、可扩展的FFT实现。在实际计算DFT之前，FFTW会预先执行一个辅助函数，通过一些列的试运行，确定在当前主机上分解FFT的最佳方式。FFTW能够针对硬件平台的缓存对程序进行自适应的调整，在任何规模上都有非常好的表现。

事实上，MATLAB自带的FFT也对FFTW进行了调用，并在这个基础上做了更多基于MATLAB特性的底层优化，将其强大的矩阵运算能力发挥到极致。

在 linux 环境下调用FFTW库完成卷积计算。编译方式详见源文件。关键代码如下：

```
#include <fftw.h>  // 需要调用相应的头文件
...
void FFTW_Conv(double *a, int a_len, double *b, int b_len, double *Result)
{
    int n = a_len + b_len - 1;
    int N = nextPowerOfTwo(n);
    fftw_complex *A = (fftw_complex *)fftw_malloc(sizeof(fftw_complex) * N);
    fftw_complex *B = (fftw_complex *)fftw_malloc(sizeof(fftw_complex) * N);
    fftw_complex *C = (fftw_complex *)fftw_malloc(sizeof(fftw_complex) * N);
    double *a_padded = (double *)fftw_malloc(sizeof(double) * N);
    double *b_padded = (double *)fftw_malloc(sizeof(double) * N);
    double *c_padded = (double *)fftw_malloc(sizeof(double) * N);
    // 填充输入数据
    for (int i = 0; i < N; i++)
    {
        a_padded[i] = (i < a_len) ? a[i] : 0;
        b_padded[i] = (i < b_len) ? b[i] : 0;
    }
    // 创建 FFTW 计划
    fftw_plan pA = fftw_plan_dft_r2c_1d(N, a_padded, A, FFTW_ESTIMATE); // 实数到虚数的转换
    fftw_plan pB = fftw_plan_dft_r2c_1d(N, b_padded, B, FFTW_ESTIMATE); // real to complex 1 dimentional
    fftw_plan pC = fftw_plan_dft_c2r_1d(N, C, c_padded, FFTW_ESTIMATE); // complex to real 1 dimentional
    // 执行 FFT
    fftw_execute(pA);
    fftw_execute(pB);
    // 点乘
    for (int i = 0; i < N; i++)
    {
        C[i][0] = A[i][0] * B[i][0] - A[i][1] * B[i][1];
        C[i][1] = A[i][0] * B[i][1] + A[i][1] * B[i][0];
    }
    // 执行逆 FFT
    fftw_execute(pC);
    // 复制结果
    for (int i = 0; i < n; i++)
    {
        Result[i] = c_padded[i] / N;
    }
    // 清理内存
    fftw_destroy_plan(pA);
```

```
    fftw_destroy_plan(pB);
    fftw_destroy_plan(pC);
    fftw_free(A);
    fftw_free(B);
    fftw_free(C);
    fftw_free(a_padded);
    fftw_free(b_padded);
    fftw_free(c_padded);
}
```

9.2 异构计算

CUDA（Compute Unified Device Architecture）是由NVIDIA开发的一种并行计算平台和编程模型，允许开发者使用图形处理单元（GPU）来进行通用计算。在FFT算法中应用GPU强大的并行计算能力，对运算速度有着极大的提升。

10 总结

11 附录与测试结果

以下测试数据来源于由`gen_data.cpp`生成的随机序列，且使用同样的数据集。除了NTT以外，代码实现上均并不区分整数、浮点数浮点序列，因而运行速率大致相等，浮点序列的运行效率略低于整数序列。

11.1 MATLAB测试

数据规模 （n+m-1=N）	数据类型	卷积耗时（s）
2+3-1=4	Int	0.2875
16+17-1=32	Int	0.00028
1024+1024-1=2047	Int	0.00819
1031+1029-1=2059	Int	0.00031
$2^{16}+2^{16}-1=2^{17}-1$	Int	0.05534
$2^{16}+1+2^{16}+1-1=2^{17}+1$	Int	0.05339
$3*2^{20}-1$	Int	25.68
$2^{22}-2$	Int	14.31
2^24	Int	222.27

11.2 FFTW测试

```
(base) tsum@LAPTOP-31P15CTK:~/CodeSpace/FFTW/build$ /home/tsum/CodeSpace/FFTW/build/Conv
Test for Double Input
Test :0, (n, m) = (2, 3), Times Cost:0.0006
Test :1, (n, m) = (16, 17), Times Cost:0.0003
Test :2, (n, m) = (1024, 1024), Times Cost:0.0037
Test :3, (n, m) = (1031, 1029), Times Cost:0.0032
Test :4, (n, m) = (65536, 65536), Times Cost:0.0141
Test :5, (n, m) = (65537, 65537), Times Cost:0.0582
Test :6, (n, m) = (1048576, 2097152), Times Cost:0.3313
Test :7, (n, m) = (1048577, 1048577), Times Cost:0.2891
Test :8, (n, m) = (16777216, 16777216), Times Cost:3.0954
Test :9, (n, m) = (4194303, 4194303), Times Cost:0.7061
```

```
(base) tsum@LAPTOP-31P15CTK:~/CodeSpace/FFTW/build$ /home/tsum/CodeSpace/FFTW/build/Conv
Test for Integer Input
Test :0, (n, m) = (2, 3), Times Cost:0.0007
Test :1, (n, m) = (16, 17), Times Cost:0.0004
Test :2, (n, m) = (1024, 1024), Times Cost:0.0041
Test :3, (n, m) = (1031, 1029), Times Cost:0.0079
Test :4, (n, m) = (65536, 65536), Times Cost:0.0119
Test :5, (n, m) = (65537, 65537), Times Cost:0.0210
Test :6, (n, m) = (1048576, 2097152), Times Cost:0.3600
Test :7, (n, m) = (1048577, 1048577), Times Cost:0.3282
Test :8, (n, m) = (16777216, 16777216), Times Cost:3.2287
Test :9, (n, m) = (4194303, 4194303), Times Cost:0.6339
```

11.3 radix-2 FFT 测试

```
Test for Integer Input
Test :0, Time Cost:0.0000
Test :1, Time Cost:0.0000
Test :2, Time Cost:0.0000
Test :3, Time Cost:0.0000
Test :4, Time Cost:0.0700
Test :5, Time Cost:0.1080
Test :6, Time Cost:2.5000
Test :7, Time Cost:2.4690
Test :8, Time Cost:4.7640
Test :9, Time Cost:10.7840
```

11.4 radix-4 FFT 测试

整数测试:

```
Test :0, Read Data Time Cost:0.0000
Test :0, (n, m) = (2, 3),Convolve Times Cost:0.0000
Test :1, Read Data Time Cost:0.0000
Test :1, (n, m) = (16, 17),Convolve Times Cost:0.0000
Test :2, Read Data Time Cost:0.0100
Test :2, (n, m) = (1024, 1024),Convolve Times Cost:0.0050
Test :3, Read Data Time Cost:0.0000
Test :3, (n, m) = (1031, 1029),Convolve Times Cost:0.0000
Test :4, Read Data Time Cost:0.0850
Test :4, (n, m) = (65536, 65536),Convolve Times Cost:0.3470
Test :5, Read Data Time Cost:0.1110
Test :5, (n, m) = (65537, 65537),Convolve Times Cost:0.4010
Test :6, Read Data Time Cost:2.1650
Test :6, (n, m) = (1048576, 2097152),Convolve Times Cost:6.7600
Test :7, Read Data Time Cost:1.2480
Test :7, (n, m) = (1048577, 1048577),Convolve Times Cost:6.5530
Test :8, Read Data Time Cost:4.9730
Test :8, (n, m) = (4194304, 4194303),Convolve Times Cost:29.9770
Test :9, Read Data Time Cost:11.5780
Test :9, (n, m) = (8388608, 8388609),Convolve Times Cost:31.3800
```

11.5 stockham FFT 测试

整数测试；

```
Test for Integer Input
Test :0, Time Cost:0.0000
Test :1, Time Cost:0.0000
Test :2, Time Cost:0.0010
Test :3, Time Cost:0.0030
Test :4, Time Cost:0.0490
Test :5, Time Cost:0.0970
Test :6, Time Cost:2.0760
Test :7, Time Cost:1.9980
Test :8, Time Cost:4.2550
Test :9, Time Cost:8.6150
```

12 References

Mixed-Radix Cooley-Tukey FFT (stanford.edu)

Mixed-Radix FFT Algorithms | SpringerLink

Self-Sorting In-Place Fast Fourier Transforms | SIAM Journal on Scientific Computing

从Cooley-Tukey FFT到Stockham FFT - Catigeart's Software Development Note

FFTNTT代码技巧_x + (x >> 31 & mod)-CSDN博客

详尽的快速傅里叶变换推导与Unity中的实现 - 知乎 (zhihu.com)

蝶形结 - 维基百科，自由的百科全书 (wikipedia.org)

快速傅里叶变换 - 维基百科，自由的百科全书 (wikipedia.org)

调库fftw 编译安装教程 | 超算小站 (mrzhenggang.com)

不同FFT 对比：jeremyfix/FFTConvolution: Some C++ codes for computing a 1D and 2D convolution product using the FFT implemented with the GSL or FFTW (github.com)

贼吊kwsp/fftconv: Fast Fourier domain 1D convolution of real vectors (github.com)

overlap 算法：Overlap–add method - Wikipedia

算法优化：深度学习 - 硬核解析FFT原理和优化策略，值得收藏！ - 个人文章 - SegmentFault 思否

用CUDA：CUDA并行算法系列之FFT快速卷积 - 张朝龙(行之) - 博客园 (cnblogs.com)

KAdamek/GPU_Overlap-and-save_convolution: Shared memory overlap-and-save method for NVIDIA GPUs using CUDA (github.com)

GPU Fast Convolution via the Overlap-and-Save Method in Shared Memory (acm.org)‘

并行计算：高性能计算学习路线（针对大二同学） - orion-orion - 博客园 (cnblogs.com)

https://blog.csdn.net/qq_41094058/article/details/116207333

FFTW cuFFT的使用记录_itk fftw cufft-CSDN博客

FFT & NTT 及其简单优化 - 樱雪喵 - 博客园 (cnblogs.com)：三次变两次优化。NTT算法

C语言使用CUDA中cufft函数做GPU加速FFT运算，与调用fftw函数的FFT做运算速度对比_cuda fft-CSDN博客

基2与基4时分FFT算法浅析及其比较_基4fft-CSDN博客

FFT-快速傅里叶变换的推导、推广与优化 | StellarWarp:有用，待看

(37 封私信 / 81 条消息) 快速傅里叶变换（FFT） N不为2的次方怎么做？ - 知乎 (zhihu.com)