Способ уменьшения числа операций в алгоритме быстрого преобразования Фурье

Е. А. Альтман
Омский государственный университет путей сообщения, Россия
Контактный e-mail: AltmanEA@gmail.com

Рассмотрен способ сокращения числа арифметических операций в алгоритме быстрого преобразования Фурье (БПФ). Способ основан на сокращении числа операций с поворачивающими множителями. Он применим для алгоритмов БПФ по основанию 2, использующих четырехточечные преобразования. Представлен новый алгоритм БПФ с меньшим по сравнению с ранее известными алгоритмами числом операций. Приведена реализация алгоритма на языке Python.

Ключевые слова: быстрое преобразование Фурье, split radix, radix-4, арифметическая сложность.

Введение

Быстрое преобразование Фурье (БПФ, Fast Fourier Transform, FFT) широко применяется при цифровой обработке сигналов. Этому преобразованию посвящено большое количество научных работ, немалая доля которых исследует его быстродействие. Хотя быстродействие программы зависит от многих факторов, наибольший интерес представляет нахождение алгоритма, требующего наименьшего числа арифметических операций. Наиболее популярная метрика эффективности алгоритма — количество флоп, т. е. операций с плавающей точкой (flop — floating-point operations).

Широкое использование БПФ началось с публикации Кули и Тьюки [1], где предлагался алгоритм, который требовал порядка $5N \log N$ flop для N-точечного преобразования. Предложено несколько его усовершенствований. Долгое время лучшим был предложенный в 1968 г. алгоритм, требующий порядка $4N \log(N)$ флоп [2]. В 1984 г. он был открыт повторно в статье [3] и получил название split-radix. Только в 2004 г. ван Баскирк предложил идею сокращения числа операций [4], с использованием которой были получены алгоритмы БПФ, имеющие асимптотическую оценку $\frac{34}{9} N \lg N$. Наиболее известная реализация этой идеи называется modified split-radix FFT [5].

В дальнейших исследованиях БПФ получены результаты, оптимизирующие известные алгоритмы, но не уменьшающие количество требуемых арифметических операций. Так, Бернштейном предложен алгоритм tangent FFT [6], реализующий вычисление арифметических операций алгоритма modified split-radix FFT без использования дополнительных объемов памяти (так называемый in-place-алгоритм), который уменьшал тем самым количество операций с памятью.

Применение идеи ван Баскирка для алгоритмов radix-2 [7] и radix-4 [8] позволило уменьшить число операций для соответствующих алгоритмов, добившись в последнем случае такого же числа операций, как и в алгоритме split-radix FFT.

В ряде работ рассмотрены модификации стандартных алгоритмов БПФ, позволяющие получить лучшие с практической точки зрения результаты при их реализации для конкретных аппаратных платформ [9], в том числе и с несколькими исполнительными устройствами [10]. С более полным библиографическим обзором можно ознакомиться в обновляемой электронной книге Fast Fourier Transforms [11].

В данной статье предлагается новый способ сокращения числа арифметических операций. С помощью этого способа строится алгоритм, который требует несколько меньшего числа операций, чем известные алгоритмы. Предлагаемый способ позволяет продвинуться в теоретическом исследовании алгоритмов БПФ в направлении уменьшения числа операций. При этом вопрос о минимальном числе операций по-прежнему остается открытым.

1. Методы

В работе рассматриваются алгоритмы БПФ типа Кули — Тьюки, основанные на вычислении преобразования с помощью нескольких БПФ с меньшим числом точек. Для учета различных аспектов алгоритмов будем описывать их тремя способами: математическими формулами, кодами программ и графами вычислительной схемы алгоритма.

Рассмотрим способы описания алгоритмов на примере алгоритма БПФ radix-2, в котором преобразование вычисляется через два БПФ половинного размера. Математически этот подход можно описать следующим образом. Обозначим поворачивающие множители $w_n^i = \exp\left(-2\pi j\frac{i}{n}\right)$, где j — мнимая единица. Формула для вычисления БПФ записывается следующим образом:

$$y_j = \sum_{i=0}^{n-1} w_n^{ij} x_i.$$

Разобьем входные данные на два подмножества четных (x_{2i}) и нечетных (x_{2i+1}) отсчетов. Для каждого подмножества вычислим БПФ:

$$\begin{aligned} z1_j &= \sum_{i=0}^{n/2-1} w_n^{(2i)j} x_{2i} = \sum_{i=0}^{n/2-1} w_{\frac{n}{2}}^{ij} x_{2i}, \\ z2_j &= \sum_{i=0}^{n/2-1} w_n^{(2i+1)j} x_{2i+1} = w_n^j \sum_{i=0}^{n/2-1} w_{\frac{n}{2}}^{ij} x_{2i+1}. \end{aligned} \tag{1}$$

Используя результаты БПФ подмножеств и свойства поворачивающих множителей, можно найти результат исходного БПФ с помощью следующей схемы, получившей название “бабочка”:

$$\begin{aligned} y_{j=0\ldots\frac{n}{2}-1} &= z1 + z2, \\ y_{j=\frac{n}{2}\ldots n-1} &= z1 - z2. \end{aligned} \tag{2}$$

Этим способом удобно описывать только один этап БПФ. Алгоритмы типа Кули — Тьюки подразумевают дальнейшее рекурсивное разбиение БПФ подмножеств на БПФ

более мелких подмножеств. Описание подобным образом нескольких этапов вычисления БПФ получается громоздким и сложным для анализа.

Более наглядным и удобным для проверки является описание алгоритма в виде программы на языке Python. Программы на этом языке не уступают в наглядности и лаконичности псевдокоду, при этом их можно непосредственно использовать для изучения свойств алгоритмов. Код приводимых в статье программ доступен в проекте [12].

Рассмотренный выше вариант БПФ приведен на листинге 1.

Листинг 1. Алгоритм БПФ radix 2

```
def fftr2(x):
    n = x.size
    if n == 1: return x
    w = array([exp(-1j*2*pi*i/n) for i in range(n//2)])
    y0 = fftr2(x[0:n:2])
    y1 = fftr2(x[1:n:2])*w
    return hstack((y0 + y1, y0 - y1))
```

Программа позволяет описать алгоритм так же точно, как и математическая формула, но не позволяет увидеть и проанализировать сразу несколько стадий вычислений. Поэтому часто для описания алгоритмов применяется наглядное визуальное представление в виде графов, пример которого для алгоритма БПФ radix-2 на восемь точек приведен на рис. 1. На схеме показано несколько этапов вычислений. “Бабочки” изображены перекрещенными линиями, штриховыми линиями обозначены точки вычислительной схемы, возникающие на отдельных этапах вычислений. При необходимости умножения точек на поворачивающие множители они указываются непосредственно на штриховых линиях.

В статье алгоритмы рассматриваются с точки зрения числа операций, необходимых для их выполнения. Расчет числа операций БПФ в общем случае является трудоемкой задачей, но для алгоритмов рассматриваемого типа ее можно упростить.

Во всех алгоритмах этого типа требуется выполнять операции комплексного сложения и вычитания, описанные в формуле (2). Для вычисления преобразования по n точкам следует выполнить одну комплексную или две вещественные операции на точку на каждой стадии вычислений (здесь и далее будем считать вещественные операции, но результаты будут справедливы для целых и других типов данных). Количество стадий равно логарифму от числа точек; таким образом, нужно выполнить $2n \log_2 n$ таких операций. Например, для восьмиточечного БПФ требуется 48 таких операций.

Для нахождения полного числа операций в алгоритме осталось подсчитать число операций, затраченных на умножение на поворачивающие множители. Обычная опе-

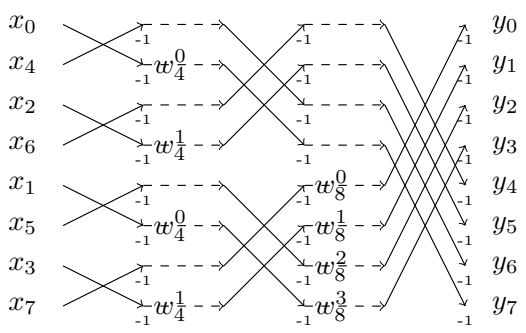

Рис. 1. Схема БПФ radix-2

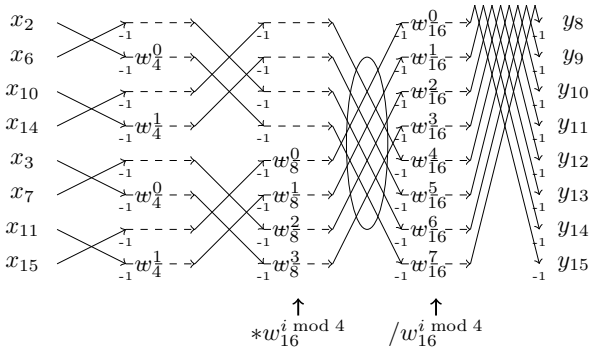

Рис. 2. Схема БПФ radix-2 на 16 точек

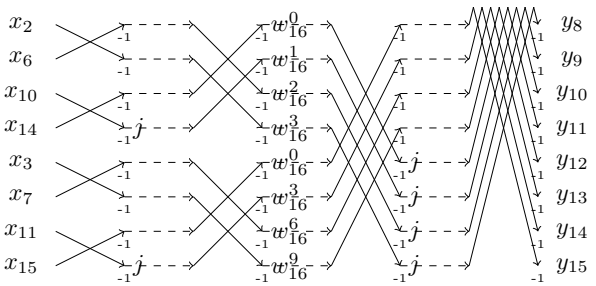

Рис. 3. Схема БПФ radix-4 на 16 точек

рация комплексного умножения требует 6 операций с вещественными числами, но есть особые числа, умножение на которые требует меньшего числа операций.

Поворачивающие множители с углом поворота (аргументом экспоненты), равным нулю или кратным $\pi/2$, будут равны одному из следующих значений: 1, j, -1 или $-j$. Умножение на такие множители не требует операций. Поворачивающие множители с углом поворота, кратным $\pi/4$, но не кратным $\pi/2$, будут иметь вид $\pm a \pm ja$. Умножение на такие множители требует четыре вещественные операции.

В алгоритме modified split-radix количество операций сокращается за счет использования масштабированных поворачивающих множителей вида $\pm 1 \pm ja$ или $\pm a \pm j1$, умножение на которые требует также 4 вещественные операции.

Восьмиточечный алгоритм БПФ содержит 8 поворачивающих множителей. При этом 6 из них (w_n^0, w_4^1 и w_8^2) не требуют выполнения операций, а два оставшихся (w_8^1 и w_8^3) вычисляются четырьмя вещественными операциями. В итоге получаем, что БПФ на 8 точек требует $48 + 4 + 4 = 56$ операций.

Алгоритмы типа Кули — Тьюки (radix-2, radix-4, split-radix и др.) различаются между собой "расстановкой" поворачивающих множителей. Обычным способом описания этих алгоритмов является применение разложения на БПФ меньшего размера с использованием формул (1) и (2) или аналогичных.

В настоящей работе будем использовать другой подход, который назовем переносом операций. Идея этого подхода основана на линейности операции БПФ. Если входные точки БПФ разделить на некоторый множитель, а выходные на него же умножить, то общий результат не изменится. Это утверждение справедливо и для любой "подБПФ". Например, рассмотрим фрагмент преобразования БПФ radix-2 на 16 точек (рис. 2, показана только нижняя часть схемы). Разделим выходные точки бабочек, обведенных эллипсом, на $w_{16}^{i \bmod 4}$. Множитель после верхних выходов бабочек становится равным 1, а множитель после нижних выходов — j. Эти множители не требуют вычислительных операций. Вычислительные операции переносятся на предыдущую стадию вычислений. При этом множители w_8^0, w_8^1, w_8^2 и w_8^3, на которые происходило умножение на предыдущей стадии, перемножаются с перенесенными значениями. Таким образом, мы сокращаем 8 операций (рис. 3). В результате такого переноса мы получили алгоритм, известный как БПФ radix-4, который обычно выводится разделением исходного БПФ на четыре БПФ меньшего размера.

2. Результаты

Рассмотрим фрагмент схемы вычисления БПФ по алгоритму radix-4 (рис. 4).

После выполнения предыдущей стадии вычислений точки нужно умножить на поворачивающие множители w_n^i для $i = 0 \dots \frac{n}{4} - 1$. При этом точки с номерами $i = 0 \dots \frac{n}{16} - 1$ попадают на первый вход четырехточечного преобразования, точки с номерами $i = \frac{n}{16} \dots \frac{2n}{16} - 1$ — на второй и т. д.

Разделим поворачивающие множители на величину $w_n^{i \bmod \frac{n}{16}}$. Множители из первой четверти станут равными 1, из второй — $w_n^{\frac{n}{16}} = w_{16}^1$, из третьей — $w_n^{\frac{2n}{16}} = w_{16}^2$, из четвертой — $w_n^{\frac{3n}{16}} = w_{16}^3$. Умножение на 1 не требует операций. Умножение на w_{16}^1 и w_{16}^3 требует шесть операций. Умножение на $w_{16}^2 = \cos\frac{\pi}{4} + j\sin\frac{\pi}{4} = \frac{\sqrt{2}}{2} + j\frac{\sqrt{2}}{2}$ — четыре операции. Всего для каждой бабочки требуется 16 операций.

Количество операций умножения на стадии перед рассматриваемой бабочкой уменьшится, но возрастет количество операций на следующей стадии — до переноса не требовалось умножать точки с первых выходов бабочки. В общем случае такое умножение не изменит число операций, а для отдельных бабочек может даже увеличить.

Дополнительно разделим поворачивающие множители на $\cos(\pi/8)$. Первый множитель становится равным $1/\cos(\pi/8)$, для умножения на него требуются две операции. Второй множитель становится равным

$$\frac{w_{16}^1}{\cos(\pi/8)} = \frac{\cos(\pi/8) + j\sin(\pi/8)}{\cos(\pi/8)} = 1 + j\tan\frac{\pi}{8},$$

для умножения на него требуются четыре операции. В третьем множителе после деления вещественная и мнимая части останутся равными между собой, и для умножения на него потребуются также четыре операции. Четвертый множитель станет равным

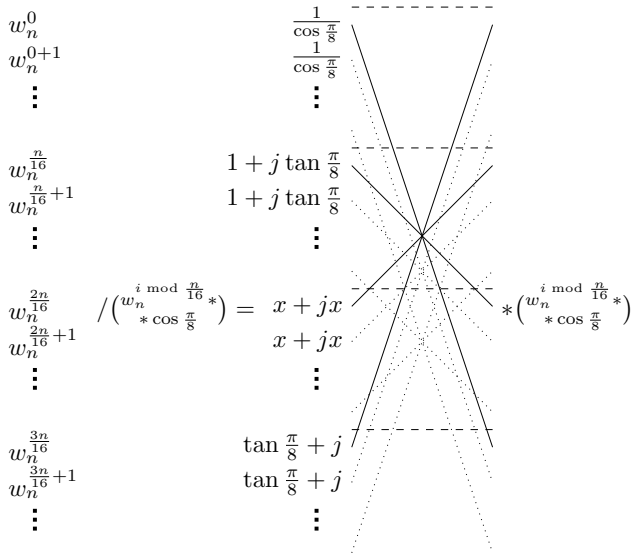

Рис. 4. Перенос вычислений на позднюю стадию

$$\frac{w_{16}^3}{\cos(\pi/8)} = \frac{\cos(3\pi/8) + j\sin(3\pi/8)}{\cos(\pi/8)} = \frac{\sin(\pi/8) + j\cos(\pi/8)}{\cos(\pi/8)} = \tan\frac{\pi}{8} + j,$$

для умножения на который требуются 4 операции. Для каждой бабочки потребуются 14 операций.

Таким образом, можно сократить две операции на бабочку. Этот прием будет действовать только для тех бабочек, после которых идет умножение на поворачивающие множители общего вида.

Количество операций можно сократить и с помощью переноса вычислений на более раннюю стадию. Для этого нужно сделать предварительное преобразование алгоритма. Воспользуемся следующим свойством БПФ:

$$y_j = \sum_{i=0}^{n-1} w_n^{ij} x_i = w_n^{-j} \sum_{i'=0}^{n-1} w_n^{i'j} x_{i'}, \tag{3}$$

где $i = i' - 1$ Подстановка $i = i' - 1$ означает циклический сдвиг входных данных на одну позицию вперед.

Формула для БПФ radix-4 выглядит следующим образом:

$$y_j = \sum_{k=0}^{3} \sum_{i=0}^{\frac{n}{4}-1} w_n^{(4i+k)j} x_{4i+k}.$$

Применим преобразование (3) к четвертому слагаемому внешней суммы:

$$y_j = \cdots + w_n^{-j} \sum_{i=0}^{\frac{n}{4}-1} w_n^{(4i-1)j} x_{4i-1}.$$

В результате множители первого (w_n^j) и второго (w_n^{-j}) слагаемых стали комплексно-сопряженными. Это позволяет использовать следующую схему сокращения операций (рис. 5).

В общем случае умножение на поворачивающие множители результатов четырехточечного преобразования требует 18 операций (первый выходной отсчет умножается на 1, остальные требуют по 6 умножений). Разделим поворачивающие множители на $\cos\dfrac{2\pi(i \bmod (n/4))}{n}$.

Множители для первых выходов примут вид $\left(\cos\dfrac{2\pi(i \bmod (n/4))}{n}\right)^{-1}$ и будут требовать две операции. Для вторых выходов — вид

$$\frac{w_n^i}{\cos\dfrac{2\pi(i \bmod (n/4))}{n}} = 1 + j\tan\left(\frac{2\pi(i \bmod (n/4))}{n}\right),$$

они требуют 4 операции. Для третьих множитель будет общего вида (6 операций). Для четвертых выходов множители примут вид $1 - j\tan\left(\dfrac{2\pi(i \bmod (n/4))}{n}\right)$ и потребуют 4 операции. Всего требуются 16 операций.

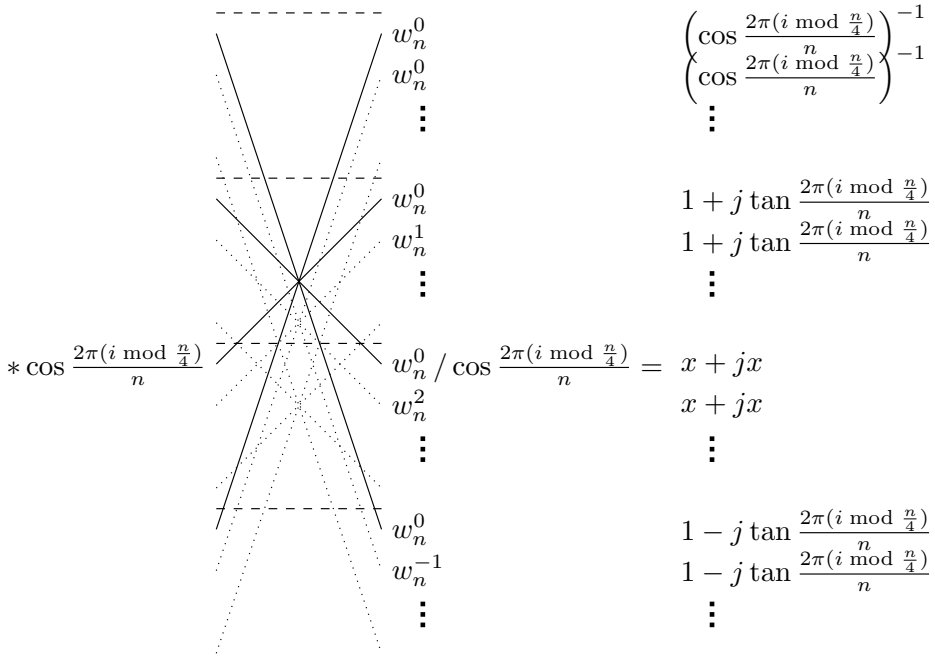

Рис. 5. Перенос вычислений на раннюю стадию

Перенос вычислений на раннюю стадию позволяет сократить число операций и в частных случаях, когда исходные поворачивающие множители не требуют полного числа операций при умножении. Например, для $n = 16$ поворачивающие множители w_0^{16}, w_1^{16}, w_2^{16} и w_3^{16} требуют $0 + 6 + 4 + 6 = 16$ операций. После деления их на $\cos(\pi/8)$ получим множители

$$(\cos(\pi/8))^{-1}, \quad 1 + j\tan(\pi/8), \quad \frac{\cos(\pi/4)}{\cos(\pi/8)} + j\frac{\sin(\pi/4)}{\cos(\pi/8)}, \quad 1 + j\cot(\pi/8),$$

которые требуют $2 + 4 + 4 + 4 = 14$ операций.

Таким образом мы получили два варианта сокращения двух операций. Эффективность способа сильно зависит от того, какие множители находятся на другой стороне четырехточечного преобразования. При наличии на другой стороне множителей, умножение на которые занимает меньше шести операций, выигрыш теряется.

Совместное применение двух вариантов переноса может дать дополнительный эффект, если в обоих случаях умножения переносят на одну стадию вычислений. Рассмотрим пример использования этого приема.

За основу возьмем следующий алгоритм вычисления БПФ на 256 точек (рис. 6). На первом этапе разобьем преобразование по схеме conjugate pair split-radix [13] на одно БПФ на 128 точек и два БПФ по 64 точки. В этой схеме поворачивающие множители перед БПФ на 64 точки с помощью сдвига точек второго БПФ делаются комплексно-сопряженными. Для вычисления БПФ на 128 точек будем использовать лучший из известных алгоритмов [5]. БПФ на 64 точки вычислим по схеме radix-4.

Быстрое преобразование Фурье radix-4 на 64 точки имеет три стадии вычисления и, соответственно, две линии умножения на поворачивающие множители. Перенесем умножения с линии поворачивающих множителей перед БПФ radix-4 на первую линию и туда же перенесем вычисления со второй линии. Переносить будем только через те бабочки, для которых перенос даст экономию операций.

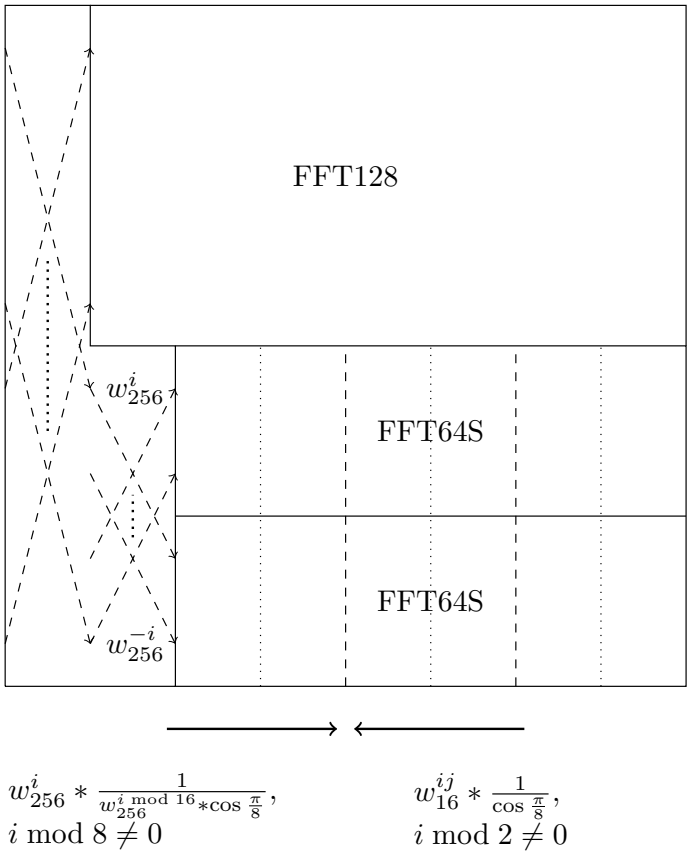

Рис. 6. Быстрое преобразование Фурье на 256 точек

Основной код программы [12] приведен на листинге 2. Функция `ncpsrfft` вычисляет БПФ на 128 точек по методу modified split-radix FFT. Функция `fft64s` реализует масштабированное БПФ на 64 точки. В ней осуществляются необходимые переносы операций. Функция `split_radix_butt` выполняет бабочки по схеме split-radix. Функция `shift_right` выполняет циклический сдвиг массива на одну позицию вправо для перехода к поворачивающим множителям, комплексно-сопряженным с множителями для первого вызова `fft64s`. В переменных `s1` и `s2` находятся переносимые множители.

Листинг 2. Алгоритм БПФ на 256 точек

```
def fft256(x):
n = 256
s1 = array([exp(1j*2*pi*i/n) for i in range(16)])/cos(pi/8)
s2 = array([exp(-1j*2*pi*i/n) for i in range(16)])/cos(pi/8)
s1[0:64:8] = 1
s2[0:64:8] = 1
w1 = array([exp(-1j*2*pi*i/n)*s1[i%16] for i in range(64)])
w2 = array([exp(1j*2*pi*i/n)*s2[i%16] for i in range(64)])
u = ncpsrfft(x[0:n:2])
z = fft64s(x[1:n:4], s1) * w1
z1= fft64s(shift_right(x[3:n:4]), s2) * w2
return split_radix_butt(u,z,z1,n)
```

Код функции `fft64s` приведен на листинге 3. Функция `even_koef` возвращает 1 для четных аргументов и $\cos(\pi/8)$ для нечетных.

Листинг 3. Масштабированное БПФ на 64 точки

```
def fft64s(x, s):
for k in range(4):
z = array(x[k:64:4])
for i in range(4):
z[i:16:4] = fft4(z[i:16:4])
w = array([[exp(-2j*pi*i*j/16)/even_koef(j)
for i in range(4)] for j in range(4)])

z = z * w.flatten()
y = zeros(16, dtype=complex)
for i in range(4):
y[i:16:4] = fft4(z[i * 4:i * 4 + 4])
x[k:64:4] = y

y = zeros(16, dtype=complex)
for i in range(4):
y[i:16:4]= fft4(z[i*4:i*4+4])
x[k:64:4] = y
w = array([[exp(-2j*pi*i*j/64)*even_koef(j)/s[j]
for i in range(4)] for j in range(16)])
x = x * w.flatten()

y = zeros(64, dtype=complex)
for i in range(16):
y[i:64:16] = fft4(x[i * 4:i * 4 + 4])
return y
```

Рассчитаем количество требуемых операций. Операций внутри бабочек требуется $2 \cdot 256 \cdot \log 256 = 4096$. Быстрое преобразование Фурье по 128 точкам требует 2792 операции, из которых $2 \cdot 128 \cdot \log 128 = 1792$ операции внутри бабочки, и 1000 требуется на поворачивающие множители.

Найдем количество умножений для множителей `w1` и `w2` (листинг 2). 64 умножения перед преобразованием `FFT64S` выполняются перед 16 бабочками на четыре точки. Для 14 из них мы выполнили перенос, и они требуют $14 \cdot 14 = 196$ операций. Из оставшихся восьми поворачивающих множителей один равен единице, второй ($w_{256}^{\pm 32}$) требует четыре операции, остальные шесть требуют шесть операций. Получаем $196 + 4 + 6 \cdot 6 = 236$ операций перед преобразованием `FFT64S`. Итого, после стадии split-radix требуются $2 \cdot 236 = 472$ операции.

Из поворачивающих множителей для стадии, на которую переносятся вычисления (`w2` на листинге 3), без переносов остались восемь множителей. Среди них шесть единиц и два поворачивающих множителя по четыре операции. Остальные 56 множителей требуют полного числа умножений. Итого, для данной стадии получаем $2 \cdot (2 \cdot 4 + 56 \cdot 6) = 688$ операций.

На последней стадии (по рисунку) точки умножаются на множители `w1` (листинг 3). `w1` — это матрица размером 4×4. В первой строке матрицы единицы, умножение на которые не требует операций. В третьей строке два множителя не требуют операций и два множителя требуют по четыре операции. Из второй и четвертой строк мы перенесли операции по описанной выше схеме, после чего множители из этих строк требуют в сумме 14 операций. Всего требуется $4 \cdot 2 + 14 \cdot 2 = 36$ операций. Умножение на эту матрицу происходит в двух вызовах функции `fft64s` по четыре раза. Итого, на данной стадии требуются $2 \cdot 4 \cdot 36 = 288$ операций.

Суммируя операции (внутри бабочек, на поворачивающие множители функции `fft128`, после стадии split-radix, на первой и второй стадиях функции `fft64s`), получим $4096 + 1000 + 472 + 688 + 288 = 6544$. Данное число на восемь операций меньше, чем в лучшем из известных алгоритмов (modified split-radix FFT, 6552 [5]).

Фактически экономия операций по сравнению с modified split-radix FFT происходит в части вычислительной схемы, обозначенной на рис. 6 как `fft64s`. Эта часть представляет собой так называемое масштабируемое преобразование Фурье (scaled DFT) на 64 точки. Предлагаемый способ позволяет сократить в таком преобразовании четыре операции.

Масштабированное преобразование на 64 точки содержит три стадии выполнения алгоритма radix-4. Благодаря переносу умножений на поворачивающие множители с первой и третьей стадий этого преобразования на вторую происходит сокращение числа операций. Для достижения аналогичного сокращения операций в масштабированных преобразованиях большего размера потребуются шесть стадий, или 2048 точек в преобразовании. Масштабированное преобразование на 2048 точек может быть использовано для уменьшения числа операций в обычном преобразовании Фурье на 8192 точки.

Таким образом, предложенный способ позволяет сократить число операций на 8 для алгоритмов БПФ с числом точек от 256 до 4096 и на 16 для алгоритмов с числом точек 8192 и выше.

Заключение

Рассмотрен теоретический способ сокращения числа операций в алгоритме БПФ, который позволяет сокращать операции в алгоритмах, использующих четырехточечные бабочки. В отдельных случаях (начиная с преобразования по 256 точкам) применение этого способа позволяет строить рекордные по количеству операций алгоритмы. При этом вопрос о теоретически возможном минимальном числе операций остается открытым.

С точки зрения практического применения нужно отметить, что во многих процессорных системах БПФ быстрее реализуются с помощью алгоритма radix-4 (меньше операций с памятью, чем у split-radix). Предложенный способ для алгоритма radix-4 дает более существенное сокращение числа операций, чем для алгоритма split-radix. Еще одним небольшим практическим плюсом предложенного способа является сокращение числа поворачивающих множителей, которые нужно хранить в памяти.

Список литературы / References

[1] **Cooley, J.W., Tukey, J.W.** An algorithm for the machine computation of the complex fourier series // Math. Computation. 1965. Vol. 19. P. 297–301.

[2] **Yavne, R.** An economical method for calculating the discrete Fourier transform // Proc. on AFIPS Fall Joint Comput. Conf. San Francisco, 1968. Vol. 33. P. 115–125.

[3] **Duhamel, P., Hollmann, H.** Split radix'FFT algorithm // Electronics Lett. 1984. Vol. 20. P. 14–16.

[4] **Van Buskirk, J.** Large FFT — NOT radix-2. `https://groups.google.com/forum/#!msg/comp.dsp/JLjf5KwuxYY/gcAiNhLnHF0J` (accessed 20.02.2018).

[5] **Johnson, S., Frigo, M.** A modified split-radix FFT with fewer arithmetic operations // IEEE Trans. Signal Proc. 2007. Vol. 55(1). P. 111–119.

[6] **Bernstein, D.** The tangent FFT // Proc. on Intern. Symp. on Appl. Algebra, Algebraic Algorithms and Error-Correcting Codes. Berlin; Heidelberg: Springer, 2007. P. 291–300.

[7] **Qadeer, S., Khan, M.Z. Ali, Sattar, S.A. et al.** A Radix-2 DIT FFT with reduced arithmetic complexity // Proc. on Intern. Conf. on Advances in Comput., Communicat. and Inform. (ICACCI, 2014). IEEE, 2014. P. 1892–1896.

[8] **Khan, M.Z. Ali, Qadeer, S.** A new variant of Radix-4 FFT // Proc. on Thirt. Intern. Conf. on Wireless and Optical Communicat. Networks (WOCN), 2016. IEEE, 2016. P. 1–4.

[9] **Korde, Ch., Malathi, P., Shelke, S.N., Sharma, M.** Design of FPGA Based Radix 4 FFT Processor using CORDIC // J. of Innovat. in Electronics and Communicat. Eng., 2015. Vol. 5. P. 56–62.

[10] **Князьков Д.Ю.** Эффективный расчет двумерного БПФ на однородном или гетерогенном вычислительном кластере // Программные системы: теория и приложения. 2017. Т. 8, № 1. С. 47–62.
Knyazkov, D.Yu. Effective computation of two-dimensional FFT on a homogeneous or heterogeneous cluster // Program Systems: Theory and Applications. 2017. Vol. 8, No. 1. P. 47–62. (In Russ.) `https://doi.org/10.25209/2079-3316-2017-8-1-47-62`

[11] **Burrus, C.** Fast Fourier Transforms (6×9 Version). `https://cnx.org/contents/Fujl6E8i@5.8:-1XsePkH@10/Preface-Fast-Fourier-Transform` (accessed 20.02.2018).

[12] **Altman, E.** DSP algorithms. `https://github.com/AltmanEA/DSP` (accessed 20.02.2018).

[13] **Kamar, I., Elcherif, Y.** Conjugate pair fast Fourier transform // Electronics Lett. 1989. Vol. 25(5). P. 324–325.

Поступила в редакцию 10 октября 2017 г.,
с доработки — 21 февраля 2018 г.

A method of reducing the number of operations in a fast Fourier transform algorithm

Altman, Evgeniy A.

Omsk State Transport University, Omsk, 644046, Russia

Corresponding author: Altman, Evgeniy A., e-mail: `AltmanEA@gmail.com`

The purpose of this work is studying of methods to reduce the number of calculations which must be performed to calculate the fast Fourier transform (FFT). These methods optimize the well-known FFT algorithms, such as Cooley–Tukey or split-radix.

The main idea of these methods is the reduction of operations due to their transfer on the graph of the algorithm through the stages of calculations This method is often used for explanation of split-radix or radix-4 algorithms. In our work, both area and distance of factor transfers are extended for 2 or more stages.

Using this idea, we found two methods of cutting down the number of multiplications. In both ways, the transfer is carried out for 4 points through 2 stages of the graph. In the first method, the transfer occurs in the direction of the calculations, in the second —

in the opposite direction. In both cases, the methods give a result if at the stage to which the transfer is carried out, there are non-trivial turning factors.

Applying these methods to radix-4 FFT algorithm allows reducing the number of calculations, but don't make them less than the number of calculations in the split-radix FFT. The record of minimal arithmetic operation for FFT algorithm was taken in applying these methods to the mixed algorithm consisting of split-radix and radix-4 stages. This approach saves 8 operations in FFT algorithm for 256 points over the modified split-radix FFT (tangent FFT).

Keywords: FFT, split radix, radix 4, arithmetic complexity.

Received 10 October 2017
Received in revised form 21 February 2018