CSE373: Data Structures and Algorithms

Divide and Conquer: The Fast Fourier Transform

Steve Tanimoto
Winter 2016

Fourier Transforms

- Joseph Fourier observed that any continuous function f(x) can be expressed as a sum of sine functions $\alpha \sin(\omega x + \phi)$, each one suitably amplified and shifted in phase.
- His object was to characterize the rate of heat transfer in materials.
- The transform named in his honor is a mathematical technique that can be used for data analysis, compression, synthesis in many fields.

Definition

- Let $f = f_0, \ldots, f_{n-1}$ be a vector of n complex numbers.
- The discrete Fourier transform of f is another vector of n complex numbers $F = F_0, \ldots, F_{n-1}$ each given by:

$$F_k = \sum_{j=0}^{n-1} f_j \, e^{\left(\frac{-2\pi i j k}{n}\right)}$$

- Here $i = \sqrt{-1}$ (the imaginary unit)

Nth roots of unity

- The factors $e^{\left(\frac{-2\pi i j k}{n}\right)}$ are nth roots of unity:
- They are solutions to the equation $x^n = 1$.
- Define $\omega = e^{\left(\frac{-2\pi i}{n}\right)}$
- This is a principal nth root of unity, meaning if $\omega^k = 1$ then k is a multiple of n.
- All the other factors are powers of ω. There are only n distinct powers that are relevant, when processing a vector of length n.

Complex exponentials as waves

- $e^{i\theta} = \cos\theta + i \sin\theta$
- $\mathrm{real}(e^{i\theta}) = \cos\theta$
- $\mathrm{imag}(e^{i\theta}) = \sin\theta$

The DFT as a Linear Transformation

$$\begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \end{bmatrix}$$

Computing the Discrete Fourier Transform

$$F_k = \sum_{j=0}^{n-1} f_j \, e^{\left(\frac{-2\pi\, i\, j\, k}{n}\right)}$$

Direct method:
Assume the complex exponentials are precomputed.

n^2 **complex multiplications**
n(n-1) **complex additions**

Divide and Conquer

- Divide the problem into smaller subproblems, solve them, and combine into the overall solution
- Solve subproblems recursively or with a direct method
- Combine the results of subproblems
- Apply dynamic programming, if possible

A Recursive Fast Fourier Transform

```
def FFT(f):
    n = len(f)
    if n==1: return [f[0]]                  # basis case
    F = n*[0]                               # Initialize results to 0.
    f_even = f[0::2]                        # Divide – even subproblem.
    f_odd  = f[1::2]                        #   "    - odd  subproblem
    F_even = FFT(f_even)                    # recursive call
    F_odd  = FFT(f_odd)                     #   "
    n2 = int(n/2)                           # Prepare to combine results
    for i in range(n2):
        twiddle = exp(-2*pi*1j*i/n)         # These could be precomputed
        oddTerm = F_odd[i] * twiddle        # Odd terms need an adjustment
        F[i]    = F_even[i] + oddTerm       # Compute a new term
        F[i+n2] = F_even[i] – oddTerm       # Compute one more new term
    return F
```


An N log N algorithm

Like in merge sort, in each recursive call, we divide the number of elements in half.

The number of levels of recursive calls is therefore $\log_2$ N.

When we combine subproblem results, we spend linear time.

Total time is bounded by c N log N.

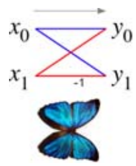
Unrolling the FFT

(more detailed views of how an FFT works)

Recursive FFT

FFT(n, $[a_0, a_1, \ldots, a_{n-1}]$):
 if n=1: return a_0
 F_{even} = FFT(n/2, $[a_0, a_2, \ldots, a_{n-2}]$)
 F_{odd} = FFT(n/2, $[a_1, a_3, \ldots, a_{n-1}]$)
 for k = 0 to n/2 – 1:
  $\omega^k = e^{2\pi i k/n}$
  $y^k = F_{even\ k} + \omega^k F_{odd\ k}$
  $y^{k+n/2} = F_{even\ k} - \omega^k F_{odd\ k}$
 return $[y_0, y_1, \ldots, y_{n-1}]$

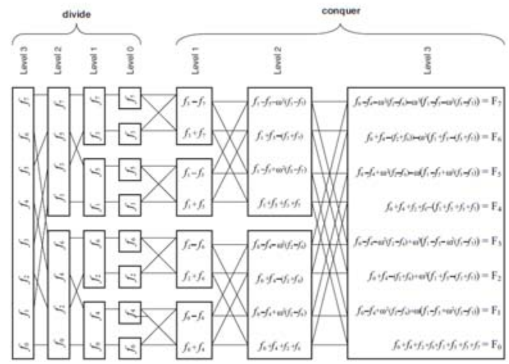
The Butterfly Step



A data-flow diagram connecting the inputs *x* (left) to the outputs *y* that depend on them (right) for a "butterfly" step of a radix-2 Cooley–Tukey FFT. This diagram resembles a butterfly.

http://en.wikipedia.org/wiki/Butterfly_diagram

Recursion Unrolled



Comments

- The FFT can be implemented:
- Iteratively, rather than recursively.
- In-place, (after putting the input in bit-reversed order)
- This diagram shows a radix-2, Cooley-Tukey, "decimation in time" FFT.
- Using a radix-4 implementation, the number of scalar multiplies and adds can be reduced by about 10 to 20 percent.

FFTs in Practice

There are many varieties of fast Fourier transforms. They typically depend on the fact that N is a composite number, such as a power of 2.

The radix need not be 2, and mixed radices can be used.

Formulations may be recursive or iterative, serial or parallel, etc.

There are also analog computers for Fourier transforms, such as those based on optical lens properties.

The Cooley-Tukey Fast Fourier Transform is often considered to be the most important numerical algorithm ever invented. This is the method typically referred to by the term "FFT."

The FFT can also be used for fast convolution, fast polynomial multiplication, and fast multiplication of large integers.