Fast Fourier Transform Algorithms (MIT IAP 2006)

Prof. Steven G. Johnson, MIT Dept. of Mathematics

10th February 2006

Fast Fourier transforms (FFTs), $O(N \log N)$ algorithms to compute a discrete Fourier transform (DFT) of size N, have been called one of the ten most important algorithms of the 20th century. They are what make Fourier transforms practical on a computer, and Fourier transforms (which express any function as a sum of pure sinusoids) are used in everything from solving partial differential equations to digital signal processing (e.g. MP3 compression) to multiplying large numbers (for computing π to 10^{12} decimal places). Although the applications are important and numerous, the FFT algorithms themselves reveal a surprisingly rich variety of mathematics that has been the subject of active research for 40 years, and into which this lecture will attempt to dip your toes. The DFT and its inverse are defined by the following relation between N inputs x_n and N outputs X_k (all *complex* numbers):

$$\text{DFT}(x_n):\;\; X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}, \tag{1}$$

$$\text{inverse DFT}(X_k):\;\; x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{+\frac{2\pi i}{N} nk} \tag{2}$$

where $i = \sqrt{-1}$, recalling Euler's identity that $e^{i\phi} = \cos\phi + i \sin\phi$. Each of the N DFT outputs $k = 0, \cdots, N-1$ is the sum of N terms, so evaluating this formula directly requires $O(N^2)$ operations.[1] The trick is to rearrange this computation to expose redundant calculations that we can factor out.

The most important FFT algorithm is called the Cooley-Tukey (C-T) algorithm, after the two authors who popularized it in 1965 (unknowingly re-inventing an algorithm known to Gauss in 1805). It works for any *composite* size $N = N_1 N_2$ by re-expressing the DFT of size N in terms of smaller DFTs of size N_1 and N_2 (which are themselves broken down, recursively, into smaller DFTs until the prime factors are reached). Effectively, C-T expresses the array $x_n$of length N as a "two-dimensional" array of size $N_1 \times N_2$ indexed by (n_1, n_2), so that $n = N_1 n_2 + n_1$ (where $n_{1,2} = 0, \cdots, N_{1,2} - 1$). Similarly, the output is expressed as a *transposed* 2d array, $N_2 \times N_1$ indexed by (k_2, k_1), so that $k = N_2 k_1 + k_2$. Substituted into the DFT above, this gives:

$$X_{N_2 k_1 + k_2} = \sum_{n_1=0}^{N_1-1} \left(\left\{ e^{-\frac{2\pi i}{N} n_1 k_2} \right\} \left[\sum_{n_2=0}^{N_2-1} e^{-\frac{2\pi i}{N_2} n_2 k_2} x_{N_1 n_2 + n_1} \right] \right) e^{-\frac{2\pi i}{N_1} n_1 k_1} \tag{3}$$

where we have used the fact that $e^{-2\pi i n_2 k_1} = 1$ (for any integers n_2 and k_1). Here, the outer sum is exactly a length-N_1 DFT of the $(\cdots)$ terms, one for each value of k_2; and the inner sum in $[\cdots]$ is a length-N_2 DFT, one for each value of n_1. The phase in the $\{\cdots\}$ is called the "twiddle factor" (honest). Assuming that N has small (bounded) prime factors, this algorithm requires $O(N \log N)$ operations when carried out recursively — the key savings coming from the fact that we have exposed a repeated calculation: the $[\cdots]$ DFTs need only be carried out *once* for *all* X_k outputs.

For a given N, there are many choices of factorizations (e.g. $12 = 3 \cdot 4$ and $4 \cdot 3$ give a different sequence of computations). Moreover, the transposition from input to output implies a data rearrangement process that can be accomplished in many ways. As a result, many different strategies for evaluating the C-T algorithm have been proposed (each with its own name), and the optimal approach is still a matter of active research. Commonly, either N_1 or N_2 is a small (bounded) constant factor, called the *radix*, and the approach is called decimation in time (DIT) for $N_1 =$ radix or frequency (DIF) for N_2. Textbook examples are typically radix-2 DIT (dividing x_n into two interleaved halves with each step), but serious implementations employ more sophisticated strategies.

The core of the DFT is the constant $\omega_N = e^{-\frac{2\pi i}{N}}$; because this is a primitive root of unity ($\omega_N^N = 1$), any exponent of ω_N is evaluated *modulo* N. That is, $\omega_N^m = \omega_N^r$ where r is the remainder when we divide m by N. A great body of number theory has been developed around such "modular arithmetic", and we can exploit it to develop FFT algorithms different from C-T. For example, Rader's algorithm (1968) allows us to compute $O(N \log N)$ FFTs of *prime* sizes N, by turning the DFT into a cyclic *convolution* of length $N-1$, which in turn is evaluated by (non-prime) FFTs. Given a_n and b_n ($n = 0, \cdots, N-1$), their

[1] Read "$O(N^2)$" as "roughly proportional, for large N." e.g. $15N^2 + 24N$ is $O(N^2)$. (Technically, I should really say $\Theta(N^2)$, but I'm going to get that formal.)

convolution c_n is defined by the sum

$$c_n = \sum_{m=0}^{N-1} a_m b_{n-m}, \tag{4}$$

where the convolution is *cyclic* if the $n - m$ subscript is "wrapped" periodically onto $0, \cdots, N-1$. This operation is central to digital filtering, differential equations, and other applications, and is evaluated in $O(N \log N)$ time by the *convolution theorem*: $c_n =$ inverse $\text{FFT}(\text{FFT}(a_n) \cdot \text{FFT}(b_n))$. Now, back to the FFT...

For prime N, there exists a *generator* g of the multiplicative group modulo N: this means that $g^p \mod$ N for $p = 0, \cdots, N-2$ produces all $n = 1, \cdots, N-1$ exactly once (but not in order!). Thus, we can write all non-zero n and k in the form $n = g^p$ and $k = g^{N-1-q}$ for some p and q, and rewrite the DFT as

$$X_0 = \sum_{n=0}^{N-1} x_n, \tag{5}$$

$$X_{k \neq 0} = X_{g^{N-1-q}} = x_0 + \sum_{p=0}^{N-2} \omega_N^{g^{p+N-1-q}} x_{g^p}, \tag{6}$$

where (6) is exactly the cyclic convolution of $a_p = x_{g^p}$ with $b_p = \omega_N^{g^{N-1-p}}$. This convolution has non-prime length $N-1$, and so we can evaluate it via the convolution theorem with FFTs in $O(N \log N)$ time (except for some unusual cases).

Many other FFT algorithms exist, from the "prime-factor algorithm" (1958) that exploits the Chinese remainder theorem for $\gcd(N_1, N_2) = 1$, to approaches that express the DFT as recursive reductions of polynomials or even multidimensional polynomials.

Further Reading

- D. N. Rockmore, "The FFT: An Algorithm the Whole Family Can Use," *Comput. Sci. Eng.* **2** (1), 60 (2000). Special issue on "top ten" algorithms of century. See: http://tinyurl.com/3wjvk and http://tinyurl.com/6l4jn

- "Fast Fourier transform," *Wikipedia: The Free Encyclopedia* (http://tinyurl.com/5c6f3). Edited by SGJ for correctness as of 10 Jan 2006 (along with subsidiary articles on C-T and other specific algorithms).

- "The Fastest Fourier Transform in the West," a free FFT implementation obviously named by arrogant MIT graduate students. http://www.fftw.org/

Homework Problems

Problem 1: Consider the Cooley-Tukey algorithm for $N = 2^{2^m}$, where instead of using a fixed radix we use radix-$\sqrt{N}$ — that is, at each step we write $N_1 = N_2 = \sqrt{N}$, recursively.

(a) Argue that the number of operations is still $O(N \log N)$.

Now, instead of the number of arithmetic operations, consider the number of *memory* operations on a system with a slow main memory that holds all of the data and a smaller, faster (idealized) *cache* that can hold some number C of the most recently accessed inputs/outputs (ignoring storage for the program itself, etc.). If the main memory is slowest part of the computer (this is typical), then the time will be dominated by the number of *cache misses*: the times that a number must be read into/out of cache from/to main memory. For a radix-2 recursive[2] C-T algorithm, the number $M(N)$ of cache misses can therefore be expressed by the following recurrence:

$$M(N) = \begin{cases} 2M(N/2) + O(N) & \text{if } N > C \\ O(N) & \text{if } N \leq C \end{cases} .$$

That is, if the problem fits in cache ($N < C$) then you just read the data into cache and no more reads are required. Otherwise, you need $M(N/2)$ twice for the two halves plus $O(N)$ to combine the two halves (with $N/2$ size-2 DFTs and twiddle factors).

(b) Argue that this recurrence leads to $M(N) = O(N \log \frac{N}{C})$ cache misses.

(c) Write down a similar $M'(N)$ recurrence for the radix-$\sqrt{N}$ algorithm. (Remember that each stage of the algorithm consists of three computations: do $\sqrt{N}$ length-$\sqrt{N}$ DFTs, multiply by N twiddle factors, then do another $\sqrt{N}$ length-$\sqrt{N}$ DFTs). Solve the recurrence to show that $M'(N) = O(???)$, and show that this is better when $\frac{N}{C} \to \infty$ than the radix-2 case $M(N)$ from (b).

(In fact, $M'(N)$ is provably optimal, but it doesn't become better in practice until N is very large. Since the algorithm itself does not depend on the cache size C, this is called an "optimal *cache-oblivious*" algorithm. See also http://tinyurl.com/77k3e)

[2]That is, we imagine the FFT is implemented in the most straightforward recursive fashion..."depth-first" traversal of the decomposition of N for you computer-science geeks. Actually, many implementations in practice use "breadth-first" traversal, which has much worse cache performance (at least in this idealized memory model).