Efficient FFT Algorithm and Programming Tricks*

Douglas L. Jones

This work is produced by OpenStax-CNX and licensed under the Creative Commons Attribution License 1.0†

Abstract

Many tricks and techniques have been developed to speed up the computation of FFTs. Significant reductions in computation time result from table lookup of twiddle factors, compiler-friendly or assembly-language programming, special hardware, and FFT algorithms for real-valued data. Higher-radix algorithms, fast bit-reversal, and special butterflies yield more modest but worthwhile savings.

The use of FFT algorithms\(^1\) such as the radix-2 decimation-in-time\(^2\) or decimation-in-frequency\(^3\) methods result in tremendous savings in computations when computing the discrete Fourier transform\(^4\). While most of the speed-up of FFTs comes from this, careful implementation can provide additional savings ranging from a few percent to several-fold increases in program speed.

1 Precompute twiddle factors

The twiddle factor\(^5\), or \(W_N^k = e^{-(i \frac{2\pi k}{N})}\), terms that multiply the intermediate data in the FFT algorithms\(^6\) consist of cosines and sines that each take the equivalent of several multiplies to compute. However, at most \(N\) unique twiddle factors can appear in any FFT or DFT algorithm. (For example, in the radix-2 decimation-in-time FFT\(^7\), only \(\frac{N}{2}\) twiddle factors \(\forall k, k = \{0, 1, 2, \ldots, \frac{N}{2} - 1\} : (W_N^k)\) are used.) These twiddle factors can be precomputed once and stored in an array in computer memory, and accessed in the FFT algorithm by table lookup. This simple technique yields very substantial savings and is almost always used in practice.

2 Compiler-friendly programming

On most computers, only some of the total computation time of an FFT is spent performing the FFT butterfly computations; determining indices, loading and storing data, computing loop parameters and other operations consume the majority of cycles. Careful programming that allows the compiler to generate efficient code can make a several-fold improvement in the run-time of an FFT. The best choice of radix in

\(^1\)http://creativecommons.org/licenses/by/1.0
\(^2\)"Overview of Fast Fourier Transform (FFT) Algorithms" <http://cnx.org/content/m12026/latest/>
\(^3\)"Decimation-in-Frequency (DIF) Radix-2 FFT" <http://cnx.org/content/m12018/latest/>
\(^4\)"DFT Definition and Properties" <http://cnx.org/content/m12019/latest/>
\(^5\)"Decimation-in-time (DIT) Radix-2 FFT" <http://cnx.org/content/m12016/latest/>
\(^6\)"Overview of Fast Fourier Transform (FFT) Algorithms" <http://cnx.org/content/m12026/latest/>
\(^7\)"Decimation-in-time (DIT) Radix-2 FFT" <http://cnx.org/content/m12016/latest/>
terms of program speed may depend more on characteristics of the hardware (such as the number of CPU registers) or compiler than on the exact number of computations. Very often the manufacturer's library codes are carefully crafted by experts who know intimately both the hardware and compiler architecture and how to get the most performance out of them, so use of well-written FFT libraries is generally recommended. Certain freely available programs and libraries are also very good. Perhaps the best current general-purpose library is the FFTW\textsuperscript{8} package; information can be found at http://www.fftw.org\textsuperscript{9}. A paper by Frigo and Johnson\textsuperscript{2} describes many of the key issues in developing compiler-friendly code.

3 Program in assembly language

While compilers continue to improve, FFT programs written directly in the assembly language of a specific machine are often several times faster than the best compiled code. This is particularly true for DSP microprocessors, which have special instructions for accelerating FFTs that compilers don't use. (I have myself seen differences of up to 26 to 1 in favor of assembly!) Very often, FFTs in the manufacturer's or high-performance third-party libraries are hand-coded in assembly. For DSP microprocessors, the codes developed by Meyer, Schuessler, and Schwarz\textsuperscript{4} are perhaps the best ever developed; while the particular processors are now obsolete, the techniques remain equally relevant today. Most DSP processors provide special instructions and a hardware design favoring the radix-2 decimation-in-time algorithm, which is thus generally fastest on these machines.

4 Special hardware

Some processors have special hardware accelerators or co-processors specifically designed to accelerate FFT computations. For example, AMI Semiconductor's\textsuperscript{10} Tocata\textsuperscript{11} ultra-low-power DSP microprocessor family, which is widely used in digital hearing aids, have on-chip FFT accelerators; it is always faster and more power-efficient to use such accelerators and whatever radix they prefer.

In a surprising number of applications, almost all of the computations are FFTs. A number of special-purpose chips are designed to specifically compute FFTs, and are used in specialized high-performance applications such as radar systems. Other systems, such as OFDM\textsuperscript{12}-based communications receivers, have special FFT hardware built into the digital receiver circuit. Such hardware can run many times faster, with much less power consumption, than FFT programs on general-purpose processors.

5 Effective memory management

Cache misses or excessive data movement between registers and memory can greatly slow down an FFT computation. Efficient programs such as the FFTW package\textsuperscript{13} are carefully designed to minimize these inefficiencies. In-place algorithms\textsuperscript{14} reuse the data memory throughout the transform, which can reduce cache misses for longer lengths.

6 Real-valued FFTs

FFTs of real-valued signals require only half as many computations as with complex-valued data. There are several methods for reducing the computation, which are described in more detail in Sorensen et al.\textsuperscript{3}

1. Use DFT symmetry properties\textsuperscript{15} to do two real-valued DFTs at once with one FFT program

http://cnx.org/content/m12021/1.6/
2. Perform one stage of the radix-2 decimation-in-time\textsuperscript{16} decomposition and compute the two length-$\frac{N}{2}$ DFTs using the above approach.
3. Use a direct real-valued FFT algorithm; see H.V. Sorensen et al.[3]

7 Special cases

Occasionally only certain DFT frequencies are needed, the input signal values are mostly zero, the signal is real-valued (as discussed above), or other special conditions exist for which faster algorithms can be developed. Sorensen and Burrus[5] describe slightly faster algorithms for pruned\textsuperscript{17} or zero-padded\textsuperscript{18} data. Goertzel’s algorithm\textsuperscript{19} is useful when only a few DFT outputs are needed. The running FFT\textsuperscript{20} can be faster when DFTs of highly overlapped blocks of data are needed, as in a spectrogram\textsuperscript{21}.

8 Higher-radix algorithms

Higher-radix algorithms, such as the radix-4\textsuperscript{22}, radix-8, or split-radix\textsuperscript{23} FFTs, require fewer computations and can produce modest but worthwhile savings. Even the split-radix FFT\textsuperscript{24} reduces the multiplications by only 33\% and the additions by a much lesser amount relative to the radix-2 FFTs\textsuperscript{25}; significant improvements in program speed are often due to implicit loop-unrolling\textsuperscript{26} or other compiler benefits than from the computational reduction itself!

9 Fast bit-reversal

Bit-reversing\textsuperscript{27} the input or output data can consume several percent of the total run-time of an FFT program. Several fast bit-reversal algorithms have been developed that can reduce this to two percent or less, including the method published by D.M.W. Evans[1].

10 Trade additions for multiplications

When FFTs first became widely used, hardware multipliers were relatively rare on digital computers, and multiplications generally required many more cycles than additions. Methods to reduce multiplications, even at the expense of a substantial increase in additions, were often beneficial. The prime factor algorithms\textsuperscript{28} and the Winograd Fourier transform algorithms\textsuperscript{29}, which required fewer multiplies and considerably more additions than the power-of-two-length algorithms\textsuperscript{30}, were developed during this period. Current processors generally have high-speed pipelined hardware multipliers, so trading multiplies for additions is often no longer beneficial. In particular, most machines now support single-cycle multiply-accumulate (MAC) operations, so balancing the number of multiplies and adds and combining them into single-cycle MACs generally results in the fastest code. Thus, the prime-factor and Winograd FFTs are rarely used today unless the application requires FFTs of a specific length.

\textsuperscript{16}"Decimation-in-time (DIT) Radix-2 FFT" [http://cnx.org/content/m12016/latest/]
\textsuperscript{17}http://www.fftw.org/pruned.html
\textsuperscript{18}"Spectrum Analysis Using the Discrete Fourier Transform": Section Zero-Padding [http://cnx.org/content/m12032/latest/#zeropad]
\textsuperscript{19}"Goertzel’s Algorithm" [http://cnx.org/content/m12024/latest/]
\textsuperscript{20}"Running FFT" [http://cnx.org/content/m12029/latest/]
\textsuperscript{21}"Short Time Fourier Transform" [http://cnx.org/content/m10570/latest/]
\textsuperscript{22}"Radix-4 FFT Algorithms" [http://cnx.org/content/m12027/latest/]
\textsuperscript{23}"Split-radix FFT Algorithms" [http://cnx.org/content/m12031/latest/]
\textsuperscript{24}"Split-radix FFT Algorithms" [http://cnx.org/content/m12031/latest/]
\textsuperscript{25}"Decimation-in-time (DIT) Radix-2 FFT" [http://cnx.org/content/m12016/latest/]
\textsuperscript{26}http://en.wikipedia.org/wiki/Loop_unrolling
\textsuperscript{27}"Decimation-in-time (DIT) Radix-2 FFT" [http://cnx.org/content/m12016/latest/]
\textsuperscript{28}"The Prime Factor Algorithm" [http://cnx.org/content/m12033/latest/]
\textsuperscript{29}"FFT's of prime length and Rader's conversion" [http://cnx.org/content/m12023/latest/]
\textsuperscript{30}"Power-of-two FFTs" [http://cnx.org/content/m12039/latest/]

http://cnx.org/content/m12021/1.6/
It is possible to implement a complex multiply with 3 real multiplies and 5 real adds rather than the usual 4 real multiplies and 2 real adds:

\[(C + iS)(X + iY) = CX - SY + i(CY + SX)\]

but alternatively

\[Z = C(X - Y)\]

\[D = C + S\]

\[E = C - S\]

\[CX - SY = EY + Z\]

\[CY + SX = DX - Z\]

In an FFT, \(D\) and \(E\) come entirely from the twiddle factors, so they can be precomputed and stored in a look-up table. This reduces the cost of the complex twiddle-factor multiply to 3 real multiplies and 3 real adds, or one less and one more, respectively, than the conventional 4/2 computation.

11 Special butterflies

Certain twiddle factors, namely \(W_0^N = 1, W_N^N, W_N^{2N}, W_N^{3N}, \) etc., can be implemented with no additional operations, or with fewer real operations than a general complex multiply. Programs that specially implement such butterflies in the most efficient manner throughout the algorithm can reduce the computational cost by up to several \(N\) multiplies and additions in a length-\(N\) FFT.

12 Practical Perspective

When optimizing FFTs for speed, it can be important to maintain perspective on the benefits that can be expected from any given optimization. The following list categorizes the various techniques by potential benefit; these will be somewhat situation- and machine-dependent, but clearly one should begin with the most significant and put the most effort where the pay-off is likely to be largest.

Methods to speed up computation of DFTs

- **Tremendous Savings** -
  a. FFT (\(\frac{N}{\log_2 N}\) savings)

- **Substantial Savings** - (2)
  a. Table lookup of cosine/sine
  b. Compiler tricks/good programming
  c. Assembly-language programming
  d. Special-purpose hardware
  e. Real-data FFT for real data (factor of 2)
  f. Special cases

- **Minor Savings** -
  a. radix-4\(^{31}\), split-radix\(^{32}\) (-10\% - +30\%)

\(^{31}\)“Radix-4 FFT Algorithms” <http://cnx.org/content/m12027/latest/>

\(^{32}\)“Split-radix FFT Algorithms” <http://cnx.org/content/m12031/latest/>
b. special butterflies

c. 3-real-multiplication complex multiply

d. Fast bit-reversal (up to 6%)

**NOTE:** On general-purpose machines, computation is only part of the total run time. Address generation, indexing, data shuffling, and memory access take up much or most of the cycles.

**NOTE:** A well-written radix-2\(^{33}\) program will run much faster than a poorly written split-radix\(^{34}\) program!

### References


\(^{33}\)“Decimation-in-time (DIT) Radix-2 FFT” <http://cnx.org/content/m12016/latest/>

\(^{34}\)“Split-radix FFT Algorithms” <http://cnx.org/content/m12031/latest/>