Implementing FFTs in Practice

Page by: Steven G. Johnson,Matteo Frigo

Books

This page is in 2 books:

- **Fast Fourier Transforms** (/contents/gua6b7go@22.1:ulXtQbN7@15
  - **Authors:** C. Sidney Burrus
  - **Revised:**Nov 18, 2012
  - Go to book (/contents/gua6b7go@22.1)
- **Fast Fourier Transforms (6x9 Version)** (/contents/Fujl6E8i@5.8:ulXtQbN7@15)
  - **Authors:** C. Sidney Burrus
  - **Revised:**Aug 16, 2012
  - Go to book (/contents/Fujl6E8i@5.8)

▶ Summary

by Steven G. Johnson (Department of Mathematics, Massachusetts Institute of Technology) and Matteo Frigo (Cilk Arts, Inc.)

Introduction

Although there are a wide range of fast Fourier transform (FFT) algorithms, involving a wealth of mathematics from number theory to polynomial algebras, the vast majority of FFT implementations in practice employ some variation on the Cooley-Tukey algorithm [link]. The Cooley-Tukey algorithm can be derived in two or three lines of elementary algebra. It can be implemented almost as easily, especially if only power-of-two sizes are desired; numerous popular textbooks list short FFT subroutines for power-of-two sizes, written in the language du jour. The implementation of the Cooley-Tukey algorithm, at least, would therefore seem to be a long-solved problem. In this chapter, however, we will argue that matters are not as straightforward as they might appear.

For many years, the primary route to improving upon the Cooley-Tukey FFT seemed to be reductions in the count of arithmetic operations, which often dominated the execution time prior to the ubiquity of fast floating-point hardware (at least on non-embedded processors). Therefore, great effort was expended towards finding new algorithms with reduced arithmetic counts [link], from Winograd's method to achieve $\Theta(n)$ multiplications[1] (at the cost of many more additions) [link], [link], [link], [link] to the split-radix variant on Cooley-Tukey that long achieved the lowest known total count of additions and multiplications for power-of-two sizes [link], [link], [link], [link], [link] (but was recently improved upon [link], [link]). The question of the minimum possible arithmetic count continues to be of fundamental theoretical interest—it is not even known whether better than $\Theta(n \log n)$ complexity is possible, since $\Omega(n \log n)$ lower bounds on the count of additions

have only been proven subject to restrictive assumptions about the algorithms [link], [link], [link]. Nevertheless, the difference in the number of arithmetic operations, for power-of-two sizes n, between the 1965 radix-2 Cooley-Tukey algorithm ($\sim 5n \log_2 n$ [link]) and the currently lowest-known arithmetic count ($\sim \frac{34}{9} n \log_2 n$ [link], [link]) remains only about 25%.

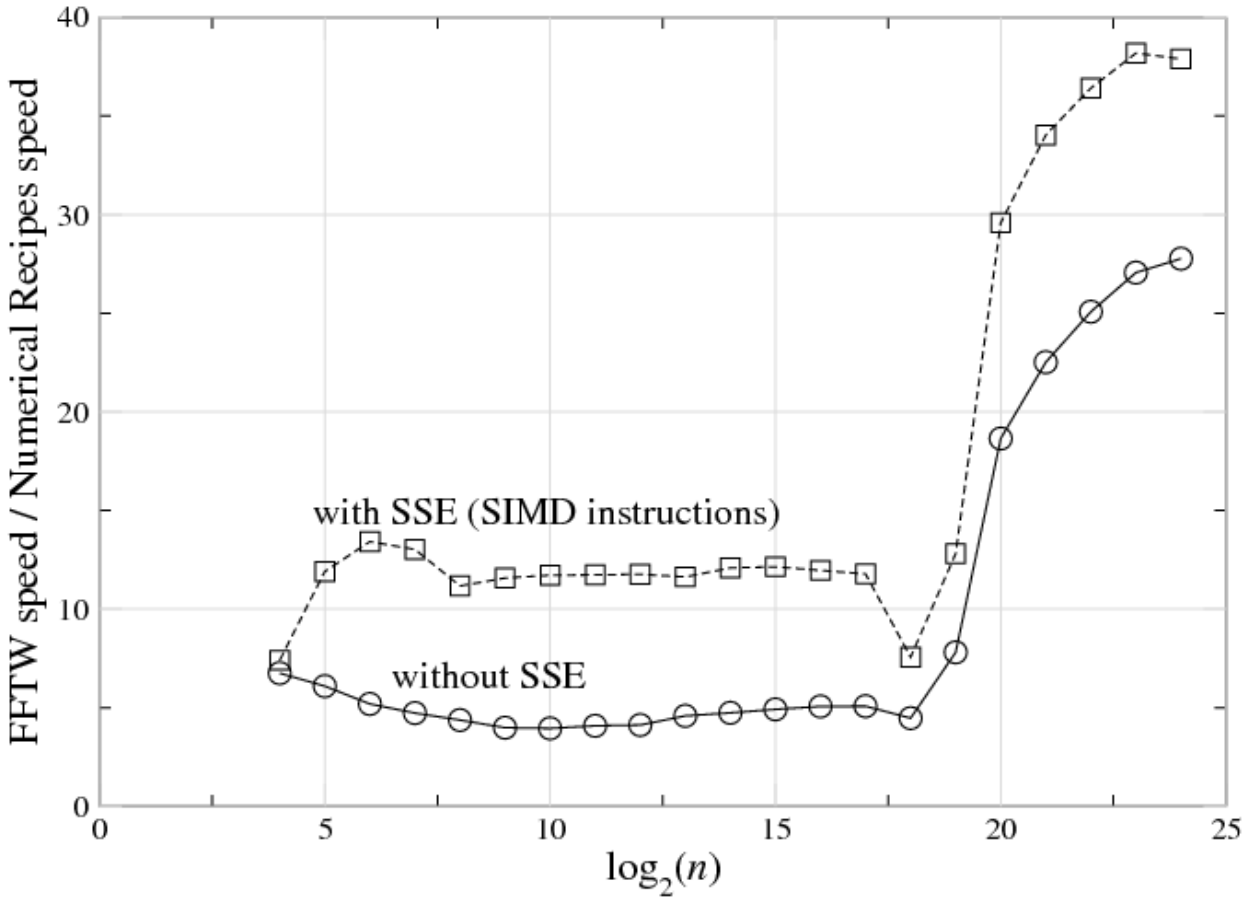

Figure 1. The ratio of speed (1/time) between a highly optimized FFT (FFTW 3.1.2 [link], [link]) and a typical textbook radix-2 implementation (*Numerical Recipes in C* [link]) on a 3 GHz Intel Core Duo with the Intel C compiler 9.1.043, for single-precision complex-data DFTs of size n, plotted versus $\log_2 n$. Top line (squares) shows FFTW with SSE SIMD instructions enabled, which perform multiple arithmetic operations at once (see section); bottom line (circles) shows FFTW with SSE disabled, which thus requires a similar number of arithmetic instructions to the textbook code. (This is not intended as a criticism of *Numerical Recipes*—simple radix-2 implementations are reasonable for pedagogy—but it illustrates the radical differences between straightforward and optimized implementations of FFT algorithms, even with similar arithmetic costs.) For $n \gtrsim 2^{19}$, the ratio increases because the textbook

code becomes much slower (this happens when the DFT size exceeds the level-2 cache).

And yet there is a vast gap between this basic mathematical theory and the actual practice—highly optimized FFT packages are often an order of magnitude faster than the textbook subroutines, and the internal structure to achieve this performance is radically different from the typical textbook presentation of the “same” Cooley-Tukey algorithm. For example, Figure plots the ratio of benchmark speeds between a highly optimized FFT [link], [link] and a typical textbook radix-2 implementation [link], and the former is faster by a factor of 5–40 (with a larger ratio as n grows). Here, we will consider some of the reasons for this discrepancy, and some techniques that can be used to address the difficulties faced by a practical high-performance FFT implementation.[2]

In particular, in this chapter we will discuss some of the lessons learned and the strategies adopted in the FFTW library. FFTW [link], [link] is a widely used free-software library that computes the discrete Fourier transform (DFT) and its various special cases. Its performance is competitive even with manufacturer-optimized programs [link], and this performance is **portable** thanks the structure of the algorithms employed, self-optimization techniques, and highly optimized kernels (FFTW's **codelets**) generated by a special-purpose compiler.

This chapter is structured as follows. First "Review of the Cooley-Tukey FFT", we briefly review the basic ideas behind the Cooley-Tukey algorithm and define some common terminology, especially focusing on the many degrees of freedom that the abstract algorithm allows to implementations. Next, in "Goals and Background of the FFTW Project", we provide some context for FFTW's development and stress that performance, while it receives the most publicity, is not necessarily the most important consideration in the implementation of a library of this sort. Third, in "FFTs and the Memory Hierarchy", we consider a basic theoretical model of the computer memory hierarchy and its impact on FFT algorithm choices: quite general considerations push implementations towards large radices and explicitly recursive structure. Unfortunately, general considerations are not sufficient in themselves, so we will explain in "Adaptive Composition of FFT Algorithms" how FFTW self-optimizes for particular machines by selecting its algorithm at runtime from a composition of simple algorithmic steps. Furthermore, "Generating Small FFT Kernels" describes the utility and the principles of automatic code generation used to produce the highly optimized building blocks of this composition, FFTW's codelets. Finally, we will briefly consider an important non-performance issue, in "Numerical Accuracy in FFTs".

Review of the Cooley-Tukey FFT

The (forward, one-dimensional) discrete Fourier transform (DFT) of an array $\mathbf{X}$ of n complex numbers is the array $\mathbf{Y}$ given by

$$\mathbf{Y}\left[k\right] = \sum_{\ell=0}^{n-1} \mathbf{X}\left[\ell\right] \omega_n^{\ell k} \ ,$$

where $0 \leq k < n$ and $\omega_n = \exp\left(-2\pi i/n\right)$ is a primitive root of unity. Implemented directly, Equation would require $\Theta\left(n^2\right)$ operations; fast Fourier transforms are $O(n \log n)$ algorithms to compute the same result. The most important FFT (and the one primarily used in FFTW) is known as the “Cooley-Tukey” algorithm, after the two authors who rediscovered and popularized it in 1965 [link], although it had been previously known as early as 1805 by Gauss as well as by later re-inventors [link]. The basic idea behind this FFT is that a DFT of a composite size $n = n_1 n_2$ can be re-expressed in terms of smaller DFTs of sizes n_1 and n_2—essentially, as a two-dimensional DFT of size $n_1 \times n_2$ where the output is **transposed**. The choices of factorizations of n, combined with the many different ways to implement the data re-orderings of the transpositions, have led to numerous implementation strategies for the Cooley-Tukey FFT, with many variants distinguished by their own names [link], [link]. FFTW implements a space of **many** such variants, as described in "Adaptive Composition of FFT Algorithms", but here we derive the basic algorithm, identify its key features, and outline some important historical variations and their relation to FFTW.

The Cooley-Tukey algorithm can be derived as follows. If n can be factored into $n = n_1 n_2$, Equation can be rewritten by letting $\ell = \ell_1 n_2 + \ell_2$ and $k = k_1 + k_2 n_1$. We then have:

$$\mathbf{Y}\left[k_1 + k_2 n_1\right] = \sum_{\ell_2=0}^{n_2-1} \left[\left(\sum_{\ell_1=0}^{n_1-1} \mathbf{X}\left[\ell_1 n_2 + \ell_2\right] \omega_{n_1}^{\ell_1 k_1} \right) \omega_n^{\ell_2 k_1} \right] \omega_{n_2}^{\ell_2 k_2} \ ,$$

where $k_{1,2} = 0, ..., n_{1,2} - 1$. Thus, the algorithm computes n_2 DFTs of size n_1 (the inner sum), multiplies the result by the so-called [link] **twiddle factors** $\omega_n^{\ell_2 k_1}$, and finally computes n_1 DFTs of size n_2 (the outer sum). This decomposition is then continued recursively. The literature uses the term **radix** to describe an n_1 or n_2 that is bounded (often constant); the small DFT of the radix is traditionally called a **butterfly**.

Many well-known variations are distinguished by the radix alone. A **decimation in time** (**DIT**) algorithm uses n_2 as the radix, while a **decimation in frequency** (**DIF**) algorithm uses n_1 as the radix. If multiple radices are used, e.g. for n composite but not a prime power, the algorithm is called **mixed radix**. A peculiar blending of radix 2 and 4 is called **split radix**, which was proposed

to minimize the count of arithmetic operations [link], [link], [link], [link], [link] although it has been superseded in this regard [link], [link]. FFTW implements both DIT and DIF, is mixed-radix with radices that are **adapted** to the hardware, and often uses much larger radices (e.g. radix 32) than were once common. On the other end of the scale, a "radix" of roughly $\sqrt{n}$ has been called a **four-step** FFT algorithm (or **six-step**, depending on how many transposes one performs) [link]; see "FFTs and the Memory Hierarchy" for some theoretical and practical discussion of this algorithm.

A key difficulty in implementing the Cooley-Tukey FFT is that the n_1 dimension corresponds to discontiguous inputs ℓ_1 in $\mathbf{X}$ but contiguous outputs k_1 in $\mathbf{Y}$, and vice-versa for n_2. This is a matrix transpose for a single decomposition stage, and the composition of all such transpositions is a (mixed-base) digit-reversal permutation (or **bit-reversal**, for radix 2). The resulting necessity of discontiguous memory access and data re-ordering hinders efficient use of hierarchical memory architectures (e.g., caches), so that the optimal execution order of an FFT for given hardware is non-obvious, and various approaches have been proposed.

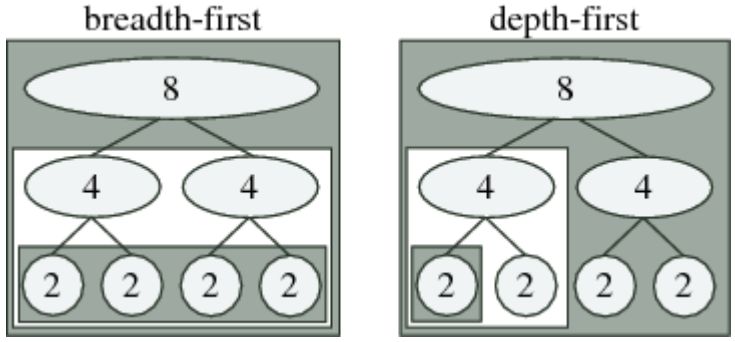

Figure 2. Schematic of traditional breadth-first (left) vs. recursive depth-first (right) ordering for radix-2 FFT of size 8: the computations for each nested box are completed before doing anything else in the surrounding box. Breadth-first computation performs all butterflies of a given size at once, while depth-first computation completes one subtransform entirely before moving on to the next (as in the algorithm below).

One ordering distinction is between recursion and iteration. As expressed above, the Cooley-Tukey algorithm could be thought of as defining a tree of smaller and smaller DFTs, as depicted in Figure; for example, a textbook radix-2 algorithm would divide size n into two transforms of size $n/2$, which are divided into four transforms of size $n/4$, and so on until a base case is reached (in principle, size 1). This might naturally suggest a recursive implementation in which the tree is traversed "depth-first" as in Figure(right) and the algorithm of Pre—one size $n/2$ transform is solved completely before processing the other one, and so on. However, most traditional FFT

implementations are non-recursive (with rare exceptions [link]) and traverse the tree “breadth-first” [link] as in Figure(left)—in the radix-2 example, they would perform n (trivial) size-1 transforms, then $n/2$ combinations into size-2 transforms, then $n/4$ combinations into size-4 transforms, and so on, thus making $\log_2 n$ passes over the whole array. In contrast, as we discuss in "Discussion", FFTW employs an explicitly recursive strategy that encompasses **both** depth-first and breadth-first styles, favoring the former since it has some theoretical and practical advantages as discussed in "FFTs and the Memory Hierarchy".

```
Y[0,...,n-1] ← recfft 2(n, X, ι):
IF n=1 THEN
    Y[0] ← X[0]
ELSE
    Y[0,...,n/2-1] ← recfft2 (n/2, X, 2ι)
    Y[n/2,...,n-1] ← recfft2 (n/2, X + ι, 2ι)
    FOR k_1 = 0 TO (n/2) - 1 DO
        t ← Y[k_1]
        Y[k_1] ← t + ω_n^{k_1} Y[k_1 + n/2]
        Y[k_1 + n/2] ← t - ω_n^{k_1} Y[k_1 + n/2]
    END FOR
END IF
```

NOT_CONVERTED_YET: caption

A depth-first recursive radix-2 DIT Cooley-Tukey FFT to compute a DFT of a power-of-two size $n = 2^m$. The input is an array $\mathbf{X}$ of length n with stride ι (i.e., the inputs are $\mathbf{X}[\ell\iota]$ for $\ell = 0,...,n-1$) and the output is an array $\mathbf{Y}$ of length n (with stride 1), containing the DFT of $\mathbf{X}$ [Equation 1]. $\mathbf{X} + \iota$ denotes the array beginning with $\mathbf{X}[\iota]$. This algorithm operates out-of-place, produces in-order output, and does not require a separate bit-reversal stage.

A second ordering distinction lies in how the digit-reversal is performed. The classic approach is a single, separate digit-reversal pass following or preceding the arithmetic computations; this approach is so common and so deeply embedded into FFT lore that many practitioners find it difficult to imagine an FFT without an explicit bit-reversal stage. Although this pass requires only $O(n)$ time [link], it can still be non-negligible, especially if the data is out-of-cache; moreover, it neglects the possibility that data reordering during the transform may improve memory locality. Perhaps the oldest alternative is the Stockham **auto-sort** FFT [link], [link], which transforms back and forth between two arrays with each butterfly, transposing one digit each time, and was popular to improve contiguity of access for vector computers [link]. Alternatively, an explicitly recursive style,

as in FFTW, performs the digit-reversal implicitly at the “leaves” of its computation when operating out-of-place (see section "Discussion"). A simple example of this style, which computes in-order output using an out-of-place radix-2 FFT without explicit bit-reversal, is shown in the algorithm of Pre [corresponding to Figure(right)]. To operate in-place with $O(1)$ scratch storage, one can interleave small matrix transpositions with the butterflies [link], [link], [link], [link], and a related strategy in FFTW [link] is briefly described by "Discussion".

Finally, we should mention that there are many FFTs entirely distinct from Cooley-Tukey. Three notable such algorithms are the **prime-factor algorithm** for $\gcd(n_1, n_2) = 1$ [link], along with Rader's [link] and Bluestein's [link], [link], [link] algorithms for prime n. FFTW implements the first two in its codelet generator for hard-coded n "Generating Small FFT Kernels" and the latter two for general prime n (sections "Plans for prime sizes" and "Goals and Background of the FFTW Project"). There is also the Winograd FFT [link], [link], [link], [link], which minimizes the number of multiplications at the expense of a large number of additions; this trade-off is not beneficial on current processors that have specialized hardware multipliers.

Goals and Background of the FFTW Project

The FFTW project, begun in 1997 as a side project of the authors Frigo and Johnson as graduate students at MIT, has gone through several major revisions, and as of 2008 consists of more than 40,000 lines of code. It is difficult to measure the popularity of a free-software package, but (as of 2008) FFTW has been cited in over 500 academic papers, is used in hundreds of shipping free and proprietary software packages, and the authors have received over 10,000 emails from users of the software. Most of this chapter focuses on performance of FFT implementations, but FFTW would probably not be where it is today if that were the only consideration in its design. One of the key factors in FFTW's success seems to have been its flexibility in addition to its performance. In fact, FFTW is probably the most flexible DFT library available:

- FFTW is written in portable C and runs well on many architectures and operating systems.
- FFTW computes DFTs in $O(n \log n)$ time for any length n. (Most other DFT implementations are either restricted to a subset of sizes or they become $\Theta\left(n^2\right)$ for certain values of n, for example when n is prime.)
- FFTW imposes no restrictions on the rank (dimensionality) of multi-dimensional transforms. (Most other implementations are limited to one-dimensional, or at most two- and three-dimensional data.)
- FFTW supports multiple and/or strided DFTs; for example, to transform a 3-component vector field or a portion of a multi-dimensional array. (Most implementations support only a single

DFT of contiguous data.)
- FFTW supports DFTs of real data, as well as of real symmetric/anti-symmetric data (also called discrete cosine/sine transforms).

Our design philosophy has been to first define the most general reasonable functionality, and then to obtain the highest possible performance without sacrificing this generality. In this section, we offer a few thoughts about why such flexibility has proved important, and how it came about that FFTW was designed in this way.

FFTW's generality is partly a consequence of the fact the FFTW project was started in response to the needs of a real application for one of the authors (a spectral solver for Maxwell's equations [link]), which from the beginning had to run on heterogeneous hardware. Our initial application required multi-dimensional DFTs of three-component vector fields (magnetic fields in electromagnetism), and so right away this meant: (i) multi-dimensional FFTs; (ii) user-accessible loops of FFTs of discontiguous data; (iii) efficient support for non-power-of-two sizes (the factor of eight difference between $n \times n \times n$ and $2n \times 2n \times 2n$ was too much to tolerate); and (iv) saving a factor of two for the common real-input case was desirable. That is, the initial requirements already encompassed most of the features above, and nothing about this application is particularly unusual.

Even for one-dimensional DFTs, there is a common misperception that one should always choose power-of-two sizes if one cares about efficiency. Thanks to FFTW's code generator (described in "Generating Small FFT Kernels"), we could afford to devote equal optimization effort to any n with small factors (2, 3, 5, and 7 are good), instead of mostly optimizing powers of two like many high-performance FFTs. As a result, to pick a typical example on the 3 GHz Core Duo processor of Figure, $n = 3600 = 2^4 \cdot 3^2 \cdot 5^2$ and $n = 3840 = 2^8 \cdot 3 \cdot 5$ both execute faster than $n = 4096 = 2^{12}$. (And if there are factors one particularly cares about, one can generate code for them too.)

One initially missing feature was efficient support for large prime sizes; the conventional wisdom was that large-prime algorithms were mainly of academic interest, since in real applications (including ours) one has enough freedom to choose a highly composite transform size. However, the prime-size algorithms are fascinating, so we implemented Rader's $O(n \log n)$ prime-n algorithm [link] purely for fun, including it in FFTW 2.0 (released in 1998) as a bonus feature. The response was astonishingly positive—even though users are (probably) never **forced** by their application to compute a prime-size DFT, it is rather inconvenient to always worry that collecting an unlucky number of data points will slow down one's analysis by a factor of a million. The prime-size algorithms are certainly slower than algorithms for nearby composite sizes, but in interactive data-analysis situations the difference between 1 ms and 10 ms means little, while educating users to

analysis situations the difference between 1 ms and 10 ms means little, while educating users to avoid large prime factors is hard.

Another form of flexibility that deserves comment has to do with a purely technical aspect of computer software. FFTW's implementation involves some unusual language choices internally (the FFT-kernel generator, described in "Generating Small FFT Kernels", is written in Objective Caml, a functional language especially suited for compiler-like programs), but its user-callable interface is purely in C with lowest-common-denominator datatypes (arrays of floating-point values). The advantage of this is that FFTW can be (and has been) called from almost any other programming language, from Java to Perl to Fortran 77. Similar lowest-common-denominator interfaces are apparent in many other popular numerical libraries, such as LAPACK [link]. Language preferences arouse strong feelings, but this technical constraint means that modern programming dialects are best hidden from view for a numerical library.

Ultimately, very few scientific-computing applications should have performance as their top priority. Flexibility is often far more important, because one wants to be limited only by one's imagination, rather than by one's software, in the kinds of problems that can be studied.

FFTs and the Memory Hierarchy

There are many complexities of computer architectures that impact the optimization of FFT implementations, but one of the most pervasive is the memory hierarchy. On any modern general-purpose computer, memory is arranged into a hierarchy of storage devices with increasing size and decreasing speed: the fastest and smallest memory being the CPU registers, then two or three levels of cache, then the main-memory RAM, then external storage such as hard disks.[3] Most of these levels are managed automatically by the hardware to hold the most-recently-used data from the next level in the hierarchy.[4] There are many complications, however, such as limited cache associativity (which means that certain locations in memory cannot be cached simultaneously) and cache lines (which optimize the cache for contiguous memory access), which are reviewed in numerous textbooks on computer architectures. In this section, we focus on the simplest abstract principles of memory hierarchies in order to grasp their fundamental impact on FFTs.

Because access to memory is in many cases the slowest part of the computer, especially compared to arithmetic, one wishes to load as much data as possible in to the faster levels of the hierarchy, and then perform as much computation as possible before going back to the slower memory devices. This is called **temporal locality**: if a given datum is used more than once, we arrange the computation so that these usages occur as close together as possible in time.

Understanding FFTs with an ideal cache

To understand temporal-locality strategies at a basic level, in this section we will employ an idealized model of a cache in a two-level memory hierarchy, as defined in [link]. This **ideal cache** stores Z data items from main memory (e.g. complex numbers for our purposes): when the processor loads a datum from memory, the access is quick if the datum is already in the cache (a **cache hit**) and slow otherwise (a **cache miss**, which requires the datum to be fetched into the cache). When a datum is loaded into the cache,[5] it must replace some other datum, and the ideal-cache model assumes that the optimal replacement strategy is used [link]: the new datum replaces the datum that will not be needed for the longest time in the future; in practice, this can be simulated to within a factor of two by replacing the least-recently used datum [link], but ideal replacement is much simpler to analyze. Armed with this ideal-cache model, we can now understand some basic features of FFT implementations that remain essentially true even on real cache architectures. In particular, we want to know the **cache complexity**, the number $Q(n; Z)$ of cache misses for an FFT of size n with an ideal cache of size Z, and what algorithm choices reduce this complexity.

First, consider a textbook radix-2 algorithm, which divides n by 2 at each stage and operates breadth-first as in Figure(left), performing all butterflies of a given size at a time. If $n > Z$, then each pass over the array incurs $\Theta(n)$ cache misses to reload the data, and there are $\log_2\ n$ passes, for $\Theta(n\ \log_2\ n)$ cache misses in total—no temporal locality at all is exploited!

One traditional solution to this problem is **blocking**: the computation is divided into maximal blocks that fit into the cache, and the computations for each block are completed before moving on to the next block. Here, a block of Z numbers can fit into the cache[6] (not including storage for twiddle factors and so on), and thus the natural unit of computation is a sub-FFT of size Z. Since each of these blocks involves $\Theta(Z\ \log\ Z)$ arithmetic operations, and there are $\Theta(n\ \log\ n)$ operations overall, there must be $\Theta\left(\frac{n}{Z}\ \log_Z\ n\right)$ such blocks. More explicitly, one could use a radix-Z Cooley-Tukey algorithm, breaking n down by factors of Z [or $\Theta(Z)$] until a size Z is reached: each stage requires n/Z blocks, and there are $\log_Z\ n$ stages, again giving $\Theta\left(\frac{n}{Z}\ \log_Z\ n\right)$ blocks overall. Since each block requires Z cache misses to load it into cache, the cache complexity Q_b of such a blocked algorithm is

$$Q_b\left(n; Z\right) = \Theta\left(n\ \log_Z\ n\right).$$

In fact, this complexity is rigorously **optimal** for Cooley-Tukey FFT algorithms [link], and immediately points us towards **large radices** (not radix 2!) to exploit caches effectively in FFTs.

However, there is one shortcoming of any blocked FFT algorithm: it is **cache-aware**, meaning that

However, there is one shortcoming of any blocked FFT algorithm: it is **cache aware**, meaning that the implementation depends explicitly on the cache size Z. The implementation must be modified (e.g. changing the radix) to adapt to different machines as the cache size changes. Worse, as mentioned above, actual machines have multiple levels of cache, and to exploit these one must perform multiple levels of blocking, each parameterized by the corresponding cache size. In the above example, if there were a smaller and faster cache of size $z < Z$, the size-Z sub-FFTs should themselves be performed via radix-z Cooley-Tukey using blocks of size z. And so on. There are two paths out of these difficulties: one is self-optimization, where the implementation automatically adapts itself to the hardware (implicitly including any cache sizes), as described in "Adaptive Composition of FFT Algorithms"; the other is to exploit **cache-oblivious** algorithms. FFTW employs both of these techniques.

The goal of cache-obliviousness is to structure the algorithm so that it exploits the cache without having the cache size as a parameter: the same code achieves the same asymptotic cache complexity regardless of the cache size Z. An **optimal cache-oblivious** algorithm achieves the **optimal** cache complexity (that is, in an asymptotic sense, ignoring constant factors). Remarkably, optimal cache-oblivious algorithms exist for many problems, such as matrix multiplication, sorting, transposition, and FFTs [link]. Not all cache-oblivious algorithms are optimal, of course—for example, the textbook radix-2 algorithm discussed above is “pessimal” cache-oblivious (its cache complexity is independent of Z because it always achieves the worst case!).

For instance, Figure(right) and the algorithm of Pre shows a way to obliviously exploit the cache with a radix-2 Cooley-Tukey algorithm, by ordering the computation depth-first rather than breadth-first. That is, the DFT of size n is divided into two DFTs of size $n/2$, and one DFT of size $n/2$ is **completely finished** before doing **any** computations for the second DFT of size $n/2$. The two subtransforms are then combined using $n/2$ radix-2 butterflies, which requires a pass over the array and (hence n cache misses if $n > Z$). This process is repeated recursively until a base-case (e.g. size 2) is reached. The cache complexity $Q_2\left(n; Z\right)$ of this algorithm satisfies the recurrence

$$Q_2\left(n; Z\right) = \begin{cases} n & n \leq Z \\ 2Q_2\left(n/2; Z\right) + \Theta\left(n\right) & \text{otherwise} \end{cases}.$$

The key property is this: once the recursion reaches a size $n \leq Z$, the subtransform fits into the cache and no further misses are incurred. The algorithm does not “know” this and continues subdividing the problem, of course, but all of those further subdivisions are in-cache because they are performed in the same depth-first branch of the tree. The solution of Equation is

$$Q_2\left(n; Z\right) = \Theta\left(n \log \left[n/Z\right]\right).$$

This is worse than the theoretical optimum $Q_b\left(n;Z\right)$ from Equation, but it is cache-oblivious (Z never entered the algorithm) and exploits at least **some** temporal locality.[7] On the other hand, when it is combined with FFTW's self-optimization and larger radices in "Adaptive Composition of FFT Algorithms", this algorithm actually performs very well until n becomes extremely large. By itself, however, the algorithm of Pre must be modified to attain adequate performance for reasons that have nothing to do with the cache. These practical issues are discussed further in "Cache-obliviousness in practice".

There exists a different recursive FFT that is **optimal** cache-oblivious, however, and that is the radix-$\sqrt{n}$ “four-step” Cooley-Tukey algorithm (again executed recursively, depth-first) [link]. The cache complexity Q_o of this algorithm satisfies the recurrence:

$$Q_o\left(n;Z\right)=\begin{cases}n & n\leq Z\\ 2\sqrt{n}Q_o\left(\sqrt{n};Z\right)+\Theta\left(n\right) & \text{otherwise}\end{cases}.$$

That is, at each stage one performs $\sqrt{n}$ DFTs of size $\sqrt{n}$ (recursively), then multiplies by the $\Theta(n)$ twiddle factors (and does a matrix transposition to obtain in-order output), then finally performs another $\sqrt{n}$ DFTs of size $\sqrt{n}$. The solution of Equation is $Q_o\left(n;Z\right)=\Theta\left(n\ \log_Z\ n\right)$, the same as the optimal cache complexity Equation!

These algorithms illustrate the basic features of most optimal cache-oblivious algorithms: they employ a recursive divide-and-conquer strategy to subdivide the problem until it fits into cache, at which point the subdivision continues but no further cache misses are required. Moreover, a cache-oblivious algorithm exploits all levels of the cache in the same way, so an optimal cache-oblivious algorithm exploits a multi-level cache optimally as well as a two-level cache [link]: the multi-level “blocking” is implicit in the recursion.

Cache-obliviousness in practice

Even though the radix-$\sqrt{n}$ algorithm is optimal cache-oblivious, it does not follow that FFT implementation is a solved problem. The optimality is only in an asymptotic sense, ignoring constant factors, $O(n)$ terms, etcetera, all of which can matter a great deal in practice. For small or moderate n, quite different algorithms may be superior, as discussed in "Memory strategies in FFTW". Moreover, real caches are inferior to an ideal cache in several ways. The unsurprising consequence of all this is that cache-obliviousness, like any complexity-based algorithm property, does not absolve one from the ordinary process of software optimization. At best, it reduces the amount of memory/cache tuning that one needs to perform, structuring the implementation to make

further optimization easier and more portable.

Perhaps most importantly, one needs to perform an optimization that has almost nothing to do with the caches: the recursion must be “coarsened” to amortize the function-call overhead and to enable compiler optimization. For example, the simple pedagogical code of the algorithm in Pre recurses all the way down to $n = 1$, and hence there are $\approx 2n$ function calls in total, so that every data point incurs a two-function-call overhead on average. Moreover, the compiler cannot fully exploit the large register sets and instruction-level parallelism of modern processors with an $n = 1$ function body.[8] These problems can be effectively erased, however, simply by making the base cases larger, e.g. the recursion could stop when $n = 32$ is reached, at which point a highly optimized hard-coded FFT of that size would be executed. In FFTW, we produced this sort of large base-case using a specialized code-generation program described in "Generating Small FFT Kernels".

One might get the impression that there is a strict dichotomy that divides cache-aware and cache-oblivious algorithms, but the two are not mutually exclusive in practice. Given an implementation of a cache-oblivious strategy, one can further optimize it for the cache characteristics of a particular machine in order to improve the constant factors. For example, one can tune the radices used, the transition point between the radix-$\sqrt{n}$ algorithm and the bounded-radix algorithm, or other algorithmic choices as described in "Memory strategies in FFTW". The advantage of starting cache-aware tuning with a cache-oblivious approach is that the starting point already exploits all levels of the cache to some extent, and one has reason to hope that good performance on one machine will be more portable to other architectures than for a purely cache-aware “blocking” approach. In practice, we have found this combination to be very successful with FFTW.

Memory strategies in FFTW

The recursive cache-oblivious strategies described above form a useful starting point, but FFTW supplements them with a number of additional tricks, and also exploits cache-obliviousness in less-obvious forms.

We currently find that the general radix-$\sqrt{n}$ algorithm is beneficial only when n becomes very large, on the order of $2^{20} \approx 10^6$. In practice, this means that we use at most a single step of radix-$\sqrt{n}$ (two steps would only be used for $n \gtrsim 2^{40}$). The reason for this is that the implementation of radix $\sqrt{n}$ is less efficient than for a bounded radix: the latter has the advantage that an entire radix butterfly can be performed in hard-coded loop-free code within local variables/registers, including the necessary permutations and twiddle factors.

Thus, for more moderate n, FFTW uses depth-first recursion with a bounded radix, similar in spirit to the algorithm of Pre but with much larger radices (radix 32 is common) and base cases (size 32

to this algorithm of FFTs but with much larger radices (radix 32 is common) and base cases (size 32 or 64 is common) as produced by the code generator of "Generating Small FFT Kernels". The self-optimization described in "Adaptive Composition of FFT Algorithms" allows the choice of radix and the transition to the radix-$\sqrt{n}$ algorithm to be tuned in a cache-aware (but entirely automatic) fashion.

For small n (including the radix butterflies and the base cases of the recursion), hard-coded FFTs (FFTW's **codelets**) are employed. However, this gives rise to an interesting problem: a codelet for (e.g.) $n = 64$ is ~ 2000 lines long, with hundreds of variables and over 1000 arithmetic operations that can be executed in many orders, so what order should be chosen? The key problem here is the efficient use of the CPU registers, which essentially form a nearly ideal, fully associative cache. Normally, one relies on the compiler for all code scheduling and register allocation, but but the compiler needs help with such long blocks of code (indeed, the general register-allocation problem is NP-complete). In particular, FFTW's generator knows more about the code than the compiler—the generator knows it is an FFT, and therefore it can use an optimal cache-oblivious schedule (analogous to the radix-$\sqrt{n}$ algorithm) to order the code independent of the number of registers [link]. The compiler is then used only for local “cache-aware” tuning (both for register allocation and the CPU pipeline).[9] As a practical matter, one consequence of this scheduler is that FFTW's machine-independent codelets are no slower than machine-specific codelets generated by an automated search and optimization over many possible codelet implementations, as performed by the SPIRAL project [link].

(When implementing hard-coded base cases, there is another choice because a loop of small transforms is always required. Is it better to implement a hard-coded FFT of size 64, for example, or an unrolled loop of four size-16 FFTs, both of which operate on the same amount of data? The former should be more efficient because it performs more computations with the same amount of data, thanks to the $\log n$ factor in the FFT's $n \log n$ complexity.)

In addition, there are many other techniques that FFTW employs to supplement the basic recursive strategy, mainly to address the fact that cache implementations strongly favor accessing consecutive data—thanks to cache lines, limited associativity, and direct mapping using low-order address bits (accessing data at power-of-two intervals in memory, which is distressingly common in FFTs, is thus especially prone to cache-line conflicts). Unfortunately, the known FFT algorithms inherently involve some non-consecutive access (whether mixed with the computation or in separate bit-reversal/transposition stages). There are many optimizations in FFTW to address this. For example, the data for several butterflies at a time can be copied to a small buffer before computing and then copied back, where the copies and computations involve more consecutive access than doing the computation directly in-place. Or, the input data for the subtransform can be copied from (discontiguous) input to (contiguous) output before performing the subtransform in-

place (see "Indirect plans"), rather than performing the subtransform directly out-of-place (as in algorithm 1). Or, the order of loops can be interchanged in order to push the outermost loop from the first radix step [the ℓ_2 loop in Equation] down to the leaves, in order to make the input access more consecutive (see "Discussion"). Or, the twiddle factors can be computed using a smaller look-up table (fewer memory loads) at the cost of more arithmetic (see "Numerical Accuracy in FFTs"). The choice of whether to use any of these techniques, which come into play mainly for moderate n ($2^{13} < n < 2^{20}$), is made by the self-optimizing planner as described in the next section.

Adaptive Composition of FFT Algorithms

As alluded to several times already, FFTW implements a wide variety of FFT algorithms (mostly rearrangements of Cooley-Tukey) and selects the “best” algorithm for a given n automatically. In this section, we describe how such self-optimization is implemented, and especially how FFTW's algorithms are structured as a composition of algorithmic fragments. These techniques in FFTW are described in greater detail elsewhere [link], so here we will focus only on the essential ideas and the motivations behind them.

An FFT algorithm in FFTW is a composition of algorithmic steps called a **plan**. The algorithmic steps each solve a certain class of **problems** (either solving the problem directly or recursively breaking it into sub-problems of the same type). The choice of plan for a given problem is determined by a **planner** that selects a composition of steps, either by runtime measurements to pick the fastest algorithm, or by heuristics, or by loading a pre-computed plan. These three pieces: problems, algorithmic steps, and the planner, are discussed in the following subsections.

The problem to be solved

In early versions of FFTW, the only choice made by the planner was the sequence of radices [link], and so each step of the plan took a DFT of a given size n, possibly with discontiguous input/output, and reduced it (via a radix r) to DFTs of size n/r, which were solved recursively. That is, each step solved the following problem: given a size n, an **input pointer** $\mathbf{I}$, an **input stride** ι, an **output pointer** $\mathbf{O}$, and an **output stride** o, it computed the DFT of $\mathbf{I}[\ell\iota]$ for $0 \leq \ell < n$ and stored the result in $\mathbf{O}[ko]$ for $0 \leq k < n$. However, we soon found that we could not easily express many interesting algorithms within this framework; for example, **in-place** ($\mathbf{I} = \mathbf{O}$) FFTs that do not require a separate bit-reversal stage [link], [link], [link], [link]. It became clear that the key issue was not the choice of algorithms, as we had first supposed, but the definition of the problem to be solved. Because only problems that can be expressed can be solved, the representation of a

solved. Because only problems that can be expressed can be solved, the representation of a problem determines an outer bound to the space of plans that the planner can explore, and therefore it ultimately constrains FFTW's performance.

The difficulty with our initial $(n, \mathbf{I}, \iota, \mathbf{O}, o)$ problem definition was that it forced each algorithmic step to address only a single DFT. In fact, FFTs break down DFTs into **multiple** smaller DFTs, and it is the **combination** of these smaller transforms that is best addressed by many algorithmic choices, especially to rearrange the order of memory accesses between the subtransforms. Therefore, we redefined our notion of a problem in FFTW to be not a single DFT, but rather a **loop** of DFTs, and in fact **multiple nested loops** of DFTs. The following sections describe some of the new algorithmic steps that such a problem definition enables, but first we will define the problem more precisely.

DFT problems in FFTW are expressed in terms of structures called I/O tensors,[10] which in turn are described in terms of ancillary structures called I/O dimensions. An **I/O dimension** d is a triple $d = (n, \iota, o)$, where n is a non-negative integer called the **length**, ι is an integer called the **input stride**, and o is an integer called the **output stride**. An **I/O tensor** $t = \{d_1, d_2, ..., d_\rho\}$ is a set of I/O dimensions. The non-negative integer $\rho = |t|$ is called the **rank** of the I/O tensor. A **DFT problem**, denoted by $\mathrm{dft}\ (\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$, consists of two I/O tensors $\mathbf{N}$ and $\mathbf{V}$, and of two **pointers** $\mathbf{I}$ and $\mathbf{O}$. Informally, this describes $|\mathbf{V}|$ nested loops of $|\mathbf{N}|$-dimensional DFTs with input data starting at memory location $\mathbf{I}$ and output data starting at $\mathbf{O}$.

For simplicity, let us consider only one-dimensional DFTs, so that $\mathbf{N} = \{(n, \iota, o)\}$ implies a DFT of length n on input data with stride ι and output data with stride o, much like in the original FFTW as described above. The main new feature is then the addition of zero or more “loops” $\mathbf{V}$. More formally, $\mathrm{dft}\ (\mathbf{N}, \{(n, \iota, o)\} \cup \mathbf{V}, \mathbf{I}, \mathbf{O})$ is recursively defined as a “loop” of n problems: for all $0 \leq k < n$, do all computations in $\mathrm{dft}\ (\mathbf{N}, \mathbf{V}, \mathbf{I} + k \cdot \iota, \mathbf{O} + k \cdot o)$. The case of multi-dimensional DFTs is defined more precisely elsewhere [link], but essentially each I/O dimension in $\mathbf{N}$ gives one dimension of the transform.

We call $\mathbf{N}$ the **size** of the problem. The **rank** of a problem is defined to be the rank of its size (i.e., the dimensionality of the DFT). Similarly, we call $\mathbf{V}$ the **vector size** of the problem, and the **vector rank** of a problem is correspondingly defined to be the rank of its vector size. Intuitively, the vector size can be interpreted as a set of “loops” wrapped around a single DFT, and we therefore refer to a single I/O dimension of $\mathbf{V}$ as a **vector loop**. (Alternatively, one can view the problem as describing a DFT over a $|\mathbf{V}|$-dimensional vector space.) The problem does not specify the order of execution of these loops, however, and therefore FFTW is free to choose the fastest or most convenient order.

DFT problem examples

A more detailed discussion of the space of problems in FFTW can be found in [link] , but a simple

A more detailed discussion of the space of problems in FFTW can be found in [link] , but a simple understanding can be gained by examining a few examples demonstrating that the I/O tensor representation is sufficiently general to cover many situations that arise in practice, including some that are not usually considered to be instances of the DFT.

A single one-dimensional DFT of length n, with stride-1 input $\mathbf{X}$ and output $\mathbf{Y}$, as in Equation, is denoted by the problem $\mathrm{dft}\ (\{(n, 1, 1)\}, \{\}, \mathbf{X}, \mathbf{Y})$ (no loops: vector-rank zero).

As a more complicated example, suppose we have an $n_1 \times n_2$ matrix $\mathbf{X}$ stored as n_1 consecutive blocks of contiguous length-n_2 rows (this is called **row-major** format). The in-place DFT of all the **rows** of this matrix would be denoted by the problem $\mathrm{dft}\ (\{(n_2, 1, 1)\}, \{(n_1, n_2, n_2)\}, \mathbf{X}, \mathbf{X})$: a length-$n_1$ loop of size-n_2 contiguous DFTs, where each iteration of the loop offsets its input/output data by a stride n_2. Conversely, the in-place DFT of all the **columns** of this matrix would be denoted by $\mathrm{dft}\ (\{(n_1, n_2, n_2)\}, \{(n_2, 1, 1)\}, \mathbf{X}, \mathbf{X})$: compared to the previous example, $\mathbf{N}$ and $\mathbf{V}$ are swapped. In the latter case, each DFT operates on discontiguous data, and FFTW might well choose to interchange the loops: instead of performing a loop of DFTs computed individually, the subtransforms themselves could act on n_2-component vectors, as described in "The space of plans in FFTW".

A size-1 DFT is simply a copy $Y[0] = X[0]$, and here this can also be denoted by $\mathbf{N} = \{\}$ (rank zero, a “zero-dimensional” DFT). This allows FFTW's problems to represent many kinds of copies and permutations of the data within the same problem framework, which is convenient because these sorts of operations arise frequently in FFT algorithms. For example, to copy n consecutive numbers from $\mathbf{I}$ to $\mathbf{O}$, one would use the rank-zero problem $\mathrm{dft}\ (\{\}, \{(n, 1, 1)\}, \mathbf{I}, \mathbf{O})$. More interestingly, the in-place **transpose** of an $n_1 \times n_2$ matrix $\mathbf{X}$ stored in row-major format, as described above, is denoted by $\mathrm{dft}\ (\{\}, \{(n_1, n_2, 1), (n_2, 1, n_1)\}, \mathbf{X}, \mathbf{X})$ (rank zero, vector-rank two).

The space of plans in FFTW

Here, we describe a subset of the possible plans considered by FFTW; while not exhaustive [link], this subset is enough to illustrate the basic structure of FFTW and the necessity of including the vector loop(s) in the problem definition to enable several interesting algorithms. The plans that we now describe usually perform some simple “atomic” operation, and it may not be apparent how these operations fit together to actually compute DFTs, or why certain operations are useful at all. We shall discuss those matters in "Discussion".

Roughly speaking, to solve a general DFT problem, one must perform three tasks. First, one must reduce a problem of arbitrary vector rank to a set of loops nested around a problem of vector rank 0, i.e., a single (possibly multi-dimensional) DFT. Second, one must reduce the multi-dimensional

0, i.e., a single (possibly multi-dimensional) DFT. Second, one must reduce the multi-dimensional DFT to a sequence of of rank-1 problems, i.e., one-dimensional DFTs; for simplicity, however, we do not consider multi-dimensional DFTs below. Third, one must solve the rank-1, vector rank-0 problem by means of some DFT algorithm such as Cooley-Tukey. These three steps need not be executed in the stated order, however, and in fact, almost every permutation and interleaving of these three steps leads to a correct DFT plan. The choice of the set of plans explored by the planner is critical for the usability of the FFTW system: the set must be large enough to contain the fastest possible plans, but it must be small enough to keep the planning time acceptable.

Rank-0 plans

The rank-0 problem $\mathrm{dft}\ (\{\}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ denotes a permutation of the input array into the output array. FFTW does not solve arbitrary rank-0 problems, only the following two special cases that arise in practice.

- When $|\mathbf{V}| = 1$ and $\mathbf{I} \neq \mathbf{O}$, FFTW produces a plan that copies the input array into the output array. Depending on the strides, the plan consists of a loop or, possibly, of a call to the ANSI C function `memcpy`, which is specialized to copy contiguous regions of memory.
- When $|\mathbf{V}| = 2, \mathbf{I} = \mathbf{O}$, and the strides denote a matrix-transposition problem, FFTW creates a plan that transposes the array in-place. FFTW implements the square transposition $\mathrm{dft}\ (\{\}, \{(n, \iota, o), (n, o, \iota)\}, \mathbf{I}, \mathbf{O})$ by means of the cache-oblivious algorithm from [link], which is fast and, in theory, uses the cache optimally regardless of the cache size (using principles similar to those described in the section "FFTs and the Memory Hierarchy"). A generalization of this idea is employed for non-square transpositions with a large common factor or a small difference between the dimensions, adapting algorithms from [link].

Rank-1 plans

Rank-1 DFT problems denote ordinary one-dimensional Fourier transforms. FFTW deals with most rank-1 problems as follows.

Direct plans

When the DFT rank-1 problem is “small enough” (usually, $n \leq 64$), FFTW produces a **direct plan** that solves the problem directly. These plans operate by calling a fragment of C code (a **codelet**) specialized to solve problems of one particular size, whose generation is described in "Generating Small FFT Kernels". More precisely, the codelets compute a loop ($|\mathbf{V}| \leq 1$) of small DFTs.

For problems of the form $\mathrm{dft}\ (\{(n, \iota, o)\}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ where $n = rm$, FFTW generates a plan that implements a radix-r Cooley-Tukey algorithm "Review of the Cooley-Tukey FFT". Both decimation-in-time and decimation-in-frequency plans are supported, with both small fixed radices (usually, $r \leq 64$) produced by the codelet generator "Generating Small FFT Kernels" and also arbitrary radices (e.g. radix-$\sqrt{n}$).

The most common case is a **decimation in time** (**DIT**) plan, corresponding to a **radix** $r = n_2$ (and thus $m = n_1$) in the notation of "Review of the Cooley-Tukey FFT": it first solves $\mathrm{dft}\ (\{(m, r \cdot \iota, o)\}, \mathbf{V} \cup \{(r, \iota, m \cdot o)\}, \mathbf{I}, \mathbf{O})$, then multiplies the output array $\mathbf{O}$ by the twiddle factors, and finally solves $\mathrm{dft}\ (\{(r, m \cdot o, m \cdot o)\}, \mathbf{V} \cup \{(m, o, o)\}, \mathbf{O}, \mathbf{O})$. For performance, the last two steps are not planned independently, but are fused together in a single "twiddle" codelet—a fragment of C code that multiplies its input by the twiddle factors and performs a DFT of size r, operating in-place on $\mathbf{O}$.

Plans for higher vector ranks

These plans extract a vector loop to reduce a DFT problem to a problem of lower vector rank, which is then solved recursively. Any of the vector loops of $\mathbf{V}$ could be extracted in this way, leading to a number of possible plans corresponding to different loop orderings.

Formally, to solve $\mathrm{dft}\ (\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$, where $\mathbf{V} = \{(n, \iota, o)\} \cup \mathbf{V}_1$, FFTW generates a loop that, for all k such that $0 \leq k < n$, invokes a plan for $\mathrm{dft}\ (\mathbf{N}, \mathbf{V}_1, \mathbf{I} + k \cdot \iota, \mathbf{O} + k \cdot o)$.

Indirect plans

Indirect plans transform a DFT problem that requires some data shuffling (or discontiguous operation) into a problem that requires no shuffling plus a rank-0 problem that performs the shuffling.

Formally, to solve $\mathrm{dft}\ (\mathbf{N}, \mathbf{V}, \mathbf{I}, \mathbf{O})$ where $|\mathbf{N}| > 0$, FFTW generates a plan that first solves $\mathrm{dft}\ (\{\}, \mathbf{N} \cup \mathbf{V}, \mathbf{I}, \mathbf{O})$, and then solves $\mathrm{dft}\ (\mathrm{copy{-}o}\ (\mathbf{N}), \mathrm{copy{-}o}\ (\mathbf{V}), \mathbf{O}, \mathbf{O})$. Here we define $\mathrm{copy{-}o}\ (t)$ to be the I/O tensor $\{(n, o, o) \mid (n, \iota, o) \in t\}$: that is, it replaces the input strides with the output strides. Thus, an indirect plan first rearranges/copies the data to the output, then solves the problem in place.

Plans for prime sizes

As discussed in "Goals and Background of the FFTW Project", it turns out to be surprisingly useful to be able to handle large prime n (or large prime factors). **Rader plans** implement the algorithm from [link] to compute one-dimensional DFTs of prime size in $\Theta(n \log n)$ time. **Bluestein plans** implement Bluestein's "chirp-z" algorithm, which can also handle prime n in $\Theta(n \log n)$ time [link], [link], [link]. **Generic plans** implement a naive $\Theta\left(n^2\right)$ algorithm (useful for $n \lesssim 100$).

Discussion

Although it may not be immediately apparent, the combination of the recursive rules in "The space of plans in FFTW" can produce a number of useful algorithms. To illustrate these compositions, we discuss three particular issues: depth- vs. breadth-first, loop reordering, and in-place transforms.

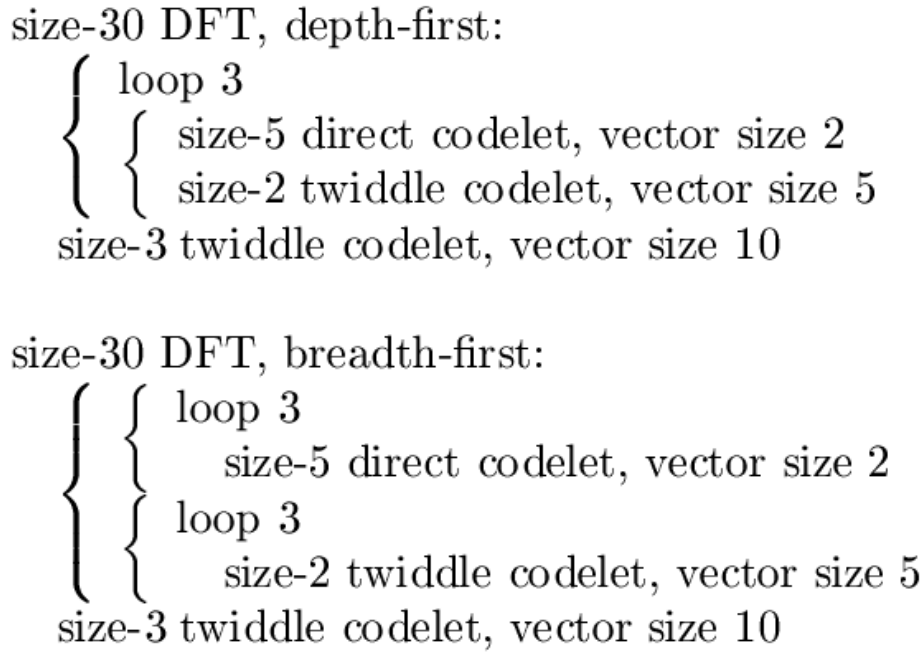

Figure 3. Two possible decompositions for a size-30 DFT, both for the arbitrary choice of DIT radices 3 then 2 then 5, and prime-size codelets. Items grouped by a "{" result from the plan for a single sub-problem. In the depth-first case, the vector rank was reduced to zero as per "Plans for higher vector ranks" before decomposing sub-problems, and vice-versa in the breadth-first case.

As discussed previously in sections "Review of the Cooley-Tukey FFT" and "Understanding FFTs with an ideal cache", the same Cooley-Tukey decomposition can be executed in either traditional

with an ideal cache", the same Cooley-Tukey decomposition can be executed in either traditional breadth-first order or in recursive depth-first order, where the latter has some theoretical cache advantages. FFTW is explicitly recursive, and thus it can naturally employ a depth-first order. Because its sub-problems contain a vector loop that can be executed in a variety of orders, however, FFTW can also employ breadth-first traversal. In particular, a 1d algorithm resembling the traditional breadth-first Cooley-Tukey would result from applying "Cooley-Tukey plans" to completely factorize the problem size before applying the loop rule "Plans for higher vector ranks" to reduce the vector ranks, whereas depth-first traversal would result from applying the loop rule before factorizing each subtransform. These two possibilities are illustrated by an example in Figure.

Another example of the effect of loop reordering is a style of plan that we sometimes call **vector recursion** (unrelated to "vector-radix" FFTs [link]). The basic idea is that, if one has a loop (vector-rank 1) of transforms, where the vector stride is smaller than the transform size, it is advantageous to push the loop towards the leaves of the transform decomposition, while otherwise maintaining recursive depth-first ordering, rather than looping "outside" the transform; i.e., apply the usual FFT to "vectors" rather than numbers. Limited forms of this idea have appeared for computing multiple FFTs on vector processors (where the loop in question maps directly to a hardware vector) [link]. For example, Cooley-Tukey produces a unit **input**-stride vector loop at the top-level DIT decomposition, but with a large **output** stride; this difference in strides makes it non-obvious whether vector recursion is advantageous for the sub-problem, but for large transforms we often observe the planner to choose this possibility.

In-place 1d transforms (with no separate bit reversal pass) can be obtained as follows by a combination DIT and DIF plans "Cooley-Tukey plans" with transposes "Rank-0 plans". First, the transform is decomposed via a radix-p DIT plan into a vector of p transforms of size qm, then these are decomposed in turn by a radix-q DIF plan into a vector (rank 2) of $p \times q$ transforms of size m. These transforms of size m have input and output at different places/strides in the original array, and so cannot be solved independently. Instead, an indirect plan "Indirect plans" is used to express the sub-problem as pq in-place transforms of size m, followed or preceded by an $m \times p \times q$ rank-0 transform. The latter sub-problem is easily seen to be m in-place $p \times q$ transposes (ideally square, i.e. $p = q$). Related strategies for in-place transforms based on small transposes were described in [link], [link], [link], [link]; alternating DIT/DIF, without concern for in-place operation, was also considered in [link], [link].

The FFTW planner

Given a problem and a set of possible plans, the basic principle behind the FFTW planner is straightforward: construct a plan for each applicable algorithmic step, time the execution of these

straightforward: construct a plan for each applicable algorithmic step, time the execution of these plans, and select the fastest one. Each algorithmic step may break the problem into subproblems, and the fastest plan for each subproblem is constructed in the same way. These timing measurements can either be performed at runtime, or alternatively the plans for a given set of sizes can be precomputed and loaded at a later time.

A direct implementation of this approach, however, faces an exponential explosion of the number of possible plans, and hence of the planning time, as n increases. In order to reduce the planning time to a manageable level, we employ several heuristics to reduce the space of possible plans that must be compared. The most important of these heuristics is **dynamic programming** [link]: it optimizes each sub-problem locally, independently of the larger context (so that the “best” plan for a given sub-problem is re-used whenever that sub-problem is encountered). Dynamic programming is not guaranteed to find the fastest plan, because the performance of plans is context-dependent on real machines (e.g., the contents of the cache depend on the preceding computations); however, this approximation works reasonably well in practice and greatly reduces the planning time. Other approximations, such as restrictions on the types of loop-reorderings that are considered "Plans for higher vector ranks", are described in [link].

Alternatively, there is an **estimate mode** that performs no timing measurements whatsoever, but instead minimizes a heuristic cost function. This can reduce the planner time by several orders of magnitude, but with a significant penalty observed in plan efficiency; e.g., a penalty of 20% is typical for moderate $n \lesssim 2^{13}$, whereas a factor of 2–3 can be suffered for large $n \gtrsim 2^{16}$ [link]. Coming up with a better heuristic plan is an interesting open research question; one difficulty is that, because FFT algorithms depend on factorization, knowing a good plan for n does not immediately help one find a good plan for nearby n.

Generating Small FFT Kernels

The base cases of FFTW's recursive plans are its **codelets**, and these form a critical component of FFTW's performance. They consist of long blocks of highly optimized, straight-line code, implementing many special cases of the DFT that give the planner a large space of plans in which to optimize. Not only was it impractical to write numerous codelets by hand, but we also needed to rewrite them many times in order to explore different algorithms and optimizations. Thus, we designed a special-purpose “FFT compiler” called **genfft** that produces the codelets automatically from an abstract description. genfft is summarized in this section and described in more detail by [link].

A typical codelet in FFTW computes a DFT of a small, fixed size n (usually, $n \leq 64$), possibly with the input or output multiplied by twiddle factors "Cooley-Tukey plans". Several other kinds of codelets can be produced by genfft , but we will focus here on this common case.

In principle, all codelets implement some combination of the Cooley-Tukey algorithm from Equation and/or some other DFT algorithm expressed by a similarly compact formula. However, a high-performance implementation of the DFT must address many more concerns than Equation alone suggests. For example, Equation contains multiplications by 1 that are more efficient to omit. Equation entails a run-time factorization of n, which can be precomputed if n is known in advance. Equation operates on complex numbers, but breaking the complex-number abstraction into real and imaginary components turns out to expose certain non-obvious optimizations. Additionally, to exploit the long pipelines in current processors, the recursion implicit in Equation should be unrolled and re-ordered to a significant degree. Many further optimizations are possible if the complex input is known in advance to be purely real (or imaginary). Our design goal for genfft was to keep the expression of the DFT algorithm independent of such concerns. This separation allowed us to experiment with various DFT algorithms and implementation strategies independently and without (much) tedious rewriting.

genfft is structured as a compiler whose input consists of the kind and size of the desired codelet, and whose output is C code. genfft operates in four phases: creation, simplification, scheduling, and unparsing.

In the **creation** phase, genfft produces a representation of the codelet in the form of a directed acyclic graph (**dag**). The dag is produced according to well-known DFT algorithms: Cooley-Tukey Equation, prime-factor [link], split-radix [link], [link], [link], [link], [link], and Rader [link]. Each algorithm is expressed in a straightforward math-like notation, using complex numbers, with no attempt at optimization. Unlike a normal FFT implementation, however, the algorithms here are evaluated symbolically and the resulting symbolic expression is represented as a dag, and in particular it can be viewed as a **linear network** [link] (in which the edges represent multiplication by constants and the vertices represent additions of the incoming edges).

In the **simplification** phase, genfft applies local rewriting rules to each node of the dag in order to simplify it. This phase performs algebraic transformations (such as eliminating multiplications by 1) and common-subexpression elimination. Although such transformations can be performed by a conventional compiler to some degree, they can be carried out here to a greater extent because genfft can exploit the specific problem domain. For example, two equivalent subexpressions can always be detected, even if the subexpressions are written in algebraically different forms, because all subexpressions compute linear functions. Also, genfft can exploit the property that **network transposition** (reversing the direction of every edge) computes the transposed linear operation

[link], in order to transpose the network, simplify, and then transpose back—this turns out to expose additional common subexpressions [link]. In total, these simplifications are sufficiently powerful to derive DFT algorithms specialized for real and/or symmetric data automatically from the complex algorithms. For example, it is known that when the input of a DFT is real (and the output is hence conjugate-symmetric), one can save a little over a factor of two in arithmetic cost by specializing FFT algorithms for this case—with genfft , this specialization can be done entirely automatically, pruning the redundant operations from the dag, to match the lowest known operation count for a real-input FFT starting only from the complex-data algorithm [link], [link]. We take advantage of this property to help us implement real-data DFTs [link], [link], to exploit machine-specific “SIMD” instructions "SIMD instructions" [link], and to generate codelets for the discrete cosine (DCT) and sine (DST) transforms [link], [link]. Furthermore, by experimentation we have discovered additional simplifications that improve the speed of the generated code. One interesting example is the elimination of negative constants [link]: multiplicative constants in FFT algorithms often come in positive/negative pairs, but every C compiler we are aware of will generate separate load instructions for positive and negative versions of the same constants.[11] We thus obtained a 10–15% speedup by making all constants positive, which involves propagating minus signs to change additions into subtractions or vice versa elsewhere in the dag (a daunting task if it had to be done manually for tens of thousands of lines of code).

In the **scheduling** phase, genfft produces a topological sort of the dag (a **schedule**). The goal of this phase is to find a schedule such that a C compiler can subsequently perform a good register allocation. The scheduling algorithm used by genfft offers certain theoretical guarantees because it has its foundations in the theory of cache-oblivious algorithms [link] (here, the registers are viewed as a form of cache), as described in "Memory strategies in FFTW". As a practical matter, one consequence of this scheduler is that FFTW's machine-independent codelets are no slower than machine-specific codelets generated by SPIRAL [link].

In the stock genfft implementation, the schedule is finally unparsed to C. A variation from [link] implements the rest of a compiler back end and outputs assembly code.

SIMD instructions

Unfortunately, it is impossible to attain nearly peak performance on current popular processors while using only portable C code. Instead, a significant portion of the available computing power can only be accessed by using specialized SIMD (single-instruction multiple data) instructions, which perform the same operation in parallel on a data vector. For example, all modern “x86” processors can execute arithmetic instructions on “vectors” of four single-precision values (SSE instructions) or two double-precision values (SSE2 instructions) at a time, assuming that the

operands are arranged consecutively in memory and satisfy a 16-byte alignment constraint. Fortunately, because nearly all of FFTW's low-level code is produced by genfft , machine-specific instructions could be exploited by modifying the generator—the improvements are then automatically propagated to all of FFTW's codelets, and in particular are not limited to a small set of sizes such as powers of two.

SIMD instructions are superficially similar to “vector processors”, which are designed to perform the same operation in parallel on an all elements of a data array (a “vector”). The performance of “traditional” vector processors was best for long vectors that are stored in contiguous memory locations, and special algorithms were developed to implement the DFT efficiently on this kind of hardware [link], [link]. Unlike in vector processors, however, the SIMD vector length is small and fixed (usually 2 or 4). Because microprocessors depend on caches for performance, one cannot naively use SIMD instructions to simulate a long-vector algorithm: while on vector machines long vectors generally yield better performance, the performance of a microprocessor drops as soon as the data vectors exceed the capacity of the cache. Consequently, SIMD instructions are better seen as a restricted form of instruction-level parallelism than as a degenerate flavor of vector parallelism, and different DFT algorithms are required.

The technique used to exploit SIMD instructions in genfft is most easily understood for vectors of length two (e.g., SSE2). In this case, we view a **complex** DFT as a pair of **real** DFTs:

$$\mathrm{DFT}(A + i \cdot B) = \mathrm{DFT}(A) + i \cdot \mathrm{DFT}(B) \ ,$$

where A and B are two real arrays. Our algorithm computes the two real DFTs in parallel using SIMD instructions, and then it combines the two outputs according to Equation. This SIMD algorithm has two important properties. First, if the data is stored as an array of complex numbers, as opposed to two separate real and imaginary arrays, the SIMD loads and stores always operate on correctly-aligned contiguous locations, even if the the complex numbers themselves have a non-unit stride. Second, because the algorithm finds two-way parallelism in the real and imaginary parts of a single DFT (as opposed to performing two DFTs in parallel), we can completely parallelize DFTs of any size, not just even sizes or powers of 2.

Numerical Accuracy in FFTs

An important consideration in the implementation of any practical numerical algorithm is numerical accuracy: how quickly do floating-point roundoff errors accumulate in the course of the

computation? Fortunately, FFT algorithms for the most part have remarkably good accuracy characteristics. In particular, for a DFT of length n computed by a Cooley-Tukey algorithm with finite-precision floating-point arithmetic, the **worst-case** error growth is $O(\log n)$ [link], [link] and the mean error growth for random inputs is only $O\left(\sqrt{\log n}\right)$ [link], [link]. This is so good that, in practical applications, a properly implemented FFT will rarely be a significant contributor to the numerical error.

The amazingly small roundoff errors of FFT algorithms are sometimes explained incorrectly as simply a consequence of the reduced number of operations: since there are fewer operations compared to a naive $O\left(n^2\right)$ algorithm, the argument goes, there is less accumulation of roundoff error. The real reason, however, is more subtle than that, and has to do with the **ordering** of the operations rather than their number. For example, consider the computation of only the output $Y[0]$ in the radix-2 algorithm of Pre, ignoring all of the other outputs of the FFT. $Y[0]$ is the sum of all of the inputs, requiring $n-1$ additions. The FFT does not change this requirement, it merely changes the order of the additions so as to re-use some of them for other outputs. In particular, this radix-2 DIT FFT computes $Y[0]$ as follows: it first sums the even-indexed inputs, then sums the odd-indexed inputs, then adds the two sums; the even- and odd-indexed inputs are summed recursively by the same procedure. This process is sometimes called **cascade summation**, and even though it still requires $n-1$ total additions to compute $Y[0]$ by itself, its roundoff error grows much more slowly than simply adding $X[0]$, $X[1]$, $X[2]$ and so on in sequence. Specifically, the roundoff error when adding up n floating-point numbers in sequence grows as $O(n)$ in the worst case, or as $O(\sqrt{n})$ on average for random inputs (where the errors grow according to a random walk), but simply reordering these n-1 additions into a cascade summation yields $O(\log n)$ worst-case and $O\left(\sqrt{\log n}\right)$ average-case error growth [link].

However, these encouraging error-growth rates **only** apply if the trigonometric "twiddle" factors in the FFT algorithm are computed very accurately. Many FFT implementations, including FFTW and common manufacturer-optimized libraries, therefore use precomputed tables of twiddle factors calculated by means of standard library functions (which compute trigonometric constants to roughly machine precision). The other common method to compute twiddle factors is to use a trigonometric recurrence formula—this saves memory (and cache), but almost all recurrences have errors that grow as $O(\sqrt{n})$, $O(n)$, or even $O\left(n^2\right)$ [link], which lead to corresponding errors in the FFT. For example, one simple recurrence is $e^{i(k+1)\theta} = e^{ik\theta}e^{i\theta}$, multiplying repeatedly by $e^{i\theta}$ to obtain a sequence of equally spaced angles, but the errors when using this process grow as $O(n)$ [link]. A common improved recurrence is $e^{i(k+1)\theta} = e^{ik\theta} + e^{ik\theta}\left(e^{i\theta} - 1\right)$, where the small quantity[1][2] $e^{i\theta} - 1 = \cos\left(\theta\right) - 1 + i\,\sin\left(\theta\right)$ is computed using $\cos\left(\theta\right) - 1 = -2\,\sin^2\left(\theta/2\right)$ [link]; unfortunately, the error using this method still grows as $O(\sqrt{n})$ [link], for worse than logarithmic

$O(\sqrt{n})$ [link], far worse than logarithmic.

There are, in fact, trigonometric recurrences with the same logarithmic error growth as the FFT, but these seem more difficult to implement efficiently; they require that a table of $\Theta(\log n)$ values be stored and updated as the recurrence progresses [link], [link]. Instead, in order to gain at least some of the benefits of a trigonometric recurrence (reduced memory pressure at the expense of more arithmetic), FFTW includes several ways to compute a much smaller twiddle table, from which the desired entries can be computed accurately on the fly using a bounded number (usually < 3) of complex multiplications. For example, instead of a twiddle table with n entries ω_n^k, FFTW can use two tables with $\Theta(\sqrt{n})$ entries each, so that ω_n^k is computed by multiplying an entry in one table (indexed with the low-order bits of k) by an entry in the other table (indexed with the high-order bits of k).

There are a few non-Cooley-Tukey algorithms that are known to have worse error characteristics, such as the “real-factor” algorithm [link], [link], but these are rarely used in practice (and are not used at all in FFTW). On the other hand, some commonly used algorithms for type-I and type-IV discrete cosine transforms [link], [link], [link] have errors that we observed to grow as $\sqrt{n}$ even for accurate trigonometric constants (although we are not aware of any theoretical error analysis of these algorithms), and thus we were forced to use alternative algorithms [link].

To measure the accuracy of FFTW, we compare against a slow FFT implemented in arbitrary-precision arithmetic, while to verify the correctness we have found the $O(n \log n)$ self-test algorithm of [link] very useful.

Concluding Remarks

It is unlikely that many readers of this chapter will ever have to implement their own fast Fourier transform software, except as a learning exercise. The computation of the DFT, much like basic linear algebra or integration of ordinary differential equations, is so central to numerical computing and so well-established that robust, flexible, highly optimized libraries are widely available, for the most part as free/open-source software. And yet there are many other problems for which the algorithms are not so finalized, or for which algorithms are published but the implementations are unavailable or of poor quality. Whatever new problems one comes across, there is a good chance that the chasm between theory and efficient implementation will be just as large as it is for FFTs, unless computers become much simpler in the future. For readers who encounter such a problem, we hope that these lessons from FFTW will be useful:

- Generality and portability should almost always come first.

- The number of operations, up to a constant factor, is less important than the order of operations.
- Recursive algorithms with large base cases make optimization easier.
- Optimization, like any tedious task, is best automated.
- Code generation reconciles high-level programming with low-level performance.

We should also mention one final lesson that we haven't discussed in this chapter: you can't optimize in a vacuum, or you end up congratulating yourself for making a slow program slightly faster. We started the FFTW project after downloading a dozen FFT implementations, benchmarking them on a few machines, and noting how the winners varied between machines and between transform sizes. Throughout FFTW's development, we continued to benefit from repeated benchmarks against the dozens of high-quality FFT programs available online, without which we would have thought FFTW was “complete” long ago.

Acknowledgements

SGJ was supported in part by the Materials Research Science and Engineering Center program of the National Science Foundation under award DMR-9400334; MF was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004. We are also grateful to Sidney Burrus for the opportunity to contribute this chapter, and for his continual encouragement—dating back to his first kind words in 1997 for the initial FFT efforts of two graduate students venturing outside their fields.

Footnotes

1 We employ the standard asymptotic notation of O for asymptotic upper bounds, Θ for asymptotic tight bounds, and Ω for asymptotic lower bounds [link].

2 We won't address the question of parallelization on multi-processor machines, which adds even greater difficulty to FFT implementation—although multi-processors are increasingly important, achieving good serial performance is a basic prerequisite for optimized parallel code, and is already hard enough!

3 A hard disk is utilized by “out-of-core” FFT algorithms for very large n [link], but these algorithms appear to have been largely superseded in practice by both the gigabytes of memory now common on personal computers and, for extremely large n, by algorithms

memory now common on personal computers and, for extremely large n, by algorithms for distributed-memory parallel computers.

4 This includes the registers: on current “x86” processors, the user-visible instruction set (with a small number of floating-point registers) is internally translated at runtime to RISC-like “μ-ops” with a much larger number of physical **rename registers** that are allocated automatically.

5 More generally, one can assume that a **cache line** of L consecutive data items are loaded into the cache at once, in order to exploit spatial locality. The ideal-cache model in this case requires that the cache be **tall**: $Z = \Omega\left(L^2\right)$ [link].

6 Of course, $O(n)$ additional storage may be required for twiddle factors, the output data (if the FFT is not in-place), and so on, but these only affect the n that fits into cache by a constant factor and hence do not impact cache-complexity analysis. We won't worry about such constant factors in this section.

7 This advantage of depth-first recursive implementation of the radix-2 FFT was pointed out many years ago by Singleton (where the “cache” was core memory) [link].

8 In principle, it might be possible for a compiler to automatically coarsen the recursion, similar to how compilers can partially unroll loops. We are currently unaware of any general-purpose compiler that performs this optimization, however.

9 One practical difficulty is that some “optimizing” compilers will tend to greatly re-order the code, destroying FFTW's optimal schedule. With GNU gcc, we circumvent this problem by using compiler flags that explicitly disable certain stages of the optimizer.

10 I/O tensors are unrelated to the tensor-product notation used by some other authors to describe FFT algorithms [link], [link].

11 Floating-point constants must be stored explicitly in memory; they cannot be embedded directly into the CPU instructions like integer “immediate” constants.

12 In an FFT, the twiddle factors are powers of ω_n, so θ is a small angle proportional to $1/n$ and $e^{i\theta}$ is close to 1.

Format:	Details	File Name:
Offline ZIP	An offline HTML copy of the content. Also includes XML, included media files, and other support files.	File not available