TITLE

Fast Fourier Transform

BYLINE

Franz Franchetti and Markus Püschel
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA
USA
{franzf, pueschel}@ece.cmu.edu

SYNONYMS

FFT, fast algorithm for the discrete Fourier transform (DFT)

DEFINITION

A fast Fourier transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) of an input vector. Efficient means that the FFT computes the DFT of an n-element vector in $O(n \log n)$ operations in contrast to the $O(n^2)$ operations required for computing the DFT by definition. FFTs exist for any vector length n and for real and higher-dimensional data. Parallel FFTs have been developed since the advent of parallel computing.

DISCUSSION

Introduction

The discrete Fourier transform (DFT) is a ubiquitous tool in science and engineering including in digital signal processing, communication, and high-performance computing. Applications include spectral analysis, image compression, interpolation, solving partial differential equations, and many other tasks.

Given n real or complex inputs $x_0, \ldots, x_{n-1}$, the DFT is defined as

$$y_k = \sum_{0 \le \ell < n} \omega_n^{k\ell} x_\ell, \quad 0 \le k < n, \tag{1}$$

with $\omega_n = \exp(-2\pi i/n)$, $i = \sqrt{-1}$. Stacking the x_ℓ and y_k into vectors $x = (x_0, \ldots, x_{n-1})^T$ and $y = (y_0, \ldots, y_{n-1})^T$ yields the equivalent form of a matrix-vector product:

$$y = \mathrm{DFT}_n\, x, \quad \mathrm{DFT}_n = [\omega_n^{k\ell}]_{0 \le k,\ell < n}. \tag{2}$$

Computing the DFT by its definition (2) requires $\Theta(n^2)$ many operations. The first fast Fourier transform algorithm (FFT) by Cooley and Tukey in 1965 reduced the runtime to $O(n \log(n))$ for two-powers n and marked the advent of digital signal processing. (It was later discovered that this FFT had already been derived and used by Gauss in the 19th century but was largely forgotten since then [9].) Since then, FFTs have been the topic of many publications and a wealth of different algorithms exist. This includes $O(n \log(n))$ algorithms for any input size n, as well as numerous variants optimized for various computing platform and

computation requirements. The by far most commonly used DFT is for two-power input sizes n, partly because these sizes permit the most efficient algorithms.

The first FFT explicitly optimized for parallelism was the Pease FFT published in 1968. Since then specialized FFT variants were developed with every new type of parallel computer. This includes FFTs for data flow machines, vector computers, shared and distributed memory multiprocessors, streaming and SIMD vector architectures, digital signal processing (DSP) processors, field-programmable gate arrays (FPGAs), and graphics processing units (GPUs). Just like Pease's FFT, these parallel FFTs are mainly for two-powers n and are adaptations of the same fundamental algorithm to structurally match the target platform.

On contemporary sequential and parallel machines it has become very hard to obtain high-performance DFT implementations. Beyond the choice of a suitable FFT, many other implementation issues have to be addressed. Up to the 1990s there were many public FFT implementations and proprietary FFT libraries available. Due to the code complexity inherent to fast implementations and the fast advances in processor design, today only a few competitive open source and vendor FFT libraries are available in the parallel computing space.

FFTs: Representation

Corresponding to the two different ways (1) and (2) of representing the DFT, FFTs are represented either as sequences of summations or as factorizations of the transform matrix DFT_n. The latter representation is adopted in Van Loan's seminal book [18] on FFTs and used in the following. To explain this representation assume as example that DFT_n in (2) can be factored into four matrices

$$\mathrm{DFT}_n = M_1 M_2 M_3 M_4. \tag{3}$$

Then (2) can be computed in four steps as

$$t = M_4 x,\ u = M_3 t,\ v = M_2 u,\ y = M_1 v.$$

If the matrices M_i are sufficiently sparse (have many zero entries) the operations count compared to a direct computation is decreased and (3) is called an FFT. For example, DFT_4 can be factorized as

$$\mathrm{DFT}_4 = \begin{bmatrix} 1 & & 1 & \\ & 1 & & 1 \\ 1 & & -1 & \\ & 1 & & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & i \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{bmatrix}, \tag{4}$$

where omitted values are zero. This example also demonstrates why the matrix-vector multiplications in (3) are not performed using a generic sparse linear algebra library, but, since the M_i are known and fixed, by a specialized program.

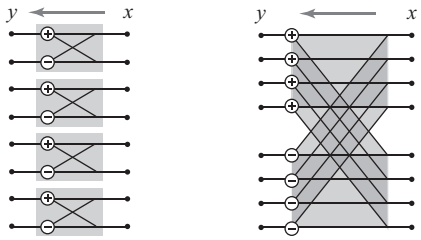
Conversely, every FFT can be written as in (3) (with varying numbers of factors). The matrices M_i in FFTs are not only sparse but also structured, as a glimpse on (4) illustrates. This structure can be efficiently expressed using a formalism based on matrix algebra and also clearly expresses the parallelism inherent to an FFT.

Matrix formalism and parallelism. The $n \times n$ identity matrix is denoted with I_n, and the *butterfly matrix* is a DFT of size 2:

$$\mathrm{DFT}_2 = \left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]. \tag{5}$$

The *Kronecker product* of matrices A and B is defined as

$$A \otimes B = [a_{k,\ell} B], \quad \text{for } A = [a_{k,\ell}].$$



(a) $y = (I_4 \otimes \mathrm{DFT}_2)x$ (b) $y = (\mathrm{DFT}_2 \otimes I_4)x$

Figure 1: Dataflow (right to left) of a block parallel and its "dual" vector parallel construct (figure from [5]).

It replaces every entry $a_{k,\ell}$ of A by the matrix $a_{k,\ell}B$. Most important for FFTs are the cases where A or B is the identity matrix. As examples consider

$$I_4 \otimes \mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 & & & & & & \\ 1 & -1 & & & & & & \\ & & 1 & 1 & & & & \\ & & 1 & -1 & & & & \\ & & & & 1 & 1 & & \\ & & & & 1 & -1 & & \\ & & & & & & 1 & 1 \\ & & & & & & 1 & -1 \end{bmatrix}, \quad \mathrm{DFT}_2 \otimes I_4 = \left[\begin{array}{cccc|cccc} 1 & & & & 1 & & & \\ & 1 & & & & 1 & & \\ & & 1 & & & & 1 & \\ & & & 1 & & & & 1 \\ \hline 1 & & & & -1 & & & \\ & 1 & & & & -1 & & \\ & & 1 & & & & -1 & \\ & & & 1 & & & & -1 \end{array}\right],$$

with the corresponding dataflows shown in Fig. 1. Note that the dataflows are from right to left to match the order of computation in (3). $I_4 \otimes \mathrm{DFT}_2$ clearly expresses block parallelism: four butterflies computing on contiguous subvectors; whereas $\mathrm{DFT}_2 \otimes I_4$ expresses vector parallelism: four butterflies operating on interleaved subvectors which is the same as one *vector* butterfly operating on *vectors* of length four as emphasized in Fig. 1(b). More precisely, consider the code for DFT_2 (i.e., $y = \mathrm{DFT}_2\, x$):

```
y[0] = x[0] + x[1];
y[1] = x[0] - x[1];
```

Then code for $\mathrm{DFT}_2 \otimes I_4$ is obtained by replacing every scalar operation by a four-way vector operation:

```
y[0:3] = x[0:3] + x[4:7];
y[4:7] = x[0:3] - x[4:7];
```

Here, `x[a:b]` denotes (Matlab or FORTRAN style) the subvector of `x` starting at `a` and ending at `b`. These examples illustrate how the tensor product captures parallelism. To summarize:

$$\text{block parallelism } (n \text{ blocks}): \qquad I_n \otimes A, \tag{6}$$

$$\text{vector parallelism } (n\text{-way}): \qquad A \otimes I_n, \tag{7}$$

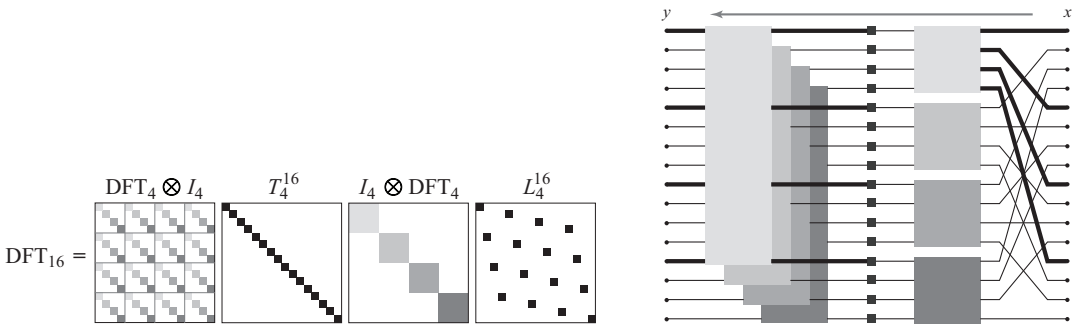
where A is any matrix.

The *stride permutation* matrix L^{mn}_m permutes the elements of the input vector as $in + j \mapsto jm + i,\ 0 \le i < m,\ 0 \le j < n$. If the vector x is viewed as an $n \times m$ matrix, stored in row-major order, then L^{mn}_m performs a transposition of this matrix. Further, if P is a permutation (matrix), then $A^P = P^{-1}AP$ is the *conjugation* of A with P.

Cooley-Tukey FFT. The fundamental algorithm at the core of the most important parallel FFTs derived in the literature is the general-radix decimation-in-time Cooley-Tukey type FFT expressed as

$$\mathrm{DFT}_n = (\mathrm{DFT}_k \otimes I_m) T^n_m (I_k \otimes \mathrm{DFT}_m) L^n_k, \quad n = km. \tag{8}$$

Here, k is called the radix and T^n_m is a diagonal matrix containing the *twiddle factors*. The algorithm factors the DFT into four factors as in (4), which shows the special case $n = 4 = 2 \times 2$. Two of the four factors in



(a) Matrix factorization (b) Data-flow graph

Figure 2: Cooley-Tukey FFT (8) for $16 = 4 \times 4$ as matrix factorization and as (complex) data-flow graph (from right to left). Some lines are bold to emphasize the strided access (figure from [5]).

(8) contain smaller DFTs; hence the algorithm is divide-and-conquer and has to be applied recursively. At each step of the recursion the radix is a degree of freedom. For two-power sizes $n = 2^\ell$, (8) is sufficient to recurse up to $n = 2$, which is computed by definition (5).

Fig. 2 shows the special case $16 = 4 \times 4$ as matrix factorization and as corresponding dataflow graph (again to be read from right to left). The smaller DFTs are represented as blocks with different shades of gray.

A straightforward implementation of (8) suggests four steps corresponding to the four factors, where two steps call smaller DFTs. However, to improve locality, the initial permutation L^n_k is usually not performed but interpreted as data access for the subsequent computation, and the twiddle diagonal T^n_m is fused with the subsequent DFTs. This strategy is chosen, for example, in the library FFTW 2.x and the code can be sketched as follows

```
void dft(int n, complex *y, complex *x) {
   int k = choose_factor(n);
   // t1 = (I_k tensor DFT_m)L(n,k)*x
   for(int i=0; i < k; ++i)
      dft_iostride(m, k, 1, t1 + m*i, x + m*i);
   // y = (DFT_k tensor I_m) diag(d(j))
   for(int i=0; i < m; ++i)
      dft_scaled(k, m, precomp_d[i], y + i, t1 + i);
}

// DFT variants needed
void dft_iostride(int n, int istride, int ostride, complex *y, complex *x);
void dft_scaled(int n, int stride, complex *d, complex *y, complex *x);
```

The DFT variants needed for the smaller DFTs are implemented similarly based on (8). There are many additional issues in implementing (8) to run fast on a non-parallel platform. The focus here is on mapping (8) to parallel platforms for two-power sizes n.

Parallel FFTs: Basic idea

The occurrence of tensor products in (8) shows that the algorithm has inherent block and vector parallelism as explained in (6) and (7). However, depending on the platform and for efficient mapping, the algorithm should exhibit one or both forms of parallelism throughout the computation to the extent possible. To achieve this, (8) can be formally manipulated using well-known matrix identities shown in Table 1.

The table makes clear that there is a virtually unlimited set of possible variants of (8), which also explains the large set of publications on FFTs. These variants hardly differ in operations count but in structure, which

$$
\begin{array}{rcl}
(BC)^\top & = & C^\top B^\top \\
(A \otimes B)^\top & = & A^\top \otimes B^\top \\
I_{mn} & = & I_m \otimes I_n, \\
A \otimes B & = & (A \otimes I_m)(I_n \otimes B) \\
I_n \otimes (BC) & = & (I_n \otimes B)(I_n \otimes C) \\
(BC) \otimes I_n & = & (B \otimes I_n)(C \otimes I_n) \\
A \otimes B & = & L^{mn}_n (B \otimes A) L^{mn}_m \\
\left(L^{mn}_m\right)^{-1} & = & L^{mn}_n \\
L^{kmn}_n & = & (L^{kn}_n \otimes I_m)(I_k \otimes L^{mn}_n) \\
L^{kmn}_{km} & = & (I_k \otimes L^{mn}_m)(L^{kn}_k \otimes I_m) \\
L^{kmn}_k & = & L^{kmn}_{km} L^{kmn}_{kn}
\end{array}
$$

Table 1: Formula identities to manipulate FFTs. A is $n \times n$, and B and C are $m \times m$. $A^\top$ is the transpose of A.

is crucial for parallelization. The remainder of this article introduces the most important parallel FFTs derived in the literature. All these FFTs can be derived from (8) using Table 1. The presentation is divided into iterative and recursive FFTs. Each FFT is visualized for size $n = 16$ in a form similar to (1) (and again from right to left) to emphasize block and vector parallelism. In these visualizations, the twiddle factors are dropped since they do not affect the dataflow and hence pose no structural problem for parallelization.

Iterative FFTs

The historically first FFTs that were developed and adapted to parallel platforms are iterative FFTs. These algorithms implement the DFT as a sequence of nested loops (usually three). The simplest are *radix-r* forms (usually $r = 2, 4, 8$), which require an FFT size of $n = r^\ell$; more complicated mixed-radix radix variants always exist. They all factor DFT_n into a product of ℓ matrices, each of which consists of a tensor product and twiddle factors. Iterative algorithms are obtained from (8) by recursive expansion, flattening the nested parentheses, and other identities in Table 1.

The most important iterative FFTs are discussed next, starting with the standard version, which is not optimized for parallelism but included for completeness. Note that the exact form of the twiddle factors differs in these FFTs, even though they are denoted with the same symbol.

Cooley-Tukey iterative FFT. The radix-r *iterative decimation-in-time FFT*

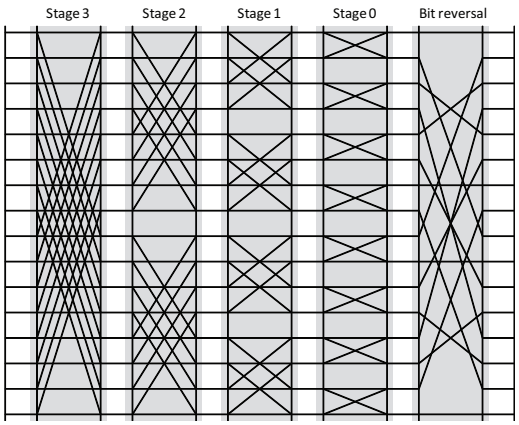
$$
\mathrm{DFT}_{r^\ell} = \left(\prod_{i=0}^{\ell-1} \left(I_{r^i} \otimes \mathrm{DFT}_r \otimes I_{r^{\ell-i-1}} \right) D^{r^\ell}_i \right) R^{r^\ell}_r, \tag{9}
$$

is the prototypical FFT algorithm and shown in Fig. 3. $R^{r^\ell}_r$ is the radix-r digit reversal permutation and the diagonal $D^{r^\ell}_i$ contains the twiddle factors in the ith stage. The radix-2 version is implemented by Numerical Recipes using a triple loop corresponding to the two tensor products (inner two loops) and the product (outer loop).

Formal transposition of (9) yields the *iterative decimation-in-frequency FFT*:

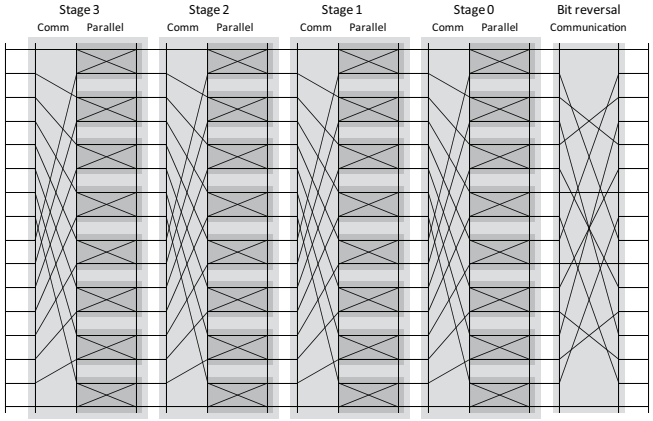
$$
\mathrm{DFT}_{r^\ell} = R^{r^\ell}_r \prod_{i=0}^{\ell-1} D^{r^\ell}_i \left(I_{r^{\ell-i-1}} \otimes \mathrm{DFT}_r \otimes I_{r^i} \right). \tag{10}
$$

Both (9) and (10) contain the bit reversal permutation $R^{r^\ell}_r$. The parallel and vector structure of the occurring butterflies depends on the stage. Thus, even though every stage is data parallel, the algorithm as is is neither



$$\Big((I_1 \otimes \mathrm{DFT}_2 \otimes I_8) D_0^{16}\Big)\Big((I_2 \otimes \mathrm{DFT}_2 \otimes I_4) D_1^{16}\Big)\Big((I_4 \otimes \mathrm{DFT}_2 \otimes I_2) D_2^{16}\Big)\Big((I_8 \otimes \mathrm{DFT}_2 \otimes I_1) D_3^{16}\Big) R_2^{16}$$

Figure 3: Iterative FFT (9) for $n = 2^4$ and $r = 2$.



$$\Big(L_2^{16}(I_8 \otimes \mathrm{DFT}_2) D_0^{16}\Big)\Big(L_2^{16}(I_8 \otimes \mathrm{DFT}_2) D_1^{16}\Big)\Big(L_2^{16}(I_8 \otimes \mathrm{DFT}_2) D_2^{16}\Big)\Big(L_2^{16}(I_8 \otimes \mathrm{DFT}_2) D_3^{16}\Big) R_2^{16}$$

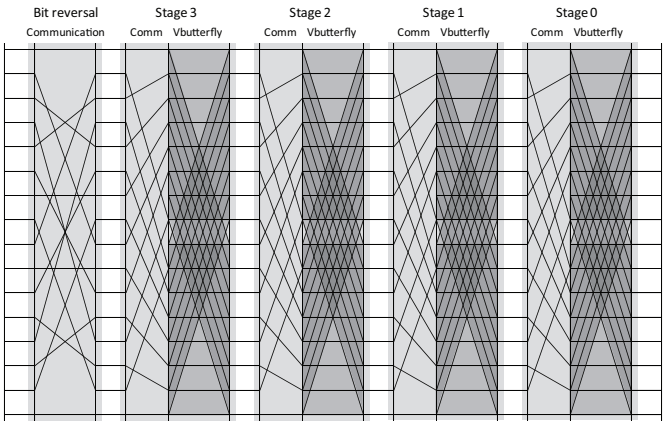
Figure 4: Pease FFT in (11) for $n = 2^4$ and $r = 2$.

well suited for machines that require block parallelism nor vector parallelism. For this reason very few parallel triple-loop implementations exist and compiler parallelization and vectorization tend not to succeed in producing any speed-up when targeting the triple-loop algorithm.

Pease FFT. A variant of (9) is the *Pease* FFT

$$\mathrm{DFT}_{r^\ell} = \left(\prod_{i=0}^{\ell-1} L_r^{r^\ell} \big(I_{r^{\ell-1}} \otimes \mathrm{DFT}_r\big) D_i^{r^\ell} \right) R_r^{r^\ell}, \tag{11}$$

shown in Fig. 4 for $r = 2$. It has constant geometry, i.e., the control flow is the same in each stage stage, and maximizes block parallelism by reducing the block sizes to r on which single butterflies are computed. However, the Pease FFT also requires the digit reversal permutation. Each stage of the Pease algorithm consists of the twiddle diagonal and a parallel butterfly block, followed by the same data exchange across parallel blocks specified through a stride permutation. The Pease FFT was originally developed for parallel computers, and its regular structure makes it a good choice for field-programmable gate arrays (FPGAs) or ASICs. Formal transposition of (11) yields a variant with the bit-reversal in the end.



$$R_2^{16}\Big(L_8^{16}D_0^{16}\big(\mathrm{DFT}_2\otimes I_8\big)\Big)\Big(L_8^{16}D_1^{16}\big(\mathrm{DFT}_2\otimes I_8\big)\Big)\Big(L_8^{16}D_2^{16}\big(\mathrm{DFT}_2\otimes I_8\big)\Big)\Big(L_8^{16}D_3^{16}\big(\mathrm{DFT}_2\otimes I_8\big)\Big)$$

Figure 5: Korn-Lambiotte FFT in (12) for $n = 2^4$ and $r = 2$. Vbutterfly = vector butterfly.

Korn-Lambiotte FFT. The *Korn-Lambiotte FFT* is given by

$$\mathrm{DFT}_{r^\ell} = R_r^{r^\ell}\left(\prod_{i=0}^{\ell-1} L_{r^{\ell-1}}^{r^\ell} D_i^{r^\ell}\big(\mathrm{DFT}_r\otimes I_{r^{\ell-1}}\big)\right), \tag{12}$$

and is the algorithm that is dual to the Pease FFT in the sense used in Fig. 1. Namely, it has also constant geometry, but maximizes vector parallelism as shown in Fig. 5 for $r = 2$. Each stage contains one vector butterfly operating on vectors of length n/r, and a twiddle diagonal. As last step it performs the digit reversal permutation. The Korn-Lambiotte FFT was developed for early vector computers. It is derived from the Pease algorithm through formal transposition followed by the translation of the tensor product from a parallel into a vector form.

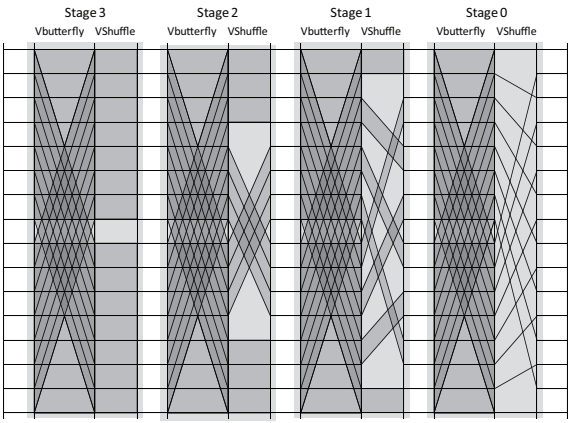
Stockham FFT. The *Stockham FFT*

$$\mathrm{DFT}_{r^\ell} = \prod_{i=0}^{\ell-1}\big(\mathrm{DFT}_r\otimes I_{r^{\ell-1}}\big)D_i^{r^\ell}\big(L_r^{r^{\ell-i}}\otimes I_{r^i}\big), \tag{13}$$

is *self-sorting*, i.e., it does not have a digit reversal permutation. It is shown Fig. 6 for $r = 2$. Like the Korn-Lambiotte FFT, it exhibits maximal vector parallelism but the permutations change across stages. Each of these permutations is a vector permutation, but the vector length increases by a factor of r in each stage (starting with 1). Thus, for most stages a sufficiently long vector length is achieved. The Stockham FFT was originally developed for vector computers. Its structure is also suitable for graphics processors (GPUs), and indeed most current GPU FFT libraries are based on the Stockham FFT. The formal transposition of (13) is also called Stockham FFT.

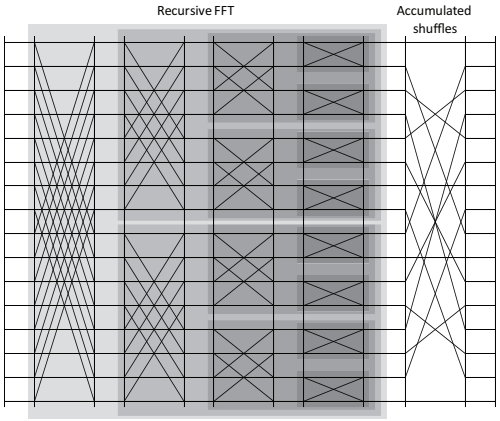
Recursive FFT Algorithms

The second class of Cooley-Tukey-based FFTs are recursive algorithms, which reduce a DFT of size $n = km$ into k DFTs of size m and m DFTs of size k. The advantage of recursive FFTs is better locality and hence better performance on computers with deep memory hierarchies. They also can be used as kernels for iterative algorithms. For parallelism, recursive algorithms are derived, for example, to maximize the block size for multicore platforms, or to obtain vector parallelism for a fixed vector length for platforms with SIMD vector extensions. The most important recursive algorithms are discussed next.



$$\Big((\mathrm{DFT}_2 \otimes I_8) D_0^{16} (L_2^2 \otimes I_8)\Big)\Big((\mathrm{DFT}_2 \otimes I_8) D_1^{16} (L_2^4 \otimes I_4)\Big)\Big((\mathrm{DFT}_2 \otimes I_8) D_2^{16} (L_2^8 \otimes I_2)\Big)\Big((\mathrm{DFT}_2 \otimes I_8) D_3^{16} (L_2^{16} \otimes I_1)\Big)$$

Figure 6: Stockham FFT in (13) for $n = 2^4$ and $r = 2$. Vbutterfly = vector butterfly, VShuffle = vector shuffle.



$$(\mathrm{DFT}_2 \otimes I_8) T_8^{16} \Big(I_2 \otimes \Big((\mathrm{DFT}_2 \otimes I_4) T_4^8 \big(I_2 \otimes \big((\mathrm{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathrm{DFT}_2) L_2^4\big)\big) L_2^8\Big)\Big) L_2^{16}$$

Figure 7: Recursive radix-2 decimation-in-time FFT for $n = 2^4$.

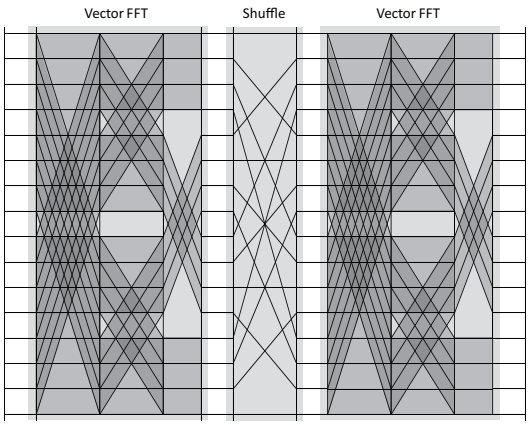
Recursive Cooley-Tukey FFT. The recursive, general-radix decimation-in-time Cooley-Tukey FFT was shown before in (8). Typically, k is chosen to be small, with values up to 64. If (8) is applied to $n = r^\ell$ recursively with $k = r$ the algorithm is called radix-r decimation-in time FFT. As explained before, the initial permutation is usually not performed but propagated as data access into the smaller DFTs. For radix-2 the algorithm is shown in Fig. 7. Note that the dataflow is equal to Fig. 3, but the order of computation is different as emphasized by the shading.

Formal transposition of (8) yields the *recursive decimation-in-frequency FFT*

$$\mathrm{DFT}_n = L_m^n (I_k \otimes \mathrm{DFT}_m) T_m^n (\mathrm{DFT}_k \otimes I_m), \quad n = km. \tag{14}$$

Recursive application of (8) and (14) eventually leads to prime sizes k and m, which are handled by a special prime-size FFT. For two-powers n the butterfly matrix DFT_2 terminates the recursion.

The implementation of (8) and (14) is more involved than the implementation of iterative algorithms, in particular in the mixed-radix case. The divide-and-conquer nature of (8) and (14) makes them good choices for machines with memory hierarchies, as at some recursion level the working set will be small enough to fit into a certain cache level, a property sometimes called *cache oblivious*. Both (8) and (14) contain both



$$\Big(\big((\mathrm{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathrm{DFT}_2) L_2^4\big) \otimes I_4\Big) L_4^{16} T_4^{16} \Big(\big((\mathrm{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathrm{DFT}_2) L_2^4\big) \otimes I_4\Big)$$

Figure 8: Four-step FFT for $n = 2^4$ and $k = m = \sqrt{n} = 4$.

vector and parallel blocks and stride permutations. Thus, despite their inherent data parallelism, they are not ideal for either parallel or vector implementations. The following variants address this problem.

Four-step FFT. The *four-step FFT* is given by

$$\mathrm{DFT}_n = (\mathrm{DFT}_k \otimes I_m) T_m^n L_k^n (\mathrm{DFT}_m \otimes I_k), \quad n = km, \tag{15}$$

and shown in Fig. 8. It is built from two stages of vector FFTs, the twiddle diagonal and a transposition. Typically, $k, m \approx \sqrt{n}$ is chosen (also called "square root decomposition"). Then, (15) results in the longest possible vector operations except for the stride permutation in the middle.

The Four Step FFT was originally developed for vector computers and the stride permutation (or transposition) was originally implemented explicitly while the smaller FFTs were expanded with some other FFT—typically iterative. The transposition can be implemented efficiently using blocking techniques. (15) can be a good choice on parallel machines that execute operations on long vectors well and on which the overhead of a transposition is not too high. Examples includes vector computers and machines with streaming memory access like GPUs.

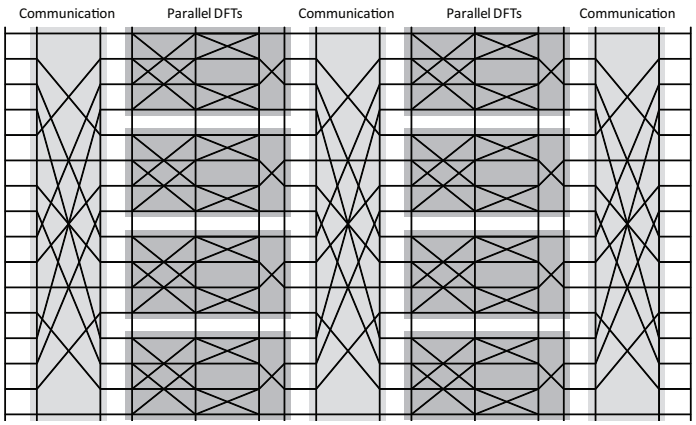
Six-step FFT. The *six-step FFT* is given by

$$\mathrm{DFT}_n = L_k^n (I_m \otimes \mathrm{DFT}_k) L_m^n T_m^n (I_k \otimes \mathrm{DFT}_m) L_k^n, \quad n = km, \tag{16}$$

and shown in Fig. 9. It is built from two stages of parallel butterfly blocks, the twiddle diagonal, and three global transpositions (all-to-all data exchanges). (16) was originally developed for distributed memory machines and out-of-core computation. Typically, $k, m \approx \sqrt{n}$ is chosen to maximize parallelism. The transposition was originally implemented explicitly as all-to-all communication while the smaller FFTs were expanded with some other FFT algorithm—typically iterative. As in (15), the required matrix transposition can be blocked for more efficient data movement. (16) can be a good choice on parallel machines that have multiple memory spaces and require explicit data movement, like message passing, offloading to accelerators (GPUs and FPGAs), and out-of-core computation.

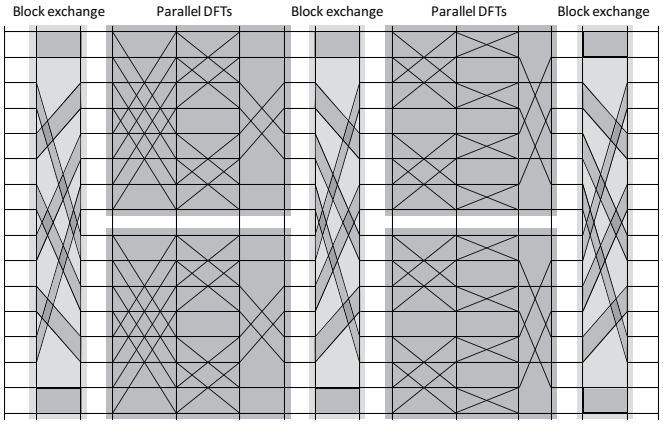
Multicore FFT. The *multicore FFT* for a platform with p cores and cache block size μ is given by

$$\begin{aligned}\mathrm{DFT}_n = \big(I_p \otimes (\mathrm{DFT}_k \otimes I_{m/p})\big)^{\left((L_p^{kp} \otimes I_{m/p\mu}) \otimes I_\mu\right)} T_m^n \\ \times \big(I_p \otimes (I_{k/p} \otimes \mathrm{DFT}_m) L_{k/p}^{n/p}\big)\big((L_p^{pm} \otimes I_{k/p\mu}) \otimes I_\mu\big), \quad n = km,\end{aligned} \tag{17}$$



$$L_4^{16}\Big(I_4 \otimes \big((\mathrm{DFT}_2 \otimes I_2)T_2^4(I_2 \otimes \mathrm{DFT}_2)L_2^4\big)\Big)L_4^{16}T_4^{16}\Big(I_4 \otimes \big((\mathrm{DFT}_2 \otimes I_2)T_2^4(I_2 \otimes \mathrm{DFT}_2)L_2^4\big)\Big)L_4^{16}$$

Figure 9: Six-step FFT for $n = 2^4$ and $k = m = \sqrt{n} = 4$.



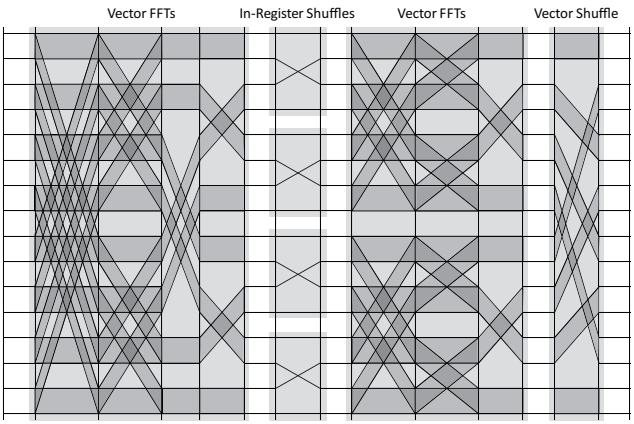
$$(L_4^8 \otimes I_2)\,\Big(I_2 \otimes \Big((\mathrm{DFT}_2 \otimes I_2)T_2^4(I_2 \otimes \mathrm{DFT}_2)L_2^4\Big) \otimes I_2\Big)\,(L_2^8 \otimes I_2)\,T_4^{16}\Big(I_2 \otimes \Big(I_2 \otimes (\mathrm{DFT}_2 \otimes I_2)T_2^4(I_2 \otimes \mathrm{DFT}_2)\Big)R_2^8\Big)\,(L_2^8 \otimes I_2)$$

Figure 10: Multicore FFT for $n = 2^4$, $k = m = 4$, $p = 2$ cores, and cache block size $\mu = 2$.

and is a version of (8) that is optimized for homogeneous multicore CPUs with memory hierarchies. An example is shown in Fig. 10. (17) follows the recursive FFT (8) closely but ensures that all data exchanges between cores and all memory accesses are performed with cache block granularity. For a multicore with cache block size μ and p cores, (17) is built solely from permutations that permute entire cache lines and p-way parallel compute blocks. This property allows for parallelization of small problem sizes across a moderate number of cores. Implementation of (17) on a cache-based system relies on the cache coherency protocol to transmit cache lines of length μ between cores and requires a global barrier. Implementation on a scratchpad based system requires explicit sending and receiving of the data packets, and depending on the communication interface additional synchronization may be required.

The smaller DFTs in (17) can be expanded, for example, with the short vector FFT (discussed next) to optimize for vector extensions.

SIMD short vector FFT. For CPUs with SIMD ν-way vector extensions like SSE and AltiVec and a



$$\Big(\big((\mathrm{DFT}_2 \otimes I_2)T_2^4(I_2 \otimes \mathrm{DFT}_2)L_2^4\big) \otimes I_2\big) \otimes I_2\Big) T_4^{16} \Big(I_2 \otimes (L_2^4 \otimes I_2)(I_2 \otimes L_2^4)\big((\mathrm{DFT}_2 \otimes I_2)T_2^4(I_2 \otimes \mathrm{DFT}_2)L_2^4\big) \otimes I_2\Big)\big(L_2^8 \otimes I_2\big)$$

Figure 11: Short vector FFT in (18) for $n = 2^4$, $k = m = 4$, and (complex) vector length $\nu = 2$.

memory hierarchy, the *short vector FFT* is defined as

$$\mathrm{DFT}_n = \big((\mathrm{DFT}_k \otimes I_{m/\nu}) \otimes I_\nu\big) T_m^n \big(I_{k/\nu} \otimes (L_\nu^m \otimes I_\nu) \times (I_{m/\nu} \otimes L_\nu^{\nu^2})(\mathrm{DFT}_m \otimes I_\nu)\big)\big(L_{k/\nu}^{n/\nu} \otimes I_\nu\big), \quad n = km, \quad (18)$$

and can be implemented using solely vector arithmetic, aligned vector memory access, and a small number of vector shuffle operations. An example is shown in Fig. 11. All compute operations in (18) have complex ν-way vector parallelism. The only operation that is not ν-way vectorized is the stride permutations $L_\nu^{\nu^2}$, which can be implemented efficiently using in-register shuffle instructions. (18) requires the support or implementation of complex vector arithmetic and packs ν complex elements into a machine vector register of width 2ν. A variant that vectorizes the real rather then the complex dataflow exists.

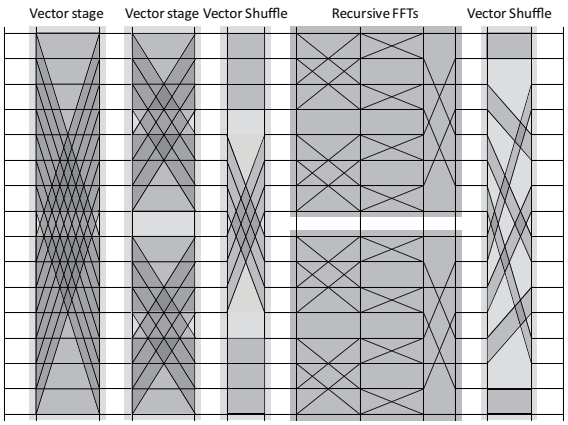
Vector recursion. The *vector recursion* performs a locality optimization for deep memory hierarchies for the first stage $(I_k \otimes \mathrm{DFT}_m)L_k^n$ of (8). Namely, in this stage DFT_m is further expanded using again (8) with $m = m_1 m_2$ and the resulting expression is manipulated to yield

$$(I_k \otimes \mathrm{DFT}_m)L_k^n = \big(I_k \otimes (\mathrm{DFT}_{m_1} \otimes I_{m_2})T_{m_2}^m\big) \times \big(L_k^{km_1} \otimes I_{m_2}\big)\Big(I_{m_1} \otimes \big(I_k \otimes \mathrm{DFT}_{m_2}\big)L_k^{km_2}\Big)\big(L_{m_1}^m \otimes I_k\big). \quad (19)$$

While the recursive FFT (8) ensures that the working set will eventually fit into any level of cache, large two-power FFTs induce large 2-power strides. For caches with lower associativity these strides result in a high number of conflict misses, which may impose a severe performance penalty. For large enough two-power sizes, in the first stage of (8) every single load will result in a cache miss. The vector recursion alleviates this problem by replacing the stride permutation in (8) by stride permutations of vectors, at the expense of an extra pass through the working set. Since (19) matches $\big(I_k \otimes \mathrm{DFT}_n\big)L_k^{kn}$, it is recursively applicable and will eventually produce child problems that fit into any cache level. The vector recursion produces algorithms that are a mix of iterative and recursive as shown in Fig. 12.

Other FFT topics

So far the discussion has focused on one-dimensional complex two-power size FFTs. Some extensions are mentioned next.



$$(\mathrm{DFT}_2 \otimes I_8) T_8^{16} \big(I_2 \otimes (\mathrm{DFT}_2 \otimes I_4) T_4^8\big) \big(L_2^4 \otimes I_4\big) \Big(I_2 \otimes \Big(I_2 \otimes \big((\mathrm{DFT}_2 \otimes I_2) T_2^4 (I_2 \otimes \mathrm{DFT}_2)\big)\Big) R_2^8\Big) \big(L_2^8 \otimes I_2\big)$$

Figure 12: Vector recursive FFT for $n = 2^4$. The vector recursion is applied once and yields vector shuffles, two recursive FFTs, and two iterative vector stages.

General size recursive FFT algorithms. DFT algorithms fundamentally different from (8) include prime-factor (n is a product of coprime factors), Rader (n is prime), and Bluestein or Winograd (any n). In practice these are mostly used for small sizes < 32, which then serve as building blocks for large composite sizes via (8). The exception is Bluestein's algorithm that is often used to compute large sizes with large prime factors or large prime numbers.

DFT variants and other FFTs. In practice, several variants of the DFT in (2) are needed including forward/inverse, interleaved/split complex format, for complex/real input data, in-place/out-of-place, and others. Fortunately, most of these variants are close to the standard DFT in (2), so fast code for the latter can be adapted. An exception is the DFT for real input data, which has its own class of FFTs.

Multidimensional FFT algorithms. The Kronecker product naturally arises in 2D and 3D DFTs, which respectively can be written as

$$\mathrm{DFT}_{m \times n} = \mathrm{DFT}_m \otimes \mathrm{DFT}_n, \tag{20}$$

$$\mathrm{DFT}_{k \times m \times n} = \mathrm{DFT}_k \otimes \mathrm{DFT}_m \otimes \mathrm{DFT}_n\,. \tag{21}$$

For a 2D DFT, applying identities from Table 1 to (20) yields the row-column algorithm

$$\mathrm{DFT}_{m \times n} = (\mathrm{DFT}_m \otimes I_n)(I_m \otimes \mathrm{DFT}_n). \tag{22}$$

The 2D vector-radix algorithm can also be derived with identities from Table 1 from (20):

$$\mathrm{DFT}_{mn \times rs} = \big(\mathrm{DFT}_{m \times r} \otimes I_{ns}\big)^{I_m \otimes L_r^{rn} \otimes I_s} \big(T_n^{mn} \otimes T_s^{rs}\big) \times \big(I_{mr} \otimes \mathrm{DFT}_{n \times s}\big)^{I_m \otimes L_r^{rn} \otimes I_s} \big(L_m^{mn} \otimes L_r^{rs}\big). \tag{23}$$

Higher-dimensional versions are derived similarly, and the associativity of $\otimes$ gives rise to more variants.

RELATED ENTRIES

FFTW
FFTE
SPIRAL
ATLAS

BIBLIOGRAPHIC NOTES AND FURTHER READING

The original Cooley-Tukey FFT algorithm can be found in [2]. The Pease FFT in [13] is the first FFT derived and represented using the Kronecker product formalism. The other parallel FFTs were derived in [10] (Korn-Lambiotte FFT), [16] (Stockham FFT), [11] (Four-Step FFT), [1] (Six-Step FFT). The vector radix FFT algorithm can be found in [8], the vector recursion in [7], the short vector FFT in [3], and the multicore FFT in [4]. A good overview on FFTs including the classical parallel variants is given in Van Loan's book [18] and the book by Tolimieri, An and Lu [17]; both are based on the formalism used here. Also excellent is Nussbaumer FFT book [12]. An overview on real FFTs can be found in [20].

At the point of writing the most important fast DFT libraries are FFTW by Frigo and Johnson [6, 7], Intel's MKL and IPP, and IBM's ESSL and PESSL. FFTE is currently used in the HPC Challenge as *Global FFT* benchmark reference implementation. Most CPU, GPU, and FPGA vendors maintain DFT libraries. Some historic DFT libraries like FFTPACK are still widely used. Numerical Recipes [14] provides C code for an iterative radix-2 FFT implementation. BenchFFT provides up-to-date FFT benchmarks of about 60 single-node DFT libraries. The Spiral system is capable of generating parallel DFT libraries directly from the tensor product based algorithm description [19, 15].

References

[1] D. H. Bailey. FFTs in external or hierarchical memory. *J. Supercomputing*, 4:23–35, 1990.

[2] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. of Computation*, 19:297–301, 1965.

[3] F. Franchetti and M Püschel. Short vector code generation for the discrete Fourier transform. In *Proc. IEEE Int'l Parallel and Distributed Processing Symposium (IPDPS)*, pages 58–67, 2003.

[4] F. Franchetti, Y. Voronenko, and M. Püschel. FFT program generation for shared memory: SMP and multicore. In *Proc. Supercomputing (SC)*, 2006.

[5] Franz Franchetti, Markus Püschel, Yevgen Voronenko, Srinivas Chellappa, and José M. F. Moura. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009.

[6] M. Frigo and S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Int'l Conf. Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, pages 1381–1384, 1998.

[7] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Adaptation".

[8] D. B. Harris, J. H. Mc Clellan, D. S. K. Chan, and H. W. Schuessler. Vector radix fast Fourier transform. In *Proc. International Conference on Acoustics, Speech, and Signal Processing. Conference Proceedings (ICASSP '77)*, pages 548–551, Los Alamitos, 1977. IEEE Comput. Soc. Press.

[9] M. T. Heidemann, D. H. Johnson, and C. S. Burrus. Gauss and the History of the Fast Fourier Transform. *Archive for History of Exact Sciences*, 34:265–277, 1985.

[10] D. G. Korn and J. J. Lambiotte, Jr. Computing the fast fourier transform on a vector computer. *Mathematics of Computation*, 33(7):977–992, 1979.

[11] A. Norton and A. J. Silberger. Parallelization and performance analysis of the Cooley-Tukey FFT algorithm for shared-memory architectures. *IEEE Trans. Comput.*, 36(5):581–591, 1987.

[12] H. J. Nussbaumer. *Fast Fourier Transformation and Convolution Algorithms*. Springer, 2nd edition, 1982.

[13] M. C. Pease. An adaptation of the fast Fourier transform for parallel processing. *Journal of the ACM*, 15(2), April 1968.

[14] W. H. Press, B. P. Flannery, Teukolsky S. A., and Vetterling W. T. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 2nd edition, 1992.

[15] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2):232–275, 2005. special issue on "Program Generation, Optimization, and Adaptation".

[16] Paul N. Schwarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5:197–210, 1987.

[17] R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transforms and Convolution*. Springer, 2nd edition, 1997.

[18] C. Van Loan. *Computational Framework of the Fast Fourier Transform*. SIAM, 1992.

[19] Y. Voronenko, F. de Mesmay, and M. Püschel. Computer generation of general size linear transform libraries. In *Proc. Code Generation and Optimization (CGO)*, pages 102–113, 2009.

[20] Yevgen Voronenko and Markus Püschel. Algebraic signal processing theory: Cooley-Tukey type algorithms for real DFTs. *IEEE Transactions on Signal Processing*, 57(1), 2009.