Recursive Cooley-Tukey FFT

\[ \text{DFT}_{km} = (\text{DFT}_k \otimes \text{I}_m)T^k_m(I_k \otimes \text{DFT}_m)L^k_m \] \quad \text{decimation-in-time} \\
\[ \text{DFT}_{km} = L^m_k(I_k \otimes \text{DFT}_m)T^k_m(\text{DFT}_k \otimes \text{I}_m) \] \quad \text{decimation-in-frequency} \\

- For powers of two \( n = 2^t \) sufficient together with base case

\[ \text{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \]
Example FFT, $n = 16$ (Recursive, Radix 4)

Fast Implementation ($\approx$ FFTW 2.x)

- Choice of algorithm
- Locality optimization
- Constants
- Fast basic blocks
- Adaptivity
1: Choice of Algorithm

- Choose recursive, not iterative

\[
DFT_{km} = (DFT_k \otimes I_m) T_m^{km} (I_k \otimes DFT_m) L_k^{km}
\]

Radix 2, recursive

Radix 2, iterative

2: Locality Improvement

\[
DFT_{16} = DFT_4 \otimes I_4 T_4^{16} I_4 \otimes DFT_4 L_4^{16}
\]

blackboard

fuse stages
DFT\(_{km} = (DFT\_k \otimes I\_m)T\_m^{km}(I\_k \otimes DFT\_m)L\_k^{km}\) 

3: Constants

- FFT incurs multiplications by roots of unity
- In real arithmetic: Multiplications by sines and cosines, e.g.,
  \[ y[i] = \sin(i \cdot \pi/128) \cdot x[i]; \]
  - Very expensive!
- **Observation**: Constants depend only on input size, not on input
- **Solution**: Precompute once and use many times
  
  ```
  d = DFT\_init(1024); // init function computes constant table
  d(x, y); // use many times
  ```
4: Optimized Basic Blocks

Just like loops can be unrolled, recursions can also be unrolled.

Empirical study: Base cases for sizes $n \leq 32$ useful (scalar code).

Needs 62 base case or “codelets” (why?)
- DFTrec, sizes 2–32
- DFTscaled, sizes 2–32

Solution: Codelet generator (codelet = optimized basic block)

```
// code sketch
void DFT(int n, cpx *x, cpx *y) {
  int k = choose_dft_radix(n); // ensure $k \leq 32$
  if (use_base_case(n))
    DFTbc(n, x, y); // use base case
  else {
    for (int i = 0; i < k; ++i)
      DFTrec(m, x + i, y + m*i, k, 1); // implemented as DFT(…)
    for (int j = 0; j < m; ++j)
      DFTscaled(k, y + j, t[j], m); // always a base case
  }
}
```

FFTW Codelet Generator

- n → FFT codelet generator → Codelet for $DFT_n$
- Twiddle codelet for $DFT_n$
- DAG generator → DAG → Simplifier → DAG → Scheduler
Small Example DAG

\[ y_{n_2 j_1 + j_2} = \sum_{k_1=0}^{n_1-1} (\omega_n^{j_2 k_1}) \left( \sum_{k_2=0}^{n_2-1} x_{n_1 k_2 + k_1} \omega_n^{j_2 k_2} \right) \omega_n^{j_1 k_1} \]

- For given \( n \), suitable FFTs are recursively applied to yield \( n \) (real) expression trees for outputs \( y_0, \ldots, y_{n-1} \)
- Trees are fused to an (unoptimized) DAG

DAG Generator

- Knows FFTs: Cooley-Tukey, split-radix, Good-Thomas, Rader, represented in sum notation
Simplifier

- **Blackboard**
- **Applies:**
  - Algebraic transformations
  - Common subexpression elimination (CSE)
  - DFT-specific optimizations
- **Algebraic transformations**
  - Simplify mults by 0, 1, -1
  - Distributivity law: kx + ky = k(x + y), kx + lx = (k + l)x
    Canonicalization: (x-y), (y-x) to (x-y), -(x-y)
- **CSE: standard**
  - E.g., two occurrences of 2x+y: assign new temporary variable
- **DFT specific optimizations**
  - All numeric constants are made positive (reduces register pressure)
  - CSE also on transposed DAG

Scheduler

- **Blackboard**
- **Determines in which sequence the DAG is unparsed to C** (topological sort of the DAG)
  
  *Goal: minimizer register spills*

  - A 2-power FFT has an operational intensity of I(n) = O(log(C)), where C is the cache size [1]
  - Implies: For R registers Ω(n log(n)/log(R)) register spills
  - FFTW’s scheduler achieves this (asymptotic) bound independent of R

First cut

4 independent components
Codelet Examples

- Notwiddle 2
- Notwiddle 3
- Twiddle 3
- Notwiddle 32

Code style:
- Single static assignment (SSA)
- Scoping (limited scope where variables are defined)

5: Adaptivity

```c
// code sketch
void DFT(int n, cpx *x, cpx *y) {
    int k = choose_dft_radix(n); // ensure k <= 32
    if (use_base_case(n))
        DFTbc(n, x, y); // use base case
    else {
        for (int i = 0; i < k; ++i)
            DFTrec(m, x + i, y + m*i, k, 1); // implemented as DFT
        for (int j = 0; j < m; ++j)
            DFTscaled(k, y + j, t[j], m); // always a base case
    }
}
```

Choices used for platform adaptation

d = DFT_init(1024); // compute constant table; search for best recursion
d(x, y); // use many times

- Search strategy: Dynamic programming
- Blackboard
<table>
<thead>
<tr>
<th></th>
<th>MMM Atlas</th>
<th>Sparse MVM Sparsity/Bebop</th>
<th>DFT FFTW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache optimization</td>
<td>Blocking</td>
<td>Blocking (rarely useful)</td>
<td>Recursive FFT, fusion of steps</td>
</tr>
<tr>
<td>Register optimization</td>
<td>Blocking</td>
<td>Blocking (changes sparse format)</td>
<td>Scheduling of small FFTs</td>
</tr>
<tr>
<td>Optimized basic blocks</td>
<td>Unrolling, scalar replacement and SSA, scheduling, simplifications (for FFT)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other optimizations</td>
<td>—</td>
<td>—</td>
<td>Precomputation of constants</td>
</tr>
<tr>
<td>Adaptivity</td>
<td>Search: blocking parameters</td>
<td>Search: register blocking size</td>
<td>Search: recursion strategy</td>
</tr>
</tbody>
</table>