Recursive Cooley-Tukey FFT

\[
\begin{align*}
\text{DFT}_{km} &= (\text{DFT}_{k} \otimes I_m)T_m^{km}(I_k \otimes \text{DFT}_m)L_k^{km} & \text{decimation-in-time} \\
\text{DFT}_{km} &= L_k^{km}(I_k \otimes \text{DFT}_m)T_m^{km}(\text{DFT}_k \otimes I_m) & \text{decimation-in-frequency}
\end{align*}
\]

- 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)*

\[
\text{DFT}_{16} = DFT_4 \otimes I_4 \quad T_4^{16} \quad I_4 \otimes \text{DFT}_4 \quad L_4^{16}
\]

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

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

- Choose recursive, not iterative

\[ DFT_{km} = (DFT_k \otimes I_m) T_{km}^{m} (I_k \otimes DFT_m) L_{km} \]

Radix 2, recursive

Radix 2, iterative

2: Locality Improvement

\[ DFT_{16} = \]

blackboard

fuse stages

Spring 2016
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?)
- $DFT_{rec}$, sizes 2–32
- $DFT_{scaled}$, 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 <= 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

**DAG:**

One possible unparsing:

\[
\begin{align*}
E_0 &= x[0] - x[3]; \\
E_1 &= x[0] + x[3]; \\
E_2 &= x[1] - x[2]; \\
E_3 &= x[1] + x[2]; \\
E_4 &= E_1 - E_3; \\
y[0] &= E_1 + E_3; \\
y[2] &= 0.7071067811865476 \times E_4; \\
E_7 &= 0.9238795325112867 \times E_0; \\
E_8 &= 0.3826834323650898 \times E_2; \\
y[1] &= E_7 + E_8; \\
f[10] &= 0.3826834323650898 \times E_0; \\
f[11] &= (-0.9238795325112867) \times E_2; \\
y[3] &= f[0] + f[1]; 
\end{align*}
\]

---

**DAG Generator**

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

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

- 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
Simplifier

- Blackboard
- Applies:
  - Algebraic transformations
  - Common subexpression elimination (CSE)
  - DFT-specific optimizations
- Algebraic transformations
  - Simplify mutls by 0, 1, -1
  - Distributivity law: \( kx + ky = k(x + y) \), \( kx + lx = (k + l)x \)
  - Canonicalization: \( (x \cdot y) \), \( (y \cdot x) \) to \( (x \cdot y) \), \( -(x \cdot y) \)
- 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 \( \Omega(n \log(n)/\log(R)) \) register spills

- FFTW’s scheduler achieves this (asymptotic) bound independent of \( R \)

typedef struct {
    double* input;
    double* output;
} spiral_t;

cast double x708[] = { 1.0, 0.9238795325112867, 0.7071067811865476, 0.3826834323650898, 
                       -0.0, -0.3826834323650898, 
               const double x709[] = { -0.0, 0.3826834323650898, 0.7071067811865476, 0.9238795325112867, 1.0, 0.9238795325112867, 0.7071067811865476 
       void staged(spiral_t* x0) {
           double* x2 = x0->output;
           double* x1 = x0->input;
           double x6 = x1[0];
           double x22 = x1[16];
           double x38 = x6 + x22;
           double x14 = x1[8];
           double x30 = x1[24];
           double x46 = x14 + x30;
           double x343 = x38 + x46;
           double x10 = x1[4];
           double x26 = x1[20];
           double x42 = x10 + x26;
           double x18 = x1[12];
           double x34 = x1[28];
           double x50 = x18 + x34;
           double x344 = x42 + x50;
           double x345 = x343 + x344;
           double x8 = x1[2];
           double x24 = x1[18];
           double x115 = x8 + x24;
           double x16 = x1[16];
           double x32 = x16 + x32;
           double x53 = x15 + x33;
           double x22 = x1[22];
           double x26 = x1[26];
           double x110 = x22 + x26;
           double x36 = x115 + x113;
           double x346 = x53 + x346;
           double x6 = x1[2];
           double x26 = x1[18];
           double x115 = x8 + x24;
           double x16 = x1[16];
           double x32 = x16 + x32;
           double x53 = x15 + x33;
           double x22 = x1[22];
           double x26 = x1[26];
           double x110 = x22 + x26;
           double x36 = x115 + x113;
           double x346 = x53 + x346;
           double x349 = x345 + x348;
           x2[0] = x349;
           double x7 = x1[1];
           double x23 = x1[17];
           double x30 = x7 + x31;
           double x15 = x1[9];
           double x31 = x1[25];
           double x47 = x15 + x31;
           double x76 = x28 + x47;
           double x11 = x1[3];
           double x27 = x1[21];
           double x43 = x11 + x27;
           double x16 = x1[15];
           double x35 = x1[29];
           double x51 = x16 + x35;
           double x80 = x43 + x51;
           double x69 = x35 + x69;

FFT, n = 16

First cut
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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Register optimization</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimized basic blocks</td>
<td>Blocking</td>
<td>Blocking (rarely useful)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Blocking (changes sparse format)</td>
<td></td>
</tr>
<tr>
<td>Other optimizations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptivity</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For MMM Atlas, Sparse MVM Sparsity/Bebop, and DFT FFTW:

- **Cache optimization**
  - Blocking
  - Blocking (rarely useful)
  - Recursive FFT, fusion of steps

- **Register optimization**
  - Blocking
  - Blocking (changes sparse format)
  - Scheduling of small FFTs

- **Optimized basic blocks**
  - Unrolling, scalar replacement and SSA, scheduling, simplifications (for FFT)

- **Other optimizations**
  - —
  - —
  - Precomputation of constants

- **Adaptivity**
  - Search: blocking parameters
  - Search: register blocking size
  - Search: recursion strategy