Advanced Systems Lab
Spring 2024
*Lecture:* Fast FFT implementation, FFTW

*Instructor:* Markus Püschel
*TA:* Tommaso Pegolotti, several more

Fast FFT: Example FFTW Library

[www.fftw.org](http://www.fftw.org)

*Frigo and Johnson, FFTW: An Adaptive Software Architecture for the FFT, ICASSP 1998*

*Frigo, A Fast Fourier Transform Compiler, PLDI 1999*

*Frigo and Johnson, The Design and Implementation of FFTW3, Proc. IEEE 93(2) 2005*
Cooley-Tukey FFT, n = 4

**Fast Fourier transform (FFT)**

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & i & -1 & -i \\
1 & -1 & 1 & -1 \\
1 & -i & -1 & i \\
\end{bmatrix} = \begin{bmatrix}
1 & 1 & \cdot & \cdot \\
\cdot & 1 & 1 & \cdot \\
\cdot & \cdot & 1 & 1 \\
\cdot & \cdot & \cdot & 1 \\
\end{bmatrix} \begin{bmatrix}
1 & 1 & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & 1 & 1 \\
\cdot & \cdot & \cdot & 1 \\
\end{bmatrix} \begin{bmatrix}
1 & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & 1 & 1 \\
\cdot & \cdot & \cdot & 1 \\
\end{bmatrix}
\]

**Representation using matrix algebra**

\[DFT_4 = (DFT_2 \otimes I_2) \text{diag}(1, 1, 1, i)(I_2 \otimes DFT_2) L_2^4\]

**Data flow graph (right to left)**

2 DFTs of size 2 at stride 2

2 DFTs of size 2

FFT, n = 16 (Recursive, Radix 4)

\[DFT_{16} = (DFT_4 \otimes I_4) T_{16}^{I4} I_4 \otimes DFT_4 L_4^{I4}\]
Recursive Cooley-Tukey FFT

\[ \text{DFT}_{km} = (\text{DFT}_k \otimes I_m)T_{km}^k (I_k \otimes \text{DFT}_m)T_{km}^m \quad \text{decimation-in-time} \]

\[ \text{DFT}_{km} = L_{km}^m (I_k \otimes \text{DFT}_m) T_{km}^m (\text{DFT}_k \otimes I_m) \quad \text{decimation-in-frequency} \]

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

\[ \text{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \]

Fast Implementation (≈ FFTW 2.x)

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

Choose recursive, not iterative

\[ \text{DFT}_{km} = (\text{DFT}_k \otimes I_m) T_m^{km} (I_k \otimes \text{DFT}_m) L_k^{km} \]

First recursive implementation we consider in this course

2: Locality Improvement

Straightforward implementation: 4 steps

- Permute
- Loop recursively calling smaller DFTs (here: 4 of size 4)
- Loop that scales by twiddle factors (diagonal elements of T)
- Loop recursively calling smaller DFTs (here: 4 of size 4)

4 passes through data: bad locality

Better: fuse some steps
2: Locality Improvement

\[ \text{DFT}_n = (\text{DFT}_k \otimes I_m) T_m^\mathrm{r} (I_k \otimes \text{DFT}_m) L_k^\mathrm{r} \]

**schematic:**
- compute \( m \) many DFT\( k \ast D \) with input stride \( m \) and output stride \( m \)
- \( D \) is part of the diagonal \( T \)
- writes to the same location then it reads from \( \rightarrow \) out-of-place

**Interface needed for recursive call:**
- \( \text{DFTrec}(m, x, y, k, 1); \)
- \( \text{DFTscaled}(k, x, d, m); \)

Cannot handle further recursion so in FFTW it is a base case of the recursion

\[
\text{DFT}_{km} = (\text{DFT}_k \otimes I_m) T_m^{km} (I_k \otimes \text{DFT}_m) L_k^{km}
\]

**one loop**

```c
// code sketch
void DFT(int n, cpx *x, cpx *y) {
    int k = choose_dft_radix(n); // ensure k small enough
    int m = n/k;
    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
}
```
3: Constants

FFT incurs multiplications by roots of unity

In real arithmetic:
Multiplications by sines and cosines, e.g.,
\[ y[i] = \sin(1\cdot\pi/128)\times x[i]; \]

Very expensive!

Observation: Constants depend only on input size, not on input

Solution: Precompute once and use many times

```c
int 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 cases or “codelets” (why?)

- \( DFTrec, \) sizes 2–32
- \( DFTscaled, \) sizes 2–32

Solution: Codelet generator (codelet = optimized basic block)
**FFTW Codelet Generator**

- **n** → FFT codelet generator
- Codelet for DFTrec
- Codelet for DFTscaled (twiddle codelet)

**Small Example DAG**

**DAG:**

- One possible unparsing:
  - \( f_0 = x[0] - x[3] \)
  - \( f_1 = x[0] + x[3] \)
  - \( f_2 = x[1] - x[2] \)
  - \( f_3 = x[1] + x[2] \)
  - \( f_4 = f_1 - f_3 \)
  - \( y[0] = f_1 + f_3 \)
  - \( y[2] = 0.7071067811865476 \times f_4 \)
  - \( f_7 = 0.9238795325112867 \times f_0 \)
  - \( f_8 = 0.3826834323650898 \times f_2 \)
  - \( y[1] = f_7 + f_8 \)
  - \( f_{10} = 0.3826834323650898 \times f_0 \)
  - \( f_{11} = (-0.9238795325112867) \times f_2 \)
  - \( y[3] = f_{10} + f_{11} \)
DAG Generator

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

\[ y_{n_2j_1+j_2} = \sum_{k_1=0}^{n_1-1} (\omega_{n_1}^{j_2k_1}) \left( \sum_{k_2=0}^{n_2-1} x_{n_1k_2+k_1} \omega_{n_2}^{j_2k_2} \right) \omega_{n_1}^{j_1k_1} \]

For given n, suitable FFTs are recursively applied to yield n (real) expression trees for outputs \( y_0, ..., y_{n-1} \)

Trees are fused to an (unoptimized) DAG

Simplifier

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

Determines in which sequence the DAG is unparsed to C (topological sort of the DAG)

*Goal: minimize register spills*

A 2-power FFT has an optimal operational intensity of \( I(n) = \Theta(\log(C)) \), 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 \( R \)


---

FFT-Specific Scheduler: Basic Idea

Cut DAG in the middle

Recurse on the connected components

*How to find the middle?*
This is a Sketched/Abstracted DAG

FFT, n = 16

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

cast double x708[] = { 1.0, 0.9238795325112867, 0.7071067811865476, 0.3826834323650898, ...}

cast const double x709[] = { -0.0, 0.3826834323650898, 0.7071067811865476, 0.9238795325112867, 1.0, 0.9238795325112867, 0.7071067811865476, 0.3826834323650898, ...};

void staged(spiral_t* x0) {
  double* x1 = x0->input;
  double* x2 = x0->output;
  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[26];
  double x123 = x16 + x115;
  double x20 = x1[30];
  double x119 = x123 + x20;
  double x127 = x119 + x127;
  double x346 = x34 + x346;
  double x347 = x346 + x347;
  double x348 = x345 + x348;
  x2[0] = x349;
  double x7 = x1[1];
  double x23 = x1[17];
  double x39 = x7 + x23;
  double x15 = x1[9];
  double x31 = x1[25];
  double x47 = x15 + x31;
  double x76 = x47 + x47;
  double x11 = x1[5];
  double x27 = x1[21];
  double x43 = x11 + x27;
  double x19 = x1[15];
  double x29 = x1[29];
  double x119 = x19 + x29;
  double x80 = x43 + x119;
  double x51 = x119 + x51;
Codelet Examples

Notwiddle 2 (DFTrec)
Notwiddle 3 (DFTrec)
Twiddle 3 (DFTscaled)
Notwiddle 32 (DFTrec)

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

5: Adaptivity

Choices used for platform adaptation

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

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

\[ d = \text{DFT\_init}(1024); \ // \text{compute constant table; search for best recursion} \]
\[ d(x, y); \ // \text{use many times} \]

Choices:

\[ \text{DFT}_{km} = (\text{DFT}_k \otimes I_m) T_{m}^{km} (I_k \otimes \text{DFT}_m) L_k^{km} \]

Base case = generated codelet is called

Exhaustive search to expensive

Solution: Dynamic programming

FFTW: Further Information

Previous Explanation: FFTW 2.x

FFTW 3.x:

- Support for SIMD/threading
- Flexible interface to handle FFT variants (real/complex, strided access, sine/cosine transforms)
- Complicates significantly the interfaces actually used and increases the size of the search space
<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>Autotuning</td>
<td>Search: blocking parameters</td>
<td>Search: register blocking size</td>
<td>Search: recursion strategy</td>
</tr>
</tbody>
</table>