Advanced Systems Lab
Spring 2021
Lecture: Optimizing FFT, FFTW

Instructor: Markus Püschel, Ce Zhang
TA: Joao Rivera, several more

Fast FFT: Example FFTW Library

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
Recursive Cooley-Tukey FFT

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

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

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

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

---

Cooley-Tukey FFT, \( n = 4 \)

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

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

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \]
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

\[ \text{DFT}_{km} = (\text{DFT}_k \otimes I_m) \tau_{km}^m (I_k \otimes \text{DFT}_m) L_{km}^k \]

**Radix 2, recursive**

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

\[ DFT_n = (DFT_k \otimes I_m) T^n_m (I_k \otimes DFT_m) L^n_k \]

schematic:

- compute \( m \) many \( DFT \), with input stride \( m \) and output stride \( m \)
- \( D \) is part of the diagonal \( T \)
- writes to the same location then it reads from \( \rightarrow \) inplace

Interface needed for recursive call:

- \( DFT_{scaled}(k, x, d, m) \);  
- \( \text{DFT size} \)
- \( \text{input} = \text{output vector} \)
- \( \text{input stride} = \text{output stride} \)
- \( \text{diagonal elements} \)

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

---

// 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 \( DFT(\_\_\_) \) is
    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.,
\[ y[i] = \sin(\frac{i\cdot\pi}{128}) \times x[i]; \]

Very expensive!

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

*Solution:* Precompute once and use many times

```c
// 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(…)
        for (int j = 0; j < m; ++j)
            DFTscaled(k, y + j, t[j], m);  // always a base case
    }
}
```

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?)

- \( DFTrec, \text{ sizes } 2-32 \)
- \( DFTscaled, \text{ sizes } 2-32 \)

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

- FFT codelet generator
- Codelet for DFTrec
- Codelet for DFTscaled (twiddle codelet)

**Small Example DAG**

**DAG:**

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

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

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

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

Trees are fused to an (unoptimized) DAG

---

Simplifier

Applies:

- **Algebraic transformations**
- **Common subexpression elimination (CSE)**
- **DFT-specific optimizations**

**Algebraic transformations**

- Simplify multis 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 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 \( R \)


FFT-Specific Scheduler: Basic Idea

Cut DAG in the middle

Recurse on the connected components

How to find the middle?

input nodes (input vector)  middle  output nodes (output vector)

\( \cdots \)  \( \rightarrow \)  \( \cdots \)

\( \cdots \)

internal nodes: adds or mults by constant

Computation DAG

© Markus Püschel
Computer Science
ETH Zurich 2021

Advanced Systems Lab
Spring 2021
typedef struct {
    double* input;
    double* output;
} spiral_t;

cast double x708[] = { 1.0, 0.826834323650898, 0.7071067811865476, 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 x119 = x16 + x115;
        double x123 = x119 + x24;
        double x20 = x1[14];
        double x36 = x1[30];
        double x127 = x20 + x36;
        double x127 = x119 + x127;
        double x127 = x36 + x34;
        double x346 = x127 + x346;
        double x347 = x346 + x347;
        double x348 = x345 + x348;
        x2[0] = x348;
        double x7 = x1[1];
        double x23 = x1[17];
        double x39 = x7 + x23;
        double x11 = x1[9];
        double x15 = x1[25];
        double x47 = x15 + x39;
        double x76 = x47 + x39;
        double x11 = x1[5];
        double x27 = x1[21];
        double x43 = x11 + x27;
        double x19 = x1[15];
        double x35 = x1[29];
        double x51 = x19 + x35;
        double x80 = x43 + x51;
        double x88 = x76 + x80;
        double x9 = x1[3];
    }
};

This is a Sketched/Abstracted DAG

4 independent components

First cut

FFT, n = 16

mults by constants

adds

subs
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

// 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(…)
        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
                   // use many times
5: Adaptivity

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

Choices:

\[ \text{DFT}_{km} = (\text{DFT}_k \otimes \text{I}_m)T_{km}(\text{I}_k \otimes \text{DFT}_m)L_{km}^{km} \]

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></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>Recursive FFT, fusion of steps</td>
</tr>
<tr>
<td>Other optimizations</td>
<td>Blocking</td>
<td>Blocking (changes sparse format)</td>
<td>Scheduling of small FFTs</td>
</tr>
<tr>
<td>Adaptivity</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>