**Advanced Systems Lab**  
Spring 2020  
*Lecture*: Optimizing FFT, FFTW

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

---

**Fast FFT: Example FFTW Library**

- [www.fftw.org](http://www.fftw.org)
- Frigo and Johnson, *The Design and Implementation of FFTW3*, Proc. IEEE 93(2) 2005
Recursive Cooley-Tukey FFT

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

\text{decimation-in-time}

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

\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, \( n = 4 \)

**Fast Fourier transform (FFT)**

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

**Representation using matrix algebra**

\[
\text{DFT}_4 = (\text{DFT}_2 \otimes I_2) \text{diag}(1, 1, 1, i)(I_2 \otimes \text{DFT}_2) L_2^4
\]

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

2 DFTs of size 2 at stride 2

\[
\begin{align*}
y_0 & \quad \text{DFT}_2 \quad x_0 \\
y_1 & \quad \text{DFT}_2 \quad x_1 \\
y_2 & \quad \text{DFT}_2 \quad x_2 \\
y_3 & \quad \text{DFT}_2 \quad x_3
\end{align*}
\]

stride 2 → stride 1
FFT, \( n = 16 \) \( (\text{Recursive, Radix 4}) \)

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

Fast Implementation \( (\approx \text{FFTW 2.x}) \)

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

- Choose recursive, not iterative

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

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

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

**schematic:**

- fuse: stage 2
  - compute \( m \) many \( \text{DFT}_k \cdot D \) with input stride \( m \) and output stride \( m \)
  - \( D \) is part of the diagonal \( T \)
  - writes to the same location then it reads from \( \to \) inplace

**Interface needed for recursive call:**

\[
\text{DFTscaled}(k, x, d, m); \\
\text{DFT size} \\
\text{input} = \text{output} \\
\text{vector} \\
\text{input stride} = \text{output stride} \\
\text{diagonal elements}
\]

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

**one loop**

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

**// code sketch**

```c
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.,
  \[ y[i] = \sin(i \cdot \pi/128) \cdot x[i]; \]
- Very expensive!

- **Observation**: Constants depend only on input size, not on input
- **Solution**: Precompute once and use many times

```c
int d = DFT_init(1024); // init function computes constant table
int 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**

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

**DAG generator** → **Simplifier** → **Scheduler**

**Small Example DAG**

**DAG:**

```
+---+---+---+---+
| e1 | e2 | e3 | e4 |
+---+---+---+---+
```

**One possible unparsing:**

```c
f0 = x[0] - x[3];
f1 = x[0] + x[3];
f2 = x[1] - x[2];
f3 = x[1] + x[2];
f4 = f1 - f3;
y[0] = f1 + f3;
y[2] = 0.7071067811865476 * f4;
f7 = 0.9238795325112867 * f0;
f8 = 0.3826834323650898 * f2;
y[3] = f7 + f8;
f10 = 0.3826834323650898 * f0;
f11 = (-0.9238795325112867) * f2;
y[3] = f10 + f11;
```
DAG Generator

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

\[
y_{n2^j_1+j_2} = \sum_{k_1=0}^{n_1-1} \left( \omega_{n_2}^{j_2k_1} \right) \left( \sum_{k_2=0}^{n_2-1} x_{n_1k_2+k_1} \omega_{n_2}^{j_2k_2} \right) \omega_{n_1}^{j_1k_1}
\]

- 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 mults 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

- 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 $R$

---


FFT-Specific Scheduler: Basic Idea

- Cut DAG in the middle

- Recurse on the connected components

---

How to find the middle?

- middle

internal nodes: adds or mults by constant

input nodes (input vector) output nodes (output vector)
typedef struct {
    double* input;
    double* output;
} spiral_t;

const double x708[] = { 1.0, 0.9238795325112867, 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[30];
    double x20 = x1[14];
    double x36 = x1[29];
    double x127 = x20 + x36;
    double x88 = x115 + x127;
    double x12 = x1[6];
    double x28 = x1[22];
    double x43 = x1[11];
    double x35 = x1[29];
    double x51 = x12 + x35;
    double x80 = x43 + x51;
    double x9 = x1[3];
                             
    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 x11 = x1[5];
    double x27 = x1[21];
    double x43 = x11 + x27;
    double x19 = x1[13];
    double x35 = x1[29];
    double x51 = x19 + x35;
    double x80 = x43 + x51;
    double x9 = x1[3] + x49;
}

FFT, n = 16
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

```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
    }
}
```

```
// compute constant table; search for best recursion
d = DFT_init(1024);
d(x, y); // 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: \( \text{DFT}_{km} = (\text{DFT}_k \otimes \text{I}_m)T_{km}^k(I_k \otimes \text{DFT}_m)L_{km}^k \)

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></td>
<td></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>

<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>