Advanced Systems Lab
Spring 2022
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
Transform Algorithms

An algorithm for $y = Tx$ is given by a factorization

\[ T = T_1 T_2 \cdots T_m \]

Namely, instead of $y = Tx$ we can compute in steps

\[
\begin{align*}
t_1 &= T_m x \\
t_2 &= T_{m-1} t_1 \\
&\vdots \\
y &= T_1 t_{m-1}
\end{align*}
\]

This reduces the op count only if:

- the $T_i$ are sparse
- $m$ is not too large

Example: Cooley-Tukey Fast Fourier Transform (FFT), size 4

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & i & -1 & -i \\
1 & -1 & 1 & -1 \\
1 & -i & 1 & i \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
\end{bmatrix}
= 
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
\end{bmatrix}
\]

12 adds
4 mults by $i$

4 adds
1 mult by $i$

4 adds
0 ops

Cooley-Tukey FFT, $n = 4$

Fast Fourier transform (FFT)

\[
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & i & -1 & -i \\
1 & -1 & 1 & -1 \\
1 & -i & 1 & i \\
\end{bmatrix}
= 
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{bmatrix}
\begin{bmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
\end{bmatrix}
\]

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)

\[ \begin{aligned}
& y_0 \\
& y_1 \\
& y_2 \\
& y_3 \\
\end{aligned} \xrightarrow{\text{DFT}_2} \begin{aligned}
& \text{DFT}_2 \\
& \text{DFT}_2 \\
\end{aligned} \xrightarrow{\text{DFT}_2} \begin{aligned}
& x_0 \\
& x_1 \\
& x_2 \\
& x_3 \\
\end{aligned} \]

2 DFTs of size 2 

at stride 2 2 DFTs of size 2 

stride 2 $\rightarrow$ stride 1
Recursive Cooley-Tukey FFT

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

decimation-in-time

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

decimation-in-frequency

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

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

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

$$DFT_{km} = (DFT_k \otimes I_m)T_m^{km}(I_k \otimes DFT_m)\beta_k^{km}$$

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

$$DFT_{16} = DFT_4 \otimes I_4$$

Interface needed for recursive call:

```
DFTrec(m, x, y, k, i);
```

Can handle further recursion (just strides change)
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
\[ d = \text{DFT}\_\text{init}(1024); \quad \text{// init function computes constant table} \]
\[ d(x, y); \quad \text{// use many times} \]
```
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?)

- $\text{DFTrec}$, sizes 2–32
- $\text{DFTscaled}$, sizes 2–32

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

---

**FTTW Codelet Generator**

![Diagram of codelet generation process]

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

- **DAG generator** -> **Simplifier** -> **Scheduler**
Small Example DAG

**DAG:**

```
```

**One possible unparsing:**

```
E0 = x[0] - x[3];
E1 = x[0] + x[3];
E2 = x[1] - x[2];
E3 = x[1] + x[2];
E4 = E1 - E3;
y[0] = E1 + E3;
y[2] = 0.7071067811865476 * E4;
y[7] = 0.9238795325112867 * E0;
y[8] = 0.3826834323650898 * E2;
y[10] = E7 + E8;
y[11] = 0.3826834323650898 * E0;
```

DAG Generator

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

\[
y_{n2j1+j2} = \sum_{k1=0}^{n1-1} \left( \omega_n^{j2k1} \right) \left( \sum_{k2=0}^{n2-1} x_{n1k2+k1} \omega_n^{j2k2} \right) \omega_n^{j1k1}
\]

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 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?

This is a Sketched/Abstracted DAG
typedef struct {
    double* input;
    double* output;
} spiral_t;

const double x708[] = { 1.0, 0.9238795325112867, 0.7071067811865476, 0.3826834323650898,
    -0.0, 0.3826834323650898, 0.7071067811865476, 0.9238795325112867, 1.0, 0.9238795325112867, 0.7071067811865476
};

void staged(spiral_t* s) {
    double* x2 = s->output;
    double* x1 = s->input;

    double x6 = x1[0];
    double x22 = x1[16];
    double x38 = x6 + x22;
    double x14 = x1[8];
    double x66 = x14 + x6;
    double x68 = x6 + x66;
    double x18 = x68 + x68;
    double x29 = x18 + x18;
    double x36 = x6 + x29;
    double x42 = x36 + x36;
    double x343 = x38 + x42;
    double x344 = x343 + x344;
    double x345 = x344 + x345;

    x2[0] = x345;
}

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(…)
        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

Choices:

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

1024 \rightarrow 128 \rightarrow 16 \rightarrow 8

1024 \rightarrow 64 \rightarrow 16 \rightarrow 8

1024 \rightarrow 32 \rightarrow 16 \rightarrow 8

1024 \rightarrow 32 \rightarrow 8 \rightarrow 4

1024 \rightarrow 32 \rightarrow 8 \rightarrow 4

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>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>
</tr>
<tr>
<td>Register optimization</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Optimized basic blocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Other optimizations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adaptivity</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MMM Atlas</td>
<td>Sparse MVM Sparsity/Bebop</td>
</tr>
<tr>
<td>----------------</td>
<td>-----------</td>
<td>---------------------------</td>
</tr>
<tr>
<td><strong>Cache optimization</strong></td>
<td>Blocking</td>
<td>Blocking (rarely useful)</td>
</tr>
<tr>
<td><strong>Register optimization</strong></td>
<td>Blocking</td>
<td>Blocking (changes sparse format)</td>
</tr>
<tr>
<td><strong>Optimized basic blocks</strong></td>
<td>Unrolling, scalar replacement and SSA, scheduling, simplifications (for FFT)</td>
<td></td>
</tr>
<tr>
<td><strong>Other optimizations</strong></td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>Adaptivity</strong></td>
<td>Search: blocking parameters</td>
<td>Search: register blocking size</td>
</tr>
</tbody>
</table>