# Advanced Systems Lab

Spring 2020

*Lecture:* Optimizing FFT, FFTW

**Instructor:** Markus Püschel, Ce Zhang

**TA:** Joao Rivera, Bojan Karlas, several more

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

## 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

*radix*

$$\mathrm{DFT}_{km} = (\mathrm{DFT}_k \otimes \mathrm{I}_m) T_m^{km} (\mathrm{I}_k \otimes \mathrm{DFT}_m) L_k^{km}$$ *decimation-in-time*

$$\mathrm{DFT}_{km} = L_m^{km} (\mathrm{I}_k \otimes \mathrm{DFT}_m) T_m^{km} (\mathrm{DFT}_k \otimes \mathrm{I}_m)$$ *decimation-in-frequency*

- **For powers of two n = $2^t$ sufficient together with base case**

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

# Cooley-Tukey FFT, n = 4

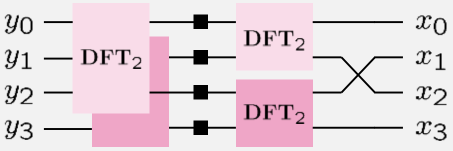
*Fast Fourier transform (FFT)*

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

*Representation using matrix algebra*

$$\mathrm{DFT}_4 = (\mathrm{DFT}_2 \otimes \mathrm{I}_2)\,\mathrm{diag}(1,1,1,i)(\mathrm{I}_2 \otimes \mathrm{DFT}_2)\,\mathrm{L}_2^4$$

*Data flow graph (right to left)*



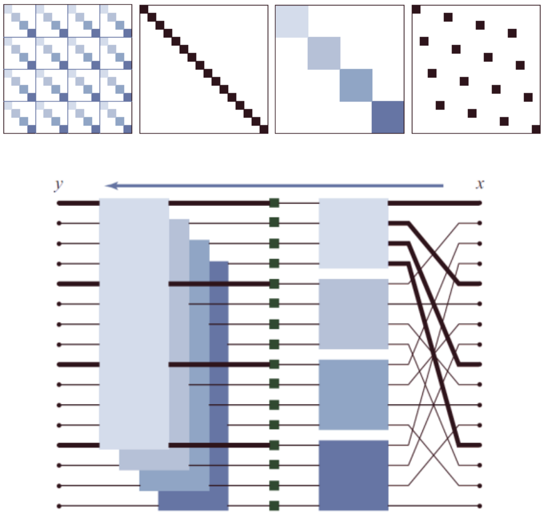
*stride 2 → stride 1*

2 DFTs of size 2
at stride 2

2 DFTs of size 2

## FFT, n = 16 *(Recursive, Radix 4)*

$$\mathrm{DFT}_{16} = \mathrm{DFT}_4 \otimes I_4 \; T_4^{16} \; I_4 \otimes \mathrm{DFT}_4 \; L_4^{16}$$



## Fast Implementation (≈ FFTW 2.x)

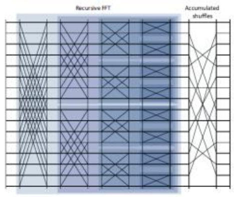
- Choice of algorithm
- Locality optimization
- Constants
- Fast basic blocks
- Adaptivity

# 1: Choice of Algorithm

- **Choose recursive, not iterative**

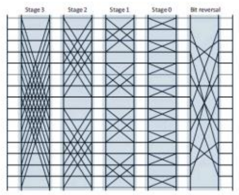
$$\mathbf{DFT}_{km} = (\mathbf{DFT}_k \otimes \mathrm{I}_m) T^{km}_m (\mathrm{I}_k \otimes \mathbf{DFT}_m) L^{km}_k$$

*Radix 2, recursive*



$(\mathrm{DFT}_2 \otimes I_8) T^{16}_8 \Big(I_2 \otimes \Big((\mathrm{DFT}_2 \otimes I_4) T^8_4 (I_2 \otimes ((\mathrm{DFT}_2 \otimes I_2) T^4_2 (I_2 \otimes \mathrm{DFT}_2) L^4_2)) L^8_2\Big)\Big) L^{16}_2$

*Radix 2, iterative*

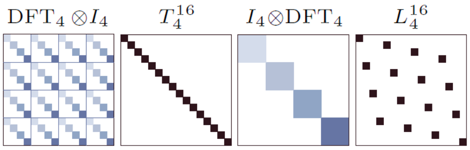


$\Big((I_1 \otimes \mathrm{DFT}_2 \otimes I_8) D^{16}_0\Big)\Big((I_2 \otimes \mathrm{DFT}_2 \otimes I_4) D^{16}_1\Big)\Big((I_4 \otimes \mathrm{DFT}_2 \otimes I_2) D^{16}_2\Big)\Big((I_8 \otimes \mathrm{DFT}_2 \otimes I_1) D^{16}_3\Big) R^{16}_2$

*First recursive implementation we consider in this course*

# 2: Locality Improvement

$\mathrm{DFT}_{16} =$

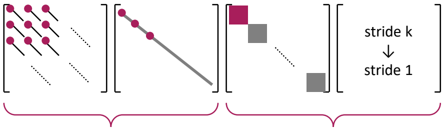


- **Straightforward implementation: 4 steps**
  - Permute
  - Loop recursively calling smaller DFTs (here: 4 of size 4)
  - Loop that scales by twiddle factors (diagonal elements of T)
  - Loop recursively calling smaller DFTs (here: 4 of size 4)
- **4 passes through data: bad locality**
- **Better: fuse some steps**

# 2: Locality Improvement

$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes \mathrm{I}_m)\, \mathrm{T}^n_m (\mathrm{I}_k \otimes \mathbf{DFT}_m)\, \mathrm{L}^n_k$$

schematic:



stride k
↓
stride 1

fuse: stage 2

- compute m many DFT$_k$*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 → inplace

*Interface needed for recursive call:*

```
DFTscaled(k, x, d, m);
```

DFT size

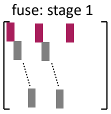
input = output vector

diagonal elements

input stride = output stride

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

fuse: stage 1



- compute k many DFT$_m$ with input stride k and output stride 1
- writes to different location then it reads from → out-of-place

*Interface needed for recursive call:*

```
DFTrec(m, x, y, k, 1);
```

DFT size

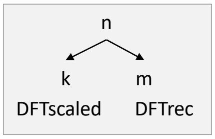
input/ output vector

input stride

output stride

Can handle further recursion (just strides change)

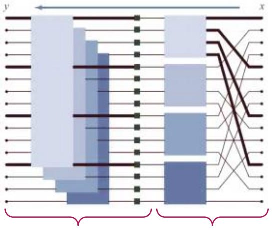
$$\mathbf{DFT}_{km} = \underbrace{(\mathbf{DFT}_k \otimes \mathrm{I}_m) T^{km}_m}_{\textit{one loop}} \underbrace{(\mathrm{I}_k \otimes \mathbf{DFT}_m) L^{km}_k}_{\textit{one loop}}$$



n

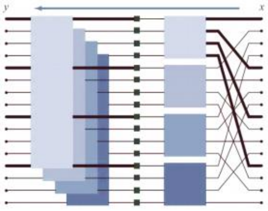
k
DFTscaled

m
DFTrec



```
// 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(i·pi/128)*x[i];
```

- Very expensive!

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

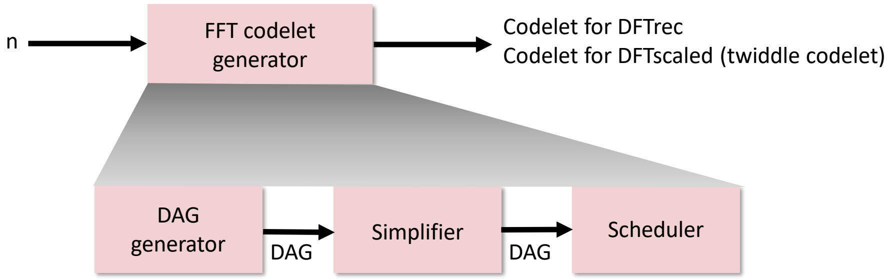
```
d = DFT_init(1024); // init function computes constant table
d(x, y);            // use many times
```

# 4: Optimized Basic Blocks

```
// 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
  }
}
```

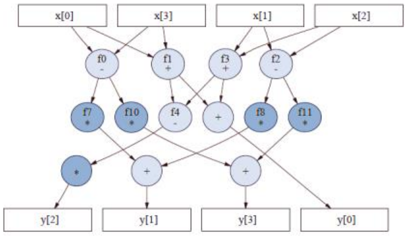
- Just like loops can be unrolled, recursions can also be unrolled
- Empirical study: Base cases for sizes n ≤ 32 useful (scalar code)
- Needs 62 base cases or "codelets" (why?)
  - DFTrec, sizes 2–32
  - DFTscaled, sizes 2–32
- *Solution:* Codelet generator (codelet = optimized basic block)

# FFTW Codelet Generator



# Small Example DAG

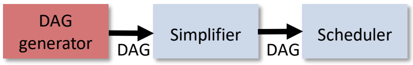
***DAG:***



***One possible unparsing:***

```
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[1] = f7 + f8;
f10 = 0.3826834323650898 * f0;
f11 = (-0.9238795325112867) * f2;
y[3] = f10 + f11;
```

# DAG Generator

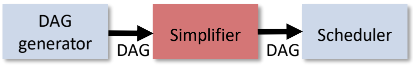


- **Knows FFTs: Cooley-Tukey, split-radix, Good-Thomas, Rader, represented in sum notation**

$$y_{n_2 j_1 + j_2} = \sum_{k_1=0}^{n_1-1} \left(\omega_n^{j_2 k_1}\right) \left( \sum_{k_2=0}^{n_2-1} x_{n_1 k_2 + k_1} \omega_{n_2}^{j_2 k_2} \right) \omega_{n_1}^{j_1 k_1}$$

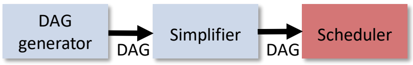
- **For given n, suitable FFTs are recursively applied to yield n (real) expression trees for outputs $y_0, \ldots, y_{n-1}$**
- **Trees are fused to an (unoptimized) DAG**

# Simplifier



- **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

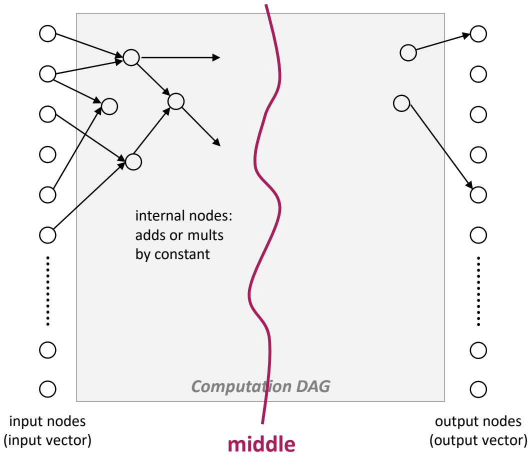


- **Determines in which sequence the DAG is unparsed to C (topological sort of the DAG)**
  *Goal: minimizer register spills*
- **A 2-power FFT has an operational intensity of I(n) = O(log(C)), where C is the cache size [1]**
- **Implies: For R registers Ω(n log(n)/log(R)) register spills**
- **FFTW's scheduler achieves this (asymptotic) bound *independent* of R**

[1] Hong and Kung: "*I/O Complexity: The red-blue pebbling game*"

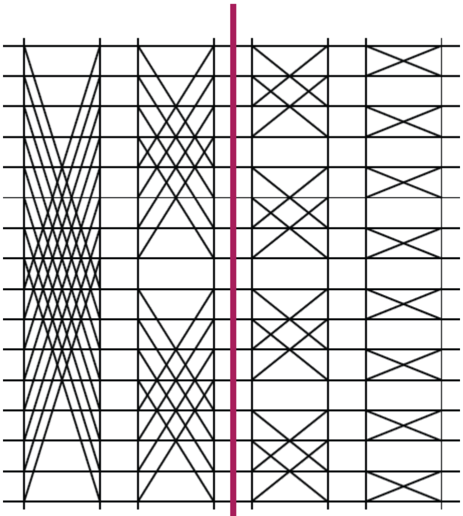
# FFT-Specific Scheduler: Basic Idea

- **Cut DAG in the middle**
- **Recurse on the connected components**


internal nodes: adds or mults by constant
Computation DAG
input nodes (input vector)
middle
output nodes (output vector)

**How to find the middle?**

# This is a Sketched/Abstracted DAG

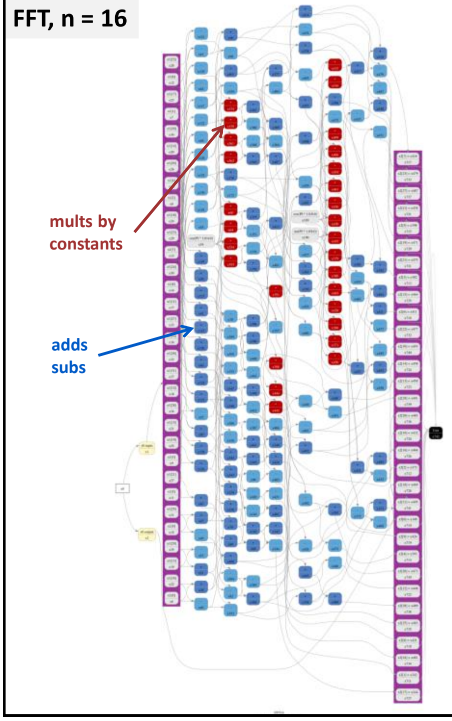


4 independent components

4 independent components

*First cut*

FFT, n = 16



mults by constants

adds subs

```
typedef struct {
                double* input;
                double* output;
} spiral_t;
const double x708[] = { 1.0, 0.9238795325112867, 0.70710678118654
const double x709[] = { -0.0, 0.3826834323650898, 0.7071067811865
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[10];
double x32 = x1[26];
double x123 = x16 + x32;
double x346 = x115 + x123;
double x12 = x1[6];
double x28 = x1[22];
double x119 = x12 + x28;
double x20 = x1[14];
double x36 = x1[30];
double x127 = x20 + x36;
double x347 = x119 + x127;
double x348 = x346 + x347;
double x349 = 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 = x39 + x47;
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 x88 = x76 + x80;
```

# 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

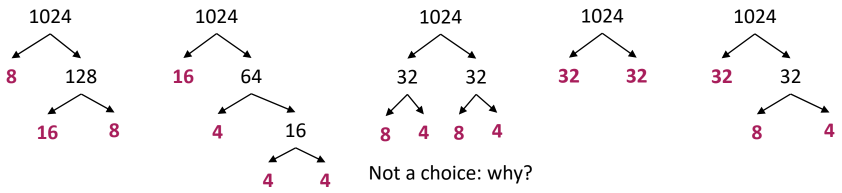
```
// code sketch
void DFT(int n, cpx *x, cpx *y) {
  if (use_base_case(n))
    DFTbc(n, x, y); // use base case
  else {
    int k = choose_dft_radix(n); // ensure k <= 32
    int m = n/k;
    for (int i = 0; i < k; ++i)
      DFTrec(m, x + i, y + m*i, k, 1); // implemented as DFT(…) is
    for (int 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 = DFT_init(1024); // compute constant table; search for best recursion
d(x, y);            // use many times
```

Choices: $$\mathbf{DFT}_{km} = (\mathbf{DFT}_k \otimes \mathrm{I}_m) T^{km}_m (\mathrm{I}_k \otimes \mathbf{DFT}_m) L^{km}_k$$



Not a choice: why?

***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

| | MMM<br>*Atlas* | Sparse MVM<br>*Sparsity/Bebop* | DFT<br>*FFTW* |
|---|---|---|---|
| **Cache optimization** | | | |
| **Register optimization** | | | |
| **Optimized basic blocks** | | | |
| **Other optimizations** | | | |
| **Adaptivity** | | | |

| | MMM<br>Atlas | Sparse MVM<br>Sparsity/Bebop | DFT<br>FFTW |
|---|---|---|---|
| **Cache optimization** | Blocking | Blocking<br>(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 |