How to Write Fast Numerical Code

Spring 2015

Lecture: Optimizing FFT, FFTW

Instructor: Markus Püschel

TA: Gagandeep Singh, Daniele Spampinato, Alen Stojanov

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Recursive Cooley-Tukey FFT

radix

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

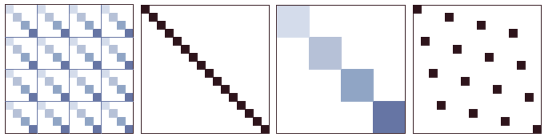
$$\mathbf{DFT}_{km} = L^{km}_m (\mathrm{I}_k \otimes \mathbf{DFT}_m) T^{km}_m (\mathbf{DFT}_k \otimes \mathrm{I}_m)$$ *decimation-in-frequency*

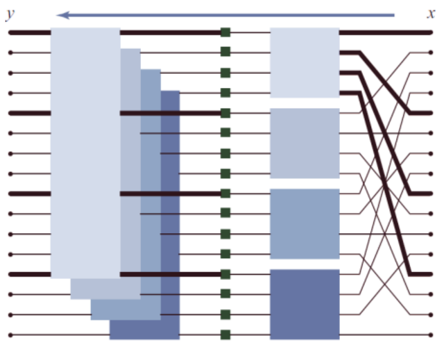
- **For powers of two n = 2^t sufficient together with base case**

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

Example 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)

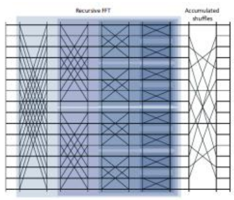
- Choice of algorithm
- Locality optimization
- Constants
- Fast basic blocks
- Adaptivity

1: Choice of Algorithm

- Choose recursive, not iterative

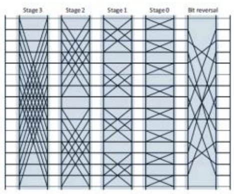
$$\mathrm{DFT}_{km} \;=\; (\mathrm{DFT}_k \otimes \mathrm{I}_m) T^{km}_m (\mathrm{I}_k \otimes \mathrm{DFT}_m) L^{km}_k$$

Radix 2, recursive



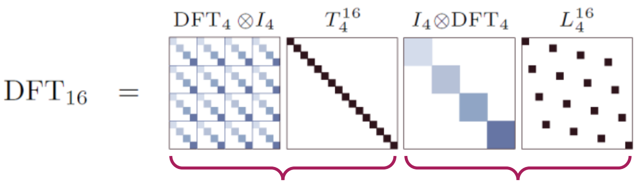
$(\mathrm{DFT}_2 \otimes I_8) T^{16}_8 \Big(I_2 \otimes \Big((\mathrm{DFT}_2 \otimes I_4) T^8_4 (I_2 \otimes ((\mathrm{DFT}_2 \otimes I_2) T^4_2 (I_2 \otimes \mathrm{DFT}_2) L^4_2)) L^8_2\Big)\Big) L^{16}_2$

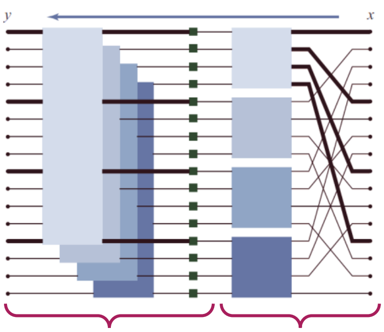
Radix 2, iterative



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

2: Locality Improvement



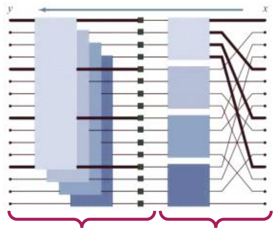


blackboard

fuse stages

ETH Eidgenössische Technische Hochschule Zürich Swiss Federal Institute of Technology Zurich

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



```
// code sketch
void DFT(int n, cpx *x, cpx *y) {
  int k = choose_dft_radix(n); // ensure k <= 32

  …
  …
    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
```

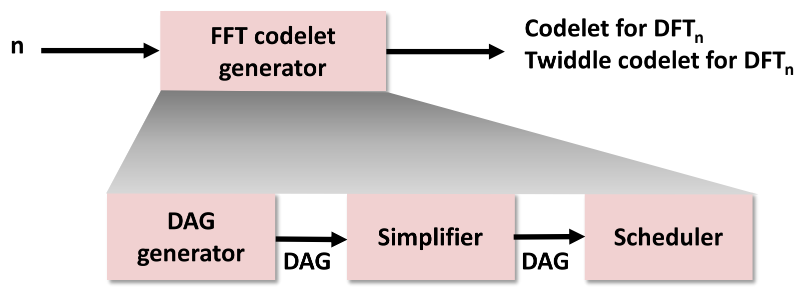
4: Optimized Basic Blocks

```
// code sketch
void DFT(int n, cpx *x, cpx *y) {
  int k = choose_dft_radix(n); // ensure k <= 32

  if (use_base_case(n))
    DFTbc(n, x, y); // use base case
  else {
    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
  }
}
```

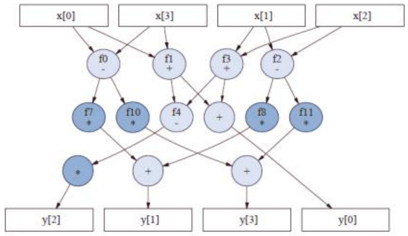
- **Just like loops can be unrolled, recursions can also be unrolled**
- **Empirical study: Base cases for sizes n ≤ 32 useful (scalar code)**
- **Needs 62 base case or "codelets" (why?)**
  - DFTrec, sizes 2–32
  - DFTscaled, sizes 2–32
- ***Solution:* Codelet generator (codelet = optimized basic block)**

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, …, y_{n-1}**
- **Trees are fused to an (unoptimized) DAG**

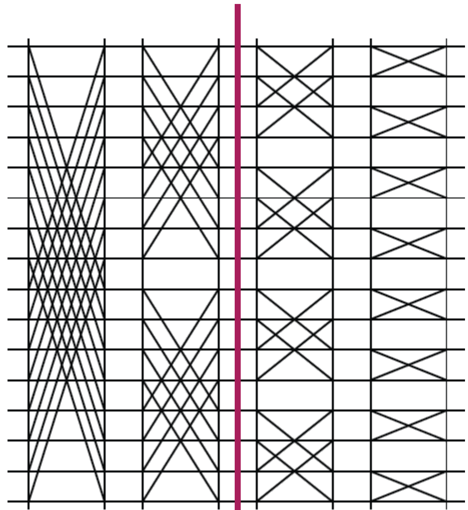
Simplifier

- **Blackboard**
- **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

- **Blackboard**
- **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**

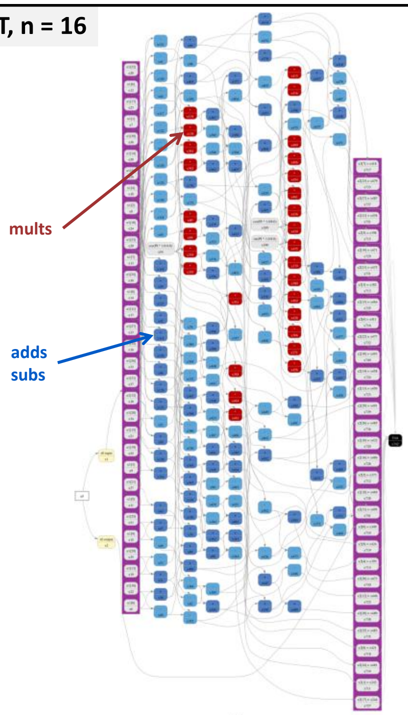
[1] Hong and Kung: "I/O Complexity: The red-blue pebbling game"



4 independent components | 4 independent components

First cut

FFT, n = 16




```
typedef struct {
                double* input;
                double* output;
} spiral_t;
const double x708[] = { 1.0, 0.9238795325112867, 0.70710678118654
const double x709[] = { -0.0, 0.3826834323650898, 0.70710678118654
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;
```

16

Codelet Examples

- Notwiddle 2
- Notwiddle 3
- Twiddle 3
- Notwiddle 32

- **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) {
  int k = choose_dft_radix(n); // ensure k <= 32

  if (use_base_case(n))
    DFTbc(n, x, y); // use base case
  else {
    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
  }
}
```

Choices used for platform adaptation

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

- **Search strategy: Dynamic programming**
- **Blackboard**

	MMM *Atlas*	Sparse MVM *Sparsity/Bebop*	DFT *FFTW*
Cache optimization			
Register optimization			
Optimized basic blocks			
Other optimizations			
Adaptivity			

	MMM Atlas	Sparse MVM Sparsity/Bebop	DFT FFTW
Cache optimization	Blocking	Blocking (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