How to Write Fast Numerical Code

Spring 2011
Lecture 22

Instructor: Markus Püschel

TA: Georg Ofenbeck

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Schedule

May 2011

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18 🟣	19	20	21
22	23 ⚫	24	25	26	27	28
29	30 🔵	31	1 🔵	2	3	4
					10 ✖	

🟣 **Today**

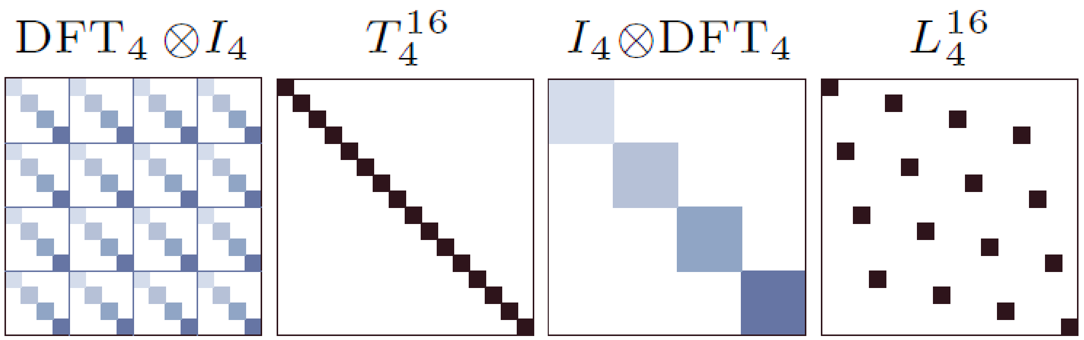
⚫ **Lecture**

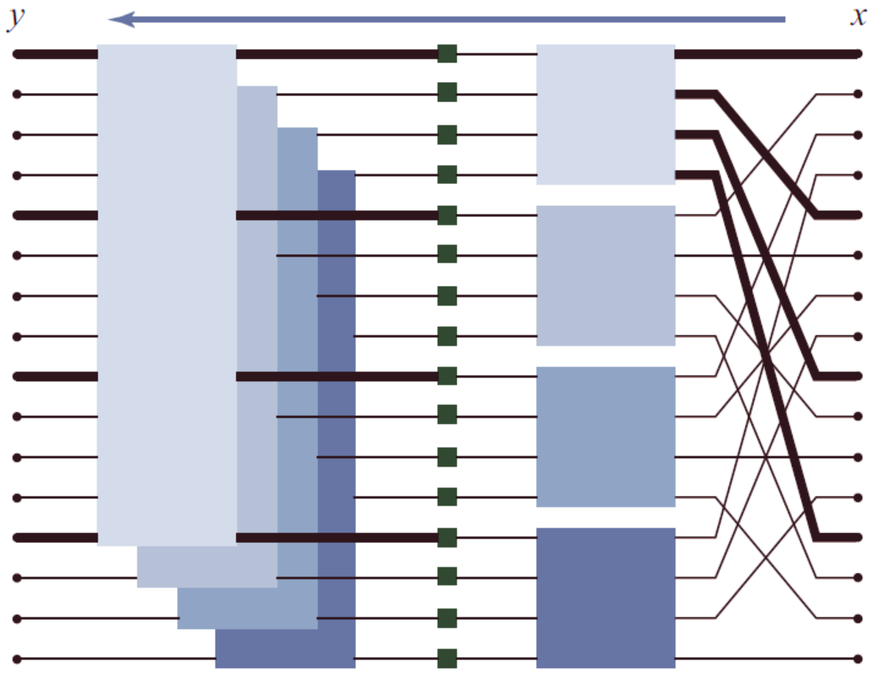
🔵 **Project presentations**
- 10 minutes each
- random order
- random speaker

Final code and paper due

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)

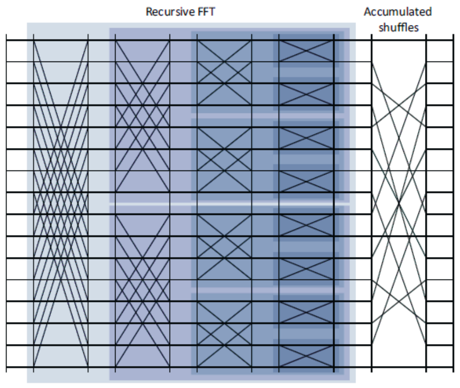
- Choice of algorithm
- Locality optimization
- Constants
- Fast basic blocks
- Adaptivity

1: Choice of Algorithm

- **Choose recursive, not iterative**

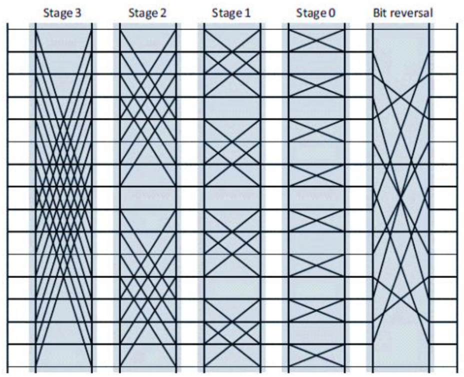
$$\mathrm{DFT}_{km} = (\mathrm{DFT}_k \quad \mathrm{I}_m) T^{km}_m (\mathrm{I}_k \quad \mathrm{DFT}_m) L^{km}_k$$

Radix 2, recursive



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

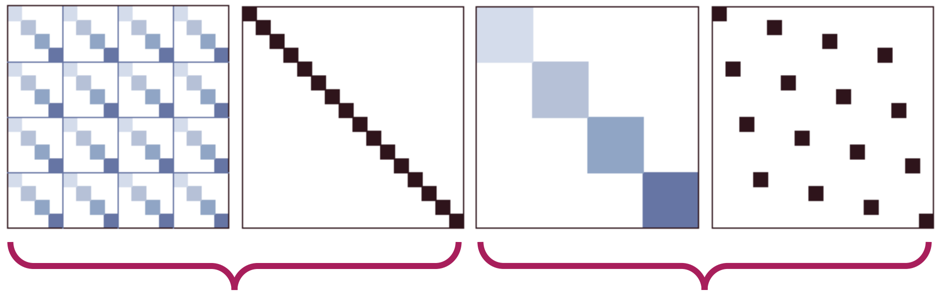
Radix 2, iterative

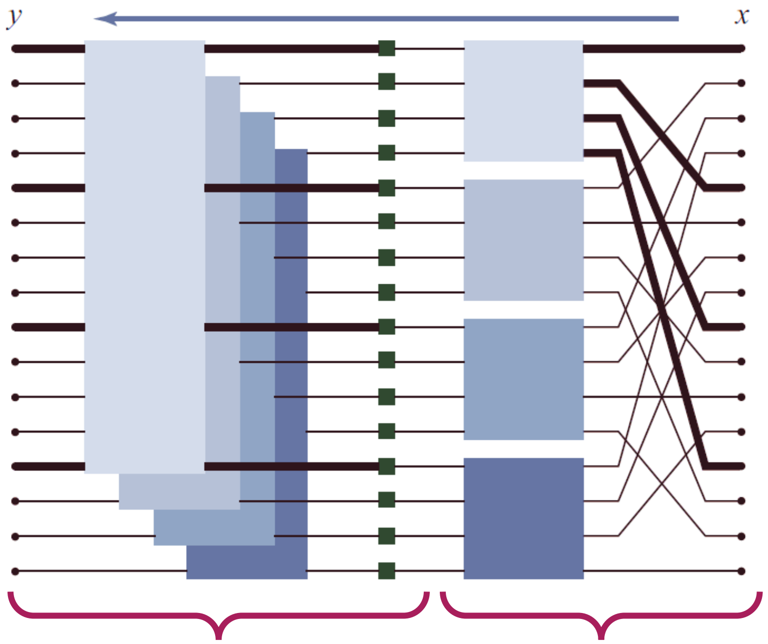


$$\Big((I_1 \otimes \mathrm{DFT}_2 \otimes I_8) D^{16}_0\Big)\Big((I_2 \otimes \mathrm{DFT}_2 \otimes I_4) D^{16}_1\Big)\Big((I_4 \otimes \mathrm{DFT}_2 \otimes I_2) D^{16}_2\Big)\Big((I_8 \otimes \mathrm{DFT}_2 \otimes I_1) D^{16}_3\Big) R^{16}_2$$

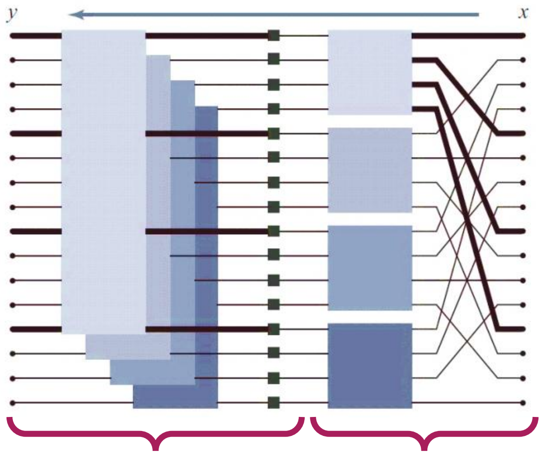
2: Locality Improvement: Fuse Stages

$$\mathrm{DFT}_{16} = \underbrace{\mathrm{DFT}_4 \otimes I_4 \quad T^{16}_4}_{} \quad \underbrace{I_4 \otimes \mathrm{DFT}_4 \quad L^{16}_4}_{}$$





$$\mathbf{DFT}_{km} = \underbrace{(\mathbf{DFT}_k \quad \mathrm{I}_m)T_m^{km}}_{}\underbrace{(\mathrm{I}_k \quad \mathbf{DFT}_m)L_k^{km}}_{}$$



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

  if (use_base_case(n))
    DFTbc(n, y, x); // use base case
  else {
    for (int i=0; i < k; ++i)
      DFTrec(m, y + m*i, x + 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!

- Solution: Precompute once and use many times

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

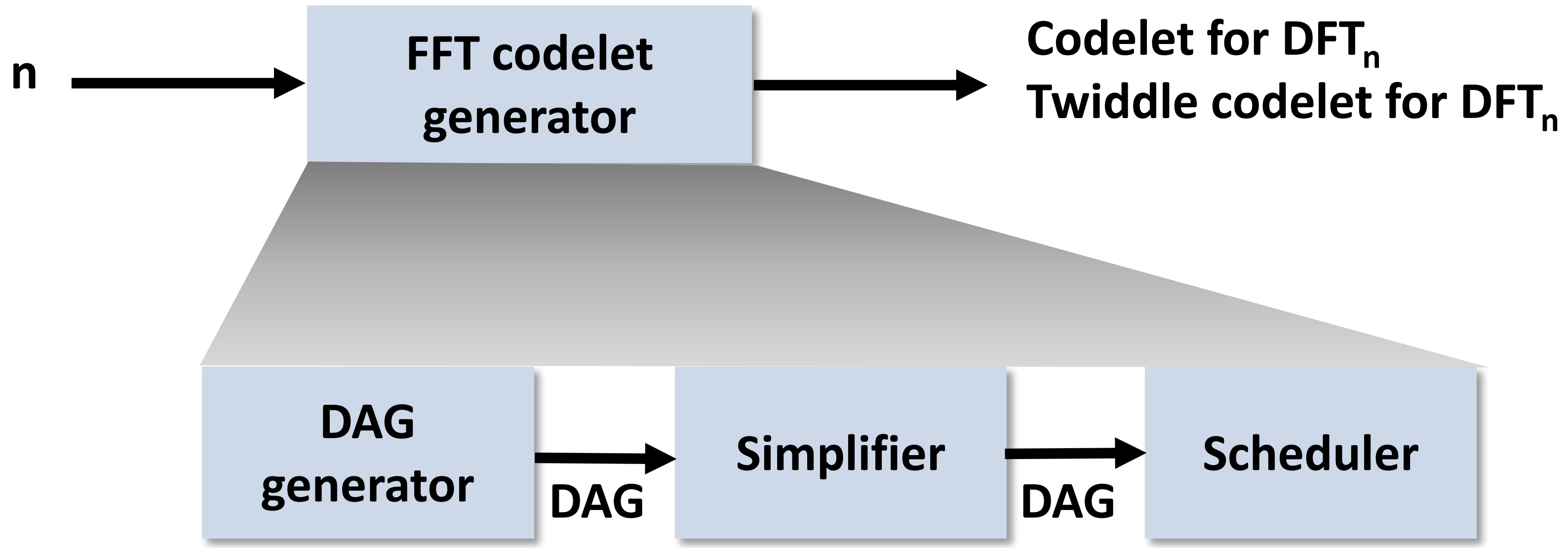
4: Optimized Basic Blocks

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

  if (use_base_case(n))
    DFTbc(n, y, x); // use base case
  else {
    for (int i=0; i < k; ++i)
      DFTrec(m, y + m*i, x + 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
  }
}
```

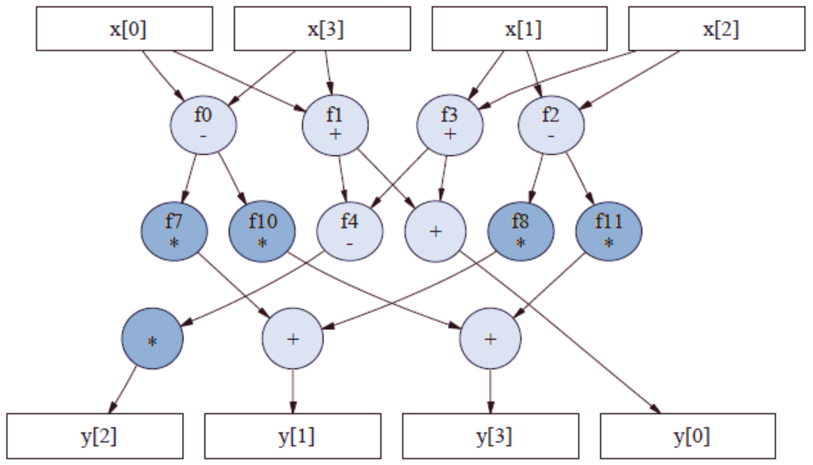
- **Empirical study: Base cases for sizes n ≤ 32 useful (scalar code)**
- **Needs 62 base case or "codelets" (why?)**
  - DFTrec, sizes 2–32
  - DFTscaled, sizes 2–32
- **Solution: Codelet generator**

FFTW Codelet Generator


n
FFT codelet generator
Codelet for DFT_n
Twiddle codelet for DFT_n
DAG generator
DAG
Simplifier
DAG
Scheduler

Small Example DAG

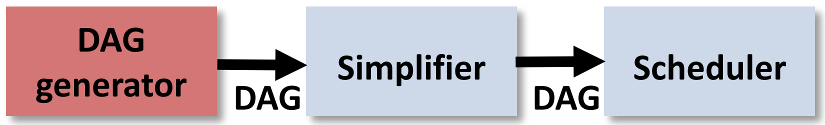
DAG:



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

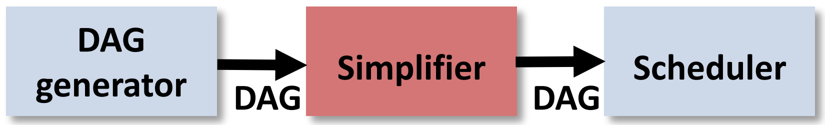


- Knows FFTs: Cooley-Tukey, split-radix, Good-Thomas, Rader, represented in sum notation

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

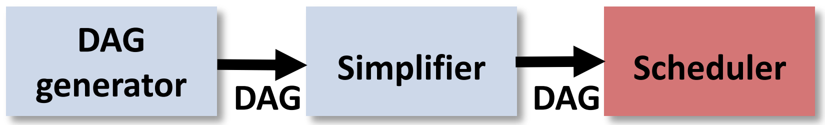
- For given n, suitable FFTs are recursively applied to yield n (real) expression trees for outputs $y_0, \ldots, y_{n-1}$
- Trees are fused to an (unoptimized) DAG

Simplifier



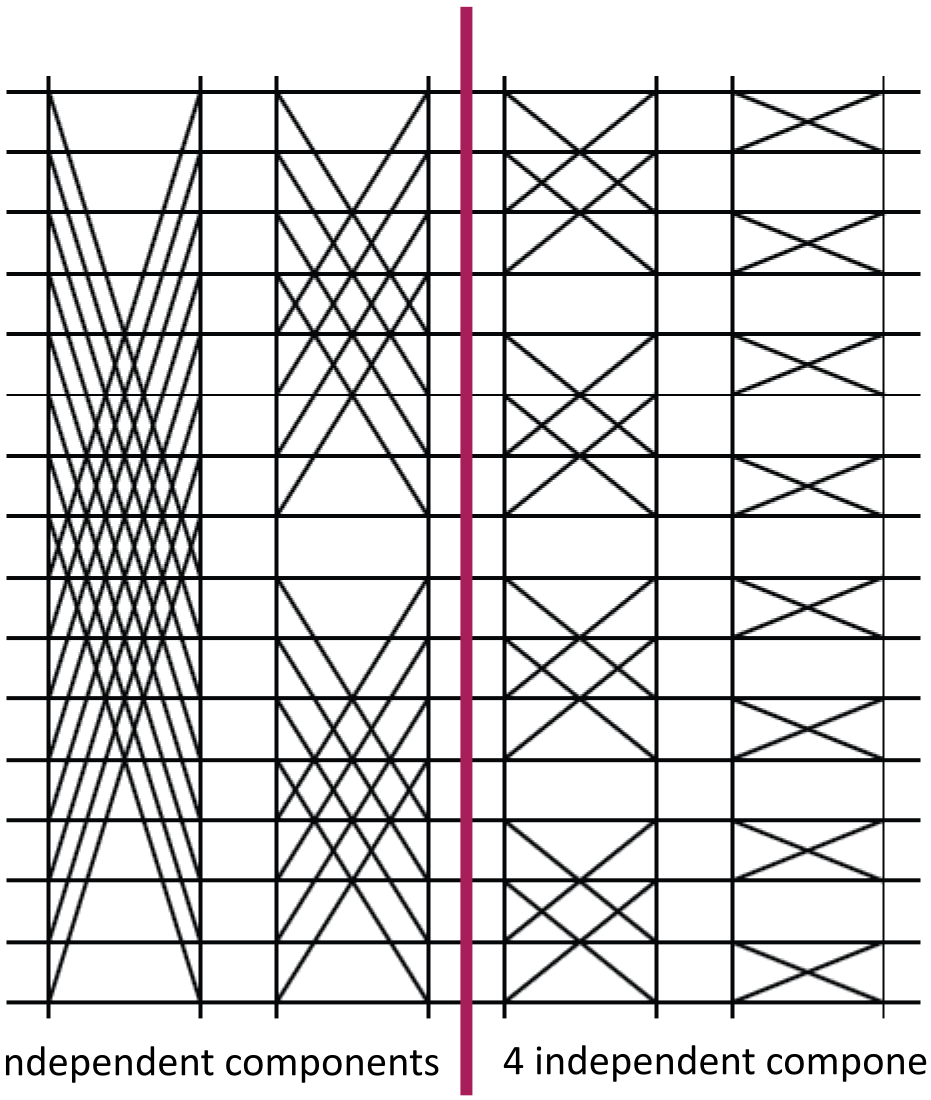
- **Applies:**
  - Algebraic transformations
  - Common subexpression elimination (CSE)
  - DFT-specific optimizations
- **Algebraic transformations**
  - Simplify mults by 0, 1, -1
  - Distributivity law: kx + ky = k(x + y), kx + lx = (k + l)x
    Canonicalization: (x-y), (y-x) to (x-y), -(x-y)
- **CSE: 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
- **If R registers are available, then a 2-power FFT needs at least Ω(nlog(n)/R) register spills [1]**
  Same holds for a fully associative cache
- **FFTW's scheduler achieves this (asymptotic) bound *independent* of R**
- **Blackboard**

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



First cut

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 *y, cpx *x) {
  int k = choose_dft_radix(n); // ensure k <= 32

  if (use_base_case(n))
    DFTbc(n, y, x); // use base case
  else {
    for (int i=0; i < k; ++i)
      DFTrec(m, y + m*i, x + 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(y, x);            // use many times
```

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

	MMM **Atlas**	**Sparse MVM** **Sparsity/Bebop**	**DFT** **FFTW**
Cache optimization	Blocking	Blocking (rarely useful)	Recursive FFT, fusion of steps
Register optimization	Blocking	Blocking (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