Fast, adaptive implementation of the Cooley-Tukey FFT (FFTW)

1.) Locality of data access

- choose recursive FFT, not iterative FFT
- $DFT_{km} = (DFT_k \otimes I_m)\, T\, (I_k \otimes DFT_m)\, L$ (DIT)

(schematic) =  4 steps

fuse and compute $DFT_k \cdot D$'s

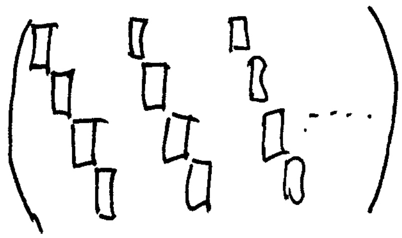
↑ part of twiddles

$\rightarrow DFT(k, *x, *t, s)$

- $*x$: in-vector = out-vector
- $*t$: twiddles
- s: in-stride = out-stride

⇑

- interface does not handle recursions
- in FFTW implemented as basic blocks (unrolled, optimized code)



- stride as parameter
- out-of-place
- $\rightarrow DFT(k, *x, *y, s\_in, s\_out)$

- k: size
- $*x$: in-vector
- $*y$: out-vector
- s_in: in-stride
- s_out: out-stride

⇑

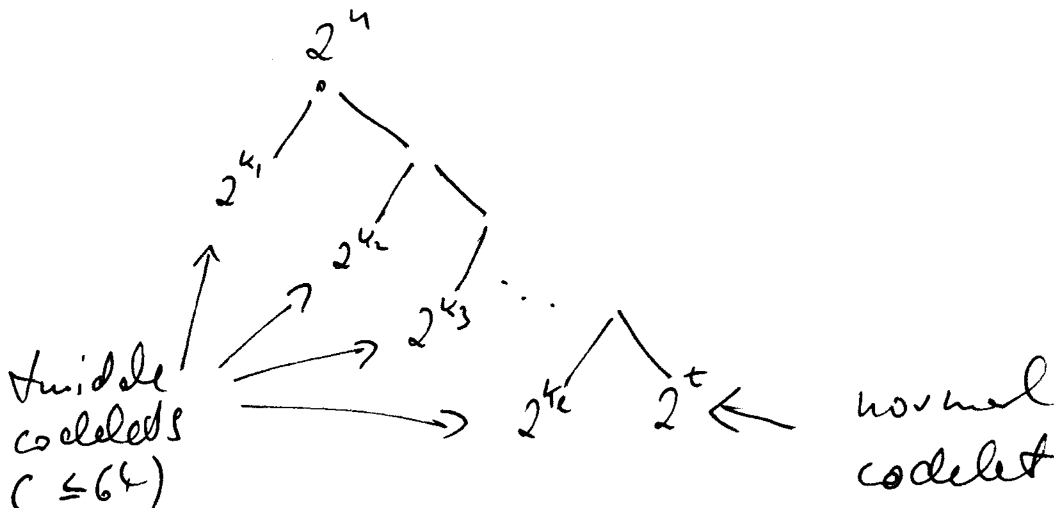
interface handles arbitrary recursions

Explain why DIT is better than DIF.

2.) Precomputing constants
- sin/cos are very expensive to compute at runtime
- solution: precompute in init(..) function and store in table

3.) Fast basic blocks for small sizes
- Slides

4.) Adaptivity
- search over relevant algorithm space



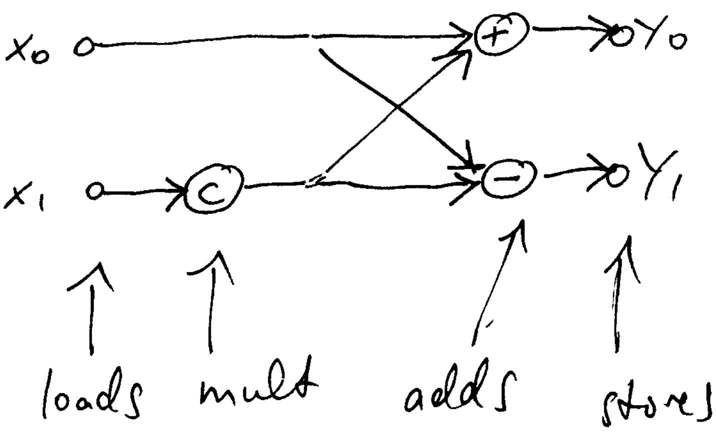
Dynamic programming search:
- Recursively, bottom up, build table of best recursions.

~~5.) Other cache optimization~~
~~after the spring break~~

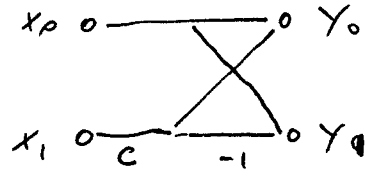
- Much faster than exhaustive search, but assumes best FFT is independent of context.

DAG example

$\mathrm{DFT}_2 \cdot \mathrm{diag}(1, c)$

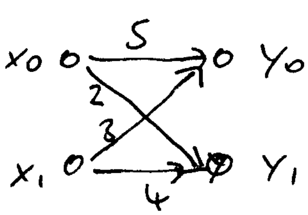


or

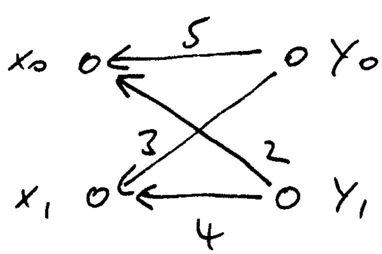


CSE on transposed DAG

DAG transposition:



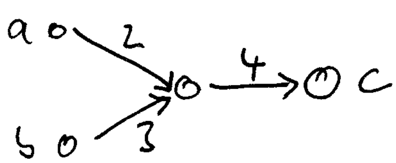
$\xrightarrow{\text{transposition}}$



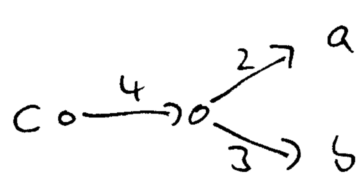
$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \underbrace{\begin{pmatrix} 5 & 3 \\ 2 & 4 \end{pmatrix}}_{= A} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \underbrace{\begin{pmatrix} 5 & 2 \\ 3 & 4 \end{pmatrix}}_{= A^T} \begin{pmatrix} y_0 \\ y_1 \end{pmatrix}$$

Example:



$\xrightarrow{\text{transpose}}$



$c = 4(2a + 3b) \rightarrow 8a + 12b$

destroys – one op
– two subexpressions

$a = 2 \cdot 4c$
$b = 3 \cdot 4c$ $\rightarrow$ $8c$, $12c$

destroys – 2 ops
– 2 subexpr.