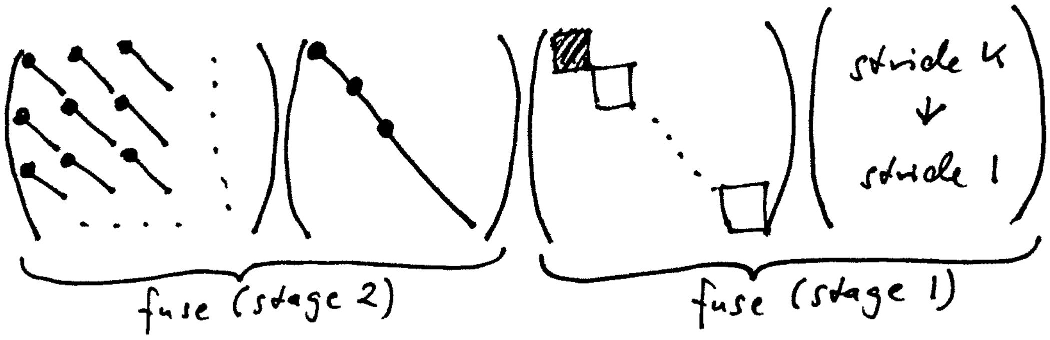
FFT, fast implementation (following FFTW 2.x)

1.) Choice of algorithm: Choose recursive FFT, not iterative FFT

2.) Locality optimization:

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

(schematic) =



compute m many

$DFT_k \cdot D$

$\uparrow$ part of diagonal T_m^{km}

at stride m (input and output)

→ writes to the same location it reads from

→ inplace

$\underset{\text{size}}{\overset{\uparrow}{}}$ DFTscaled(k, *x, *d, stride)

- k: size
- *x: input = output vector
- *d: diagonal elements

this interface cannot handle arbitrary recursions

→ in FFTW a base case

compute k many DFT_m with input stride k and output stride 1.

→ writes to different location it reads from

→ out-of-place

DFTrec(m, *x, *y, instride, outstri

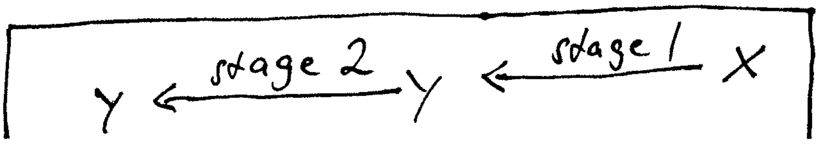
- m: size
- *x: input vector
- *y: output vector

this interface can handle arbitrary recursion

Pseudocode:

```
DFT(n, x, y) = DFTrec(n, x, y, 1, 1)
  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
```

↑ precomputed twiddles



3.) Constants:

The matrix T_m^{km} yields multiplications by constants:

$$y_i = \underbrace{\omega_n^7}_{\text{some root of unity}} x_i$$

which in the code, on real numbers, gives multiplications by sines and cosines

$$y_i = \sin\left(\frac{i\pi}{128}\right) x_i \quad \text{etc....}$$

Problem: Computing sin(..) is very expensive (HW 2)

Solution:
- precompute once
- reuse many times
- assumes a transform for one size is used many times

Changes library interface:

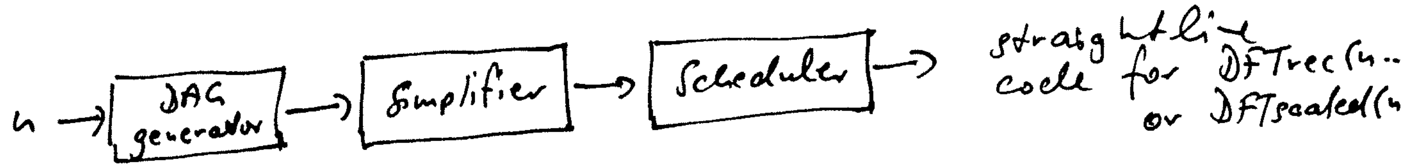
```
d = dft-plan(1024);     // precomputes constants
d(*x, *y);              // computes DFT, size 1024
```

4.) Fast basic blocks

We do not want to recurse all the way to n=2
- function call overhead
- suboptimal register use

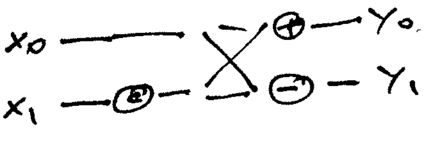
Solution:
- unroll recursion for small enough n
- practice shows $n \leq 32$ is sufficient
- requires 62 functions! Why?

FFTW: "codelet" generator for small size FFT



a.) DAG generator
- generates DAG recursively from stored algorithms
- DAGs have only adds/subs/mults by const

Example:



$$\longleftrightarrow \begin{pmatrix}y_0\\y_1\end{pmatrix} = \begin{pmatrix}1&1\\1&-1\end{pmatrix}\begin{pmatrix}1&\\&c\end{pmatrix}\begin{pmatrix}x_0\\x_1\end{pmatrix}$$

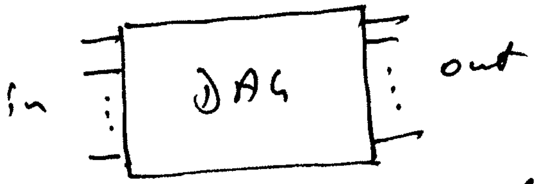
b.) Simplifier

- simplifies mults by 0, 1, -1
- distributivity law: $kx + ky = k(x+y)$
- canonicalization: $x-y, y-x \rightarrow x-y, -(x-y)$
- common subexpression elimination (CSE)
- all constants are made positive: reduces register pressure
- CSE also on transposed DAG

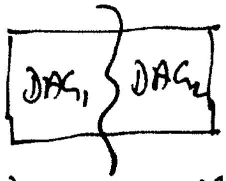
c.) Scheduler:

Theoretical result: 2-power FFT needs $\Omega(\frac{n\log(n)}{R})$ register spills for R registers

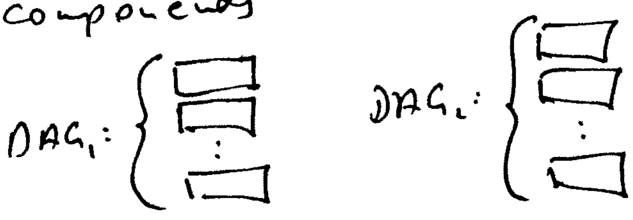
The following algorithm achieves that:



step 1: cut DAG in middle (how to do that



step 2: DAG1, DAG2 decompose into independent components



schedule these recursively

Finally: output straightline, SSA code