A Fast Fourier Transform Compiler

Paper by: Matteo Frigo

MIT Laboratory for Computer Science. February 16, 1999

Presented by: Marco Poltera. November 16, 2011

Software Engineering Seminar

Introduction and motivation

/ Computation of Discrete Fourier transform (DFT) required by many real world applications

real world application, i.e. JPEG compression —uses→ **DFT program,** i.e. FFTW → **result,** i.e. compressed image

Goal

- Look at the internals of FFTW
- Argue that a specialized compiler is a valuable tool

Recap: DFT

- linear transform: $y = Tx$

- DFT:

  $Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{-ij}$ with $\omega_n = e^{2\pi\sqrt{-1}/n}$ (primitive n-root of unity)

  $y = DFT_n x$

- FFT: We can compute $y = Tx = \ (T_1(T_2..(T_m x)))$

Recap: DFT

- $\mathrm{DFT}_4 =$

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -j & -1 & j \\ 1 & -1 & 1 & j \\ 1 & j & -1 & -j \end{pmatrix} = \begin{pmatrix} 1 & & 1 & \\ & 1 & & 1 \\ 1 & & -1 & \\ & 1 & & -1 \end{pmatrix} \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & j \end{pmatrix} \begin{pmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & & & \\ & & 1 & \\ & 1 & & \\ & & & 1 \end{pmatrix}$$

Example from: How to write fast numerical code. Markus Püschel. Carnegie Mellon University. Course 18-645. Lecture 17.

FFTW

- FFTW consists of three parts:

Compiler (genfft)

- run once
- output: codelets

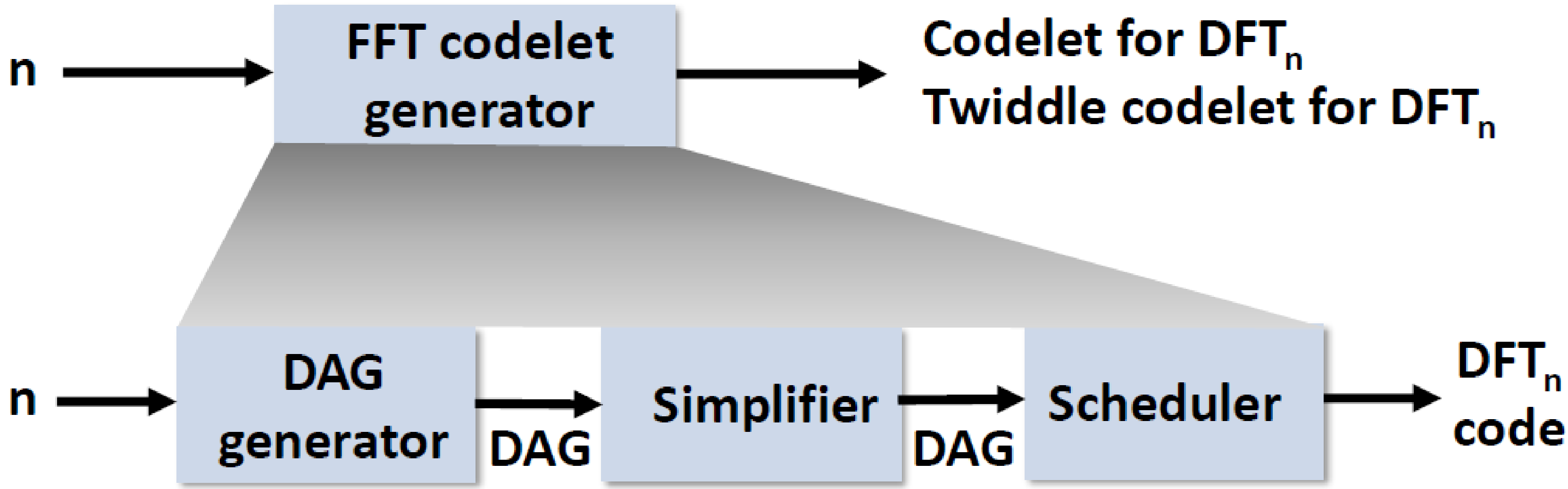
Planner

- run once for every transform size
- hardware adaption
- output: plan
- reusable

Executor

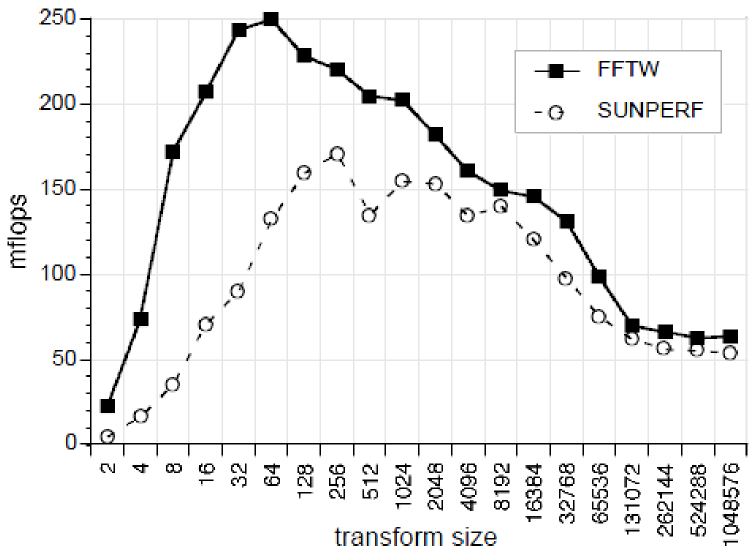
- computes the DFT
- output: transformed vector

FFTW

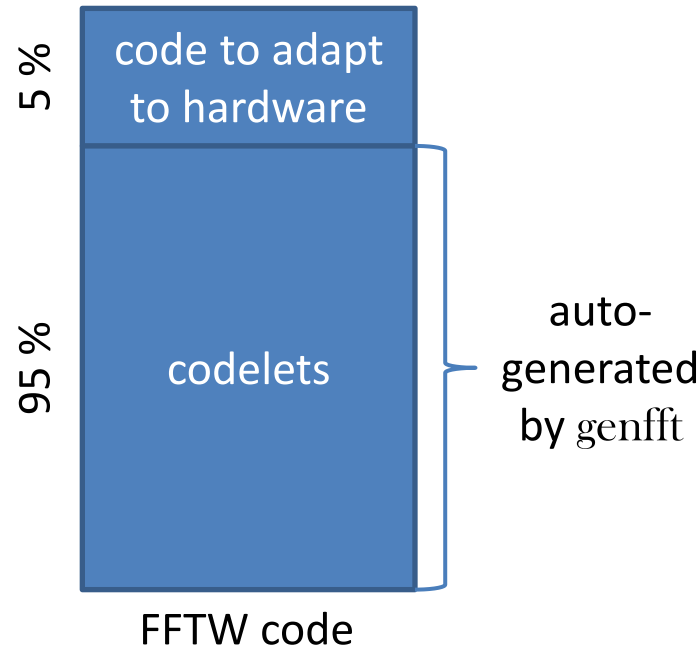


graphic from: How to write fast numerical code. Markus Püschel. Carnegie Mellon University. Course 18-645. Lecture 19.

FFTW

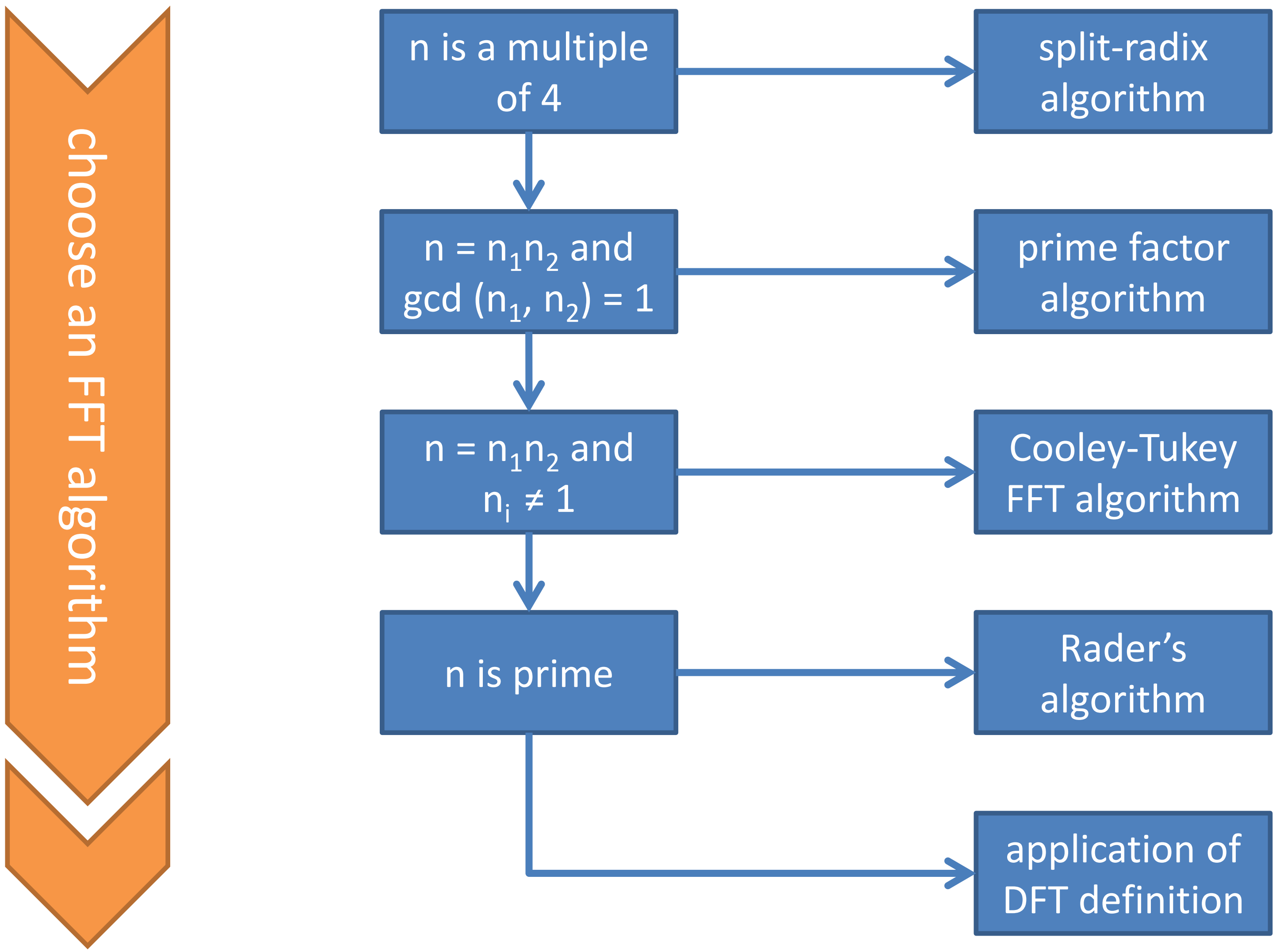


graph from paper



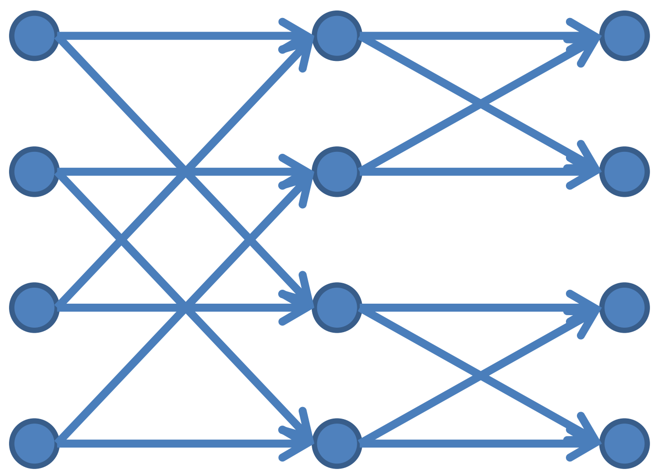
The four phases of genfft

Creation phase



Creation phase

generate dag according to FFT



Creation phase

- Example: Cooley-Tukey algorithm

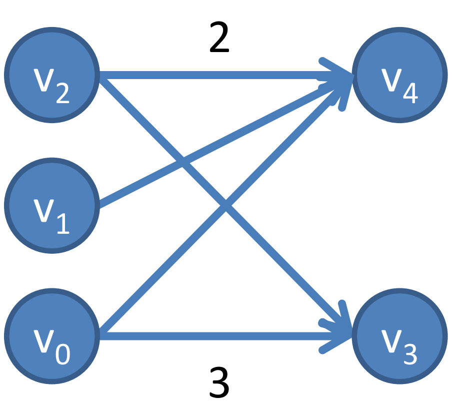
$$Y[i_1 + i_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] \omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right] \omega_{n_2}^{-i_2 j_2}$$

```
let rec cooley_tukey n1 n2 input sign =
    let tmp1 j2 = fftgen n1 (fun j1 -> input (j1*n2+j2)) sign in
    let tmp2 i1 j2 = exp n (sign*i1*j2) @* tmp1 j2 i1 in
    let tmp3 i1= fftgen n2 (tmp2 i1) sign
    in (fun i -> tmp3 (i mod n1) (i/n1))
```

Creation phase

- DAG representation

Type node = Num of Number.number | Load of Variable.variable | Store of Variable.variable * node | Plus of node list | Times of node * node | Uminus of node



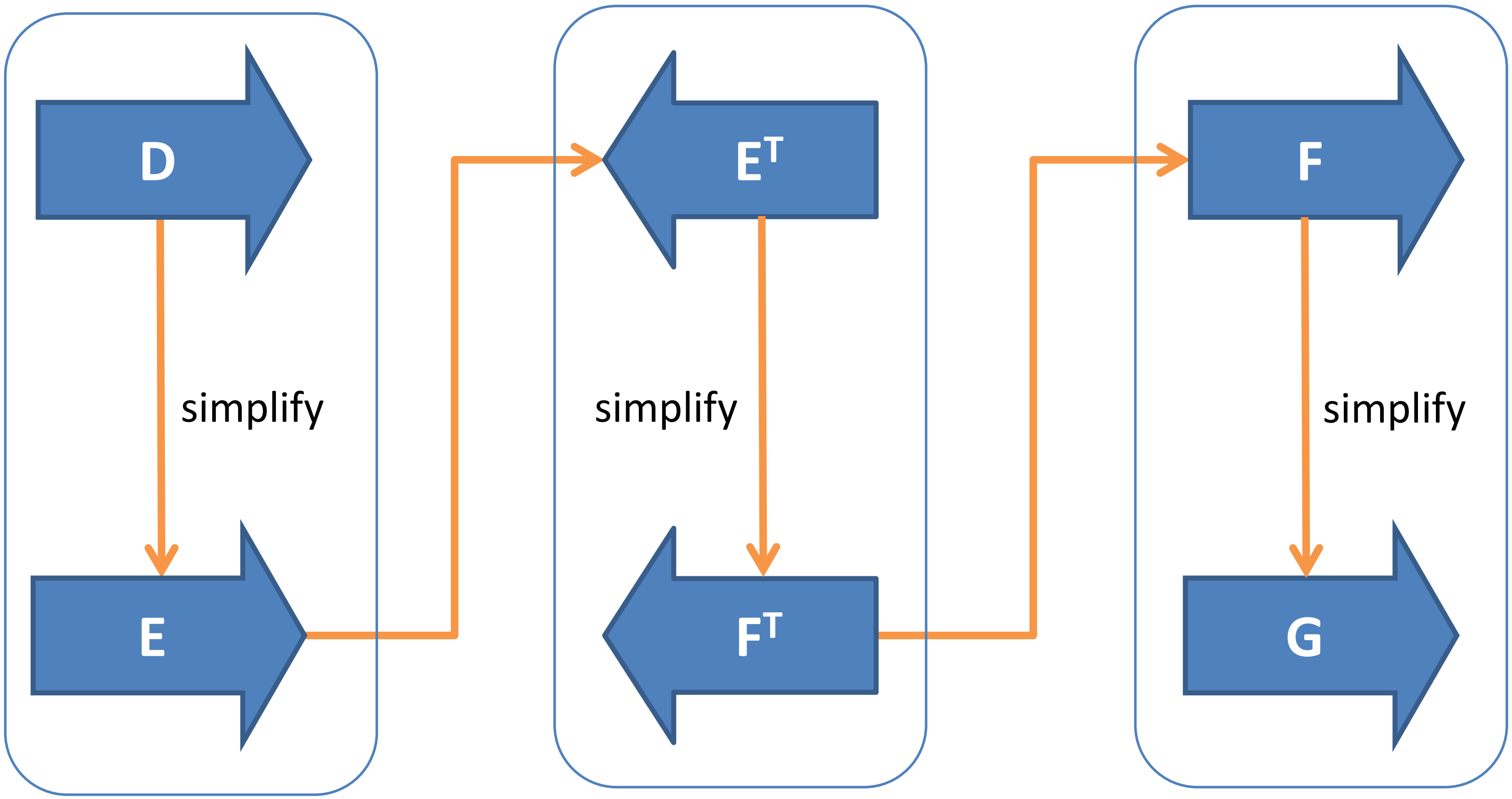
v_3 = Plus [v_2; Times (Num 3, v_0)]
v_4 = Plus [Times (Num 2, v_2); v_1; v_0]

Simplifier

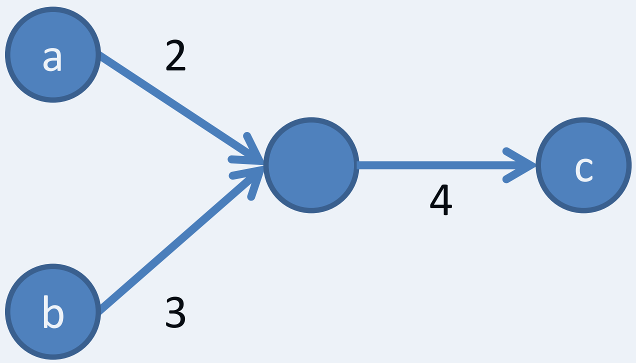
- algebraic transformations
  - i.e. apply distributive property: $kx + ky \rightarrow k(x + y)$
- common-subexpressions
- DFT-specific improvements
  - make numeric constants positive
  - dag transposition

Simplifier: DAG transposition

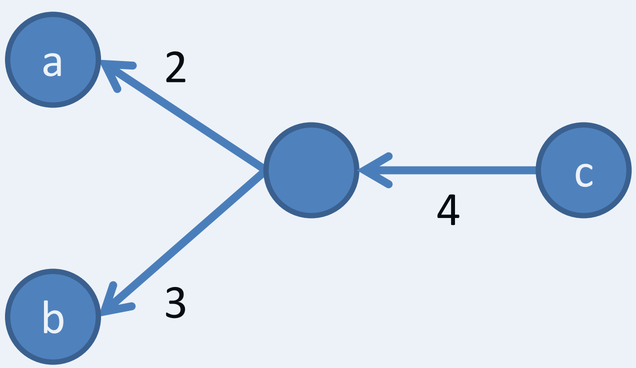
- three passes:



Simplifier: DAG transposition



$$c = 4(2a + 3b)$$



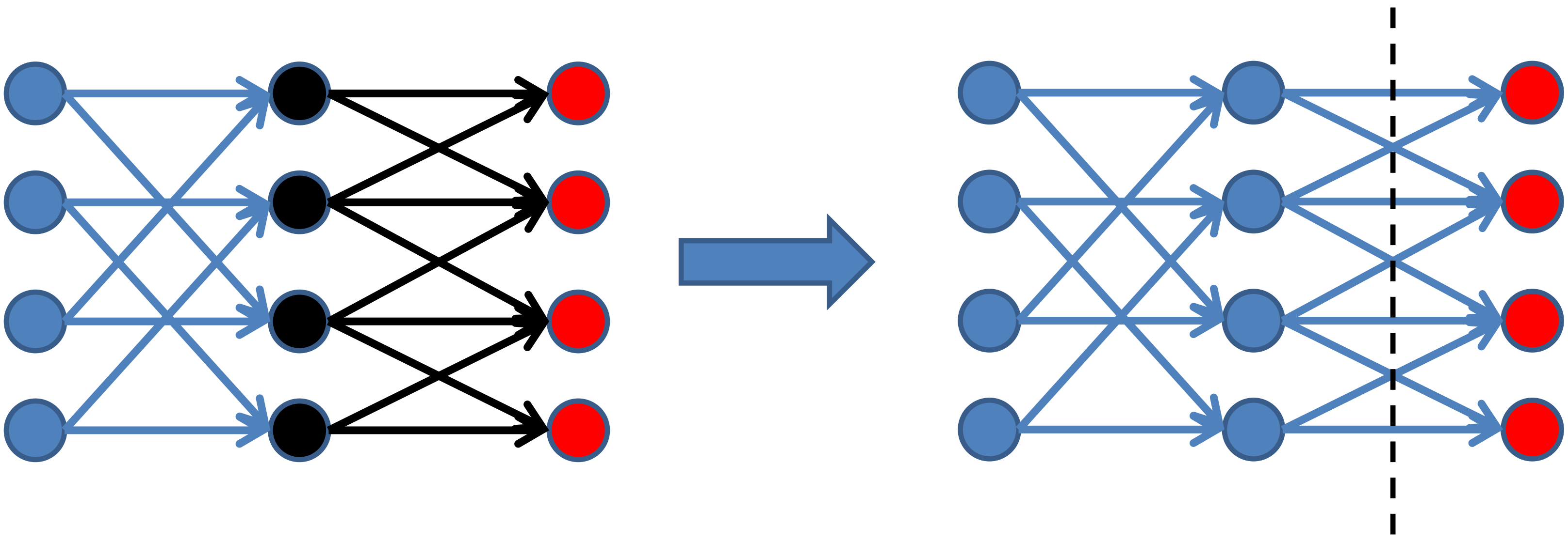
$$\mathrm{a} = 2 * 4c$$
$$b = 3 * 4c$$

Scheduler

Goal
- maximize register usage

/ schedule is cache-oblivious

Scheduler



Scheduler

- #register spills = $\Theta(n \log(n) / \log(R))$

Unparser

- Schedule is unparsed to C

Conclusion

- performance
- rapid turnaround
- effectiveness
- derived new algorithms
- not reduced to a specific language such as C

Further information

- Download FFTW: www.fftw.org

- Details on FFTW: “FFTW: An Adaptive Software Architecture For The FFT” by M. Frigo/S. Johnson (1998)

Usage of FFTW

```
#include <fftw3.h>
...
{
        fftw_complex *in, *out;
        fftw_plan p;
        ...
        in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
        out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * N);
        p = fftw_plan_dft_1d(N, in, out, FFTW_FORWARD, FFTW_ESTIMATE);
        ...
        fftw_execute(p); /* repeat as needed */
        ...
        fftw_destroy_plan(p);
        fftw_free(in); fftw_free(out);
}
```

from the tutorial included in the FFTW distribution 3.3

DFT

- FFT refers to
  - any $O(N \log N)$ algorithm or
  - the specific Cooley-Tukey algorithm
- computing a DFT of N points takes
  - in the naive way, using the definition, $O(N^2)$ arithmetical operations
  - $O(N \log N)$ operations for a FFT

FFTW and Parallelism

- Parallel versions are available for
  - Cilk
  - Posix threads
  - MPI

Simplifier

- Implementation:
  - simplifier written as if it was an expression *tree*
  - mapping from trees to DAGs accomplished by memoization which is performed implicitly by a monad

Pragmatic aspects of genfft

- running time

- memory requirements