FFTW:

An Adaptive Software Architecture for the FFT

Steven G. Johnson
Condensed Matter Theory
MIT Physics Department

Matteo Frigo
Supercomputing Technologies Group
MIT Laboratory for Computer Science

The Fastest Fourier Transform in the West

Steven G. Johnson
Condensed Matter Theory
MIT Physics Department

Matteo Frigo
Supercomputing Technologies Group
MIT Laboratory for Computer Science

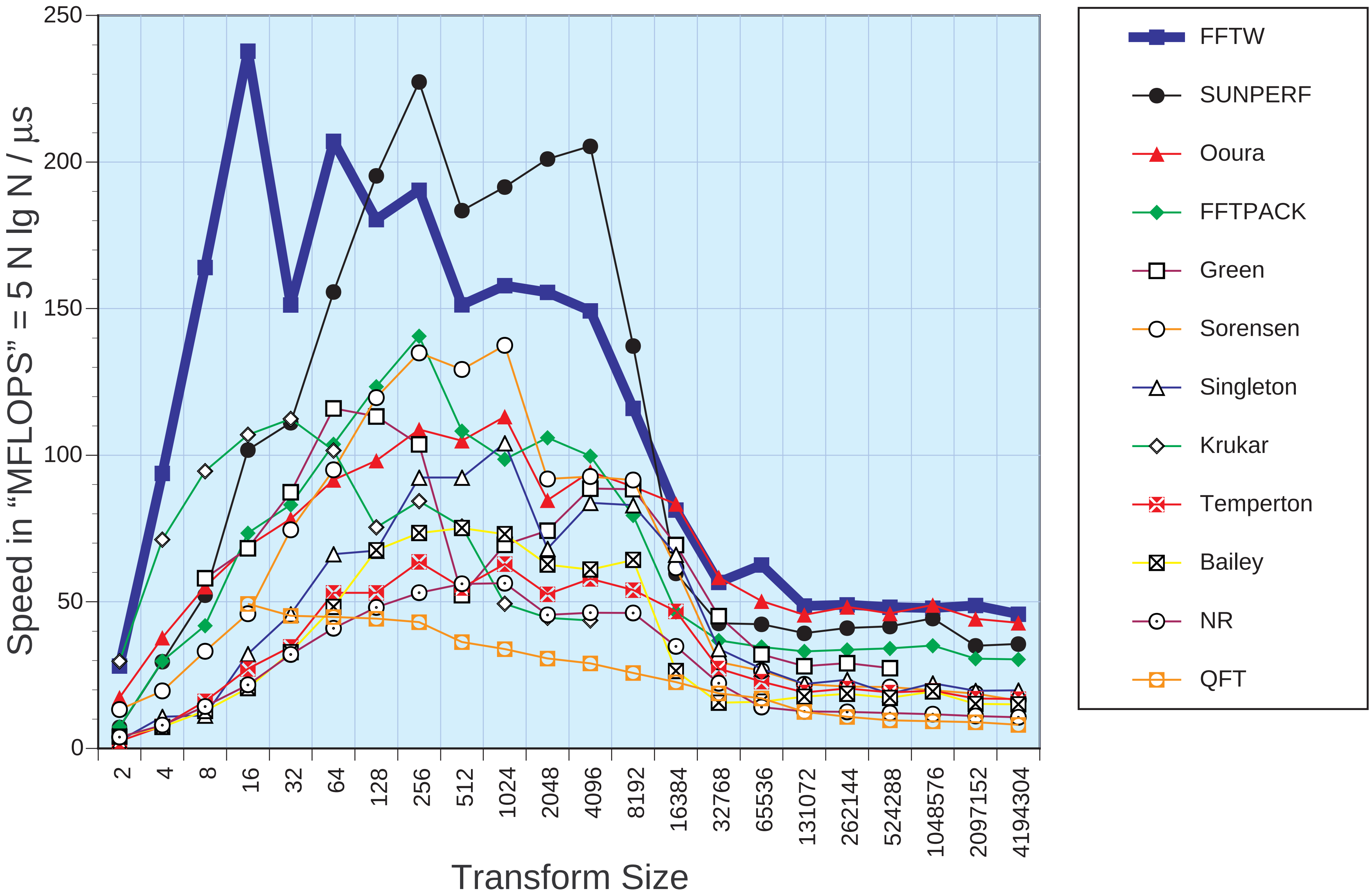
FFTW is:

- A C library for computing the FFT
  - in one or more dimensions
  - includes parallel transforms
- Fast
  - competitive with vendor-tuned libraries
  - FFTW's performance is portable
- Here today (since 3/97)
  - ...with thousands of happy users
  - `http://theory.lcs.mit.edu/~fftw`

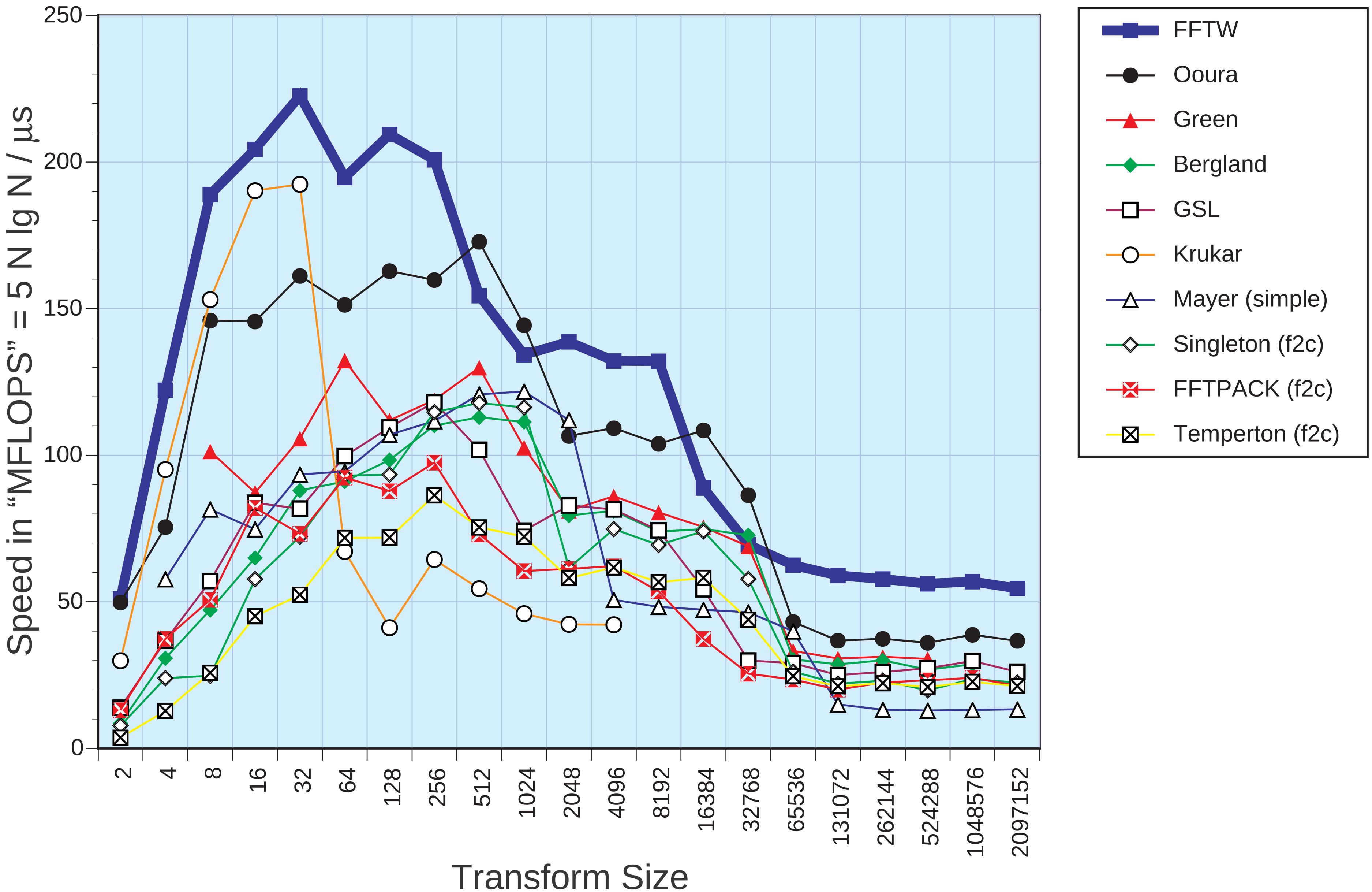
FFTW is Not:

- **Always the fastest code**
  - **but is the fastest more often than any other program**
- **A new FFT algorithm per se**
  - **it is a new way of implementing known algorithms**

FFT Benchmark on a 167MHz UltraSPARC-I (xolas)



FFT Benchmark on a 300MHz Pentium II



We Also Have Results From...

- Over 40 FFT codes
- 19 different machines
- Both 1D and 3D transforms
- Transform sizes not a power of 2

Why is FFTW So Fast?

- **The Runtime Planner**
  - **optimizes FFTW for your CPU, your cache size, etcetera**
- **The Codelets**
  - **composable blocks of optimized code**
  - **computer generated**
- **The Executor**
  - **interprets the plan to compose the codelets**
  - **includes a few tricks of its own**
- **But first, a review of the FFT...**

FFTW

The Cooley-Tukey FFT Algorithm

- **Computes a DFT of size N = N1 * N2**
  - **first, does N1 transforms of size N2**
  - **then, multiplies by “twiddle factors”**
  - **finally, does N2 transforms of size N1**
- **Performance is O(N lg N)**
- **Base cases of recursion are optimized small transforms**

The Planner

- For a given N, there are many factorizations
  - not clear a priori which is best
- The planner tries them "all" and picks the best one
  - uses actual runtime timing measurements
  - result is encoded in a "plan"
- Uses dynamic programming to reduce number of possible plans
  - remembers optimal sub-plans for small sizes

FFTW

Example Plans

- **N=32768 on Alpha and Pentium II**

Alpha:

```
RADIX 16: 32768 -> 16*2048
RADIX  8:  2048 -> 8*256
RADIX  8:   256 -> 8*32
SOLVE 32
```

Pentium II:

```
RADIX 64: 32768 -> 64*512
RADIX 16:   512 -> 16*32
SOLVE 32
```

The Codelet Generator

- **Generates highly-optimized transforms of small sizes ("codelets")**
  - **with and without twiddle factors**
  - **form the base cases of the FFT recursion**
- **Written in the Caml-Light dialect of ML**
- **Manipulates abstract syntax tree which is unparsed to C**
  - **knows about complex arithmetic, etcetera**

Advantages of Generating Codelets

- Long, unrolled code takes advantage of:
  - optimizing compilers
    - instruction scheduling, etcetera
  - large register sets
- Applies tedious optimizations
- Easy to experiment with different algorithms
  - prime factor, split-radix, Rader
    - express the algorithm once, abstractly
  - various optimization hacks
- You only have to get it right once

The Expression Simplifier

- Here is a fragment that helps simplify multiplications:

```
simplify_times = fun
            (Real a) (Real b) -> (Real (a *. b))
           | a (Real b) -> simplify_times (Real b) a
           | (Uminus a) b -> simplify (Uminus (Times (a,b)))
           | (Real a) (Times ((Real b), c)) ->
               simplify (Times ((Real (a *. b)), c))
           | (Real a) b -> if (almost_equal a 0.0) then (Real 0.0
                           else if (almost_equal a 1.0) then b
                           else if (almost_equal a (-1.0))then
                               simplify (Uminus b)
                           else Times ((Real a), b)
           | ...
```

FFTW

Tricky Optimizing Rules

- **Quiz: which of the following is faster?**

```
a = 0.5 * b;
c = 0.5 * d;
e = 1.0 + a;
f = 1.0 - c;
```

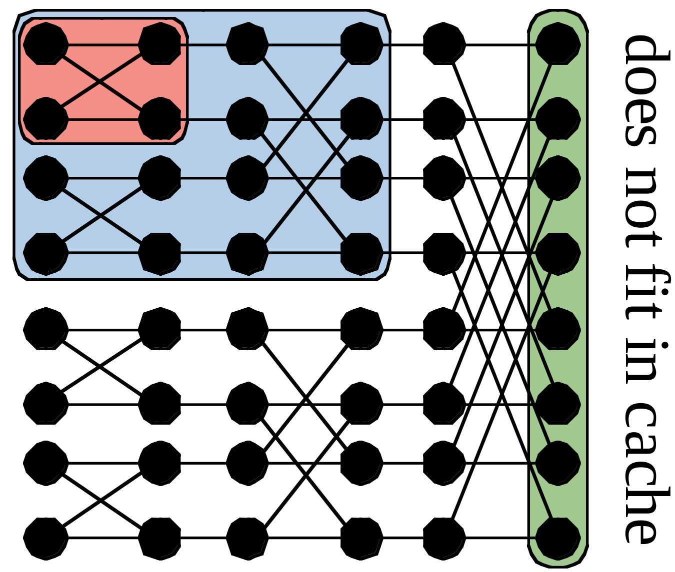
```
a = 0.5 * b;
c = -0.5 * d;
e = 1.0 + a;
f = 1.0 + c;
```

Answer: the fragment on the left.

The number of floating-point constants should be minimized.

The Executor

- **Executes the plan by composing codelets**
- **Explicit recursion**
  - **divide-and-conquer uses all levels of the memory hierarchy**
- **Novel storage for the twiddle factors**
  - **store them in the order they are used**



FFTW

FFTW is Easy to Use

- **Complexity is abstracted from the user:**

```
COMPLEX A[n], B[n];
fftw_plan plan;

/* create the plan: */
plan = fftw_create_plan(n);

/* use the plan: */
fftw(plan,A);

/* re-use the plan: */
fftw(plan,B);
```

Parting Thought: This is Ridiculous!

- All this for an FFT?
- Modern architectures are invalidating conventional wisdom about what is fast
  - no new wisdom is emerging
- In the name of performance, designers have sacrificed:
  - predictability
  - repeatability
  - composability
- Hand-optimization of programs is becoming impractical