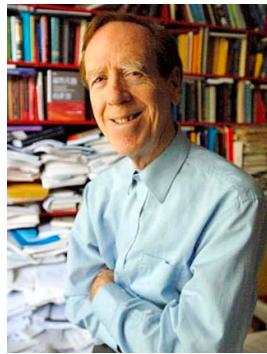


Fast Fourier Transforms (FFT algorithms)

“the most important numerical algorithm of our lifetime”
— Gil Strang, MIT



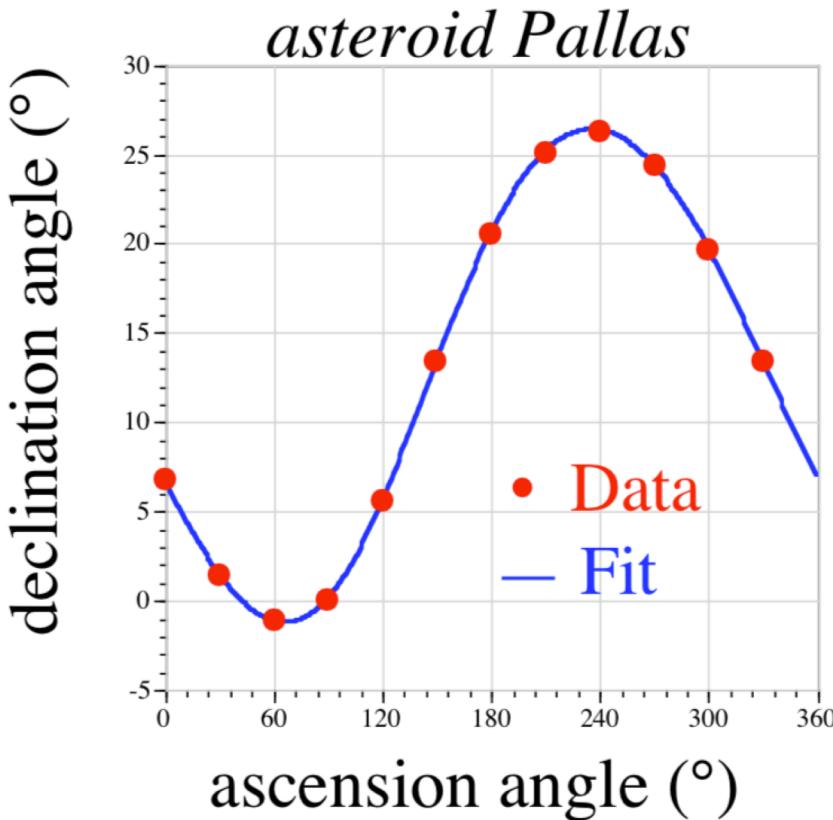
one of

- “Top 10 Algorithms of 20th Century”
- Jack Dongarra & F. Sullivan, *Computing in Science and Engineering* (2000)



MIT course 18.335, spring 2021
Steven G. Johnson MIT Applied Math

In the beginning (c. 1805): Carl Friedrich Gauss



discrete Fourier transform (DFT):
(before Fourier)

trigonometric interpolation:

$$y_j = \sum_{k=0}^{n-1} c_k e^{i \frac{2\pi}{n} kj}$$

generalizing work
of Clairaut (1754)
and Lagrange (1762)

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} y_j e^{-i \frac{2\pi}{n} kj}$$

Gauss' DFT notation:

From “*Theoria interpolationis methodo nova tractata*”

Quum haec formula indefinite pro valore quocunque ipsius t locum habeat, manifestum est, si producta sinuum in numeratoribus in cosinus sinusque arcuum multiplicium evolvantur, id quod provenit cum

$$\begin{aligned} \alpha + \alpha' \cos t + \alpha'' \cos 2t + \alpha''' \cos 3t + \text{etc.} \\ + \beta' \sin t + \beta'' \sin 2t + \beta''' \sin 3t + \text{etc.} \end{aligned}$$

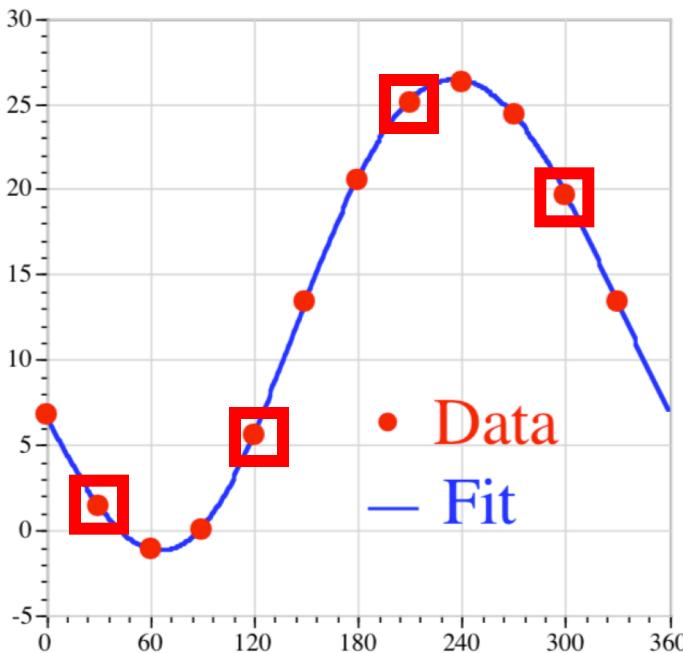
identicum esse debere, unde coëfficientes $\alpha, \alpha', \beta', \alpha'', \beta''$ etc. innotescunt. Ceterum formula pro T , ut hic exhibita est, ita est comparata, ut sponte et sine calculo pateat, substitutis pro t resp. a, b, c, d etc. valoribus propositis A, B, C, D etc. probe satisfieri.

Kids: don't try this at home!

Gauss' fast Fourier transform (FFT)

how do we compute: $c_k = \frac{1}{n} \sum_{k=0}^{n-1} y_j e^{-\frac{2\pi}{n} k j}$?

- not directly: $O(n^2)$ operations ... for Gauss, $n=12$



Gauss' insight: “*Distribuamus hanc periodum primo in tres periodos quaternionorum terminorum.*”

= We first distribute this period [$n=12$] into 3 periods of length 4 ...

Divide and conquer.
(any composite n)

But how fast was it?

“illam vero methodum calculi mechanici taedium magis minuere”

= “truly, this method greatly reduces
the tedium of mechanical calculation”

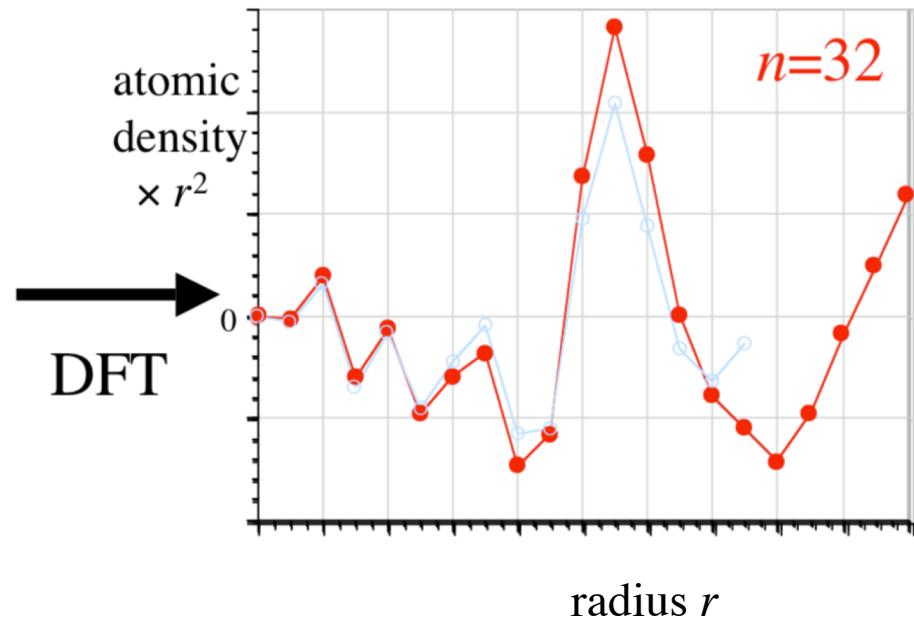
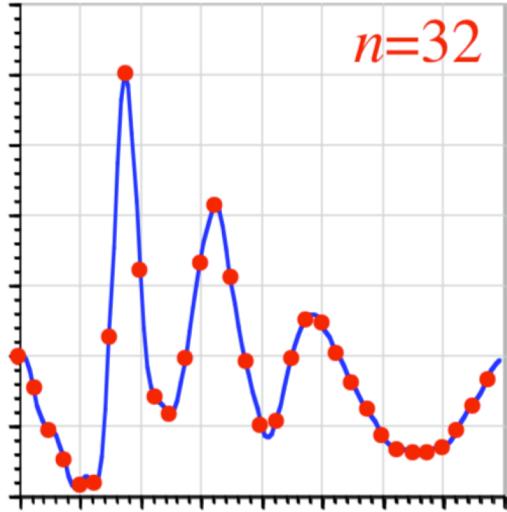
(For Gauss, being less boring was good enough.)

two (of many) re-inventors: Danielson and Lanczos (1942)

[*J. Franklin Inst.* **233**, 365–380 and 435–452]

Given Fourier transform of density (X-ray scattering) find density:
discrete sine transform (DST-1) = **DFT** of real, odd-symmetry

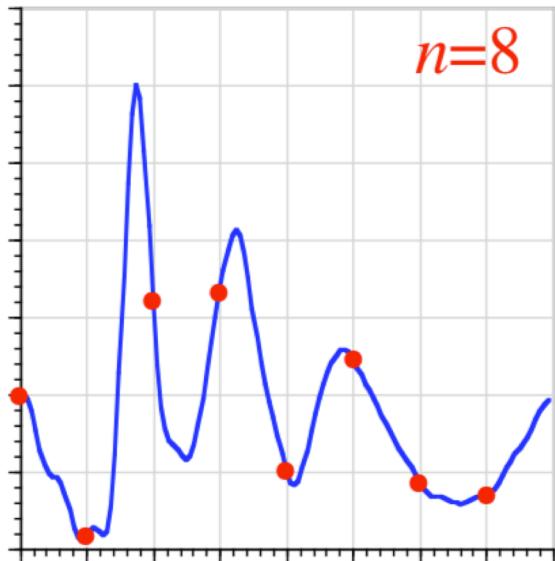
sample
the spectrum
at n points:



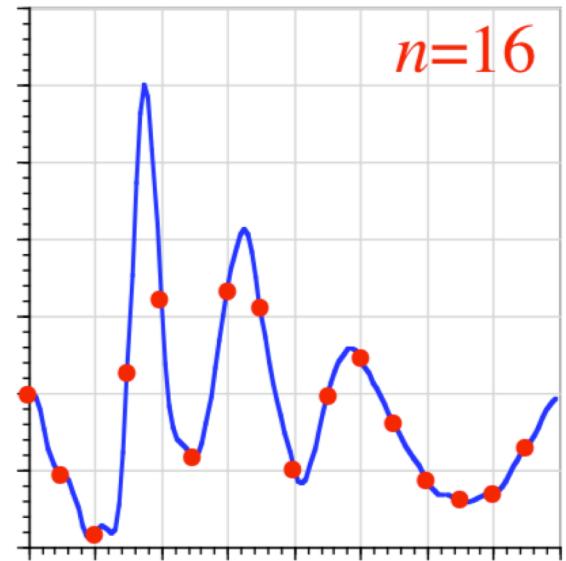
...double sampling until density (DFT) converges...

Gauss' FFT *in reverse*: Danielson and Lanczos (1942)

[*J. Franklin Inst.* **233**, 365–380 and 435–452]



→
double sampling
re-using results



“By a certain transformation process, it is possible to double the number of ordinates with only slightly more than double the labor.”

from
 $O(n^2)$ to ???

64-point DST in *only 140 minutes!*

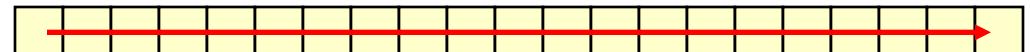
re-inventing Gauss (for the last time)

[*Math. Comp.* **19**,
297–301]

Cooley and Tukey (1965)

$$N = N_1 N_2$$

1d DFT of size N :



= \sim 2d DFT of size $N_1 \times N_2$

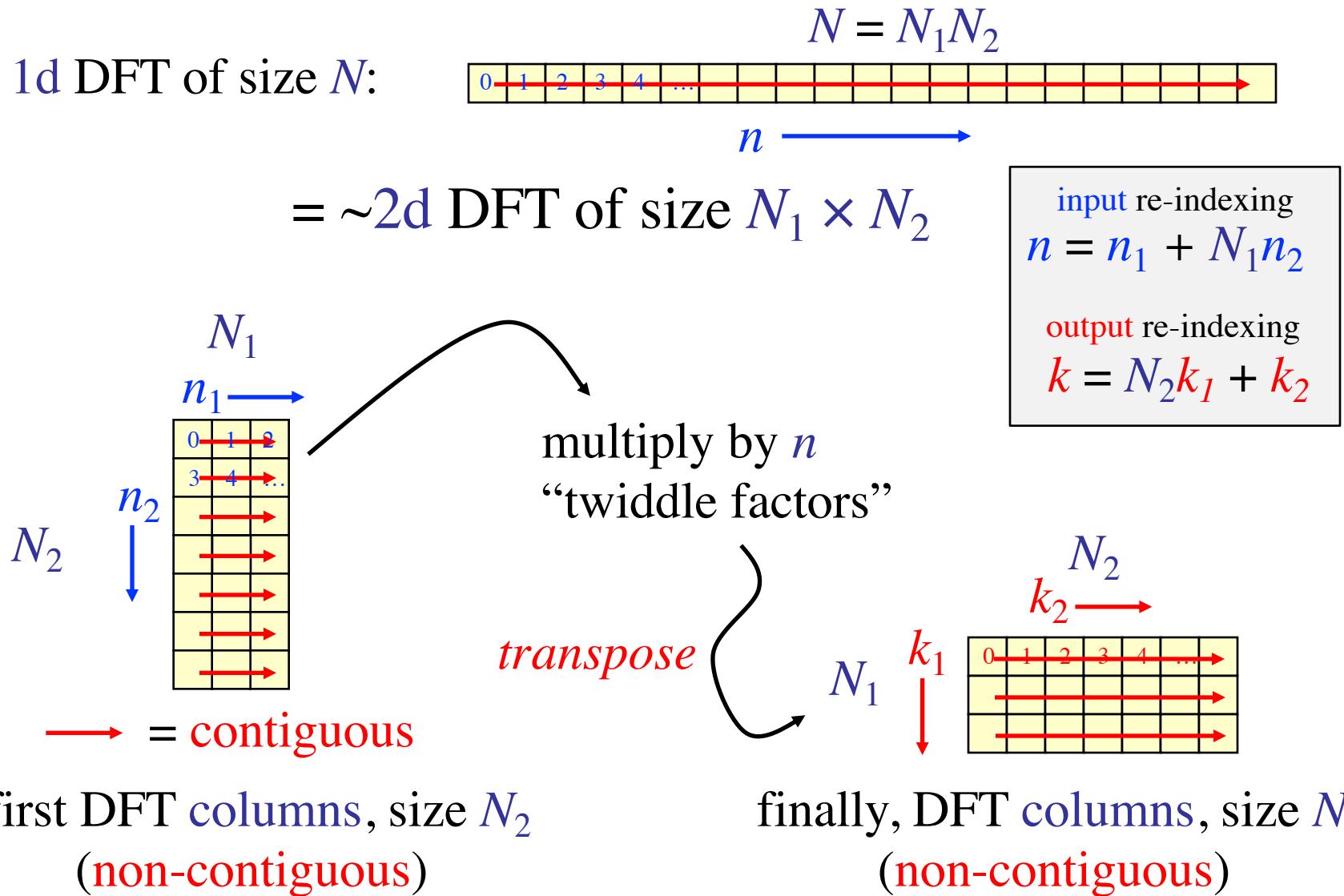
(+ phase rotation by twiddle factors)

= Recursive DFTs of sizes N_1 and N_2

$$\mathcal{O}(N^2) \longrightarrow \mathcal{O}(N \log N)$$

$n=2048$, IBM 7094, 36-bit float: 1.2 seconds
($\sim 10^6$ speedup vs. Dan./Lanc.)

The “Cooley-Tukey” FFT Algorithm



“Cooley-Tukey” FFT, in math

Recall the definition of the DFT:

$$y_k = \sum_{n=0}^{N-1} \omega_N^{nk} x_n \quad \text{where} \quad \omega_N = e^{-\frac{2\pi i}{N}}$$

Trick: if $N = N_1 N_2$, re-index $n = n_1 + N_1 n_2$ and $k = N_2 k_1 + k_2$:

$$\begin{aligned} y_{N_2 k_1 + k_2} &= \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \omega_N^{n_1 N_2 k_1} \omega_N^{n_1 k_2} \omega_N^{N_1 n_2 N_2 k_1} \omega_N^{N_1 n_2 k_2} x_{n_1 + N_1 n_2} \\ &= \sum_{n_1=0}^{N_1-1} \omega_{N_1}^{n_1 k_1} \omega_N^{n_1 k_2} \left(\sum_{n_2=0}^{N_2-1} \omega_{N_2}^{n_2 k_2} x_{n_1 + N_1 n_2} \right) \end{aligned}$$

size- N_1 DFTs twiddles size- N_2 DFTs

... repeat recursively.

Cooley–Tukey terminology

- Usually N_1 or N_2 is small, called *radix r*
 - N_1 is radix: “decimation in time” (DIT)
 - N_2 is radix: “decimation in frequency” (DIF)
- Size- r DFTs of radix: “**butterflies**”
 - Cooley & Tukey *erroneously* claimed $r=3$ “optimal”: they thought butterflies were $\Theta(r^2)$
 - In fact, $r \approx \sqrt{N}$ is optimal cache-oblivious
- “Mixed-radix” uses different radices at different stages (different factors of n)

Why is Cooley–Tukey $\Theta(N \log N)$?

Take the **radix-2 DIT** case, and suppose $N=2^m$.

Size- N DFT is expressed as **2 size- $N/2$ DFTs** +
multiplication by **$\Theta(N)$ twiddle factors** + **$N/2$ size-2 DFTs**:

flop count for
size- $N=2^m$ DFT:

$$\begin{aligned} T(N) &= 2T\left(\frac{N}{2}\right) + \Theta(N) + \frac{N}{2}T(2) \\ &\quad \overbrace{\qquad\qquad\qquad}^{\#N} \\ &\approx 2T\left(\frac{N}{2}\right) + \approx 2T\left(\frac{N}{2}\right) \\ &= 2 \left[2T\left(\frac{N}{4}\right) + \# \frac{N}{2} \right] + \#N = 4T\left(\frac{N}{4}\right) + \#N(1+1) \\ &= \dots = NT(1) + \underbrace{\#N(1+1+\dots+1)}_{m \text{ times}} = \#\text{N} \log_2 N + O(N) \end{aligned}$$

...similarly for *any radices*, as
long as the **base cases**
(= the prime factors of N)
are **bounded**.

Note that the **radices** do **not** need to be bounded!
It is a good exercise to show that **radix-** \sqrt{N} also
gives a $\Theta(N \log N)$ algorithm.
(this algorithm turns out to be the **optimal**
cache-oblivious FFT, actually.)

Many other FFT algorithms

- Prime-factor algorithm: $N = N_1 N_2$ where N_1 and N_2 are co-prime: re-indexing based on Chinese Remainder Theorem with no twiddle factors.
- Rader's algorithm: for prime N , re-index using generator of multiplicative group to get a convolution of size $N-1$, do via FFTs.
- Bluestein's algorithm: re-index using $nk = -\frac{1}{2}(k-n)^2 + \frac{n^2}{2} + \frac{k^2}{2}$ to get convolution of size N , do via zero-padded FFTs.
- Many others...
- Specialized versions for real x_n , real-symmetric/antisymmetric x_n (DCTs and DSTs), etc.

...but how do we make it faster?

We (probably) cannot do better than $\Theta(n \log n)$.

(the proof of this remains an open problem
... can prove it only under restrictive assumptions,
e.g. linear algorithms with bounded multiplicative constants)

[unless we give up exactness]

We're left with the “constant” factor...

The Next 30 Years...

Assume “time”

~~= # multiplications~~

multiplications + # additions (= flops)

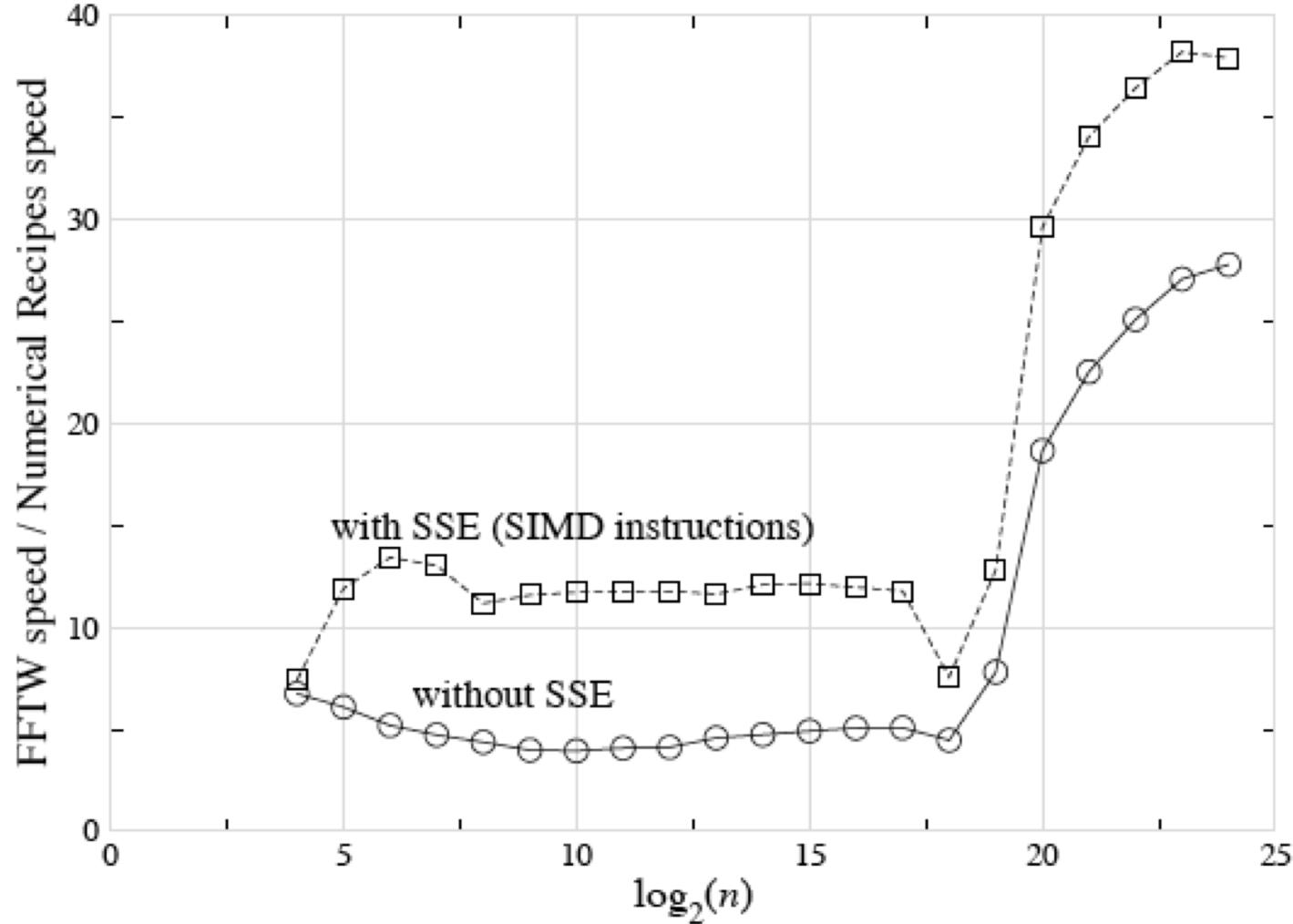
Winograd (1979): # multiplications = $\Theta(n)$

(...realizable bound! ... but costs too many additions)

Yavne (1968): split-radix FFT, saves 20% over radix-2 flops

[unsurpassed until last 2007, another ~6% saved
by Lundy/Van Buskirk and Johnson/Frigo]

Are arithmetic counts so important?



The Next 30 Years...

Assume “time”

= ~~# multiplications~~

multiplications + # additions (= flops)

Winograd (1979): # multiplications = $\Theta(n)$

(...realizable bound! ... but costs too many additions)

Yavne (1968): split-radix FFT, saves 20% over radix-2 flops
[unsurpassed until last 2007, another ~6% saved]

last 15+ years: flop count (varies by ~20%)

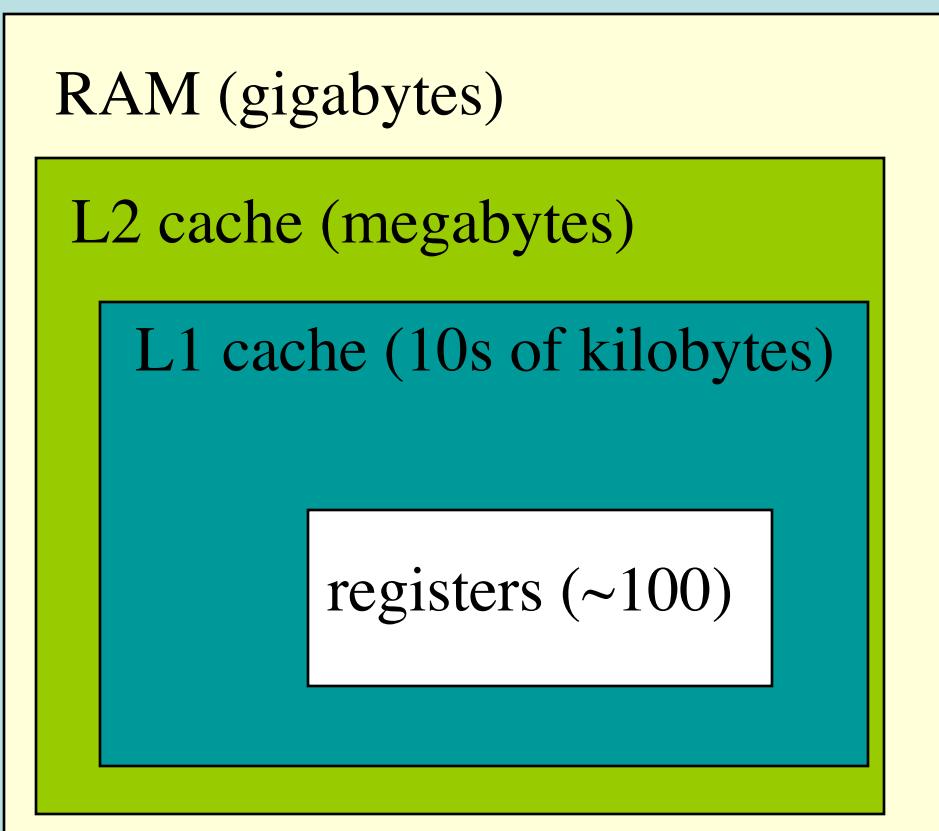
no longer determines speed (varies by factor of ~10+)

a basic question:

If arithmetic no longer dominates,
what does?

The Memory Hierarchy (not to scale)

disk (out of core) / remote memory (parallel)
(terabytes)



...what matters is not how much work you do, but *when* and *where* you do it.

the name of the game:

- do as much work as possible before going out of cache

...difficult for FFTs
...many complications
...continually changing

What's the fastest algorithm for _____?

(computer science = math + time = math + \$)

- 1 Find best asymptotic complexity
naïve DFT to FFT: $O(n^2)$ to $O(n \log n)$
- 2 ~~Find best exact operation count?~~
- 3 Find variant/implementation that runs fastest
hardware-dependent — **unstable answer!**

Better to change the question...

A question with a more stable answer?

What's the smallest
set of “simple” algorithmic steps
whose compositions ~always
span the ~fastest algorithm?

Lots of choices in FFT implementation

- What is the **base case**? $N=1$ is elegant but inefficient.
- What **factorization of N** (“radices”)?
- **Ordering** of loops, memory layout?
- **Other FFT algorithms** besides Cooley–Tukey?
 - Cooley–Tukey is only for composite N (often just 2^m), but there are FFT algorithms even for prime N .



the “Fastest
Fourier Transform
in the West”

S. G. Johnson
& Matteo Frigo
(1997)

- C library for real & complex FFTs (arbitrary size/dimensionality)
(+ parallel versions for threads & MPI)
- Computational kernels (80% of code) automatically generated
- Self-optimizes for your hardware (picks best composition of steps)
= portability + performance

free software: <http://www.fftw.org>

Julia interface: **FFTW** package
Matlab: it's the default `fft(...)`



Started (1997) with a practical need and a benchmark

- I was solving Maxwell's equations with FFTs and needed:
 $3d$, not just 2^m , parallel, portable.
- My friend Matteo suggested his Cilk FFT ...
- I downloaded a bunch of other FFTs, benchmarked them, and put up a web page with results ... no clear winner. Matteo:
- Could we do better? Key principles:
 - Benchmark against every competitor you can — otherwise you can't know if you did a good job.
 - Make it fast and portable, but...
 - Don't sacrifice generality / practicality.



(& I turned it in as final project
in 18.337 taught by ...
... Alan Edelman)

Why is FFTW fast?

FFTW implements many FFT algorithms:

A planner picks the **best composition** (*plan*)
by **measuring** the speed of different combinations.

Three ideas:

- 1 A recursive framework enhances locality.
- 2 Computational kernels (“codelets”)
should be automatically generated.
- 3 Self-optimizes algorithm compositions ...
the unit of composition is critical.

What does the planner compose?

- The Cooley-Tukey algorithm presents many choices:
 - which factorization? what order? memory reshuffling?

Find simple steps that combine without restriction
to form many different algorithms.

... steps to do WHAT?

FFTW 1 (1997): steps solve out-of-place DFT of size **n**

“Composable” Steps in FFTW 1

SOLVE — Directly solve a small DFT by a **codelet**

CT-FACTOR[r] — Radix- r Cooley-Tukey step =
execute loop of r sub-problems of size n/r

- X** Many algorithms difficult to express via simple steps.
- e.g. expresses **only depth-first** recursion
(loop is *outside* of sub-problem)
 - e.g. **in-place without bit-reversal**
requires combining
two CT steps (DIT + DIF) + transpose

What does the planner compose?

- The Cooley-Tukey algorithm presents many choices:
 - which factorization? what order? memory reshuffling?

Find simple steps that combine without restriction
to form many different algorithms.

... steps to do WHAT?

FFTW 1 (1997): steps solve out-of-place DFT of size n

Steps cannot solve problems that cannot be expressed.

What does the planner compose?

- The Cooley-Tukey algorithm presents many choices:
 - which factorization? what order? memory reshuffling?

Find simple steps that combine without restriction
to form many different algorithms.

... steps to do WHAT?

FFTW 3 (2003):

steps solve a problem, specified as a DFT(input/output, \mathbf{v}, \mathbf{n}):
multi-dimensional “vector loops” \mathbf{v} of multi-dimensional transforms \mathbf{n}

{sets of (size, input/output strides)}

Key component: The “Codelet” Generator

a domain-specific FFT “compiler”

- Generates fast hard-coded C for FFT of a given size

Necessary to give the planner a large space of base cases (“codelets”) to experiment with (any factorization).

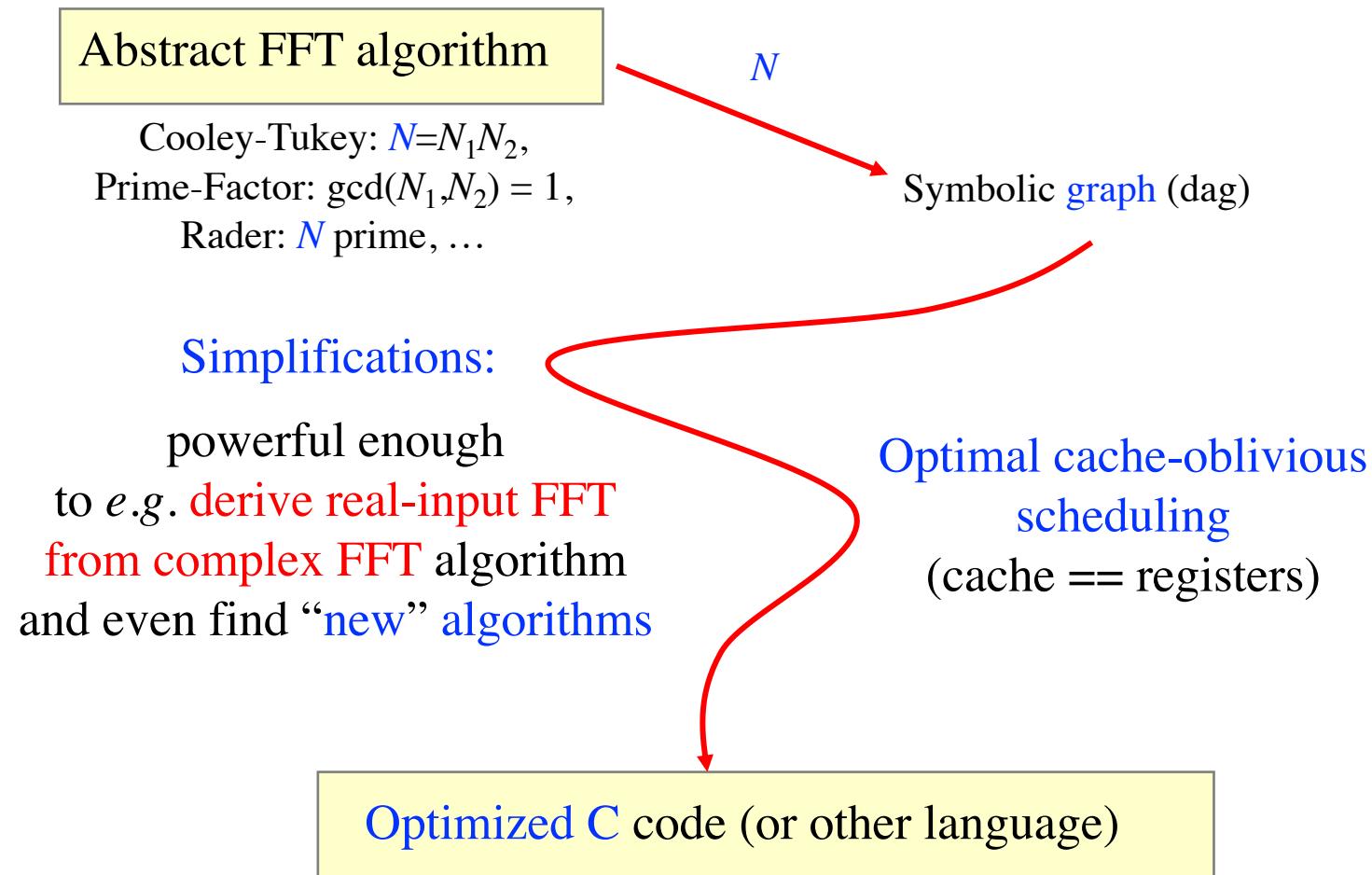
Exploits modern CPU deep pipelines & large register sets.

Allows easy experimentation with different optimizations & algorithms.
+ CPU-specific hacks (SIMD)...

(& large base case negates recursion overhead: “recursion coarsening”)

The Codelet Generator

written in Objective Caml [Leroy, 1998], an ML dialect

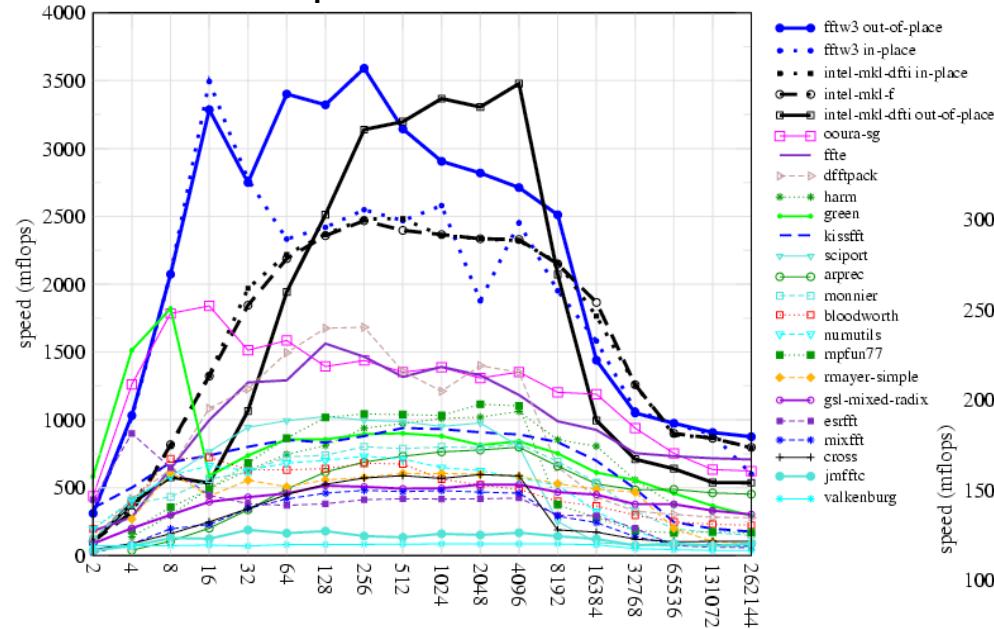


FFTW performance

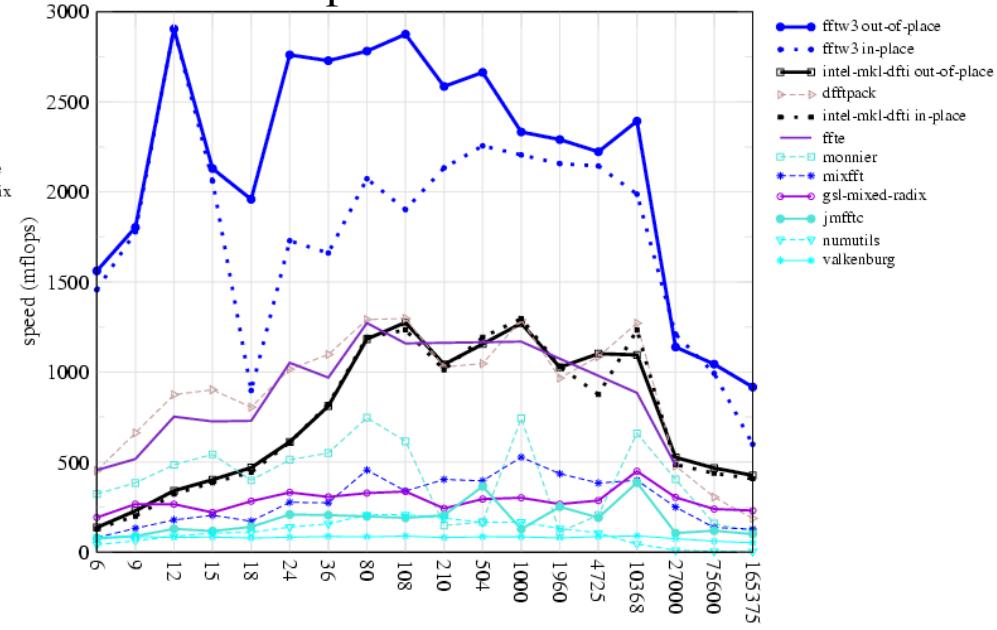
double precision, most recent benchmarks on fftw.org

... a 2.8GHz Pentium IV Prescott (2004)
Maybe I need to re-run the benchmark?

powers of two



non-powers-of-two

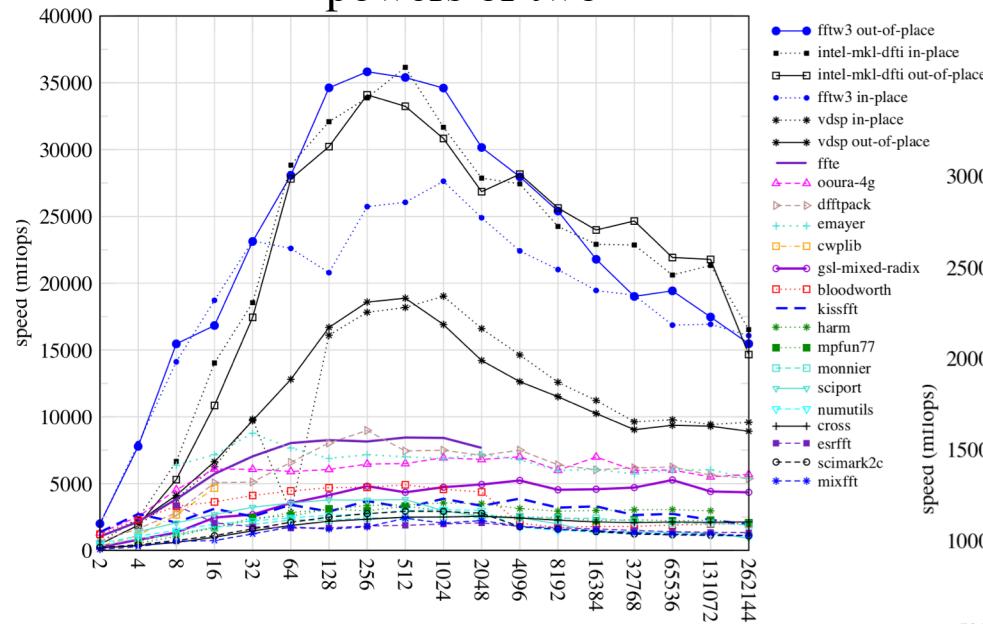


(If you don't hand code or generate SIMD instructions,
you lose ... compilers won't save you for FFTs.)

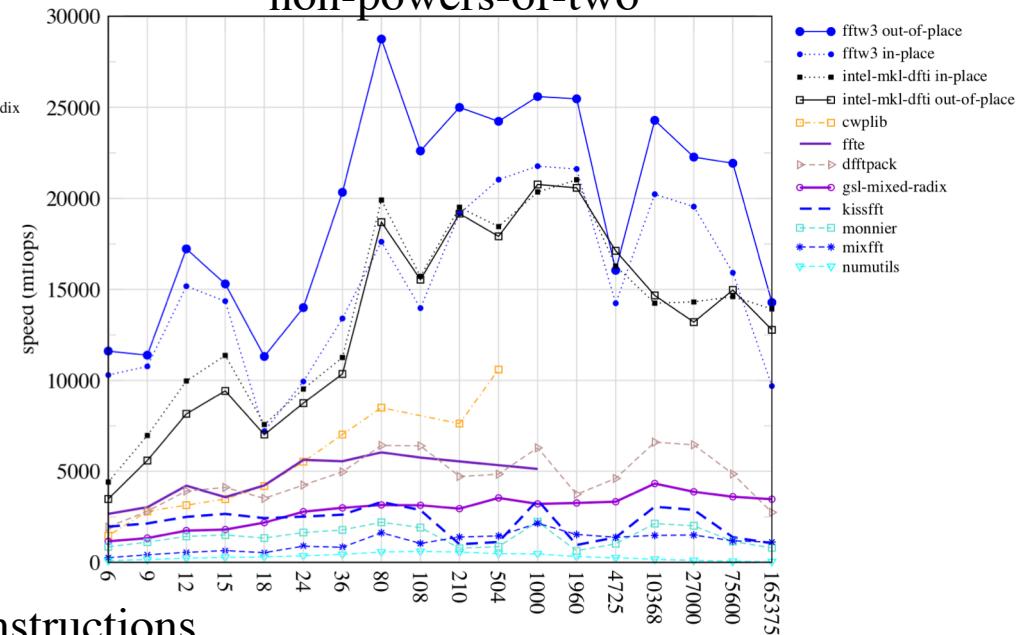
Updated FFTW performance

double precision, 3GHz Intel Core i5 (2018 Mac mini)

powers of two



non-powers-of-two



(If you don't hand code or generate SIMD instructions,
you really lose ... AVX2 etc. getting really wide.)

We've Come a Long Way?

- In the name of performance, computers have become complex & unpredictable.
- Optimization is hard: simple heuristics (*e.g.* fewest flops) no longer work.

- One solution is to avoid the details, not embrace them:
(Recursive) composition of simple modules
+ feedback (self-optimization)

High-level languages (not C) & code generation
are a powerful tool for high performance.

A closing suggestion for numerical software

- Performance **gets people's attention.**
 - ... they don't take you seriously without it.
 - ... the “last factor of 2” is *really* hard to get,
& not worth it for most problems.
- **Usability** & generality **keeps** the attention.

FFTW was the first widely available FFT library that supported arbitrary (even prime) sizes, arbitrary dimensions, strides, real/complex data, DCTs/DSTs, & parallelism ... **without** that, it would have just been a **hack for benchmark bragging**.