FFT:
The “Fastest Fourier Transform in the West”

Steven G. Johnson, MIT Applied Mathematics
Matteo Frigo, Cilk Arts Inc.
In the beginning (c. 1805):
Carl Friedrich Gauss

\[
\begin{align*}
y_j &= \sum_{k=0}^{n-1} c_k e^{i\frac{2\pi}{n} kj} \\
c_k &= \frac{1}{n} \sum_{k=0}^{n-1} y_j e^{-i\frac{2\pi}{n} kj}
\end{align*}
\]

generalizing work of Clairaut (1754) and Lagrange (1762)
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 quaternorum 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) = \text{DFT of real, odd-symmetry}

sample the spectrum at \( n \) points:

\( \ldots \text{double sampling until density (DFT) converges} \ldots \)
Gauss’ FFT in reverse:
Danielson and Lanczos (1942)


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

64-point DST in *only 140 minutes*!
re-inventing Gauss (for the last time)  

Cooley and Tukey (1965)  

[ *Math. Comp.* 19, 297–301 ]

1d DFT of size $n$:

$$n = pq$$

$$= \sim 2d \text{ DFT of size } p \times q$$

$$= \text{ Recursive DFTs of sizes } p \text{ and } q$$

$$O(n^2) \quad \rightarrow \quad O(n \log n)$$

$n=2048$, IBM 7094, 36-bit float: 1.2 seconds

($\sim 10^6$ speedup vs. Dan./Lanc.)
The “Cooley-Tukey” FFT Algorithm

1d DFT of size $n$: $n = pq$

= ~2d DFT of size $p \times q$

first DFT columns, size $q$ (non-contiguous)

finally, DFT columns, size $p$ (non-contiguous)
“Cooley-Tukey” FFT, in math

Recall the definition of discrete Fourier transform:

\[ y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk}, \quad \text{where } \omega_n = e^{-2\pi \sqrt{-1}/n}. \]

**Trick:** If \( n = pq \), write \( j = pj_1 + j_2 \) and \( k = k_1 + qk_2 \).

\[
y_{k_1+qk_2} = \sum_{j_2=0}^{p-1} \sum_{j_1=0}^{q-1} x_{pj_1+j_2} \omega_n^{pj_1k_1} \cdot \omega_n^{j_2k_1} \cdot \omega_n^{pqj_1k_2} \cdot \omega_n^{qj_2k_2}
\]

\[
= \sum_{j_2=0}^{p-1} \left[ \left( \sum_{j_1=0}^{q-1} x_{pj_1+j_2} \omega_n^{j_1k_1} \right) \omega_n^{j_2k_1} \right] \omega_n^{qj_2k_2}.
\]

size-\( p \) DFTs  size-\( q \) DFTs  twiddles
...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) [unless we give up exactness]

We’re left with the “constant” factor...
Choice of factorizations: the “radix”

1d DFT of size $n$:

- $n = pq$
- $\approx 2d$ DFT of size $p \times q$
- $p$ DFTs of size $q$,
  ... then $q$ DFTs of size $p$

Usually, either $p$ or $q$ is small = “radix” $r$

- e.g. $p=2$ is “radix-2 decimation in time”

Cooley & Tukey’s error:

thought radix 3 was optimal (closest to $e$)
— they forgot that size-$r$ DFTs can also use FFT
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)

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

A question with a more stable answer?
• **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/](http://www.fftw.org/)
FFTW performance
power-of-two sizes, double precision
unusual: non-power-of-two sizes receive as much optimization as powers of two

...because we let the code do the optimizing
FFTW performance

double precision, 2.8GHz Pentium IV: 2-way SIMD (SSE2)

powers of two

...because we let the code write itself

exploiting CPU-specific SIMD instructions (rewriting the code) is easy

non-powers-of-two
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. Determining the unit of composition is critical.
FFTW is easy to use

```c
{
    complex x[n];
    plan p;

    p = plan_dft_1d(n, x, x, FORWARD, MEASURE);
    ...
    execute(p); /* repeat as needed */
    ...
    destroy_plan(p);
}
```

Key fact: usually, many transforms of same size are required.
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. Determining the unit of composition is critical.
Why is FFTW slow?

1965 Cooley & Tukey, IBM 7094, 36-bit single precision:
size 2048 DFT in 1.2 seconds

2003 FFTW3+SIMD, 2GHz Pentium-IV 64-bit double precision:
size 2048 DFT in 50 microseconds (24,000x speedup)

(= 30% improvement per year)

(= doubles every ~30 months)

Moore’s prediction: 30 nanoseconds

FFT’s are hard: don’t get “peak” CPU speed especially for large n, unlike e.g. dense matrix multiply
Discontiguous Memory Access

1d DFT of size $n$:

$$n = pq$$

$= \sim 2d$ DFT of size $p \times q$

- first DFT columns, size $q$
  (non-contiguous)

- multiply by $n$ “twiddle factors”

- transpose

- finally, DFT columns, size $p$
  (non-contiguous)
Cooley-Tukey is Naturally Recursive

But traditional implementation is non-recursive, breadth-first traversal:

$\log_2 n$ passes over whole array
Traditional cache solution: **Blocking**

breadth-first, but with *blocks* of size = cache optimal choice: radix = cache size

radix >> 2

...requires program specialized for cache size

...multiple levels of cache = multilevel blocking
Recursive Divide & Conquer is Good
(deepth-first traversal)

Size 8 DFT

$p = 2$ (radix 2)

Size 4 DFT

Size 4 DFT

Size 2 DFT

Size 2 DFT

Size 2 DFT

Size 2 DFT

eventually small enough to fit in cache

…no matter what size the cache is
Cache Obliviousness

- A cache-oblivious algorithm does not know the cache size — for many algorithms [Frigo 1999], can be provably “big-O” optimal for any machine & for all levels of cache simultaneously

… but this ignores e.g. constant factors, associativity, …

**cache-obliviousness is a good beginning, but is not the end of optimization**

we’ll see: FFTW combines *both styles* (breadth- and depth-first) with self-optimization
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. Determining the unit of composition is critical.
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 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) feasible

(& negates recursion overhead)
The Codelet Generator
written in Objective Caml [Leroy, 1998], an ML dialect

Abstract FFT algorithm
Cooley-Tukey: $n=pq$,
Prime-Factor: $\gcd(p,q) = 1$,
Rader: $n$ prime, …

Simplifications
powerful enough
to e.g. derive real-input FFT
from complex FFT algorithm
and even find “new” algorithms

Symbolic graph (dag)

Optimal cache-oblivious scheduling
(cache .EQ. registers)

Optimized C code (or other language)
The Generator Finds Good/New FFTs

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\text{FFTW (adds+mults)}$</th>
<th>$\text{literature (adds+mults)}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\text{complex}$</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>$176 + 68 = 244$</td>
<td>$172 + 90 = 262$ [LCT93]</td>
</tr>
<tr>
<td></td>
<td>$156 + 56 = 212$</td>
<td>$188 + 40 = 228$ [SB96]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$162 + 50 = 212$ [BP85]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$162 + 36 = 198$ [BP85]</td>
</tr>
<tr>
<td>64</td>
<td>$912 + 248 = 1160$</td>
<td>$964 + 196 = 1160$ [Yavne68]</td>
</tr>
<tr>
<td></td>
<td>$\text{real}$</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>$64 + 25 = 89$</td>
<td>$67 + 25 = 92$ [HBJ84]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$67 + 17 = 84$ [SJHB87]</td>
</tr>
<tr>
<td>64</td>
<td>$394 + 124 = 518$</td>
<td>$420 + 98 = 518$ [SJHB87]</td>
</tr>
<tr>
<td></td>
<td>$\text{real symmetric (even)}$</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>$26 + 9 = 35$</td>
<td>$30 + 5 = 35$ [Duhamel86]</td>
</tr>
<tr>
<td>64</td>
<td>$172 + 67 = 239$</td>
<td>$190 + 49 = 239$ [Duhamel86]</td>
</tr>
</tbody>
</table>
Symbolic Algorithms are Easy

Cooley-Tukey in OCaml

DSP book:

\[ y_k = \sum_{j=0}^{n-1} x_j \omega_n^j = \sum_{j_2=0}^{p-1} \left[ \left( \sum_{j_1=0}^{q-1} x_{p j_1 + j_2} \omega_q^{j_1 k_1} \right) \omega_n^{j_2 k_1} \right] \omega_p^{j_2 k_2}, \]

where \( n = pq \) and \( k = k_1 + q k_2 \).

OCaml code:

```ocaml
let cooley_tukey n p q x =
  let inner j2 = fftgen q
    (fun j1 -> x (p * j1 + j2)) in
  let twiddle k1 j2 =
    (omega n (j2 * k1)) @* (inner j2 k1) in
  let outer k1 = fftgen p (twiddle k1) in
  (fun k -> outer (k mod q) (k / q))
```
Simple Simplifications

Well-known optimizations:

- Algebraic simplification, *e.g.* $a + 0 = a$
- Constant folding
- Common-subexpression elimination
Symbolic Pattern Matching in OCaml

The following actual code fragment is solely responsible for simplifying multiplications:

\[
stimesM = \text{function}
\quad | (\text{Uminus } a, b) \rightarrow stimesM (a, b) >>= \text{suminusM}
\quad | (a, \text{Uminus } b) \rightarrow stimesM (a, b) >>= \text{suminusM}
\quad | (\text{Num } a, \text{Num } b) \rightarrow \text{snumM (Number.mul } a \text{ } b)
\quad | (\text{Num } a, \text{Times } (\text{Num } b, c)) \rightarrow
\quad \quad \text{snumM (Number.mul } a \text{ } b) >>= \text{fun } x \rightarrow stimesM (x, c)
\quad | (\text{Num } a, b) \ \text{when } \text{Number.is_zero } a \rightarrow \text{snumM Number.zero}
\quad | (\text{Num } a, b) \ \text{when } \text{Number.is_one } a \rightarrow \text{makeNode } b
\quad | (\text{Num } a, b) \ \text{when } \text{Number.is_mone } a \rightarrow \text{suminusM } b
\quad | (a, b) \ \text{when } \text{is_known_constant } b \ \&\& \ \text{not } (\text{is_known_constant } a) \rightarrow
\quad \quad \text{stimesM } (b, a)
\quad | (a, b) \rightarrow \text{makeNode } (\text{Times } (a, b))
\]

(Common-subexpression elimination is implicit via “memoization” and monadic programming style.)
Simple Simplifications

Well-known optimizations:

Algebraic simplification, *e.g.* \( a + 0 = a \)

Constant folding

Common-subexpression elimination

FFT-specific optimizations:

Network transposition (transpose + simplify + transpose)

_________________ negative constants…
A Quiz: Is One Faster?

Both compute the same thing, and have the same number of arithmetic operations:

\[
\begin{align*}
  a &= 0.5 \times b; \\
  c &= 0.5 \times d; \\
  e &= 1.0 + a; \\
  f &= 1.0 - c;
\end{align*}
\]

\[
\begin{align*}
  a &= 0.5 \times b; \\
  c &= -0.5 \times d; \\
  e &= 1.0 + a; \\
  f &= 1.0 + c;
\end{align*}
\]

Faster because no separate load for \(-0.5\)

10–15% speedup
Non-obvious transformations require experimentation.
Quiz 2: Which is Faster?

accessing strided array inside codelet (amid dense numeric code), nonsequential

array[\text{stride} \times i]

array[\text{strides}[i]]

using precomputed stride array:

\text{strides}[i] = \text{stride} \times i

...namely, Intel Pentia:

integer multiplication conflicts with floating-point

up to \sim 10–20\% speedup

This is faster, of course!

Except on brain-dead architectures…

...even better to bloat:

pregenerate various constant strides
Machine-specific hacks are feasible if you just generate special code:

- **stride** precomputation
- **SIMD** instructions (SSE, Altivec, 3dNow!)
- **fused multiply-add** instructions…
The Generator Finds Good/New FFTs

<table>
<thead>
<tr>
<th>$n$</th>
<th><strong>FFTW (adds+mults)</strong></th>
<th><strong>literature (adds+mults)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>complex</strong></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>176 + 68 = 244</td>
<td>172 + 90 = 262</td>
</tr>
<tr>
<td></td>
<td>188 + 40 = 228</td>
<td>[SB96]</td>
</tr>
<tr>
<td>15</td>
<td>156 + 56 = 212</td>
<td>162 + 50 = 212</td>
</tr>
<tr>
<td></td>
<td>162 + 36 = 198</td>
<td>[BP85]</td>
</tr>
<tr>
<td>64</td>
<td>912 + 248 = 1160</td>
<td>964 + 196 = 1160</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Yavne68]</td>
</tr>
<tr>
<td></td>
<td><strong>real</strong></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>64 + 25 = 89</td>
<td>67 + 25 = 92</td>
</tr>
<tr>
<td></td>
<td>67 + 17 = 84</td>
<td>[SJHB87]</td>
</tr>
<tr>
<td>64</td>
<td>394 + 124 = 518</td>
<td>420 + 98 = 518</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[SJHB87]</td>
</tr>
<tr>
<td></td>
<td><strong>real symmetric (even)</strong></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>26 + 9 = 35</td>
<td>30 + 5 = 35</td>
</tr>
<tr>
<td></td>
<td>30 + 5 = 35</td>
<td>[Duhamel86]</td>
</tr>
<tr>
<td>64</td>
<td>172 + 67 = 239</td>
<td>190 + 49 = 239</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[Duhamel86]</td>
</tr>
</tbody>
</table>
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. Determining 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 \( n/r \)

---

✗ 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, \(v,n\)):
multi-dimensional “vector loops” \(v\) of multi-dimensional transforms \(n\)

{sets of (size, input/output strides)}
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SOLVE</strong></td>
<td>Directly solve a small DFT by a codelet</td>
</tr>
<tr>
<td><strong>CT-FACTOR</strong></td>
<td>Radix-(r) Cooley-Tukey step = (r) (loop) sub-problems of size (n/r) &amp; recombine with size-(r) twiddle codelet</td>
</tr>
<tr>
<td><strong>VECLOOP</strong></td>
<td>Perform one vector loop (can choose any loop, i.e. loop reordering)</td>
</tr>
<tr>
<td><strong>INDIRECT</strong></td>
<td>DFT = copy + in-place DFT (separates copy/reordering from DFT)</td>
</tr>
<tr>
<td><strong>TRANSPOSE</strong></td>
<td>solve in-place (m \times n) transpose</td>
</tr>
</tbody>
</table>
Many Resulting “Algorithms”

• **INDIRECT + TRANSPOSE** gives *in-place* DFTs,
  — bit-reversal = product of transpositions
  … no separate bit-reversal “pass”
  [ Johnson (unrelated) & Burrus (1984) ]

• **VECLOOP** can push topmost loop to “leaves”
  — “vector” FFT algorithm [ Swarztrauber (1987) ]

• **CT-FACTOR** *then* **VECLOOP**(s) gives “breadth-first” FFT,
  — erases iterative/recursive distinction
Many Resulting “Algorithms”

• INDIRECT + TRANSPOSE gives in-place DFTs,
  — bit-reversal = product of transpositions
  … no separate bit-reversal “pass”
  [ Johnson (unrelated) & Burrus (1984) ]

• VECLOOP can push topmost loop to “leaves”
  — “vector” FFT algorithm [ Swarztrauber (1987) ]

• CT-FACTOR then VECLOOP(s) gives “breadth-first” FFT,
  — erases iterative/recursive distinction
Depth- vs. Breadth- First
for size $n = 30 = 3 \times 5 \times 2$

A “depth-first” plan:

CT-FACTOR[3]
VECLOOP $\times 3$
CT-FACTOR[2]
SOLVE[2, 5]

A “breadth-first” plan:

CT-FACTOR[3]
CT-FACTOR[2]
VECLOOP $\times 3$
SOLVE[2, 5]

(Note: both are executed by explicit recursion.)
Many Resulting “Algorithms”

- **INDIRECT** + **TRANSPOSE** gives in-place DFTs,
  - bit-reversal = product of transpositions
  - … no separate bit-reversal “pass”
    [ Johnson (unrelated) & Burrus (1984) ]

- **VECLOOP** can push topmost loop to “leaves”
  - “vector” FFT algorithm [ Swarztrauber (1987) ]

- **CT-FACTOR then VECLOOP(s)** gives “breadth-first” FFT,
  - erases iterative/recursive distinction
In-place plan for size $2^{14} = 16384$

(2 GHz PowerPC G5, double precision)

Radix-32 DIT + Radix-32 DIF = 2 loops = transpose

… where leaf SOLVE ~ “radix” 32 x 1
Out-of-place plan for size $2^{19}=524288$
(2GHz Pentium IV, double precision)

CT-FACTOR[4] (buffered variant)
CT-FACTOR[32] (buffered variant)

VECLOOP (reorder) $\times 32$

CT-FACTOR[64]

INDIRECT

+ VECLOOP (reorder) $\times 64$

VECLOOP $\times 4$
COPY[64]

VECLOOP $\times 4$
SOLVE[64, 64]

~2000 lines hard-coded C!

Unpredictable: (automated) experimentation is the only solution.
Dynamic Programming

the assumption of “optimal substructure”

Try all applicable steps:

\[
\text{DFT}(16) = \text{fastest of:}\quad \begin{align*}
\text{CT-FACTOR}[2]: & \quad 2 \quad \text{DFT}(8) \\
\text{CT-FACTOR}[4]: & \quad 4 \quad \text{DFT}(4)
\end{align*}
\]

\[
\text{DFT}(8) = \text{fastest of:}\quad \begin{align*}
\text{CT-FACTOR}[2]: & \quad 2 \quad \text{DFT}(4) \\
\text{CT-FACTOR}[4]: & \quad 4 \quad \text{DFT}(2) \\
\text{SOLVE}[1,8]
\end{align*}
\]

If exactly the same problem appears twice,
assume that we can re-use the plan.
— i.e. ordering of plan speeds is assumed independent of context
Planner Unpredictability

double-precision, power-of-two sizes, 2GHz PowerPC G5

heuristic: pick plan with fewest adds + multiplies + loads/stores

Classic strategy: minimize op’s fails badly

another test:
Use plan from: another machine? e.g. Pentium-IV? ... lose 20–40%
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.