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

trigonometric interpolation:
\[ y_j = \sum_{k=0}^{n-1} c_k e^{\frac{i2\pi kj}{n}} \]

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

c_k = \frac{1}{n} \sum_{k=0}^{n-1} y_j e^{-\frac{i2\pi kj}{n}}

discrete Fourier transform (DFT): (before Fourier)
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

\[
\alpha + \alpha' \cos t + \alpha'' \cos 2t + \alpha''' \cos 3t + \text{ etc.}
\]
\[
+ \delta' \sin t + \delta'' \sin 2t + \delta''' \sin 3t + \text{ etc.}
\]

\textit{identicum} esse debere, unde coëfficientes \( \alpha, \alpha', \delta', \alpha'', \delta'' \) etc. innotescent. 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.

\textit{Kids: don’t try this at home!}
Gauss’ fast Fourier transform (FFT)

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

— 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) = 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

“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)

\[ N = N_1 N_2 \]

1d DFT of size \( N \):

\[
= \sim 2d \text{ DFT of size } N_1 \times N_2
\]

(\(+\) phase rotation by twiddle factors)

\[ = \text{Recursive DFTs of sizes } N_1 \text{ and } N_2 \]

\[ O(N^2) \quad \rightarrow \quad O(N \log N) \]

\[ n=2048, \text{ IBM 7094, 36-bit float: } 1.2 \text{ seconds} \]

(\(~10^6 \text{ speedup vs. Dan./Lanc.}~\))
The “Cooley-Tukey” FFT Algorithm

1d DFT of size $N$: $N = N_1 N_2$

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

input re-indexing $n = n_1 + N_1 n_2$

output re-indexing $k = N_2 k_1 + k_2$

$N_1$

$n_1 \rightarrow$

multiply by $n$

“twiddle factors”

$N_2$

$n_2 \downarrow$

$\rightarrow = \text{contiguous}$

first DFT columns, size $N_2$

(non-contiguous)

transpose

$N_1$

$k_1 \downarrow$

finally, DFT columns, size $N_1$

(non-contiguous)
“Cooley-Tukey” FFT, in math

Recall the definition of the DFT:

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

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

$$y_{N_2 k_1 + k_2} = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} \omega_N^{n_1 N_2 k_1} \omega_N^{n_1 k_2} \omega_N^{N_1 n_2 N_2 k_1} \omega_N^{N_1 n_2 k_2} x_{n_1 + N_1 n_2}$$

$$= \sum_{n_1=0}^{N_1-1} \omega_N^{n_1 k_1} \omega_N^{n_1 k_2} \left( \sum_{n_2=0}^{N_2-1} \omega_N^{n_2 k_2} x_{n_1 + N_1 n_2} \right)$$

size-$N_1$ DFTs \hspace{1cm} twiddles \hspace{1cm} 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 $n$)
Many other FFT algorithms

- **Prime-factor algorithm**: $N = N_1 N_2$ where $N_1$ and $N_2$ are coprime: 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)

[ unless we give up exactness ]

We’re left with the “constant” factor...
The Next 30 Years...

Assume “time”

\[
\text{# multiplications} = \text{# additions} (= \text{flops})
\]

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)
- RAM (gigabytes)
- L2 cache (megabytes)
- L1 cache (10s of kilobytes)
- registers (~100)

…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
FFT W:
The “Fastest Fourier Transform in the West”

Steven G. Johnson, MIT Applied Mathematics
Matteo Frigo, Oracle; formerly MIT LCS (CSAIL)
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 \(\sim\)always span the \(\sim\)fastest algorithm?
• **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

833 MHz Alpha EV6

2 GHz PowerPC G5

2 GHz AMD Opteron

500 MHz Ultrasparc IIe
FFT performance

non-power-of-two sizes, double precision

unusual: non-power-of-two sizes receive as much optimization as powers of two

2 GHz AMD Opteron

833 MHz Alpha EV6

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

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

powers of two

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

non-powers-of-two

...because we let the code write itself
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.
FFT W is easy to use

{ 
    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)

FFTs 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 $n$:

$$n = pq$$

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

```
\rightarrow = \text{ contiguous}
```

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

```
multiply by $n$ “twiddle factors”
```

```
\rightarrow = \text{ contiguous}
```

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

But traditional implementation is non-recursive, breadth-first traversal:
\[
\log_2 n \text{ passes over whole array}
\]
Traditional cache solution: **Blocking**

Size 8 DFT

\[ p = 2 \text{ (radix 2)} \]

- Size 4 DFT
  - Size 2 DFT
  - Size 2 DFT

- Size 4 DFT
  - Size 2 DFT
  - Size 2 DFT

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
(depth-first traversal)  [Singleton, 1967]

$p = 2$ (radix 2)

Eventually small enough to fit in cache
…no matter what size the cache is
• 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, …

Symbolic graph (dag)

Simplifications

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

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>FFTW (adds+mults)</th>
<th>literature (adds+mults)</th>
<th>complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>176 + 68 = 244</td>
<td>172 + 90 = 262</td>
<td>[LCT93]</td>
</tr>
<tr>
<td></td>
<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>
<td>[BP85]</td>
</tr>
<tr>
<td></td>
<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>
<td>[Yavne68]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>real</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>64</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>real symmetric (even)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
<tr>
<td>64</td>
</tr>
</tbody>
</table>
Symbolic Algorithms are Easy

**Cooley-Tukey in OCaml**

**DSP book:**

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk} = \sum_{j_2=0}^{p-1} \left[ \left( \sum_{j_1=0}^{q-1} x_{pj_1+j_2} \omega_q^{jk_1} \right) \omega_n^{jk_1} \right] \omega_p^{j_2k_2},$$

where $n = pq$ and $k = k_1 + qk_2$.

**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:

```ocaml
stimesM = function
  | (Uminus a, b) -> stimesM (a, b) >>= suminusM
  | (a, Uminus b) -> stimesM (a, b) >>= suminusM
  | (Num a, Num b) -> snumM (Number.mul a b)
  | (Num a, Times (Num b, c)) ->
    snumM (Number.mul a b) >>= fun x -> stimesM (x, c)
  | (Num a, b) when Number.is_zero a -> snumM Number.zero
  | (Num a, b) when Number.is_one a -> makeNode b
  | (Num a, b) when Number.is_mone a -> suminusM b
  | (a, b) when is_known_constant b && not (is_known_constant a) ->
    stimesM (b, a)
  | (a, b) -> makeNode (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:

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

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[stride * i]
```

This is faster, of course!

Except on brain-dead architectures…

```
array[strides[i]]
```

using precomputed stride array:

```
strides[i] = stride * i
```

…namely, Intel Pentia:

integer multiplication conflicts with floating-point

up to $\sim 10\text{--}20\%$ speedup

(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>FFTW (adds+mults)</th>
<th>literature (adds+mults)</th>
<th>complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>176 + 68 = 244</td>
<td>172 + 90 = 262</td>
<td>[LCT93]</td>
</tr>
<tr>
<td></td>
<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>
<td>[BP85]</td>
</tr>
<tr>
<td></td>
<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>
<td>[Yavne68]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>real</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>64 + 25 = 89</td>
<td>67 + 25 = 92</td>
<td>[HBJ84]</td>
</tr>
<tr>
<td></td>
<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>
<td>[SJHB87]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>real symmetric (even)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>26 + 9 = 35</td>
<td>30 + 5 = 35</td>
<td>[Duhamel86]</td>
</tr>
<tr>
<td>64</td>
<td>172 + 67 = 239</td>
<td>190 + 49 = 239</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)\}
Some Composable Steps (out of ~16)

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

**CT-FACTOR** \[ r \]  —  Radix-\( r \) Cooley-Tukey step =
\[
  r \text{ (loop) sub-problems of size } n/r
\]
(& recombine with size-\( r \) twiddle codelet)

**VECLOOP**  —  Perform one vector loop
(can choose any loop, i.e. loop reordering)

**INDIRECT**  —  DFT = copy + in-place DFT
(separates copy/reordering from DFT)

**TRANSPOSE**  —  solve in-place \( m \times n \) transpose
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 x3
- CT-FACTOR[2]
- SOLVE[2, 5]

A “breadth-first” plan:
- CT-FACTOR[3]
- CT-FACTOR[2]
- VECLOOP x3
- 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 $\sim$ “radix” 32 x 1
Out-of-place plan for size $2^{19} = 524288$
(2GHz Pentium IV, double precision)

\[
\begin{align*}
\text{CT-FACTOR}[4] & \quad \text{(buffered variant)} \\
\text{CT-FACTOR}[32] & \quad \text{(buffered variant)} \\
\text{VECLOOP (reorder)} \times 32 & \\
\text{INDIRECT} & \\
+ & \\
\text{VECLOOP (reorder)} & \\
( + \ldots ) & \\
= & \\
\text{huge improvements} & \\
\text{for large 1d sizes} & \\
\end{align*}
\]

\[
\begin{align*}
\text{INDIRECT} & \\
\text{CT-FACTOR}[64] & \\
\text{INDIRECT} & \\
\text{VECLOOP (reorder)} \times 64 & \\
\text{VECLOOP x} 4 & \\
\text{COPY}[64] & \\
\text{VECLOOP x} 4 & \\
\text{SOLVE}[64, 64] & \\
\sim 2000 \text{ lines hard-coded C!} &
\end{align*}
\]

Unpredictable: (automated) experimentation is the only solution.
Dynamic Programming
the assumption of “optimal substructure”

Try all applicable steps:

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

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

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

Classical strategy: minimize op’s
fails badly

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

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

FFT W 3
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.
For information about citing these materials or our Terms of Use, visit: https://ocw.mit.edu/terms.