# The Fastest Fourier Transform in the West

Matteo Frigo

*Supercomputing Technologies Group*
*MIT Laboratory for Computer Science*

Steven G. Johnson

*Condensed Matter Theory*
*MIT Physics Department*

# FFTW is available now

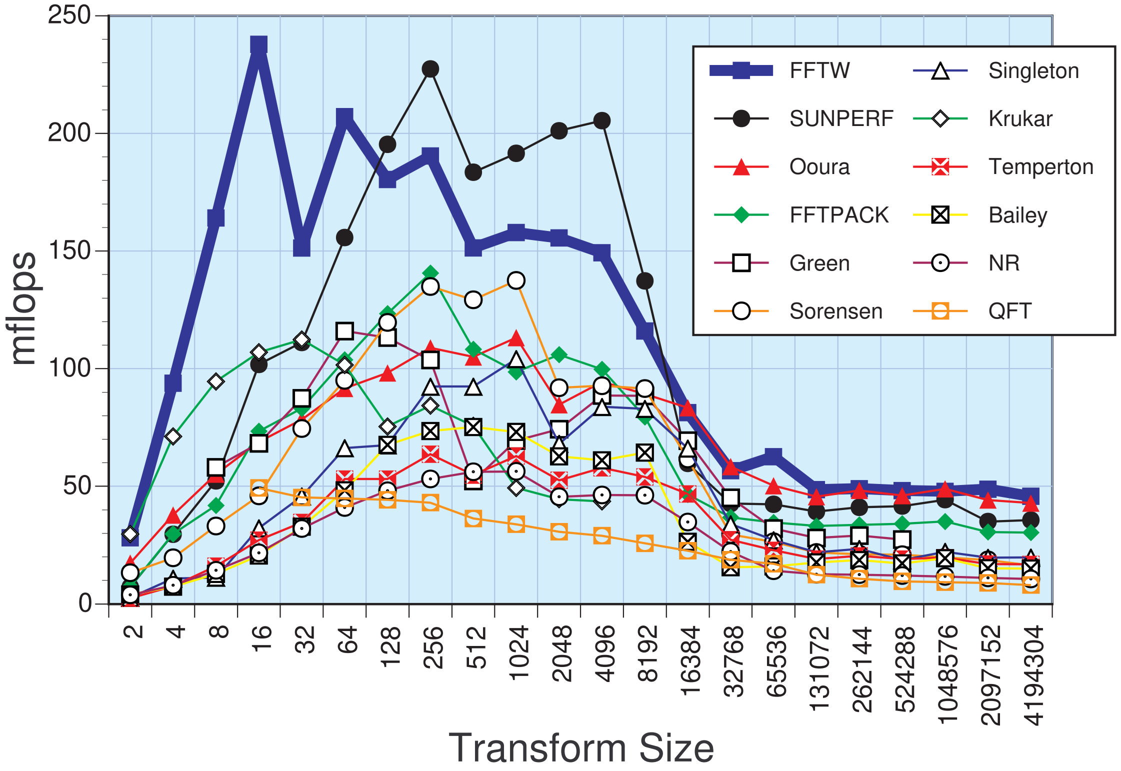
- FFTW is a package that computes one- and multidimensional complex discrete Fourier transforms (DFT) of arbitrary size.
- FFTW achieves superior performance by means of an *adaptive* software architecture, which achieves *speed*, *portability*, and *elegance* at the same time.
- Parallel versions are available for Cilk, Posix threads and MPI.
- Code, documentation and benchmark results at

  `http://theory.lcs.mit.edu/~fftw`

- FFTW will be part of the forthcoming *SPECmbc* multimedia benchmark (end of 1998).

# FFTW's performance

FFTW is faster than all other portable FFT codes, and it is comparable with machine-specific libraries provided by vendors.




Comparison of FFTW versus other codes on a 167-MHz Sun UltraSPARC I.

# Why FFTW is fast

The transform is computed by an *executor*, composed of highly optimized, composable blocks of C code called *codelets*.

- At runtime, a *planner* finds an efficient way to compose codelets: it *measures* the speeds of different plans and chooses the best using a dynamic programming algorithm.
- The executor interprets the plan with negligible overhead.
- Codelets are generated automatically and are fast. A high-level *codelet generator* allowed the authors to experiment quickly with many code sequences and choose the fastest.

*FFTW leads to a new model of high-performance programs in which performance-critical code is machine-generated.*

# FFTW is easy to use

FFTW's internal complexity is not visible to the user.

```
fftw_plan plan;
int n = 1024;
COMPLEX A[n], B[n];

/* plan the computation */
plan = fftw_create_plan(n);

/* execute the plan */
fftw(plan, A);

/* the plan can be reused */
fftw(plan, B);
```

The user only needs to call the planner once, and the executor as many times as needed. The plan depends only on the input size, and can be used to computed many transforms.

# Outline

1. Algorithms for the Fourier transform.
2. The executor. (With a digression about divide and conquer.)
3. The planner.
4. The codelet generator.
5. Conclusions.

# References

[0] Gauss, 1805. Predates Fourier himself (1807).

[1] I. J. Good, *The Interaction Algorithm and Practical Fourier Analysis*, J. Royal Stat. Soc., B-20, 361-372, 1958.

[2] J. W. Cooley and J. W. Tukey, *An Algorithm for the Machine Calculation of Complex Fourier Series*, Math. Computat., vol. 19, pp. 297-301, 1965.

[3] C. M. Rader, *Discrete Fourier Transform When the Number of Data Samples Is Prime*, Proc. IEEE, 56, 1107-1108, June 1968.

[4] S. Winograd, *On Computing the Discrete Fourier Transform*, Mathematics of Computation, vol. 32, pp. 175-199, January 1978.

[5] H. V. Sorensen, C. S. Burrus, and M. T. Heideman, *Fast Fourier Transform Database*. Boston: PWS Publishing, 1995. Contains more than 3400 references.

# Cooley-Tukey FFT algorithms

Recall the definition of discrete Fourier transform:

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

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

$$\begin{aligned} y_{k_1+qk_2} &= \sum_{j_2=0}^{p-1} \sum_{j_1=0}^{q-1} x_{pj_1+j_2} \omega_n^{pj_1k_1} \cdot \omega_n^{j_2k_1} \cdot \omega_n^{pqj_1k_2} \cdot \omega_n^{qj_2k_2} \\ &= \sum_{j_2=0}^{p-1} \left[ \left( \sum_{j_1=0}^{q-1} x_{pj_1+j_2} \omega_q^{j_1k_1} \right) \omega_n^{j_2k_1} \right] \omega_p^{j_2k_2}. \end{aligned}$$

$O(n \log n)$ *recursive divide-and-conquer algorithms*.

The *twiddle factors* are annoying.

# The executor

# The executor

The executor implements the recursive divide and conquer Cooley-Tukey FFT algorithm.

- The executor is composed of many optimized code sequences called *codelets*.
- Codelets come in two flavors. *Nontwiddle codelets* compute transforms of small sizes. *Twiddle codelets* combine small transforms to compute bigger transforms.
- The executor invokes the codelets as dictated by the plan.
- Recursive divide-and-conquer maximizes *memory locality*. The executor does not need to know the size of the cache.

# What a plan looks like

```
FFT(128) =
    DIVIDE-AND-CONQUER(128, 4)
    DIVIDE-AND-CONQUER(32, 8)
    SOLVE(4)
```

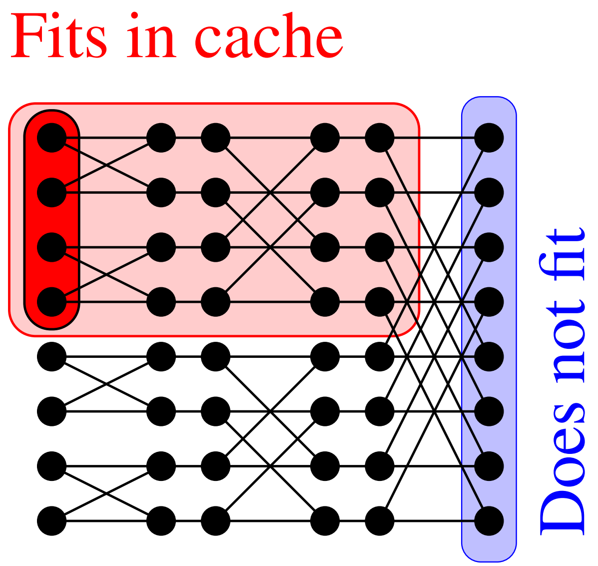
The plan is a sequence of instructions, either

**SOLVE**$(n)$ : compute the FFT of size $n$ using a nontwiddle codelet.

**DIVIDE-AND-CONQUER**$(n, p)$ : solve $p$ problems of size $n/p$ using the rest of the plan. Then combine the results using a twiddle codelet.

# Executor is designed for locality

- Recursive divide-and-conquer exploits all levels of the memory hierarchy.
- Divide-and-conquer is faster than loops!

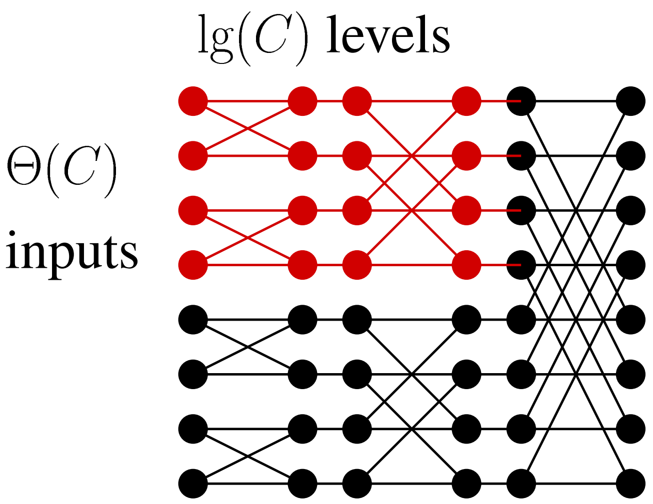


# Digression about divide-and-conquer

# Out-of-cache FFTs

Suppose we want to compute the FFT of $n$ points, and a computer has an ideal cache of size $C$, where $C < n$.

**Lower bound.** *(Hong-Kung, 1981.)* Performing the $n$-point FFT with a cache of size $O(C)$ requires at least $\Omega(n \lg n / \lg C)$ cache misses.




**Upper bound I.** Partition the butterfly into blocks of size $\Theta(C) \times \Theta(\lg C)$. Each block can be executed with $O(C)$ cache misses. The total number of misses is

$$O\left(C \cdot \frac{n \lg n}{C \lg C}\right) = O\left(\frac{n \lg n}{\lg C}\right) .$$

This algorithm requires knowledge of $C$, however.

# Cache-oblivious FFT algorithm

**Upper bound II.** Execute the Cooley-Tukey algorithm with $p = q = \sqrt{n}$, i.e.:

1. Perform $\sqrt{n}$ FFTs of size $\sqrt{n}$, recursively.
2. Multiply $O(n)$ elements by the twiddle factors.
3. Perform $\sqrt{n}$ FFTs of size $\sqrt{n}$, recursively.

The number of cache misses satisfies the recurrence

$$M(n) = \begin{cases} O(n) & \text{if } n \le C \ ; \\ 2\sqrt{n}M(\sqrt{n}\,) + O(n) & \text{if } n > C \ , \end{cases}$$

which has the optimal solution

$$M(n) \le O(n \lg n / \lg C) \ .$$

# D&C thinking

- D&C leads to optimal cache-oblivious algorithms for matrix multiplication, LU decomposition, etc.
- A cache-oblivious algorithm does not know $C$. Consequently, it is automatically optimal for all levels of the memory hierarchy.
- If you are accustomed to FORTRAN loops, *think different*.

---

In practice, the cache-oblivious FFT algorithm does not appear to perform better than fixed-radix algorithms w.r.t. the L1 cache. However, the cache-oblivious algorithm is faster w.r.t. registers.

# The planner

# The planner

The planner finds a fast way to compute the DFT of a given size $n$.

- The planner produces many plans, *measures* their speed and picks the best.
- The planner uses *dynamic programming*.
- The planner can produce a “reasonable” plan quickly if desired.
- The planner collects information about the machine, which can be stored on disk and reused at a later time.

# How the planner works

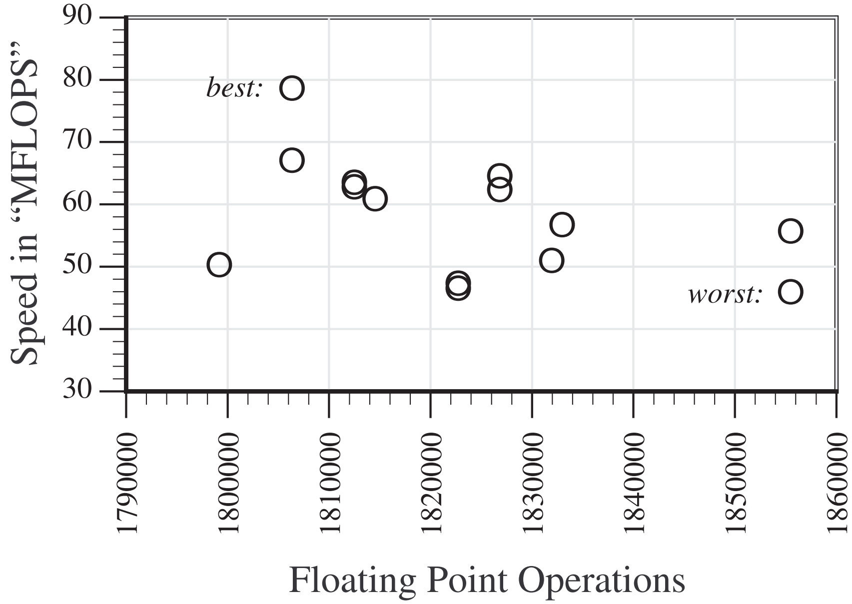
- Try all ways to solve a problem of size $n$ or to factor $n$ using the available codelets.

$$\texttt{FFT}(16) = \begin{array}{c}\text{fastest}\\ \text{of}\end{array} \begin{cases} \texttt{SOLVE}(16) \\ \texttt{DIVIDE-AND-CONQUER}(16,2) + 2 \cdot \texttt{FFT}(8) \\ \texttt{DIVIDE-AND-CONQUER}(16,4) + 4 \cdot \texttt{FFT}(4) \\ \texttt{DIVIDE-AND-CONQUER}(16,8) + 8 \cdot \texttt{FFT}(2) \end{cases}$$

$$\texttt{FFT}(8) = \begin{array}{c}\text{fastest}\\ \text{of}\end{array} \begin{cases} \texttt{SOLVE}(8) \\ \texttt{DIVIDE-AND-CONQUER}(8,2) + 2 \cdot \texttt{FFT}(4) \\ \texttt{DIVIDE-AND-CONQUER}(8,4) + 4 \cdot \texttt{FFT}(2) \\ \texttt{DIVIDE-AND-CONQUER}(8,8) + 8 \cdot \texttt{FFT}(1) \end{cases}$$

*Can store solutions of the subproblems in a table and reuse them.*

# Why adaptive programs?




Performance of different plans can vary by a factor of $2$, and the fastest plan is not the one with the fewest operations. (Transform of size $n = 32768$ on a 167-MHz Ultrasparc I.)

Present-day machines lack predictability, and experimentation appears to be the only way to achieve high performance. FFTW's planner automates this experimentation.

# Does the planner need to be so complicated?

# Is the planner necessary?

**Q:** Can't you simply count the number of arithmetic operations and pick the cheapest codelet sequence?

**A:** Forget it. These days, performance has little to do with the number of operations.

*For example, fast code must exploit the instruction-level parallelism of the processor. It is important to write code in such a way that the C compiler can schedule it efficiently.*

# Can the planner be simpler?

**Q:** Can't you just measure the execution time of the individual codelets and sum them?

**A:** No way. The performance of codelets is not additive. It is not even monotonic.

*The performance of a codelet depends on how the previous codelet stored data into memory: for example, if the data are in cache the second codelet runs faster. Therefore, codelets are not independent of each other.*

# Are plans optimal?

**Q:** Does your planner produce optimal plans?

**A:** No. However, my plans are good enough.

*The best plan for, say,* `FFT`*(16) depends on whether the plan is being used alone or as a subplan for a bigger problem, but my planner ignores this fact.*

# Can you plan offline?

**Q:** Can you just precompute all the plans you care about for a given operating environment and store them in a file?

**A:** This works (almost).

*Performance can also depend on the set of physical pages my program obtains from the OS at load time.*

# Summary of the planner

- Experimentation appears to be the only way to find a good plan.
- The planner does all the experiments automatically.

# The codelet generator

# The codelet generator

The codelet generator is a domain-specific FFT compiler.

- Generates C code for transforms of small sizes, with and without twiddle factors.
- Written in Objective Caml, a dialect of ML. Early versions were written in Caml-Light and Scheme.
- The generator produces a directed acyclic graph (dag) of the codelet, then applies *algebraic simplifications*, *constant folding* and *code scheduling*.
- Knowledge of the application allows us to apply more optimizations than can be performed safely by an ordinary compiler.

*A domain-specific compiler is the key for quick experimentation of new ideas. For example, we could generate codelets for real to complex transforms changing only one line of the generator.*

# Domain-specific compilers

The codelet generator contains several FFT algorithms expressed at a very high level.

---

The function `fftgen` generates a symbolic expression that computes the DFT. The function call `(input` $j$`)` returns an expression containing the $j$-th input.

```
let rec fftgen_prime n input sign =
  let sum i =
    sigma 0 n (fun j ->
      let coeff = exp n (sign * i * j)
      in [times coeff (input j)])
  in
    sum
```

Later stages of the generator unroll and simplify the high-level algorithm, and produce optimized C code.

# The expression simplifier

*The simplifier is rule-based.*

```
let 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)))
   | a (Uminus 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)
   | a b -> Times (a, b)
```

(This code simplifies trees. Using *memoization* and *monadic programming*, almost the same code also works on dags.)

# The scheduler

The generator first produces a computation graph of the FFT, and then simplifies it. After these steps, the generator produces a butterfly.

*How do we generate code from the butterfly?*

We use the cache-oblivious divide-and-conquer algorithm. The generated code runs well on machines *with any number of registers*.

# Techniques used by the CG

- Producing good C code is tricky.
- I shall now test your knowledge of C with two quizzes. I shall present you pairs of equivalent C fragments. Your goal is to guess which fragment is faster.

# Quiz 1

```
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 number of floating-point constants should be minimized.

*The codelet generator automatically makes all constants positive and propagates signs throughout the rest of the program.*

# Quiz 2

```
{
   double tmp;
   tmp = 2.0 * b;
   x += tmp;
   tmp = 3.0 * c;
   x += tmp;
}
```

```
{
   double tmp;
   tmp = 2.0 * b;
   x += tmp;
}
{
   double tmp;
   tmp = 3.0 * c;
   x += tmp;
}
```

Answer: reusing temporary variables introduces unnecessary dependencies.

*The codelet generator produces nested blocks of code and writes to variables only once.*

# Summary of the codelet generator

- Can express algorithms in an abstract way. Ugly details are hidden. *Invent the best language to express your problem.*
- Easy to experiment with new ideas and make changes.
- Can generate nonobvious and tricky code.
- You have to get it right only once.
- Generator can be reused for other problems.
- ML is `fun`.

# Testimonials

Thousands of people downloaded FFTW so far.

> *"It's the best FFT package I have ever seen."*

> *"It performs my standard 256 iterations of 1024pt complex fft about 20 times faster than the previous one I used."*

> *"FFTW is the best thing since microwave popcorn."*

> *"[FFTW] is written in C, very easy to use and extremely fast. It involves some very clever programming, which is, however, transparent to the user."*

FFTW is used in the *Virtual Radio* [Guttag et al., 1998].

# Future work

- FFTW will be part of the forthcoming *SPECmbc* multimedia benchmark (end of 1998).
- We are implementing *real to complex transforms*.
- I plan to develop of the idea of a general-purpose *metacompiler* for high-performance computing. The programmer should be able to specify both a program to be compiled, and the transformations that the metacompiler applies to the program. The FFTW codelet generator is already a powerful metacompiler, but it is not yet general enough.