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).

  Let $X = (x_0, x_1, \ldots, x_{n-1})$ be a vector of complex numbers. The (one-dimensional) *discrete Fourier transform* of X is the complex vector Y given by

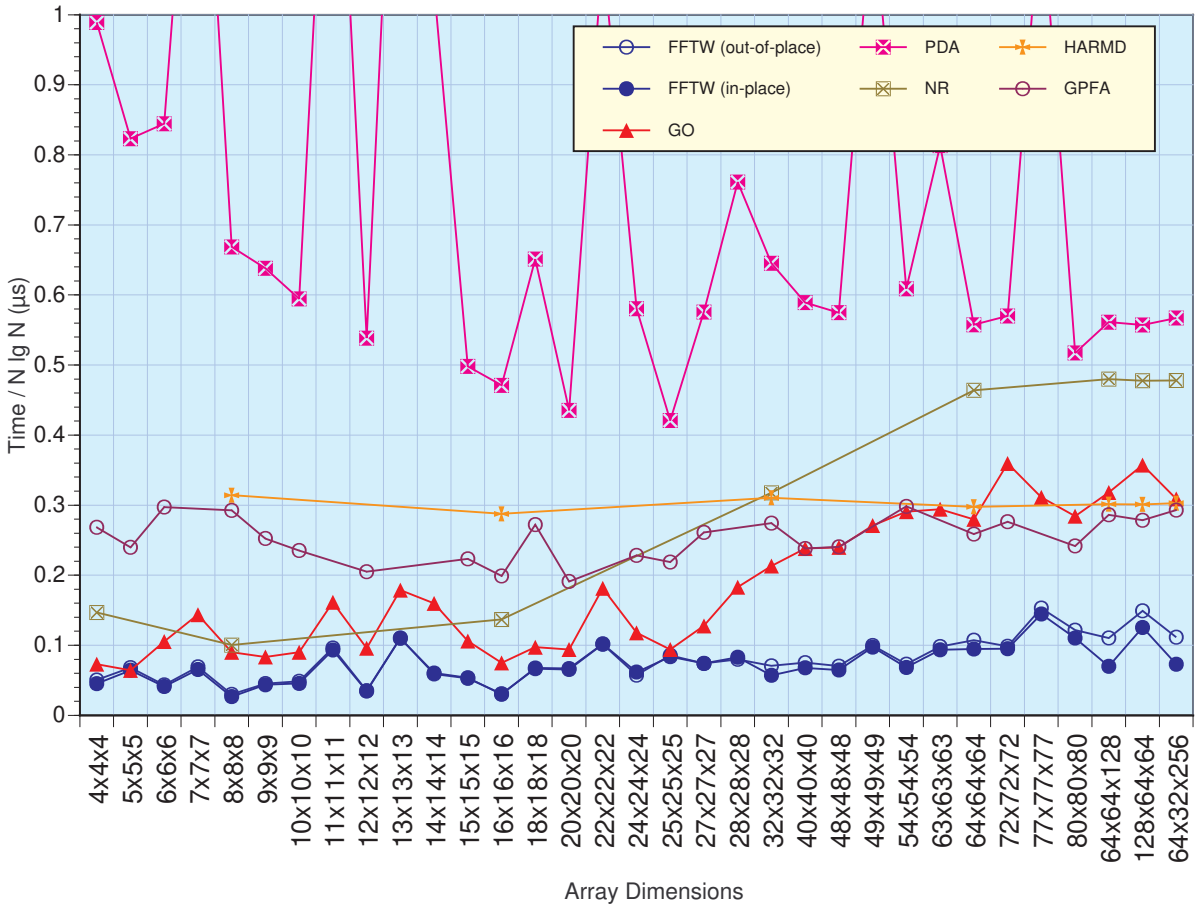
$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk} ,$$

  where $k \in \{0, 1, \ldots, n-1\}$, and $\omega_n = e^{-2\pi\sqrt{-1}/n}$ is a *primitive root of unity*.

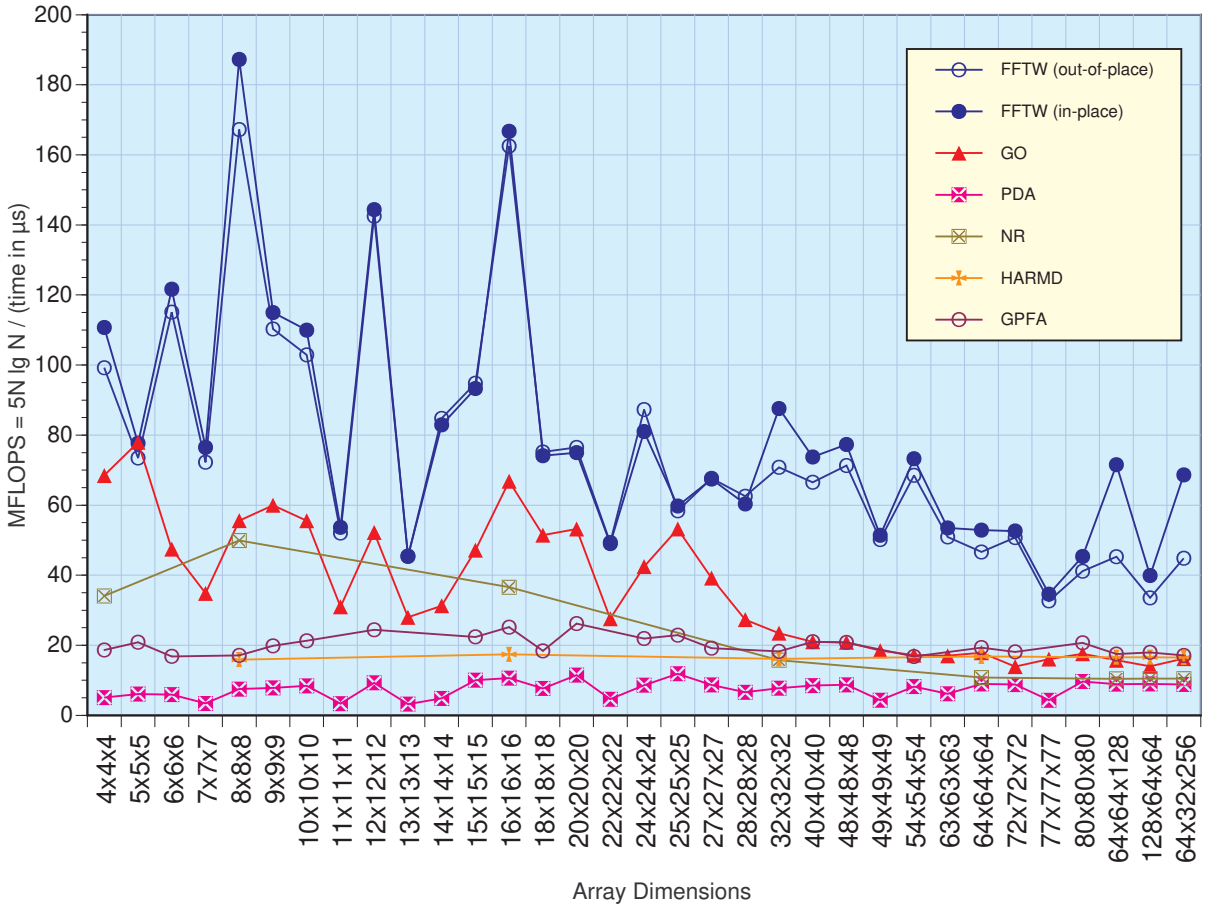
- Code and benchmark results at

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

FFTW's performance



FFTW's performance (MFLOPS)



FFTW is easy to use

```
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 for other
   inputs of size n */
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.

Why FFTW is fast

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

- The 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 fast. A high-level *codelet generator* allowed the authors to experiment quickly with many code sequences and choose the fastest.

Code is fast, and yet elegant, modular, and readable.

Outline

1. Algorithms for the Fourier transform.
2. The executor.
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:

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk} \ .$$

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 Cooley-Tukey algorithm.

- The executor is composed of many codelets that compute transforms of small sizes.
- Codelets come in two flavors: with and without twiddle factors.
- The executor invokes the codelets as dictated by the plan.

What does a plan look 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 codelet.

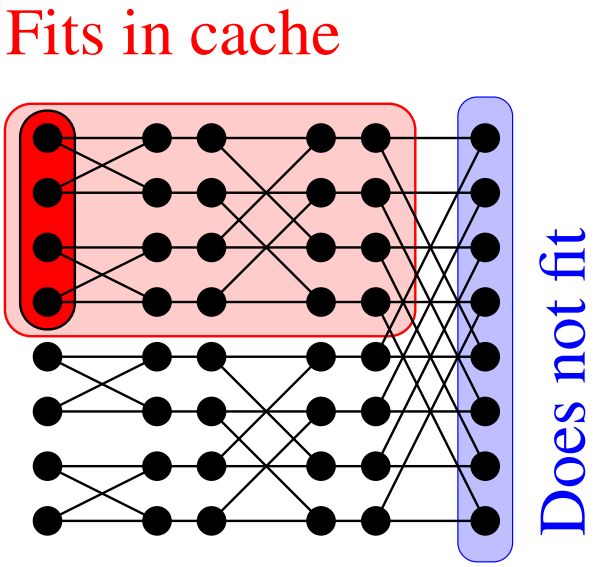
DIVIDE-AND-CONQUER(n, p) : solve p problems of size n/p using the rest of the plan. Then combine the results using a codelet with twiddle factors.

Executor is designed for locality

- Recursive divide-and-conquer exploits all levels of the memory hierarchy.

  Divide-and-conquer is faster than loops.

- Novel storage scheme for the twiddle factors: store them in the order they are used.



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.

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.

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

- Generates C code for transforms of small sizes, with and without twiddle factors.
- Written in the Caml-Light dialect of ML (early prototype written in Scheme).
- Works on a subset of C's abstract syntax tree (AST).
- Knows about complex numbers and complex arithmetic.
- Unparses the AST to C.

A fragment of the CG

(input j**)** : returns an expression containing the j-th input.

(output k x**)** : returns an piece of code that stores the expression x into the k-th output variable.

(simplify x**)** : performs constant folding and other simplifications on an expression.

```
let fftgen_prime n input output =
    let expr k = simplify
                   (Sigma 0 n (fun j ->
                       let coeff = CExp(k * j / n)
                       in CTimes coeff (input j)))
    and FFT =
        forall 0 n (fun k ->
            (output k (expr k)))
    in
       Block ([], FFT)
;;
```

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)
```

Techniques used by the CG

- Producing good C code is tricky.
- I shall now test your knowledge of C with three 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.

Quiz 3

```
a = b + (c + d);
e = f + (c + d);
```

```
{
  double tmp = c + d;
  a = b + tmp;
  e = f + tmp;
}
```

Answer: C compilers eliminate common subexpressions effectively, unless a memory operation is involved.

The codelet generator does not bother to eliminate common subexpressions, unless they involve a memory operation.

Summary of the codelet generator

- Can express algorithms in an abstract way. Ugly details are hidden. *Invent the best language to express your problem (6.001)*.
- 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`.

Summary

FFTW is a *self-optimizing* program.

- FFTW is composed of fast machine-generated codelets.
- FFTW generates and exploits locality at all levels.
- The codelet generator contains knowledge about C compilers.
- Runtime interpretation can be fast.
- The abstraction power of high-level languages can be used without runtime costs.

Conclusions

- More than 600 jubilant users in the first month after the release:

  "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 compiles perfectly and runs extremely fast in this environment [Windows '95]."

- Parallel version using Cilk is coming soon.

- Can we optimize the plan *incrementally*?

You can find FFTW at

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

Your speaker is not jubilant

- What? I need *all this* just to compute an FFT?
- Modern computer architectures and compilers are invalidating traditional wisdom about what is fast.
- No new wisdom is emerging.
- In the name of performance, performance is lost (as well as predictability, repeatability, and composability).
- The situation is getting worse.