A Fast Fourier Transform Compiler

Matteo Frigo

Supercomputing Technologies Group
MIT Laboratory for Computer Science

`genfft`

`genfft` is a domain-specific FFT compiler.

- Generates fast C code for Fourier transforms of arbitrary size.
- Written in Objective Caml 2.0 [Leroy 1998], a dialect of ML.
- “Discovered” previously unknown FFT algorithms.
- From a complex-number FFT algorithm it derives a real-number algorithm [Sorensen et al., 1987] automatically.

`genfft` used in FFTW

`genfft` produced the "inner loop" of FFTW (95% of the code).

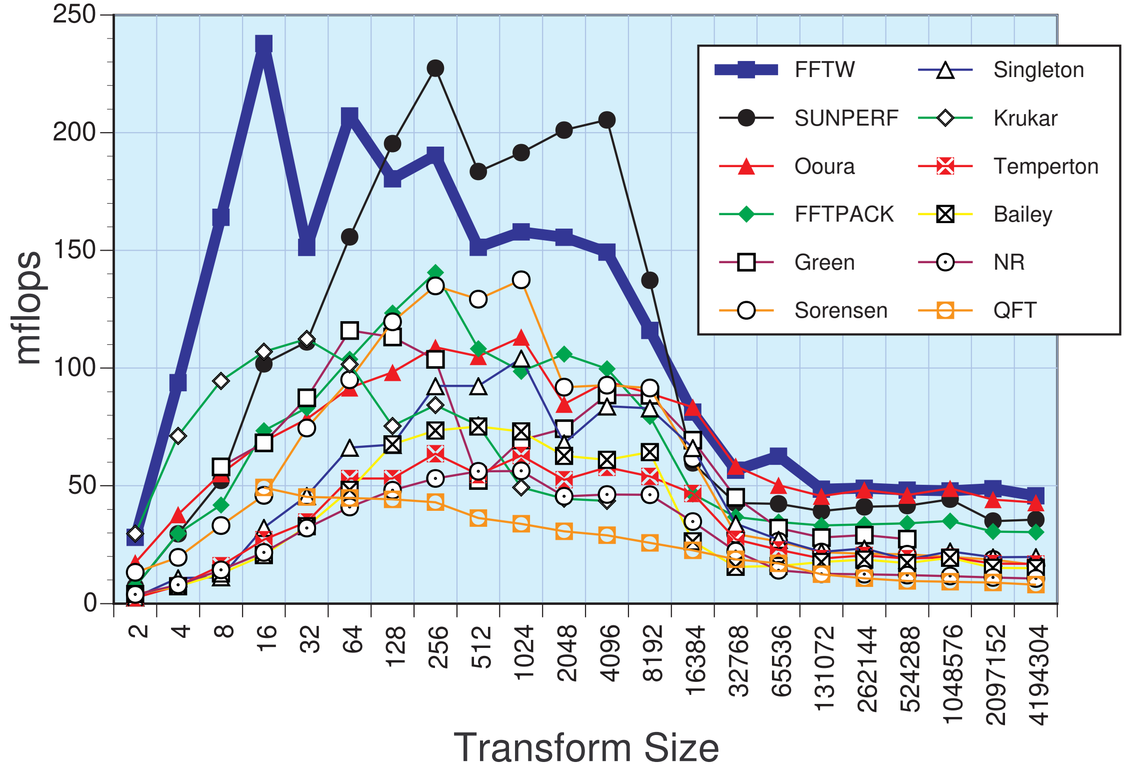
- FFTW [Frigo and Johnson, 1997–1999] is a library for computing one- and multidimensional real and complex discrete Fourier transforms of arbitrary size.
- FFTW ***adapts*** itself to the hardware automatically, giving ***portability*** and ***speed*** at the same time.
- Parallel versions are available for Cilk, Posix threads and MPI.
- Code, documentation and benchmark results at

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

`genfft` makes FFTW possible.

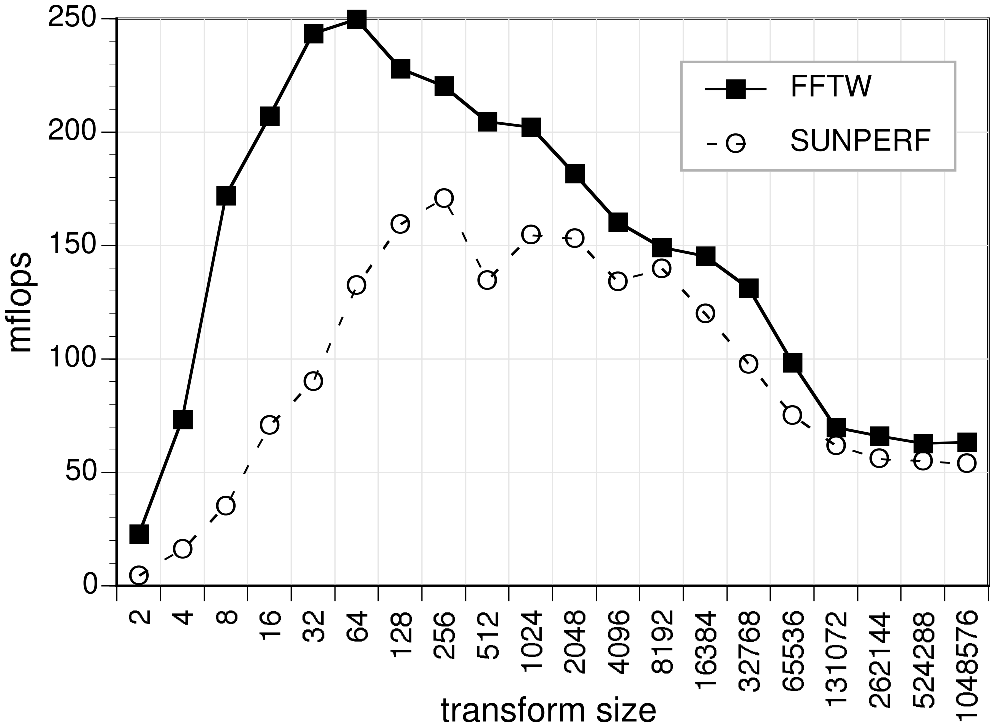
FFTW's performance

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



Benchmark of FFT codes on a 167-MHz Sun UltraSPARC I, double precision.

FFTW's performance



FFTW versus Sun's Performance library on a 167-MHz Sun Ultra-SPARC I, single precision.

Why FFTW is fast

FFTW does not implement a single fixed algorithm. Instead, the transform is computed by an ***executor***, composed of highly optimized, composable blocks of C code called ***codelets***. Roughly speaking, a codelet computes a Fourier transform of small size.

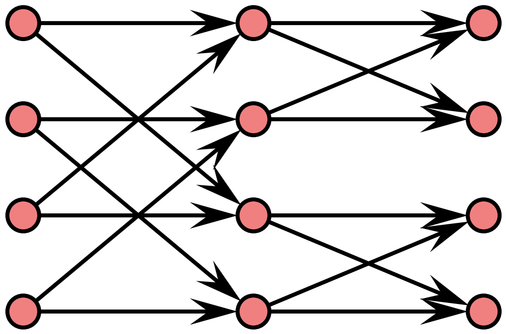
At runtime, a ***planner*** finds a fast composition of codelets. The planner *measures* the speed of different plans and chooses the fastest.

FFTW contains 120 codelets for a total of 56215 lines of *optimized* code.

Real FFTs are complex

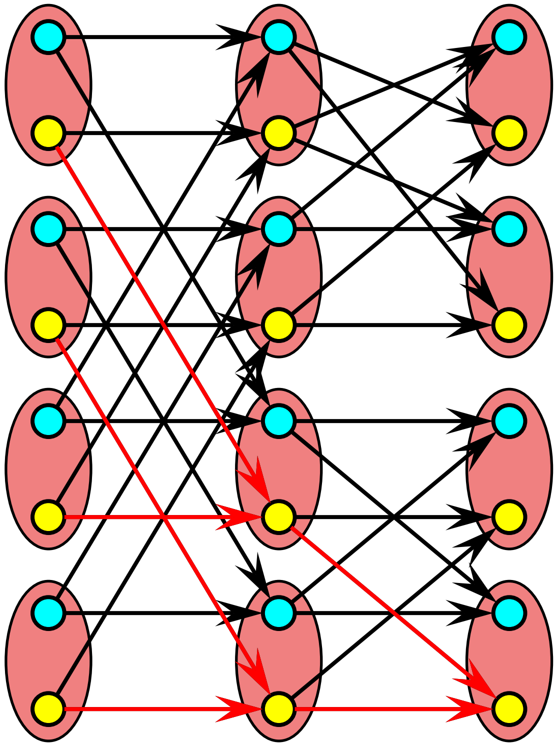
The data-flow graph of the Fourier transform is not a butterfly, it is a mess!

Complex butterfly



The FFT is even messier when the size is not a power of 2, or when the input consists of real numbers.

Real and imaginary parts



`genfft`'s compilation strategy

1. ***Create a dag*** for an FFT algorithm in the simplest possible way. Do not bother optimizing.

2. ***Simplify*** the dag.

3. ***Schedule*** the dag, using a register-oblivious schedule.

4. ***Unparse*** to C. (Could also unparse to FORTRAN or other languages.)

The case for `genfft`

Why not rely on the C compiler to optimize codelets?

- FFT algorithms are best expressed recursively, but C compilers are not good at unrolling recursion.
- The FFT needs *very* long code sequences to use all registers effectively. (FFTW on Alpha EV56 computes FFT(64) with 2400 lines of straight C code.)
- Register allocators choke if you feed them with long code sequences. In the case of the FFT, however, we know the (asymptotically) optimal register allocation.
- Certain optimizations only make sense for FFTs.

Since C compilers are good at instruction selection and instruction scheduling, I did not need to implement these steps in `genfft`.

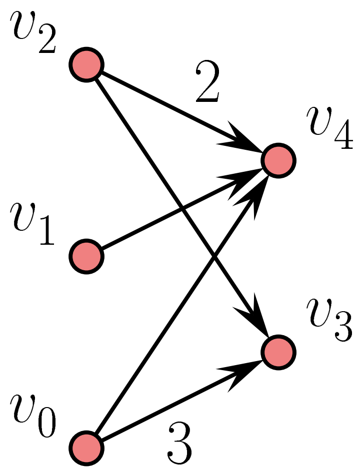
Outline

- Dag creation.
- The simplifier.
- Register-oblivious scheduling.
- Related work.
- Conclusion.

Dag representation

A dag is represented as a data type `node`. (Think of a syntax tree or a symbolic algebra system.)

```
type node =
  Num of Number.number  | Load of Variable.variable
| Plus of node list | Times of node * node | ...
```



$v_3 = \texttt{Plus [}v_2\texttt{; Times (Num 3, }v_0\texttt{)]}$
$v_4 = \texttt{Plus [Times (Num 2, }v_2\texttt{); }v_1\texttt{; }v_0\texttt{]}$

(All numbers are real.)

An abstraction layer implements complex arithmetic on pairs of dag nodes.

Dag creation

The function `fftgen` performs a complete symbolic evaluation of an FFT algorithm, yielding a dag.

No single FFT algorithm is good for all input sizes n. `genfft` contains many algorithms, and `fftgen` dispatches the most appropriate.

- Cooley-Tukey [1965] (works for $n = pq$).
- Prime factor [Good, 1958] (works for $n = pq$ when $\gcd(p, q) = 1$).
- Split-radix [Duhamel and Hollman, 1984] (works for $n = 4k$).
- Rader [1968] (works when n is prime).

Cooley-Tukey FFT algorithm

DSP book:

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk} = \sum_{j_2=0}^{p-1} \left[\left(\sum_{j_1=0}^{q-1} x_{pj_1+j_2} \omega_q^{j_1 k_1} \right) \omega_n^{j_2 k_1} \right] \omega_p^{j_2 k_2} ,$$

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

OCaml code:

```
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))
```

Outline

- Dag creation.
- The simplifier.
- Register-oblivious scheduling.
- Related work.
- Conclusion.

The simplifier

Well-known optimizations:

- Algebraic simplification, e.g., $a + 0 \to a$.
- Constant folding.
- Common-subexpression elimination. (In the current implementation, CSE is encapsulated into a *monad*.)

Specific to the FFT:

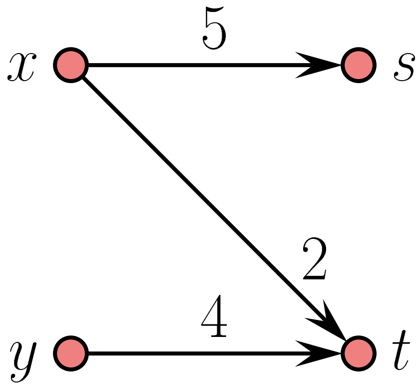
- (Elimination of negative constants.)
- Network transposition.

Why did I implement these well-known optimizations, which are implemented in C compilers anyway?
Because the the specific optimizations can only be done after the well-known optimizations.

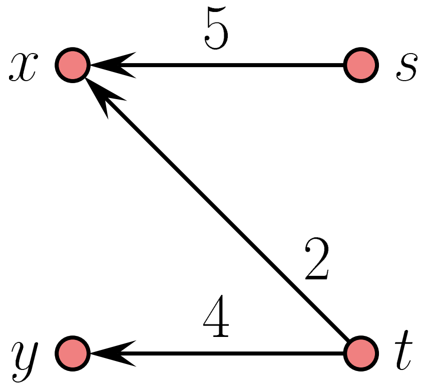
Network transposition

A network is a dag that computes a linear function [Crochiere and Oppenheim, 1975]. (Recall that the FFT is linear.)

Reversing all edges yields a ***transposed*** network.



$$\begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} 5 & 0 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 5 & 2 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix}$$

If a network computes $X = AY$, the transposed network computes $Y = A^T X$.

Optimize the transposed dag!

`genfft` employs this dag optimization algorithm:

$$
\begin{aligned}
&\textsc{Optimize}(G) = \\
&\quad G_1 := \textsc{Simplify}(G) \\
&\quad G_2^T := \textsc{Simplify}(G_1^T) \\
&\quad \textbf{return } \textsc{Simplify}(G_2)
\end{aligned}
$$

(Iterating the transposition has no effect.)

Benefits of network transposition

size	genfft adds	genfft muls *original*	genfft muls *transposed*	genfft savings	**literature** adds	**literature** muls
complex to complex						
5	32	16	12	25%	34	10
10	84	32	24	25%	88	20
13	176	88	68	23%	214	76
15	156	68	56	18%	162	50
real to complex						
5	12	8	6	25%	17	5
10	34	16	12	25%	?	?
13	76	44	34	23%	?	?
15	64	31	25	19%	67	25

Outline

- Dag creation.
- The simplifier.
- Register-oblivious scheduling.
- Related work.
- Conclusion.

How to help register allocators

`genfft`'s scheduler reorders the dag so that the register allocator of the C compiler can minimize the number of register spills.

`genfft`'s recursive partitioning heuristic is asymptotically optimal for $n = 2^k$.

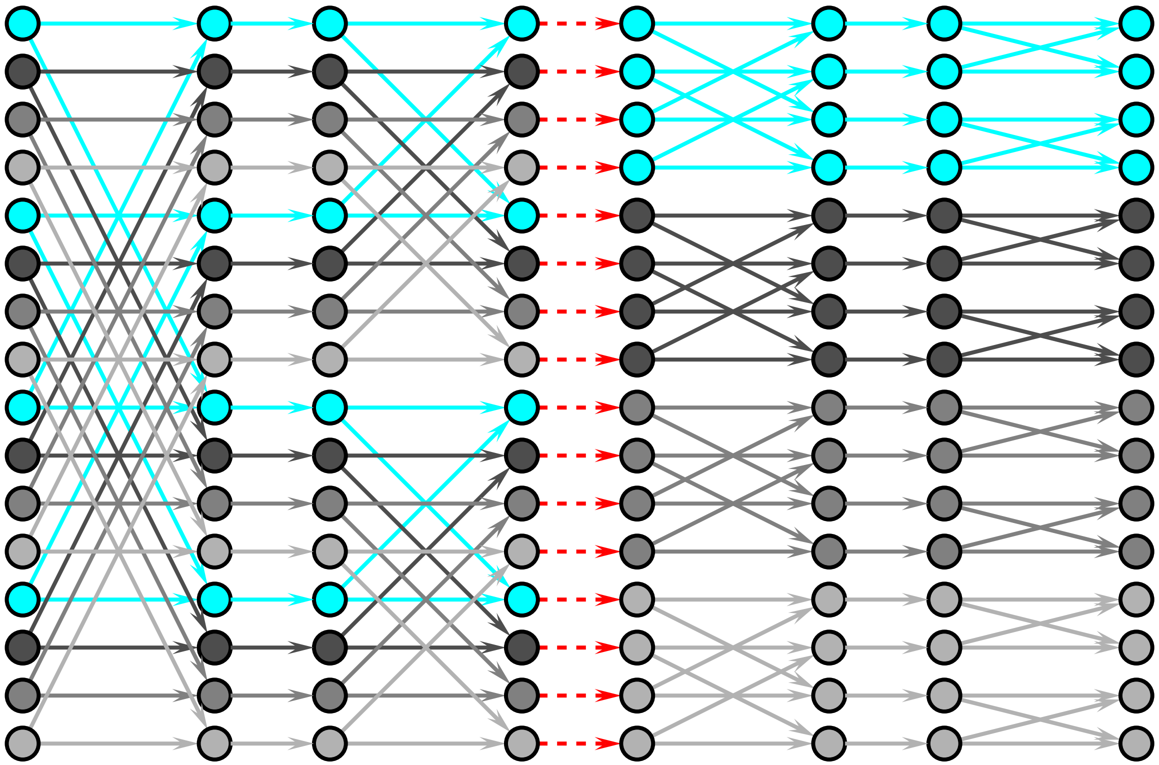
`genfft`'s schedule is ***register-oblivious***: This schedule does not depend on the number R of registers, and yet it is optimal for all R.

Certain C compilers insist on imposing their own schedule. With `egcs 1.1`/SPARC, FFTW is 50-100% faster if I use `-fno-schedule-insns`.

(How do I persuade the SGI compiler to respect my schedule?)

Recursive partitioning

The scheduler partitions the dag by cutting it in the "middle." This operation creates connected components that are scheduled recursively.



Register-oblivious scheduling

Assume that we want to compute the FFT of n points, and a computer has R registers, where $n > R$.

Lower bound [Hong and Kung, 1981]. If $n = 2^k$, the n-point FFT requires at least $\Omega(n \lg n / \lg R)$ register spills.

Upper bound. If $n = 2^k$, `genfft`'s output program for n-point FFT incurs at most $O(n \lg n / \lg R)$ register spills.

A more general theory of cache-oblivious algorithms exists [Frigo, Leiserson, Prokop, and Ramachandran, 1999].

Related work

- FOURGEN [Maruhn, 1976]. Written in PL/I, generates FORTRAN transforms of size 2^k.
- [Johnson and Burrus, 1983] automate the design of DFT programs using dynamic programming.
- [Selesnick and Burrus, 1986] generate MATLAB programs for FFTs of certain sizes.
- [Perez and Takaoka, 1987] generated prime-factor FFT programs.
- [Veldhuizen, 1995] uses a template metaprograms technique to generate C++.
- EXTENT [Gupta et al., 1996] compiles a tensor-product language into FORTRAN.

Conclusion

A domain-specific compiler is a valuable tool.

- For the FFT on modern processors, *performance* requires straight-line sequences with hundreds of instructions.
- *Correctness* was surprisingly easy.
- *Rapid turnaround*. (About 15 minutes to regenerate FFTW.)
- We can implement *problem-specific* code improvements.
- `genfft` derives *new algorithms*.