A Fast Fourier Transform Compiler

Matteo Frigo

MIT Laboratory for Computer Science

February 16, 1999

Presented by Tam Le

October 25, 2011

The Fast Fourier Transform

"The FFT has been called the most important numerical algorithm of our lifetime..." [Ken02]

The Discrete Fourier Transform Defined

- The **forward** DFT:

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{-ij}$$

$$\text{where } \omega_n = e^{2\pi\sqrt{-1}/n} \text{ and } 0 \leq i < n$$

- In case where X is real, the transform Y has *hermitian symmetry*:

$$Y[n-i] = Y^*[i]$$

$$\text{where } Y^*[i] \text{ is the complex conjugate}$$

The Discrete Fourier Transform Defined

- The **backward** DFT flips the sign in the exponent of ω_n and is defined as:

$$Y[i] = \sum_{j=0}^{n-1} X[j]\omega_n^{ij}$$

- Backward DFT is the "scaled inverse" of the forward DFT, i.e. backward transform of forward transform computes the original array multiplied by n

Cooley-Tukey [CT65]

- If n can be factored to $n = n_1 n_2$, rewrite DFT:

$$Y[i_1 + i_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 n_2 + j_2] \omega_{n_1}^{-i_1 j_2} \right) \omega_n^{-i_1 j_2} \right]$$

$$\text{where } j = j_1 n_2 + j_2 \text{ and } i = i_1 + i_2 n_1$$

- Divide and conquer scheme recursively breaks down DFT of size n into smaller DFTs of sizes n_1 and n_2
- $\omega_{n_1}^{-i_1 j_2}$ called *twiddle factors*

Prime Factor [OS89]

- Works for $n = n_1 n_2$ when $gcd(n_1, n_2) = 1$
- Avoids recursive multiplication of twiddle factors in place of more involved computations of indices

Fast Fourier Transform Algorithms

Split-Radix [DV90]

- Works for $n = 4k$
- Can lead to some saving of operations when compared with Cooley-Tukey

Rader's [Rad68]

- Works when n is prime
- Re-expresses DFT as "cyclic convolution" of size $n - 1$
- A special case of *Winograd* algorithm [Win78]

Computational Bounds

- Calculating DFT using straight-forward application of definition requires $O(n^2)$ arithmetical operations
- Calculating DFT using FFT algorithms have upper bound time complexities of $O(n \log n)$

The Fastest Fourier Transform in the West (FFTW)

- Original 1999 paper covers FFTW revision 2.0
- Latest version (3.0) will be discussed later
- Website: http://www.fftw.org/

What is FFTW?

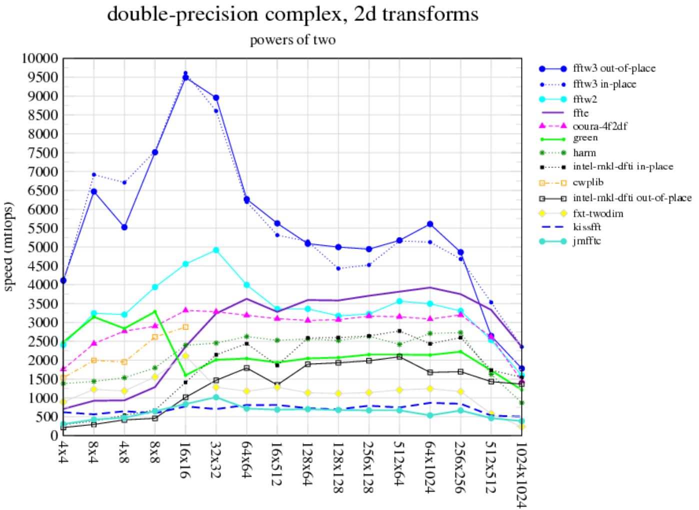
- Software library of fast C routines to compute one and multi-dimensional real and complex DFTs of arbitrary size
- Currently fastest FFT algorithm available upheld by regular benchmarks
- Speed advantage due to two distinguishing features:
  - FFTW's computational routines *adapts automatically* to the hardware providing for *portability* and *speed*
  - Inner loop of FFTW generated by a special-purpose compiler written in *Objective Caml*

`genfft`: a domain-specific FFT compiler

- `genfft` compiler is magic behind FFTW
- Written in *Objective Caml* 2.0
- From a complex number FFT algorithm, automatically derives a real number algorithm [Soren87]
- Automatic generation of inner loop of FFTW which comprises 95% of total code base

Benchmark: Just how fast compared to other FFTs?

Test System: 3.0 GHz Intel Core Duo, Intel compilers, 32-bit mode



Reasons for Speed Advantage?

- FFTW does not implement any single fixed DFT algorithm
- Instead, DFT is computed using a structured library of highly optimized blocks of C code called **codelets** which can be composed in many ways
- Composition of codelets is called a **plan** that determines which codelet should be executed in what order

Reasons for Speed Advantage?

- At runtime FFTW finds optimal composition of codelets by measuring speed of different plans, choosing the fastest
- FFTW contains 120 codelets with total of approximately 55,000 lines of optimized code to compute forward, backward, real to complex, and complex to real transforms

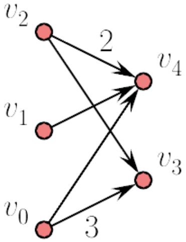
Outline of the Compilation Strategy

1. **Creation**: `genfft` produces a *directed acyclic graph (dag)* of the codelet according to an algorithm for the DFT; FFTW contains a number of such algorithms and applies the most appropriate
2. **Simplification**: `genfft` applies rewriting rules to each *dag* node in order to simplify the node

3. **Scheduling**: `genfft` applies a topological sort of the *dag* which minimizes the number of register spills "no matter how many registers the target machine has..."
4. **Unparsing**: `genfft` finally unparses to C (or to any other language by swapping out the unparser)

dag Representation

```
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.)

Definition of the `node` data type which represents an arithmetic expression *dag*. Cited [Aho86] for syntax tree representation.

dag Creation

- Function `fftgen` produces the expression *dag*
- `fftgen` performs symbolic evaluation of FFT algorithm to produce the *dag* for DFT of size n
- No single FFT algorithm is optimal for all size n so `genfft` contains many algorithms and `fftgen` chooses most appropriate
  - For example, for complex transform of size $n = 13$, generator employs Rader's algorithm in a variant formulated by Tolimieri et al. [Tol97]. However, that algorithm performs 214 real floating point additions and 76 real multiplications while generated FFTW code executes only 176 additions and 68 multiplications—`genfft` found simplifications overlooked by the authors!

dag Creation

- For FFTW version 2.0, `fftgen` implemented:
  1. **Cooley-Tukey** for $n = n_1 n_2$ where $n \neq 1$
  2. **Split-Radix** for n muliple of 4
  3. **Prime Factor** if n factors into $n_1 n_2$, $n \neq 1$, and $gcd(n_1, n_2) = 1$
  4. **Rader's** for prime length if $n = 5$ or $n \geq 13$
  5. Direct application of DFT definition

dag Creation

```
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))
```

OCaml code for **Cooley-Tukey** FFT algorithm. The infix operator `@*` computes the complex product while the function `exp n k` computes the constant $exp(2\pi k\sqrt{-1}/n)$.

Simplification

- Simplifier traverses *dag* bottom-up and applies series of "improvements" at every node
- Common, well-known optimizations [Aho86]:
  1. *Algebraic Transformations*: constant folding and simplify multiplication by $0, 1, -1$ and addition by 0
  2. *Common-Subexpression Elimination (CSE)*: simplifier implemented in *monadic* style [Wad97] in which the *monad* performs CSE

Simplification

- DFT-specific:
  1. *Eliminate negative constants.* Constants generally appear as pairs in a DFT *dag*; C compiler would store values in program text and then load both constants into a register at runtime. Thus, making all constants positive reduces load by factor of two, speeding up generated codelets by 10-15%
  2. *Network transposition.* Based on fact that network is a *dag* that computes a linear function [Cro75]

Simplification

`genfft`'s simplifier performs three passes over the *dag*:

$$
\begin{aligned}
\textsc{Optimize}(G) = \\
E &:= \textsc{Simplify}(G) \\
F^T &:= \textsc{Simplify}(E^T) \\
\textbf{RETURN}\ &\textsc{Simplify}(F)
\end{aligned}
$$

Summary of *dag* transposition benefits

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

Scheduling

- The `genfft` scheduler produces a topological sort of the *dag* so register allocator of C compiler can minimize number of register spills
- Proven [HK81] that for DFTs of size power of 2 ($n = 2^k$), there exists a schedule that is asymptotically optimal

Scheduling

- `genfft's` schedule is *cache-oblivious*, i.e. not dependent on the number R of registers on a machine and yet optimal for every R
- In fact, execution of FFT *dag* of size $n = 2^k$ on a machine of R registers where $R \leq n$ has:
  1. lower bound of $\Omega(n \log n / \log R)$ register spills
  2. upper bound in which `gennfft's` output program incurs at most $O(n \log n / \log R)$ register spills

Runtime & Memory Footprint

- Takes approximately 75 seconds for DFT of size $n = 64$ to run FFTW generated C code on a 200MHz Pentium Pro machine running Linux 2.2
- `genfft` needs less than 3 MB of memory to complete generation which resulted in a codelet containing 912 additions and 248 multiplications
- Regeneration of whole FFTW system can be done in approximately 15 minutes

Some Conclusions to Draw

- *Optimal Performance*: Main goal of project achieved since up-to-date benchmarks show FFTW's performance still ahead of other competing FFTs

Some Conclusions to Draw

- *Correctness*: In words of author: "surprisingly easy." Since DFT algorithms in `genfft` were encoded using a straight-forward, high-level language (*OCaml*), *simplification* phase of the compiler transforms algorithms into optimized code via application of simple algebraic rules which are easy to verify

Some Conclusions to Draw

- *Rapid Turnaround*: Just around 15 minutes (back in 1999) to regenerate FFTW form scratch

Some Conclusions to Draw

- *Domain-specific code enhancements*: Topological sort in *scheduling* phase is effective only for DFT *dags* and perform poorly for other computations while *simplification* performs certain improvements which rely on DFT being a *linear* transformation
- `genfft` "derived" or "discovered" *new algorithms*, as in case of $n = 13$ discussed earlier

FFTW v3.0

- Released April 2003
- Latest stable release: v3.3, Jul 26, 2011
- Major enhancements:
  1. Complete rewrite adding new algorithms and FFTs (*Bluestein's, etc.*)
  2. Improved speed: programs often 20% faster than comparable FFTW 2.x code
  3. New set of APIs to support more general semantics
  4. Single Instruction, Multiple Data (SIMD) support for parallel processing CPUs (SSE, SSE2, 3DNow!, Altivec)
  5. Read release notes for full list of improvements and bug fixes: http://www.fftw.org/release-notes.html

Awards & Recognition

- 1999 J. H. Wilkinson Prize for Numerical Software (awarded every 4 years)
- 2009 Most Influential PLDI Paper Award (http://sigplan.org/award-pldi.htm)

Questions & Answers?

References

- [Ken02] Kent, Ray D. and Read, Charles (2002). *Acoustic Analysis of Speech*. ISBN 0-7693-0112-6. Cites Strang, G. (1994)/MayJune). Wavelets. *American Scientist*, 82, 250-255.
- [CD65] J. W. Cooley and J.W. Tukey. An algorithm for the machine computation of the complex Fourier series. *Mathematics of Computation*, 19:297301, April 1965.
- [OS89] A. V. Oppenheim and R. W. Schafer. *Discrete-time Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ 07632, 1989.
- [DV90] P. Duhamel and M. Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Processing*, 19:259299, April 1990.

References

- [Rad68] C. M. Rader. Discrete Fourier transforms when the number of data samples is prime. *Proc. of the IEEE*, 56:11071108, June 1968.
- [Win78] S. Winograd. On computing the discrete Fourier transform. *Mathematics of Computation*, 32(1):175199, January 1978.
- [Aho86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques, and tools*. Addison-Wesley, March 1986.
- [Tol97] Richard Tolimieri, Myoung An, and Chao Lu. *Algorithms for Discrete Fourier Transform and Convolution*. Springer Verlag, 1997.

References

- [Wad97] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240263, September 1997.
- [Cro75] R. E. Crochiere and A. V. Oppenheim. Analysis of linear digital networks. *Proceedings of the IEEE*, 63:581595, April 1975.
- [Soren87] H. V. Sorensen, D. L. Jones, M. T. Heideman, and C. S. Burrus. Real-valued fast Fourier transform algorithms. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(6):849863, June 1987.