The secrets of FFTW: the Fastest Fourier Transform in the West

Tomasi Maurizio

INAF

January, 2012

What is FFTW?

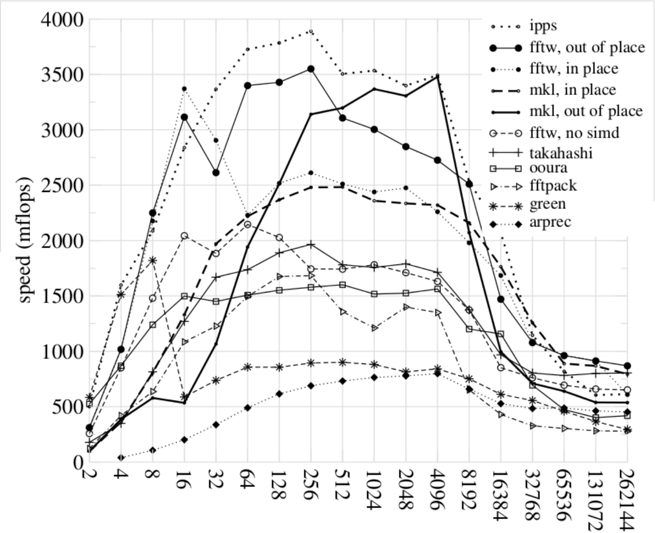
"FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data (as well as of even/odd data, i.e. the discrete cosine/sine transforms or DCT/DST)."

`http://www.fftw.org/`

Part I

The Fourier Transform

Benchmarks



speed (mflops)
4000
3500
3000
2500
2000
1500
1000
500
0
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
131072
262144
ipps
fftw, out of place
fftw, in place
mkl, in place
mkl, out of place
fftw, no simd
takahashi
ooura
fftpack
green
arprec

The Fourier transform

Discrete FT formula $x \to y$:

$$y[i] = \sum_{j=0}^{n-1} x[j] \omega_n^{-ij},$$

with $\omega_n = e^{2\pi i/n}$. This is a $O(N^2)$ algorithm, which means it does not scale well.

The Fast Fourier transform

In 1965 Cooley and Turkey proved that if $n = n_1 n_2$ then

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

yields the same results.

The Fast Fourier transform

In 1965 Cooley and Turkey proved that if $n = n_1 n_2$ then

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

yields the same results.
Since the inner sum is a DFT, the procedure can be recursive. If $N = 2^k$, then the algorithm is $O(N \log N)$.

The Fast Fourier transform

Cool! Our problems are solved!

The Fast Fourier transform

Cool! Our problems are solved!

Not so fast, mister…

Problems in writing a FFT library (1/4)

To compute the FT of a vector of n elements you can use:

1. Cooley-Tuckey's algorithm (if $n = n_1 n_2$);

Problems in writing a FFT library (1/4)

To compute the FT of a vector of n elements you can use:

1. Cooley-Tuckey's algorithm (if $n = n_1 n_2$);
2. Cooley-Tuckey's prime factor algorithm (as above, but $\gcd(n_1, n_2) = 1$);

Problems in writing a FFT library (1/4)

To compute the FT of a vector of n elements you can use:

1. Cooley-Tuckey's algorithm (if $n = n_1 n_2$);
2. Cooley-Tuckey's prime factor algorithm (as above, but $\gcd(n_1, n_2) = 1$);
3. Split-radix algorithm (if n is a multiple of 4);

Problems in writing a FFT library (1/4)

To compute the FT of a vector of n elements you can use:

1. Cooley-Tuckey's algorithm (if $n = n_1 n_2$);
2. Cooley-Tuckey's prime factor algorithm (as above, but $\gcd(n_1, n_2) = 1$);
3. Split-radix algorithm (if n is a multiple of 4);
4. Rader's algorithm (if n is prime);

Problems in writing a FFT library (1/4)

To compute the FT of a vector of n elements you can use:

1. Cooley-Tuckey's algorithm (if $n = n_1 n_2$);
2. Cooley-Tuckey's prime factor algorithm (as above, but $\gcd(n_1, n_2) = 1$);
3. Split-radix algorithm (if n is a multiple of 4);
4. Rader's algorithm (if n is prime);
5. Plain definition of the FT (any n)

Problems in writing a FFT library (1/4)

To compute the FT of a vector of n elements you can use:

1. Cooley-Tuckey's algorithm (if $n = n_1 n_2$);
2. Cooley-Tuckey's prime factor algorithm (as above, but $\gcd(n_1, n_2) = 1$);
3. Split-radix algorithm (if n is a multiple of 4);
4. Rader's algorithm (if n is prime);
5. Plain definition of the FT (any n)
6. ... and many others!

Problems in writing a FFT library (2/4)

Need to support:

1. Real and complex data
2. Single precision and double precision
3. Forward ($\rightarrow$) and backward ($\leftarrow$) transforms

Thus, $2^3 = 8$ combinations for each algorithm you want to implement.

Problems in writing a FFT library (2/4)

Need to support:

1. Real and complex data
2. Single precision and double precision
3. Forward ($\rightarrow$) and backward ($\leftarrow$) transforms

Thus, $2^3 = 8$ combinations for each algorithm you want to implement.

(And this does not consider multidimensional transforms...)

Problems in writing a FFT library (3/4)

Sometimes you can rewrite a mathematical formula in a way that is computationally more efficient, e.g.:

$$y = ax^4 + bx^3 + cx^2 + dx + e$$

(10 multiplications, 4 additions) can be rewritten as

$$y = x(x(x(ax + b) + c) + d) + e$$

(4 multiplications, 4 additions).

Problems in writing a FFT library (3/4)

Sometimes you can rewrite a mathematical formula in a way that is computationally more efficient, e.g.:

$$y = ax^4 + bx^3 + cx^2 + dx + e$$

(10 multiplications, 4 additions) can be rewritten as

$$y = x(x(x(ax + b) + c) + d) + e$$

(4 multiplications, 4 additions). Again, you have to do this optimization for all the algorithms/variants you want to implement!

Problems in writing a FFT library (4/4)

One algorithm can be more efficient than another on some CPU, and vice versa on a different architecture.

Problems in writing a FFT library (4/4)

One algorithm can be more efficient than another on some CPU, and vice versa on a different architecture.

For instance, an algorithm requires 3 sums and 2 multiplications, another one 5 sums and 1 multiplication. Which one do you choose?

Problems in writing a FFT library (4/4)

One algorithm can be more efficient than another on some CPU, and vice versa on a different architecture.

For instance, an algorithm requires 3 sums and 2 multiplications, another one 5 sums and 1 multiplication. Which one do you choose?

This applies to FFT, as e.g., if $N = 24$ you can either use Cooley-Tuckey (since $N = 3 \times 2^3$) or the split-radix algorithm (since $N = 4n$).

To recap

1. One definition of FT, but many algorithms and ways of coding them.
2. Each one must be optimized;
3. Not clear which one is the best if you do not know *a priori* the architecture you're going to run your program on.

Part II

FFTW's approach

Problems and solutions

1. One definition of FT, but many algorithms and ways of coding them.

2. Each one must be optimized;

3. Not clear which one is the best. . .

Problems and solutions

1. One definition of FT, but many algorithms and ways of coding them. $\rightarrow$ Specify the algorithms in some high-level language, then automatically translate them.
2. Each one must be optimized;
3. Not clear which one is the best…

Problems and solutions

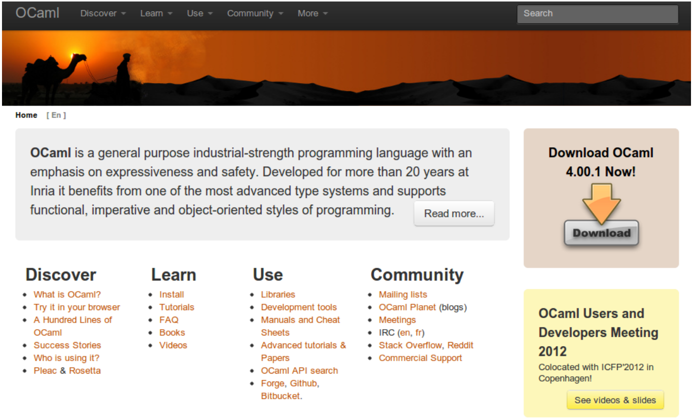
1. One definition of FT, but many algorithms and ways of coding them. $\rightarrow$ Specify the algorithms in some high-level language, then automatically translate them.
2. Each one must be optimized; $\rightarrow$ Make an optimizing compiler do the translation.
3. Not clear which one is the best...

Problems and solutions

1. One definition of FT, but many algorithms and ways of coding them. $\rightarrow$ Specify the algorithms in some high-level language, then automatically translate them.
2. Each one must be optimized; $\rightarrow$ Make an optimizing compiler do the translation.
3. Not clear which one is the best... $\rightarrow$ Profile each algorithm at runtime, before actually using the library (create a plan).

FT algorithms in FFTW

FFTW specifies FT algorithms using OCaml (`http://www.ocaml.org`), a high-level functional language with some neat features.



Example in OCaml

To see how the features of OCaml can be useful for writing FT algorithms, we'll first show how to solve a simple problem using OCaml:

How would you write a function that calculates derivatives?

Differentiation in C

```c
/* derivative.c
   cc -o derivative derivative.c -lm */
#include <float.h>
#include <math.h>
#include <stdio.h>

typedef double fn_t (double);
double derivative(fn_t * f, double x)
{
    const double eps = 1e-6;
    return ((*f)(x + eps) - (*f)(x)) / eps;
}

void main(void)
{
    printf("The derivative of cos(x) in x=1 is %f\n",
           derivative(cos, 1));
}
```

Differentiation in OCaml

```
(* derivative.ml
   ocamlopt -o derivative derivative.ml *)

(* There's no need to specify types,
   as the compiler will infer them *)
let derivative f x =
  let eps = 1e-6
  in (f (x +. eps) -. f x) /. eps;;

Printf.printf "The derivative of cos(x) in x=1 is %f\n"
              (derivative cos 1.0);;
```

Differentiation: improvements

Can we do better?

Differentiation: improvements

Computing the derivative symbolically would make us safe from rounding errors (why using 10^{-6} for `eps` instead of 10^{-8}?).

Differentiation: improvements

Computing the derivative symbolically would make us safe from rounding errors (why using 10^{-6} for `eps` instead of 10^{-8}?).

It would also allow to make a few optimizations, e.g.:

```
double function(double x)
{
  double constant = extremely_slow_function();
  return x + constant;
}
```

Differentiation: improvements

Computing the derivative symbolically would make us safe from rounding errors (why using 10^{-6} for `eps` instead of 10^{-8}?).

It would also allow to make a few optimizations, e.g.:

```
double function(double x)
{
  double constant = extremely_slow_function();
  return x + constant;
}
```

However, it is extremely hard to do this in C/C++/Python. . . .

Differentiation in OCaml: expressions

Let's see how to do this in OCaml. We'll follow a tutorial by Jon Harrop, the author of "OCaml for Scientists"

`http://www.ffconsultancy.com/ocaml/benefits/symbolic.html`.

The idea

- To compute a derivative, we need to know the inner structure of a function;

The idea

- To compute a derivative, we need to know the inner structure of a function;
- But a C/OCaml function like `sin` is a $\mathbb{R} \to \mathbb{R}$ "black box";

The idea

- To compute a derivative, we need to know the inner structure of a function;
- But a C/OCaml function like `sin` is a $\mathbb{R} \to \mathbb{R}$ "black box";
- We therefore need to specify functions symbolically, by means of an ad-hoc type;

The idea

- To compute a derivative, we need to know the inner structure of a function;
- But a C/OCaml function like `sin` is a $\mathbb{R} \to \mathbb{R}$ "black box";
- We therefore need to specify functions symbolically, by means of an ad-hoc type;
- We need to define some mathematical operators on this type, as well as their properties;

The idea

- To compute a derivative, we need to know the inner structure of a function;
- But a C/OCaml function like `sin` is a $\mathbb{R} \to \mathbb{R}$ "black box";
- We therefore need to specify functions symbolically, by means of an ad-hoc type;
- We need to define some mathematical operators on this type, as well as their properties;
- Last but not least, we need to specify how to compute derivatives!

Differentiation in OCaml: expressions

```
type expr =
   | Add of expr * expr (* Sum of two expressions *)
   | Mul of expr * expr (* Product of two expressions *)
   | Int of int         (* Integer constant *)
   | Var of string      (* Named variable, like "x" *)
   | Sin of expr        (* Sine *)
   | Cos of expr ;;     (* Cosine *)
```

Differentiation in OCaml: expressions

```
type expr =
  | Add of expr * expr (* Sum of two expressions *)
  | Mul of expr * expr (* Product of two expressions *)
  | Int of int         (* Integer constant *)
  | Var of string      (* Named variable, like "x" *)
  | Sin of expr        (* Sine *)
  | Cos of expr ;;     (* Cosine *)
```

Example: $\sin(3x + 1) + 2x$ becomes

```
let x = Var("x") in
    Add(Sin(Add(Mul(Int 3, x),
                Int 1)),
        Mul(Int 2, x))
```

Differentiation in OCaml: operations

Defining expressions in this way is boring!

Differentiation in OCaml: operations

Defining expressions in this way is boring!
We define a nice shorthand for `Add` by defining a new mathematical operator, `+:`, and using OCaml's powerful pattern matching:

```
let rec ( +: ) f g = match f, g with
    | Int n, Int m           -> Int (n + m)
    | Int 0, f | f, Int 0    -> f
    | f, Add(g, h)           -> f +: g +: h
    | f, g when f > g        -> g +: f
    | f, g                   -> Add(f, g) ;;
```

Differentiation in OCaml: operations

We do the same for `Mul`:

```
(* Rules for multiplication *)
let rec ( *: ) f g = match f, g with
    | Int n, Int m          -> Int (n * m)
    | Int 0, _ | _, Int 0 -> Int 0
    | Int 1, f | f, Int 1 -> f
    | f, Mul(g, h)          -> f *: g *: h
    | f, g when f > g       -> g *: f
    | f, g                  -> Mul(f, g) ;;
```

Differentiation in OCaml: operations

Now $\sin(3x + 1) + 2x$ can be written as

```
let x = Var("x") in
    Sin(Int 3 *: x +: Int 1) +: Int 2 *: x
```

The OCaml compiler will translate it into

```
let x = Var("x") in
    Add(Sin(Add(Mul(Int 3, x),
                Int 1)),
        Mul(Int 2, x))
```

(but now it's able to do simplifications, e.g., multiplying by 1).

Differentiation in OCaml: the core

This is the implementation of `d`, the differential operator.

```
let rec d f x = match f with
  | Var y when x=y -> Int 1
  | Var _ | Int _ -> Int 0
  | Add(f, g) -> d f x +: d g x
  | Mul(f, g) -> f *: d g x +: g *: d f x
  | Sin(f) -> Cos(f) *: d f x
  | Cos(f) -> Int (-1) *: Sin(f) *: d f x ;;
```

Pretty-printing

```
open Format;;
let rec print_expr ff = function
  | Int n -> fprintf ff "%d" n
  | Var v -> fprintf ff "%s" v
  | Sin(f) -> fprintf ff "sin(%a)" print_expr f
  | Cos(f) -> fprintf ff "cos(%a)" print_expr f
  | Add(f, g) -> fprintf ff "%a +@;<1 2>%a"
                         print_expr f print_expr g
  | Mul(Add _ as f, g) ->
      fprintf ff "(@[%a@])@;<1 2>%a"
                 print_expr f print_expr g
  | Mul(f, g) -> fprintf ff "%a@;<1 2>%a"
                         print_expr f print_expr g;;
#install_printer print_expr;;
```

(Run these commands at the OCaml prompt.)

Example

Run this at the OCaml prompt (#):

```
# let     a = Var "a"
      and b = Var "b"
      and c = Var "c"
      and x = Var "x" ;;
# let expr = a*:x*:x +: b*:x +: x*:Sin(Int 2 *: x)
# expr ;;
- : expr = a x x + b x + x sin(2 x)
# d expr "x" ;;
- : expr = a x + a x + b + 2 x cos(2 x) + sin(2 x)
```

$$D_x\left(ax^2+bx+x\sin 2x\right) = 2ax+b+2x\cos 2x+\sin 2x.$$

Lessons learned

To recap:

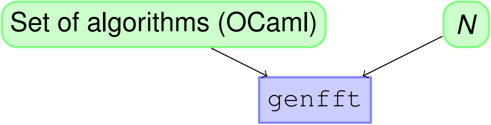
- We specify the algorithm (derivation) symbolically;
- We specify how to perform optimizations on the expressions;
- We translate one symbolic expression (function to be derived) into another one (derivative).
- (This required 27 lines of code!)

How does this apply to FFTW?

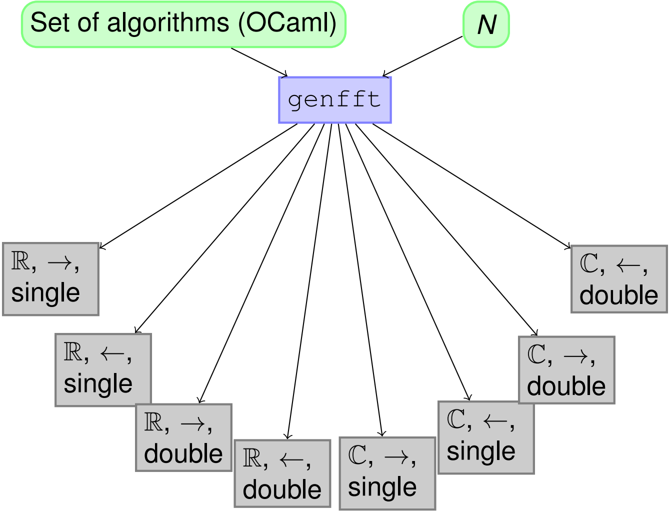
FFTW uses the same idea to manipulate FT algorithms:

- Define a data type (like our `expr`) that represents a Fourier Transform;
- Define a function, called `genfft`, that transforms such data types (like our function `d`);
- The output of `genfft` is a stream of characters which make the source code of a set of C functions.

The workflow of `genfft`



The workflow of `genfft`



Example: Cooley-Tukey

The formula:

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

The code passed as input to `genfft`:

```
let rec cooley_tukey n1 n2 input sign =
  let tmp1 j2 = fftgen n1
      (fun j1 -> input (j1 * n2 + j2)) sign in
  let tmp2 i1 j2 =
      exp n (sign * i1 * j2) @* tmp1 j2 i1)) in
  let tmp3 i1 = fftgen n2 (tmp2 i1) sign in
     (fun i -> tmp3 (i mod n1) (i / n1)) ;;
```

Example output from `genfft` (1/2)

```
/* This function contains 4 FP additions,
 * 0 FP multiplications, (or, 4 additions,
 * 0 multiplications, 0 fused multiply/add),
 * 5 stack variables, 0 constants, and 8
 * memory accesses */
void n1_2(const R *ri, const R *ii, R *ro, R *io,
          stride is, stride os, INT v, INT ivs,
          INT ovs) {
  INT i;
  for (i = v; i > 0; i = i - 1, ri = ri + ivs,
       ii = ii + ivs, ro = ro + ovs, io = io + ovs,
       MAKE_VOLATILE_STRIDE(is),
       MAKE_VOLATILE_STRIDE(os)) {
     E T1, T2, T3, T4;
     T1 = ri[0];
     T2 = ri[WS(is, 1)];
     /* (continue...) */
```

Example output from `genfft` (2/2)

```
      T3 = ii[0];
      T4 = ii[WS(is, 1)];
      ro[0] = T1 + T2;
      ro[WS(os, 1)] = T1 - T2;
      io[0] = T3 + T4;
      io[WS(os, 1)] = T3 - T4;
   }
}
```

References

- M. Frigo, *A Fast Fourier Transform Compiler*. Proceedings of the 1999 ACM SIGPLAN (May 1999).
- M. Frigo, *The Design and Implementation of FFTW3*, Proceedings of the IEEE 93 (2), 216231 (2005)
- The OCaml website, `http://ocaml.org`.
- J. Harrop, *OCaml for scientists*, `http://www.ffconsultancy.com/products/ocaml_for_scientists`.

Imperative vs. functional

Imperative machine

- Turing's work: 1936-37
- First high-level language: Fortran (1954)
- C/C++, C#, Pascal, Ada, Python...

λ-calculus

- Church's papers: 1933, 1935
- First language: LISP (1958)
- OCaml, Haskell, Scala, F#...

The two concepts are equivalent. See

`http://www.infoq.com/presentations/Y-Combinator`.

Project Euler's Problem 34

Quiz: write the sum of all the numbers *n* between 10 and 10^7 that are equal to the factorials of their digits (e.g., $145 = 1! + 4! + 5!$).

Project Euler's Problem 34

Quiz: write the sum of all the numbers *n* between 10 and 10^7 that are equal to the factorials of their digits (e.g., $145 = 1! + 4! + 5!$).

(The answer is 40 730.)

Problem 34 in Python

```python
def fact(n):
    if n < 2: return 1
    else:
        result = 1
        for i in xrange(2, n + 1): result = result * i
        return result

FAST_FACT = tuple ([fact(x) for x in xrange(0, 10)])

def digits (n):
    return [int(x) for x in list(str(n))]

def test_number (n):
    return n == sum([FAST_FACT[digit]
                     for digit in digits(n)])

print sum([num for num in xrange(10, 10000000)
           if test_number(num)])
```

Problem 34 in OCaml (1/2)

```
(* Array with the factorials of the 10 digits *)
let fact =
    let rec f n = if n > 1 then n * f (n-1) else 1
    in Array.map f [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|];;

let sum list_of_nums =
    List.fold_left (+) 0 list_of_nums;;

(* Return a list with the digits of 'num' *)
let digits num =
  let rec f num result =
    if num < 10 then num :: result
    else f (num / 10) ((num mod 10) :: result)
  in f num [];;

let test_number num =
  num == sum (List.map (fun x->fact.(x)) (digits num));;
```

Problem 34 in OCaml (2/2)

```
let calc_sum max =
  let rec helper start cumul =
    if start >= max then
      cumul
    else
      (* Tail call *)
      helper (start + 1)
             (if test_number start then
                (cumul + start)
             else
                cumul)
  in helper 10 0 ;;

print_endline (string_of_int (calc_sum 10000000));
```

Problem 34 in Haskell

```haskell
-- File problem-34.hs
--
-- Compile it with
--   ghc -o problem-34 problem-34.hs

import Data.Char (digitToInt)

main = print (sum ([x|x <- [10..100000],
                      x == sum (map (\n -> product [1..n])
                                    (map (digitToInt)
                                         (show x)))]))
```

Benchmarks

Language	LOC	Running time
Python	18	59.0 s
OCaml	27	2.7 s
Haskell	6	0.2 s

Benchmarks

Language	LOC	Running time
Python	18	59.0 s
OCaml	27	2.7 s
Haskell	6	0.2 s

Haskell is 300 times faster than Python

Benchmarks

Language	LOC	Running time
Python	18	59.0 s
OCaml	27	2.7 s
Haskell	6	0.2 s

Haskell is 300 times faster than Python and three times more concise.

Benchmarks

Language	LOC	Running time
Python	18	59.0 s
OCaml	27	2.7 s
Haskell	6	0.2 s

Haskell is 300 times faster than Python and three times more concise.
In this example OCaml is more verbose than Python, but still much faster.