The Design and Implementation of FFTW3

Matteo Frigo and Steven G. Johnson

(Invited Paper)

***Abstract*— FFTW is an implementation of the discrete Fourier transform (DFT) that adapts to the hardware in order to maximize performance. This paper shows that such an approach can yield an implementation that is competitive with hand-optimized libraries, and describes the software structure that makes our current FFTW3 version flexible and adaptive. We further discuss a new algorithm for real-data DFTs of prime size, a new way of implementing DFTs by means of machine-specific "SIMD" instructions, and how a special-purpose compiler can derive optimized implementations of the discrete cosine and sine transforms automatically from a DFT algorithm.**

***Index Terms*— FFT, adaptive software, Fourier transform, cosine transform, Hartley transform, I/O tensor.**

I. Introduction

FFTW [1] is a widely used free-software library that computes the discrete Fourier transform (DFT) and its various special cases. Its performance is competitive even with vendor-optimized programs, but unlike these programs, FFTW is not tuned to a fixed machine. Instead, FFTW uses a *planner* to adapt its algorithms to the hardware in order to maximize performance. The input to the planner is a *problem*, a multi-dimensional loop of multi-dimensional DFTs. The planner applies a set of rules to recursively decompose a problem into simpler sub-problems of the same type. "Sufficiently simple" problems are solved directly by optimized, straight-line code that is automatically generated by a special-purpose compiler. This paper describes the overall structure of FFTW as well as the specific improvements in FFTW3, our latest version.

FFTW is fast, but its speed does not come at the expense of flexibility. In fact, FFTW is probably the most flexible DFT library available:

- FFTW is written in portable C and runs well on many architectures and operating systems.
- FFTW computes DFTs in $O(n \log n)$ time for any length n. (Most other DFT implementations are either restricted to a subset of sizes or they become $\Theta(n^2)$ for certain values of n, for example when n is prime.)
- FFTW imposes no restrictions on the rank (dimensionality) of multi-dimensional transforms. (Most other implementations are limited to one-dimensional, or at most two- and three-dimensional data.)
- FFTW supports multiple and/or strided DFTs; for example, to transform a 3-component vector field or a portion of a multi-dimensional array. (Most implementations support only a single DFT of contiguous data.)
- FFTW supports DFTs of real data, as well as of real symmetric/antisymmetric data (also called discrete cosine/sine transforms).

The interaction of the user with FFTW occurs in two stages: planning, in which FFTW adapts to the hardware, and execution, in which FFTW performs useful work for the user. To compute a DFT, the user first invokes the FFTW *planner*, specifying the *problem* to be solved. The problem is a data structure that describes the "shape" of the input data—array sizes and memory layouts—but does not contain the data itself. In return, the planner yields a *plan*, an executable data structure that accepts the input data and computes the desired DFT. Afterwards, the user can execute the plan as many times as desired.

The FFTW planner works by measuring the actual run time of many different plans and by selecting the fastest one. This process is analogous to what a programmer would do by hand when tuning a program to a fixed machine, but in FFTW's case no manual intervention is required. Because of the repeated performance measurements, however, the planner tends to be time-consuming. In performance-critical applications, many transforms of the same size are typically required, and therefore a large one-time cost is usually acceptable. Otherwise, FFTW provides a mode of operation where the planner quickly returns a "reasonable" plan that is not necessarily the fastest.

The planner generates plans according to rules that recursively decompose a problem into simpler sub-problems. When the problem becomes "sufficiently simple," FFTW produces a plan that calls a fragment of optimized straight-line code that solves the problem directly. These fragments are called *codelets* in FFTW's lingo. You can envision a codelet as computing a "small" DFT, but many variations and special cases exist. For example, a codelet might be specialized to compute the DFT of real input (as opposed to complex). FFTW's speed depends therefore on two factors. First, the decomposition rules must produce a space of plans that is rich enough to contain "good" plans for most machines. Second, the codelets must be fast, since they ultimately perform all the real work.

FFTW's codelets are generated automatically by a special-purpose compiler called `genfft`. Most users do not interact with `genfft` at all: the standard FFTW distribution contains a set of about 150 pre-generated codelets that cover the most common uses. Users with special needs can use `genfft` to generate their own codelets. `genfft` is useful because of the following features. From a high-level mathematical description of a DFT algorithm, `genfft` derives an optimized implementation automatically. From a complex-DFT algo-

M. Frigo is with the IBM Austin Research Laboratory, 11501 Burnet Road, Austin, TX 78758. He was supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004.

S. G. Johnson is with the Massachusetts Institute of Technology, 77 Mass. Ave. Rm. 2-388, Cambridge, MA 02139. He was supported in part by the Materials Research Science and Engineering Center program of the National Science Foundation under award DMR-9400334.