FFTW: AN ADAPTIVE SOFTWARE ARCHITECTURE FOR THE FFT

Matteo Frigo

MIT Laboratory for Computer Science
545 Technology Square NE43-203
Cambridge, MA 02139
athena@theory.lcs.mit.edu

Steven G. Johnson

Massachusetts Institute of Technology
77 Massachusetts Avenue, 12-104
Cambridge, MA 02139
stevenj@alum.mit.edu

ABSTRACT

FFT literature has been mostly concerned with minimizing the number of floating-point operations performed by an algorithm. Unfortunately, on present-day microprocessors this measure is far less important than it used to be, and interactions with the processor pipeline and the memory hierarchy have a larger impact on performance. Consequently, one must know the details of a computer architecture in order to design a fast algorithm. In this paper, we propose an adaptive FFT program that tunes the computation automatically for any particular hardware. We compared our program, called FFTW, with over 40 implementations of the FFT on 7 machines. Our tests show that FFTW's self-optimizing approach usually yields significantly better performance than all other publicly available software. FFTW also compares favorably with machine-specific, vendor-optimized libraries.

1. INTRODUCTION

The discrete Fourier transform (DFT) is an important tool in many branches of science and engineering [1] and has been studied extensively [2]. For many practical applications, it is important to have an implementation of the DFT that is as fast as possible. In the past, speed was the direct consequence of clever algorithms [2] that minimized the number of arithmetic operations. On present-day general-purpose microprocessors, however, the performance of a program is mostly determined by complicated interactions of the code with the processor pipeline, and by the structure of the memory. Designing for performance under these conditions requires an intimate knowledge of the computer architecture. In this paper, we address this problem by means of a novel *adaptive* approach, where the program itself adapts the computation to the details of the hardware. We developed FFTW, an adaptive, high performance implementation of the Cooley-Tukey fast Fourier transform (FFT) algorithm [3], written in C. We have compared many C and Fortran implementations of the DFT on several machines, and our experiments show that FFTW typically yields significantly better performance than all other publicly available DFT software. More interestingly, while retaining complete portability, FFTW is competitive with or faster than proprietary codes such as Sun's Performance Library and IBM's ESSL library that are highly tuned for a single machine. Such encouraging results raise the hope that similar adaptive techniques could be applied successfully to other problems.

```
fftw_plan plan;
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);
```

Figure 1: Simplified example of FFTW's use. The user must first create a plan, which can be then used at will.

In FFTW, the computation of the transform is accomplished by an *executor* that consists of highly optimized, composable blocks of C code called *codelets*. A codelet is a specialized piece of code that computes part of the transform. The combination of codelets applied by the executor is specified by a special data structure called a *plan*. The plan is determined at runtime, before the computation begins, by a *planner* which uses a dynamic programming algorithm [4, chapter 16] to find a fast composition of codelets. The planner tries to minimize the actual *execution time*, and not the *number of floating point operations*, since, as we show in Section 2, there there is little correlation between these two performance measures. Consequently, the planner measures the run time of many plans and selects the fastest. In the current implementation, plans can also be saved to disk and used at a later time.

The speed of the executor depends crucially on the efficiency of the codelets, but writing and optimizing them is a tedious and error-prone process. For this reason, we found it convenient to generate the codelets automatically by means of a special-purpose compiler. FFTW's *codelet generator*, written in the Caml Light dialect of the functional language ML [5], is a sophisticated program that first produces a representation of the codelet in the form of abstract C syntax tree, and then "optimizes" the codelet by applying well known transformations such as constant folding and algebraic identities. The main advantages of generating code are that it is simple to experiment with different algorithms or coding strategies, and it is easy to produce many long blocks of unrolled, optimized code.

FFTW's internal complexity is not visible to the user, however. The user interacts with FFTW only through the planner and

Matteo Frigo was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant N00014-94-1-0985, and by a Digital Equipment Corporation Fellowship. Steven G. Johnson was supported in part by a DoD NDSEG Fellowship, an MIT Karl Taylor Compton Fellowship, and by the Materials Research Science and Engineering Center program of the National Science Foundation under award DMR-9400334.

the executor. (See Figure 1.) The codelet generator is not used after compile time, and the user does not need to know Caml Light or have a Caml Light compiler. FFTW provides a function that creates a plan for a transform of a specified size, and once the plan has been created it can be used as many times as needed.

The FFTW library (currently at version 1.2) is publicly available at our WWW page [6]. FFTW is *not* a toy system, but a production-quality library that already enjoys many hundreds of users. FFTW performs one- and multidimensional transforms, and it is not restricted to input sizes that are powers of 2. A parallel version of the executor, written in Cilk [7], also exists.

The rest of the paper is organized as follows. In Section 2 we outline the runtime structure of FFTW, consisting of the executor and the planner. In Section 3 we briefly describe the compile-time structure of FFTW—that is, the codelet generator. In Section 4 we present part of the performance measurements we collected during the development of FFTW. Finally, in Section 5 we give some concluding remarks.

2. FFTW'S RUNTIME STRUCTURE

In this section we describe the executor, which is the part of FFTW that actually computes the transform. We also discuss how FFTW builds a plan (a sequence of instructions that specifies the operation of the executor). Finally, we present evidence that FFTW's adaptive architecture is a good idea.

The executor implements the Cooley-Tukey FFT algorithm [3], which centers around factoring the size N of the transform into $N = N_1 N_2$. The algorithm recursively computes N_1 transforms of size N_2, multiplies the results by certain constants traditionally called *twiddle factors*, and finally computes N_2 transforms of size N_1. The executor consists of a C function that implements the algorithm just outlined, and of a library of *codelets* that implement special cases of the Cooley-Tukey algorithm. Specifically, codelets come in two flavors. *Normal* codelets compute the DFT of a fixed size, and are used as the base case for the recursion. *Twiddle* codelets are like normal codelets, but in addition they multiply their input by the twiddle factors. Twiddle codelets are used for the internal levels of the recursion. The current FFTW release contains codelets for all the integers up to 16 and all the powers of 2 up to 64, covering a wide spectrum of practical applications.

The executor takes as input the array to be transformed, and also a *plan*, which is a data structure that specifies the factorization of N as well as which codelets should be used. For example, here is a high-level description of a possible plan for a transform of length $N = 128$:

```
DIVIDE-AND-CONQUER(128, 4)
DIVIDE-AND-CONQUER(32, 8)
SOLVE(4)
```

In response to this plan, the executor initially computes 4 transforms of size 32 recursively, and then uses the twiddle codelet of size 4 to combine the results of the subproblems. In the same way, the problems of size 32 are divided into 8 problems of size 4, which are solved directly using a normal codelet (as specified by the last line of the plan) and are then combined using a size-8 twiddle codelet.

The executor works by explicit recursion, in contrast with the traditional loop-based implementations [1, page 608]. We chose an explicitly recursive implementation because of theoretical evidence that divide-and-conquer algorithms improve locality [8]. For example, as soon as a subproblem fits into the cache, no further cache misses are needed in order to solve that subproblem. We have not yet determined experimentally the relative advantages of the loop-based and recursive approaches, however. Since a codelet performs a significant amount of work, the overhead of the recursion is negligible. Moreover, recursion is easier to code and allows codelets to perform a well defined task that is independent of the context in which the codelet is used.

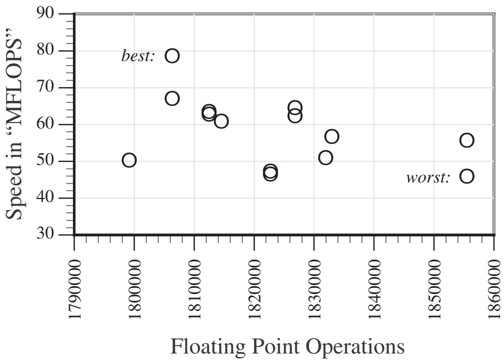

Figure 2: Speeds vs. flops of various plans considered by the planner for $N = 32768$. The units of speed ("MFLOPS") and the machine are described in Section 4. Notice that the fastest plan is not the one that performs the fewest operations.

How does one construct a good plan? FFTW's strategy is to measure the execution time of many plans and to select the best. Ideally, FFTW's *planner* should try all possible plans. This approach, however, is not practical due to the combinatorial explosion of the number of plans. Instead, the planner uses a dynamic-programming algorithm [4, chapter 16] to prune the search space. In order to use dynamic-programming, we assumed *optimal substructure* [4]: if an optimal plan for a size N is known, this plan is still optimal when size N is used as a subproblem of a larger transform. This assumption is in principle false because of the different states of the cache in the two cases. In practice, we tried both approaches and the simplifying hypothesis yielded good results.

In order to demonstrate the importance of the planner, as well as the difficulty of predicting the optimal plan, in Figure 2 we show the speed of various plans (measured and reported as in Section 4) as a function of the number of floating point operations (flops) required by each plan. (The planner computes an exact count of the operations.) There are two important phenomena that we can observe in this graph. First, different compositions of the codelets result in a wide range of performance, and it is important to choose the right combination. Second, the total number of flops is inadequate as a predictor of the execution time, at least for the relatively small variations in the flops that obtain for a given N.

We have found that the optimal plan depends heavily on the processor, the memory architecture, and the compiler. For example, for double-precision complex transforms, $N = 1024$ is factored into $1024 = 8 \cdot 8 \cdot 16$ on an UltraSPARC and into $1024 = 32 \cdot 32$ on an Alpha. We currently have no theory that predicts the optimal plan, other than some heuristic rules of the form "radix X seems to work best on machine Y."

3. THE CODELET GENERATOR

In this section we describe the codelet generator, which produces optimized fragments of C code ("codelets") specialized to compute the transform of a fixed size. In the limited space available, we shall try to give the reader a flavor of how the generator works and why such a tool is important.

```
let simplify_times = fun
  (Real a) (Real b) -> (Real (a *. b))
| (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)
```

Figure 3: Example of the rules that constitute the optimizer. The function shown in the figure simplifies the product of two factors. If both factors are real numbers, the optimizer replaces the multiplication by a single real number. Multiplications by constants can be simplified when the constant is 0, 1 or -1. The actual generator contains other rules that are not shown here.

The codelet generator accepts as input an integer N and produces a normal or a twiddle codelet that computes the Fourier transform of size N (either the forward or backward transform). The generator is written in the Caml Light dialect of ML [5]. Caml is an applicative, polymorphic, and strongly typed functional language with first-class functions, algebraic data types, and pattern matching.

The generator operates on a subset of the abstract syntax tree (AST) of the C language. First, the generator produces an AST for a naïve program that computes the transform. Then, it applies local optimizations to the AST in order to improve the program. Finally, it unparses the AST to produce the desired C code.

The AST generation phase creates a crude AST for the desired codelet. This AST contains some useless code, such as multiplications by 0 and 1, but the code is polished by the following optimization phase. The current version of the AST generator contains knowledge of many DFT algorithms, including Cooley-Tukey (in the form presented in [1, page 611]), a prime factor algorithm (as described in [1, page 619]), a split-radix algorithm [2], and Rader's algorithm for transforms of prime length [9]. Our first implementation of the Cooley-Tukey AST generator consisted of 60 lines of Caml code. The prime factor and split-radix algorithms were added using about 20 additional lines of code each. (To avoid confusion, it is worth remarking that the *codelet generator* uses a variety of algorithms for producing codelets, but the *executor* is currently only capable of composing codelets according to the Cooley-Tukey algorithm.)

The AST generator builds the syntax tree recursively. At any stage of the recursion, several algorithms are applicable, and it is not clear which one should be used. The AST generator chooses the algorithm that minimizes a certain cost function which depends on the arithmetic complexity of the codelet and its memory traffic. Experimentally, we achieved the best results by minimizing the function $4v + f$, where v is the number of stack variables generated, and f is the number of floating-point operations. (The coefficient 4 is not critical.) This choice of the cost function yielded a typical improvement of about 20% over our first generator that just implemented radix-2 Cooley-Tukey. Typically with this function, if the prime factor algorithm is not applicable, the generator tends to use a radix-$\sqrt{N}$ Cooley-Tukey algorithm.

The optimizer transforms a raw AST into an equivalent one that executes much faster. The optimizer consists of a set of rules that are applied locally to all nodes of the AST. A fragment of the optimizer appears in Figure 3. The example shows that the pattern-matching features of Caml are useful for writing the optimizer.

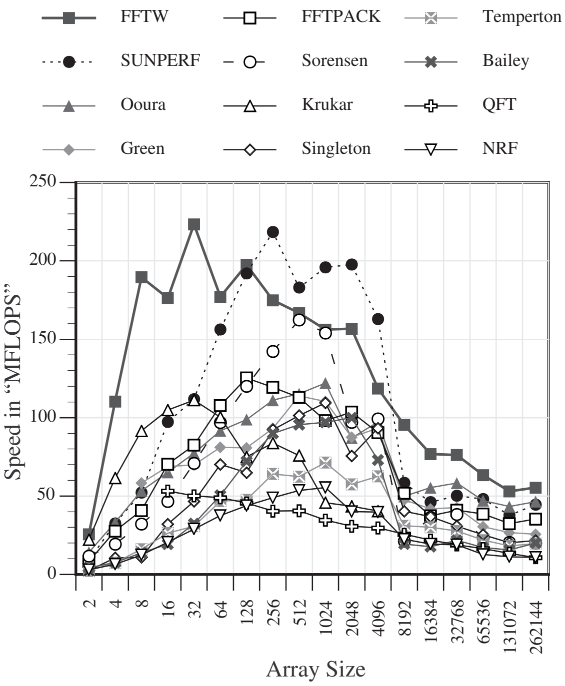

Figure 4: Comparison of double precision 1D complex FFTs on a Sun HPC 5000 (167MHz UltraSPARC-I). Compiled with `[cc|f77] -native -fast -xO5 -dalign`. SunOS 5.5.1, Sun WorkShop Compilers version 4.2.

Most simplifying rules are obvious, such as "$a + 0 \Rightarrow a$", but other rules are more subtle. For example, because of the usual trigonometric identities, a codelet contains many floating-point constant coefficients a that come paired with $-a$. We found that a rule making all constants positive, propagating the minus sign accordingly, typically yielded a speed improvement of about 10–15%, because floating-point constants are typically not part of the program code, but are loaded from memory. If the same constant appears twice in a C program, the compiler recycles the first memory load.

We believe that tools like our codelet generator will become increasingly important as processors grow more and more complex and their performance becomes, practically speaking, unpredictable. While in principle it is always possible to write an assembly program that is faster than the program generated automatically, in practice this option is seldom viable. For example, FFTW tends to use high radices (such as 32 or 64) on processors with a rich set of registers. The codelet of size 64 contains 928 additions, 248 multiplications, and 156 stack variables. Hand-coding such a subroutine correctly would be a formidable task even for the most talented programmer. Instead, not only does the generator produce the correct code automatically, but it also allows hacks such as the propagation of the minus sign to be implemented with just a couple of lines of code. Moreover, as we briefly mentioned earlier, it is not clear *a priori* which algorithm and coding style will lead to the best performance. Using the generator, however, we were able to produce code quickly and experiment with new ideas.

4. PERFORMANCE RESULTS

We have compared FFTW with over 40 other complex FFT implementations on 7 platforms, but due to space constraints we can only present a small, characteristic selection of that data here. (For more results, see [6].) In Figure 4, the performance is shown for the 8 fastest codes on an UltraSPARC, along with that of 4 other interesting or well-known programs. Speed is measured in "MFLOPS," defined for a transform of size N as $(5N \log_2 N)t$, where t is the time in μs (see [10, page 45]). The codes are listed in the legend under the author's name (or by program name if it is more well-known), and are sorted by average relative performance. They include the Sun Performance Library version 1.2 (SUNPERF); public-domain code by T. Ooura (Fortran, 1996), J. Green (C, 1996), and R. H. Krukar (C, 1990); the Fortran FFTPACK library [11]; a Fortran split-radix FFT by Sorensen [12]; a Fortran FFT by Singleton [13]; Temperton's Fortran GPFA code [14]; Bailey's "4-step" FFT implementation [15]; Sitton's QFT code [16]; and the `four1` routine from [17] (NRF).

We get similar numbers on other machines. For example, on an IBM RS/6000, FFTW ranges from 55% faster than IBM's ESSL library for $N = 64$, to 12% slower for $N = 16384$, to again 7% faster for $N = 131072$.

5. CONCLUSION

We believe computer architectures have become so complex that manually optimizing software is difficult to the point of impracticality. Our FFTW system is a method of dealing with such complexity. Similar ideas have been incorporated by other researchers [18] into an interesting system called EXTENT which uses a tensor product framework to synthesize Fortran FFTs for multiprocessors. Like FFTW, EXTENT generates code optimized for speed, but unlike FFTW, the generated program only works for one transform size. The idea of using ML as a metalanguage for generating C applications first appeared, to the best of our knowledge, in [19]. Other automatic systems for the generation of FFT programs include [20], which describes the generation of FFT programs for prime sizes. [21] presents a generator of Pascal programs implementing a prime factor FFT algorithm. Johnson and Burrus [22] applied dynamic programming to the design of optimal DFT modules. These systems all try to minimize the arithmetic complexity of the transform rather than its execution time.

Adaptive techniques such as the ones we have used appear very attractive, but much work remains to be done. The executor should be extended to use other DFT algorithms (prime factor, split-radix). Currently, the development of an FFTW-like system requires knowledge about programming languages and compilers. We plan to develop a system for program generation that could also be used by people with no specific competence in these fields.

ACKNOWLEDGEMENTS

We are grateful to SUN Microsystems Inc., which donated the cluster of 9 8-processor Ultra HPC 5000 SMPs that served as the primary platform for the development of FFTW.

Prof. Charles E. Leiserson of MIT provided continuous support and encouragement, and suggested that we use the fashionable buzzword "codelets."

6. REFERENCES

[1] A. V. Oppenheim and R. W. Schafer, *Discrete-time Signal Processing*. Englewood Cliffs, NJ 07632: Prentice-Hall, 1989.

[2] P. Duhamel and M. Vetterli, "Fast Fourier transforms: a tutorial review and a state of the art," *Signal Processing*, vol. 19, pp. 259–299, Apr. 1990.

[3] J. W. Cooley and J. W. Tukey, "An algorithm for the machine computation of the complex Fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, Apr. 1965.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, Massachusetts: The MIT Press, 1990.

[5] X. Leroy, *The Caml Light system release 0.71*. Institut National de Recherche en Informatique at Automatique (INRIA), Mar. 1996.

[6] M. Frigo and S. G. Johnson. `http://theory.lcs.mit.edu/~fftw`.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, (Santa Barbara, California), pp. 207–216, July 1995.

[8] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall, "An analysis of dag-consistent distributed shared-memory algorithms," in *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, (Padua, Italy), pp. 297–308, June 1996.

[9] C. M. Rader, "Discrete Fourier transforms when the number of data samples is prime," *Proc. of the IEEE*, vol. 56, pp. 1107–1108, June 1968.

[10] C. V. Loan, *Computational Frameworks for the Fast Fourier Transform*. Philadelphia: SIAM, 1992.

[11] P. N. Swarztrauber, "Vectorizing the FFTs," *Parallel Computations*, pp. 51–83, 1982. G. Rodrigue ed.

[12] H. V. Sorensen, M. T. Heideman, and C. S. Burrus, "On computing the split-radix FFT," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 34, pp. 152–156, Feb. 1986.

[13] R. C. Singleton, "An algorithm for computing the mixed radix fast Fourier transform," *IEEE Transactions on Audio and Electroacoustics*, vol. AU-17, pp. 93–103, June 1969.

[14] C. Temperton, "A generalized prime factor FFT algorithm for any $n = 2^p 3^q 5^r$," *SIAM Journal on Scientific and Statistical Computing*, vol. 13, pp. 676–686, May 1992.

[15] D. H. Bailey, "A high-performance FFT algorithm for vector supercomputers," *Intl. Journal on Supercomputing Applications*, vol. 2, no. 1, pp. 82–87, 1988.

[16] H. Guo, G. A. Sitton, and C. S. Burrus, "The quick discrete fourier transform," in *Proc. IEEE Int. Conf. Acoust., Speech, and Signal Proc.*, Apr. 1994.

[17] W. H. Press, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in Fortran: The Art of Scientific Computing*. New York, NY: Cambridge University Press, 1992.

[18] S. K. S. Gupta, C. Huang, P. Sadayappan, and R. W. Johnson, "A framework for generating distributed-memory parallel programs for block recursive algorithms," *Journal of Parallel and Distributed Computing*, vol. 34, pp. 137–153, 1 May 1996.

[19] S. Kamin, "Standard ML as a meta-programming language." Unpublished technical report, available from `s-kamin@uiuc.edu`, Oct. 1996.

[20] I. Selesnick and C. S. Burrus, "Automatic generation of prime length FFT programs," *IEEE Transactions on Signal Processing*, pp. 14–24, Jan. 1996.

[21] F. Perez and T. Takaoka, "A prime factor FFT algorithm implementation using a program generation technique," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 35, pp. 1221–1223, August 1987.

[22] H. W. Johnson and C. S. Burrus, "The design of optimal DFT algorithms using dynamic programming," *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. 31, pp. 378–387, Apr. 1983.