The Fastest Fourier Transform in the West

(MIT-LCS-TR-728)

Matteo Frigo[1] Steven G. Johnson[2]

Massachusetts Institute of Technology

September 11, 1997

Matteo Frigo was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant N00014-94-1-0985. 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.

This paper is a technical report of the MIT Labroratory for Computer Science MIT-LCS-TR-728

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

[2]Massachusetts Institute of Technology, 77 Massachusetts Avenue, 12-104, Cambridge, MA 02139. `stevenj@mit.edu`

Abstract

This paper describes FFTW, a portable C package for computing the one- and multidimensional complex discrete Fourier transform (DFT). FFTW is typically faster than all other publicly available DFT software, including the well-known FFTPACK and the code from Numerical Recipes. More interestingly, FFTW is competitive with or better than proprietary, highly-tuned codes such as Sun's Performance Library and IBM's ESSL library. FFTW implements the Cooley-Tukey fast Fourier transform, and is freely available on the Web at `http://theory.lcs.mit.edu/~fftw`.

Three main ideas are the keys to FFTW's performance. First, the computation of the transform is performed by an executor consisting of highly-optimized, composable blocks of C code called codelets. Second, at runtime, a planner finds an efficient way (called a 'plan') to compose the codelets. Through the planner, FFTW adapts itself to the architecture of the machine it is running on. Third, the codelets are automatically generated by a codelet generator written in the Caml Light dialect of ML. The codelet generator produces long, optimized, unreadable code, which is nevertheless easy to modify via simple changes to the generator.

Keywords: Fast Fourier transform, high performance, ML, code generation.

1 Introduction

This paper describes FFTW, a portable C package for computing the one- and multidimensional complex discrete Fourier transform (DFT). Extensive benchmarking demonstrates that FFTW is typically faster than all other publicly available DFT software, including the well-known FFTPACK [1] and the code from Numerical Recipes [2]. More interestingly, FFTW is competitive with or better than proprietary, highly-tuned codes such as Sun's Performance Library and IBM's ESSL library. FFTW is an implementation of the Cooley-Tukey [3] fast Fourier transform (FFT), and is freely available on the World Wide Web at the URL `http://theory.lcs.mit.edu/~fftw`.

Three main ideas are the keys to FFTW's performance. First, the computation of the transform is performed by an *executor* consisting of highly-optimized, composable blocks of C code called *codelets*. Second, at runtime, a *planner* finds an efficient way (called a *plan*) to compose the codelets. Through the planner, FFTW adapts itself to the architecture of the machine it is running on. In this way, FFTW is a *single* program that performs efficiently on a variety of architectures. Third, the codelets are automatically generated by a *codelet generator* written in the Caml Light dialect of ML [4]. The codelet generator produces long, optimized, unreadable code, which is nevertheless easy to modify via simple changes to the generator.

Despite its internal complexity, FFTW is easy to use. (See Figure 1.) The user interacts with FFTW only through the planner and the executor; the codelet generator is not used after compile-time. FFTW provides a function that creates a plan for a transform of a certain size. Once the user has created a plan, she can use the plan as many times as needed. FFTW is not restricted to transforms whose size is a power of 2. A parallel version of the executor, written in Cilk [5], also exists.

The executor implements the well-known Cooley-Tukey algorithm [3], which works by factoring the size N of the transform into $N = N_1 N_2$. The algorithm then recursively computes N_1 transforms of size N_2 and N_2 transforms of size N_1. The base case of the recursion is handled by the codelets, which are hard-coded transforms for various small sizes. We emphasize that the executor works by explicit recursion, in sharp contrast with the traditional loop-based implementations [6, page 608]. The recursive divide-and-conquer approach is superior on modern machines, because it exploits all levels of the memory hierarchy: as soon as a subproblem fits into cache, no further cache misses are needed in order to solve that subproblem. Our results contradict the folk theorem that recursion is slow. Moreover, the divide-and-conquer approach parallelizes naturally in Cilk.

The Cooley-Tukey algorithm allows arbitrary choices for the factors N_1 and N_2 of N. The best choice depends upon hardware details like the number of registers, latency and throughput of instructions, size and associativity of caches, structure of the processor pipeline,

```
fftw_plan plan;
int n = 1024;
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 the actual code, there are a few other parameters that specify the direction, dimensionality, and other details of the transform.

etc. Most high-performance codes are tuned for a particular set of these parameters. In contrast, FFTW is capable of optimizing itself at runtime through the planner, and therefore the *same* code can achieve good performance on many architectures. We can imagine the planner as trying all factorizations of N supported by the available codelets, measuring their execution times, and selecting the best. In practice, the number of possible plans is too large for an exhaustive search. In order to prune the search, we assume that the optimal solution for a problem of size N is still optimal when used as a subroutine of a larger problem. With this assumption, the planner can use a dynamic-programming [7, chapter 16] algorithm to find a solution that, while not optimal, is sufficiently good for practical purposes. The solution is expressed in the form of byte-code that can be interpreted by the executor with negligible overhead. Our results contradict the folk theorem that byte-code is slow.

The codelet generator produces C subroutines (codelets) that compute the transform of a given (small) size. Internally, the generator itself implements the Cooley-Tukey algorithm in symbolic arithmetic, the result of which is then simplified and unparsed to C. The simplification phase applies to the code many transformations that an experienced hacker would perform by hand. The advantage of generating codelets in this way is twofold. First, we can use much higher radices than are practical to implement manually (for example, radix-32 tends to be faster than smaller radices on RISC processors). Second, we can easily experiment with diverse optimizations and algorithmic variations. For example, it was easy to add support for prime factor FFT algorithms (see [8] and [6, page 619]) within the codelets.

A huge body of previous work on the Fourier transform exists, which we do not have space to reference properly. We limit ourselves to mention some references that are important to the present paper. A good tutorial on the FFT can be found in [9] or in classical textbooks such as [6]. Previous work exists on automatic generation of FFT programs: [10] describes the generation of FFT programs for prime sizes, and [11] presents a generator of Pascal programs implementing a Prime Factor algorithm. Johnson and Burrus [12] first applied dynamic programming to the design of optimal DFT modules. Although these papers all deal with the arithmetic complexity of the FFT, we are not aware of previous work where these techniques are used to maximize the actual performance of a program. The behavior of the FFT in the presence of nonuniform memory was first studied by [13]. Savage [14] gives an asymptotically optimal strategy for minimizing the memory traffic of the FFT under very general conditions. Our divide-and-conquer strategy is similiar in spirit to Savage's approach. The details of our implementation are asymptotically suboptimal but faster in practice. Some other theoretical evidence in support of recursive divide-and-conquer algorithms for improving locality can be found in [15].

In this short paper we do not have space to give more details about the planner and the executor. Instead, in Section 2 we present performance comparisons between FFTW and various other programs. In Section 3, we discuss the codelet generator and its optimization strategy. Finally, in Section 4 we give some concluding remarks.

2 Performance results

In this section, we present the result of benchmarking FFTW against many public-domain and a few proprietary codes. From the results of our benchmark, FFTW appears to be the fastest portable FFT implementation for almost all transform sizes. Indeed, its performance is competitive with that of the vendor-optimized Sun Performance and ESSL libraries on the UltraSPARC and the RS/6000, respectively. At the end of the section we describe our benchmark methodology.

It is impossible to include the benchmark results for all machines here. Instead, we present the data from two machines: a 167-MHz Sun UltraSPARC-I and an IBM RS/6000 Model 3BT (Figures 2 through 6). Data from other machines are similar and can be found on our web site, as well as results for transforms whose size is not a power of 2. The performance results are given as a graph of the speed of the transform in MFLOPS versus array size for both one and three dimensional transforms. The MFLOPS count is computed by postulating the number of floating point operations to be $5N \log_2 N$, where N is the number of complex values being transformed (see [16, page 23] and [17, page 45]). This metric is imprecise because it refers only to radix-2 Cooley-Tukey algorithms. Nonetheless, it allows our numbers to be compared with other results in the literature [1]. Except where otherwise noted, all benchmarks were performed in double precision. A complete listing of the FFT implementations included in the benchmark is given in Table 1.

Figure 2 shows the results on a 167MHz UltraSPARC-I. FFTW outperforms the Sun Performance Library for large transforms in double precision, although Sun's software is faster for sizes between 128 and 2048. In single precision (Figure 4) FFTW is superior over the entire range. On the RS/6000 FFTW is always comparable or faster than IBM's ESSL library, as shown in Figures 3 and 6. The high priority that was given to memory locality in FFTW's design is evident in the benchmark results for large, one-dimensional transforms, for which the cache size is exceeded. Especially dramatic is the factor of three contrast on the RS/6000 (Figure 3) between FFTW and most of the other codes (with the exception of CWP, discussed below, and ESSL, which is optimized for this machine). This trend also appeared on most of the other hardware that we benchmarked.

A notable program is the one labelled 'CWP' in the graphs, which sometimes surpasses the speed of FFTW for large transforms. Unlike all other programs we tried, CWP uses a prime-factor algorithm [18, 19] instead of the Cooley-Tukey FFT. CWP works only on a restricted set of transform sizes. Consequently, the benchmark actually times it for a transform whose size (chosen by CWP) is slightly larger than that used by the rest of the codes. We chose to include it on the graph since, for many applications, the exact size of the transform is unimportant. The reader should be aware that the point-to-point comparison of CWP with other codes may be meaningless: CWP is solving a bigger problem and, on the other hand, it is choosing a problem size it can solve efficiently.

The results of a particular benchmark run were never entirely reproducible. Usually, the differences from run to run were 5% or less, but small changes in the benchmark could produce much larger variations in performance, which proved to be very sensitive to the alignment of code and data in memory. We were able to produce changes of up to 10% in the benchmark results by playing with the data alignment (e.g. by adding small integers to the array sizes). More alarmingly, changes to a single line of code of one FFT could occasionally affect the performance of another FFT by more than 10%. The most egregious offender in this respect was one of our Pentium Pro machines running Linux 2.0.17 and the `gcc` 2.7.2 compiler. On this machine, the insertion of a single line of code into FFTW slowed down a completely unrelated FFT (CWP) by almost a factor of twenty. Consequently, we do not dare to publish any data from this machine. We do not completely understand why the performance Pentium Pro varies so dramatically. Nonetheless, on the other machines we tried, the overall trends are consistent enough to give us confidence in the qualitative results of the benchmarks.

Our benchmark program works as follows. The benchmark uses each FFT subroutine to compute the DFT many times, measures the elapsed time, and divides by the number

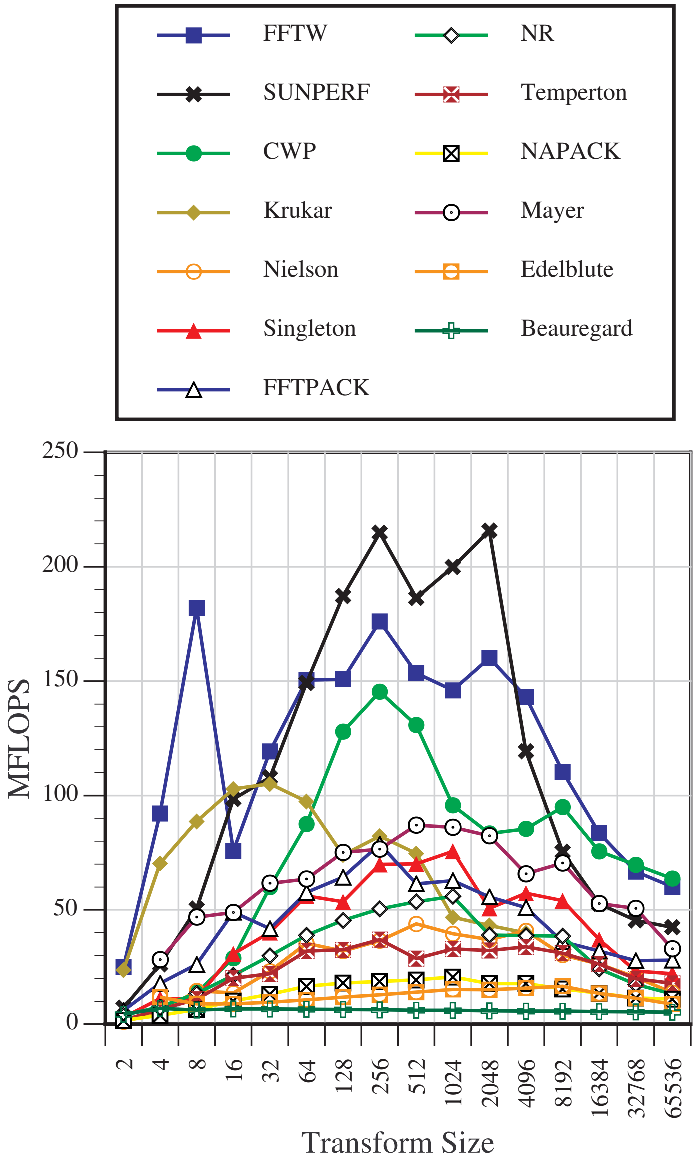

Figure 2: Comparison of 1D FFTs on a Sun HPC 5000 (167MHz UltraSPARC-I). Compiled with `cc -native -fast -xO5 -dalign`. SunOS 5.5.1, `cc` version 4.0.

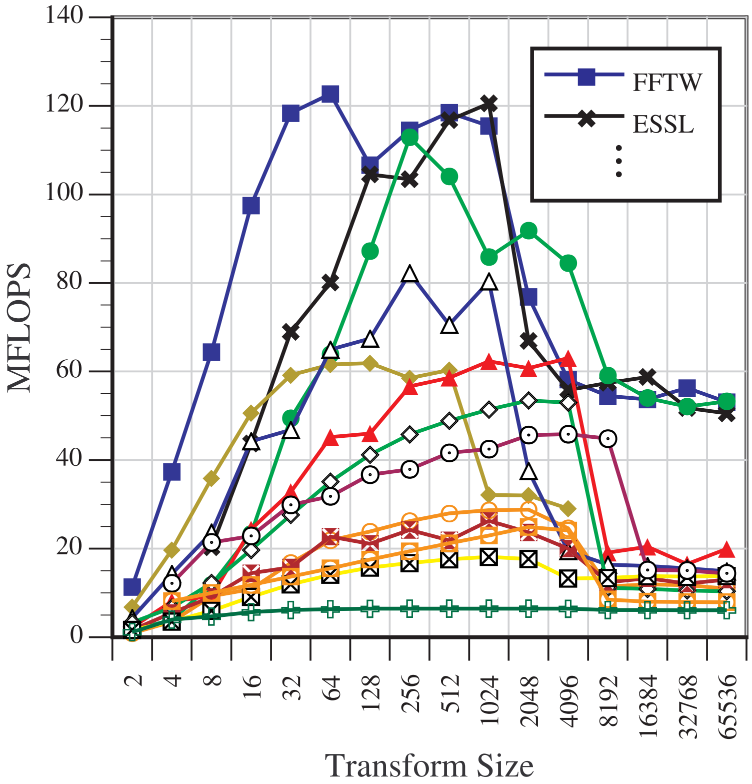

Figure 3: Comparison of 1D FFTs on an IBM RS/6000, model 3BT, running AIX version 3.2. Compiled with `cc -O3 -qarch=pwrx -qtune=pwrx`.

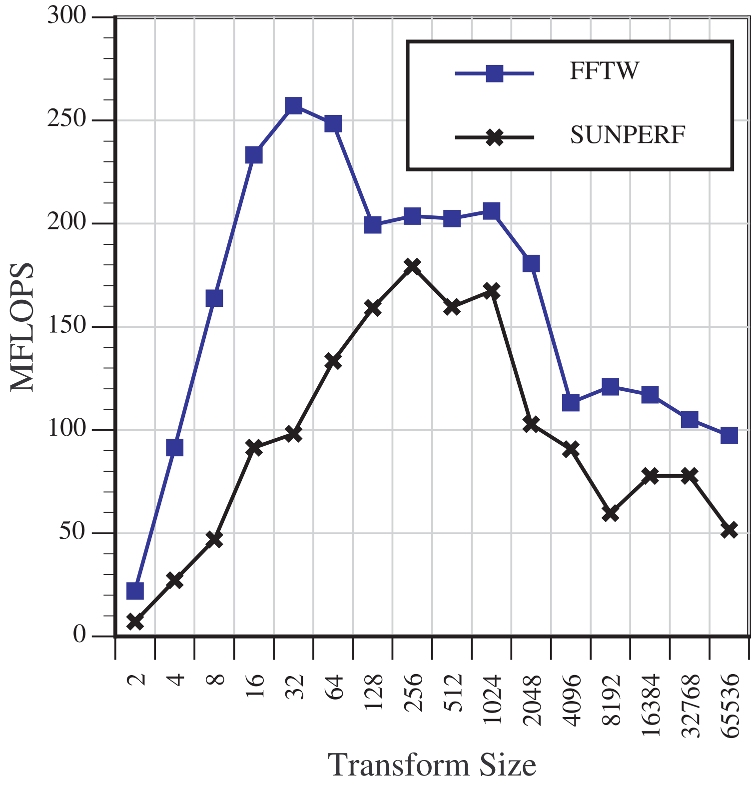

Figure 4: Comparison of FFTW with the Sun Performance Library on the UltraSPARC for single precision 1D transforms.

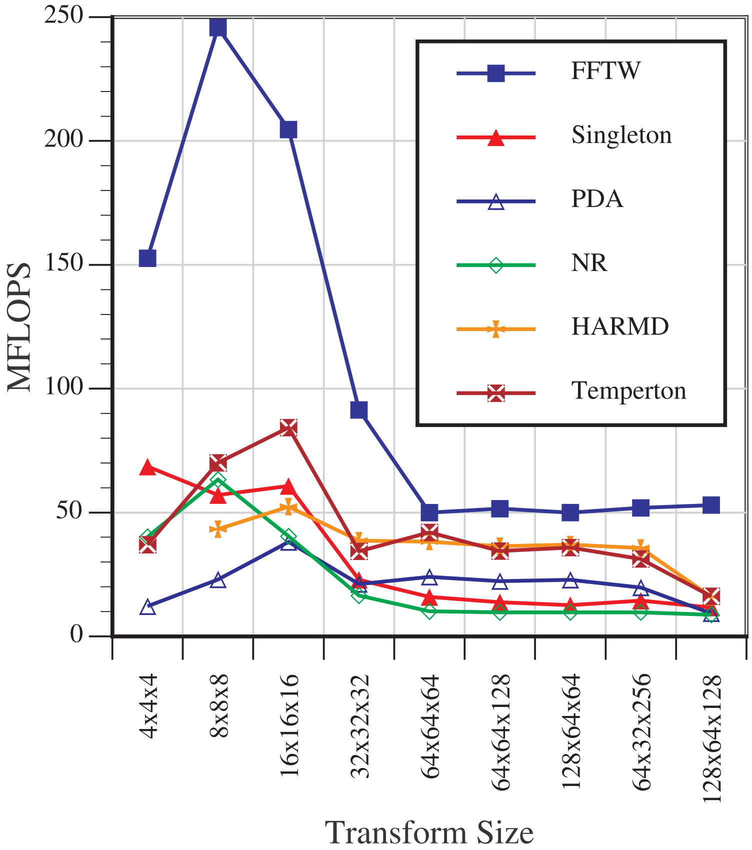

Figure 5: Comparison of 3D FFTs on the UltraSPARC.

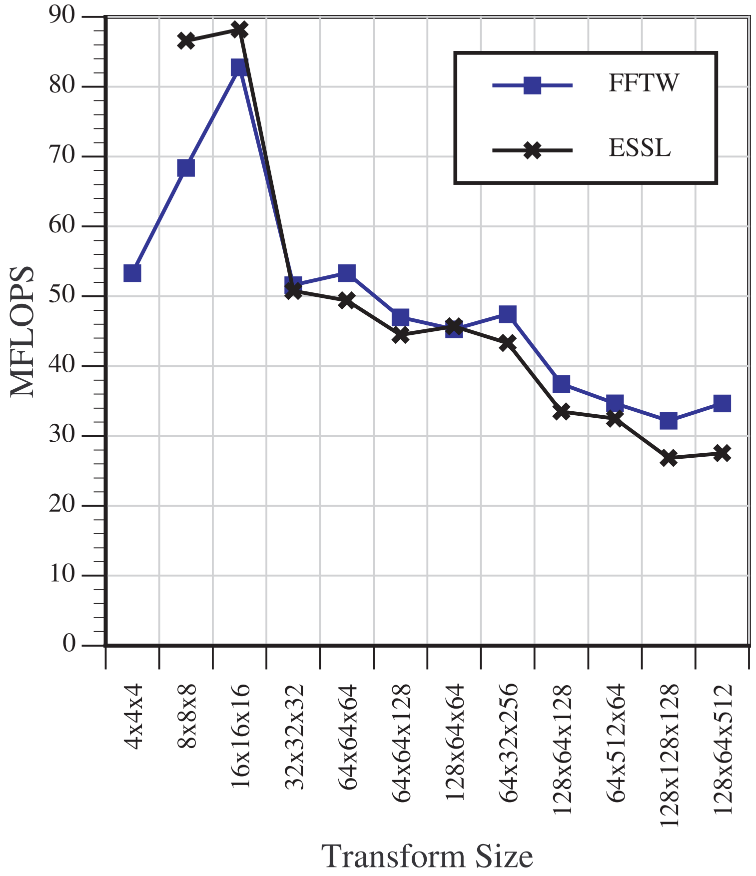

Figure 6: Comparison of 3D transforms from FFTW and the ESSL library on the RS/6000, in single precision.

FFTW	The Fastest Fourier Transform in the West
*__SUNPERF__	1D FFT from the Sun Performance Library (UltraSPARC version)
*__ESSL__	1D and 3D FFTs from IBM's ESSL library for the RS/6000. Only the single precision version of the 3D transform was available to us.
CWP	A prime-factor FFT implementation by D. Hale in a C numerical library from the Colorado School of Mines.
Krukar	1D C FFT by R. H. Krukar.
Nielson	Mixed-radix, C FFT by J. J. Nielson.
Singleton	Mixed-radix, multidimensional, Fortran FFT by R. C. Singleton [20].
FFTPACK	Fortran 1D FFT library by P. N. Swarztrauber [1].
PDA	3D FFT from the Public Domain Algorithms library. Uses FFTPACK for its 1D FFTs.
NR	C FFTs in one or more dimensions from Numerical Recipes in C [2].
Temperton	Fortran FFT in one and three dimensions by C. Temperton [21].
NAPACK	Fortran FFT from the free NAPACK library.
Mayer	1D C FFT by R. Mayer.
Edelblute	1D C FFT by D. Edelblute and R. Mayer.
Beauregard	1D C FFT by G. Beauregard.
HARMD	3D Fortran FFT, author unknown.

Table 1: Description of the programs benchmarked. All codes are generally available except for the entries marked with an asterisk, which are proprietary codes optimized for particular machines.

of iterations to get the time required for a single transform. After each FFT, however, it is necessary to reinitialize the array being transformed (iteration of the DFT is a diverging process). The time for these reinitializations is measured separately and subtracted from the elapsed time mentioned above.

Instead of reinitializing the input array after each transform, one could alternatively follow the transform by the inverse transform in every iteration. Many FFT implementations compute an unnormalized DFT, however, and thus it would be necessary to have an additional loop to scale the data properly. We did not want to measure this additional cost, since in many applications the scale factor can easily be absorbed into other multiplicative factors to avoid the extra multiplications and loads.

It was our intention to benchmark C FFTs, but much of the public-domain software was written in Fortran. These codes were converted to C via the free `f2c` software [22]. This raises some legitimate concerns about the quality of the automatic translation performed by `f2c`, as well as the relative quality of Fortran and C compilers. Accordingly, we compared the original Fortran FFTPACK with the `f2c` version. On average, the C code was 16% slower than the Fortran version on the RS/6000 and 27% slower on the UltraSPARC. The Fortran code was never faster than FFTW. We will give more results for native Fortran software in the final paper.

3 The codelet generator

In this section we describe the codelet generator, that produces optimized C codelets. It is written in Caml Light [4] because it is easy to express symbolic manipulations in that language. Because of this automatic generation process, it is easy to produce and experiment with long straight-line code. The generator performs many optimizations such as constant folding and minus-sign propagation.

The codelet generator accepts as input an integer n, and produces a fragment of C code that computes the Fourier transform of size n (a *codelet*). Depending on the options with which the generator is invoked, the codelet computes either the forward or the backward transform, and can optionally multiply its input by a precomputed set of *twiddle factors*. The codelet generator is written in the Caml Light dialect of ML [4]. Caml is an applicative, polymorphic, and strongly-typed functional language with first-class functions, algebraic data types, and pattern matching.

The codelet generator operates on a subset of C's abstract syntax tree (AST). It first produces an AST for a naïve program that computes the transform, and then applies local optimizations to the AST in order to improve the program. Finally, it unparses the AST to produce the desired C code. The interface between the first phase (generation of the AST) and

```
let fftgen_prime N input output =
    let expr k = (Sigma 0 N (fun n ->
            let coeff = CExp(k * n / N)
            in CTimes coeff (input n)))
    and FFT =
        forall 0 N (fun k ->
            (output k (expr k)))
    in  FFT ;;
```

Figure 7: Caml program that generates the AST of a transform of size N, when N is prime. The kth output (denoted by `(expr k)` in the program) is the sum (`Sigma`) for n ranging from 0 to $N-1$ of the nth input multiplied by a certain coefficient (`coeff`). The AST `FFT` contains a program that outputs the kth element of the transform, for all k ranging from 0 to $N-1$. The operator `forall` concatenates many ASTs into one.

the following phases is such that the AST can be expressed in terms of complex arithmetic, and the reduction to an AST that uses only real arithmetic is performed automatically.

In the rest of the section, we shall describe the AST generation phase and the optimizer. The unparser is rather uninteresting, and we will not bother to describe it.

The AST generation phase creates a crude AST for the desired codelet. This AST contains a lot of useless code, such as multiplications by 0 and 1, but the code is polished by the following optimization phase. The AST generator uses the Cooley-Tukey algorithm [3] in the form presented by [6, page 611]. We assume that the reader is familiar with this well-known algorithm. The actual implementation of the AST generator consists of about 60 lines of code. With 20 additional lines of code our generator can also produce an AST for the Prime Factor algorithm [8] as described in [6, page 619].

Recall that the Cooley-Tukey algorithm reduces a transform of size $N = N_1 N_2$ to N_1 transforms of size N_2, followed by some multiplications by certain complex constants called *twiddle factors*, followed by N_2 transforms of size N_1. If N is prime, the algorithm computes the transform directly according to the definition. The AST generator is an almost literal transcription of this algorithm. It consists of a recursive function `genfft` that takes three arguments: the size `N` of the transform and two functions `input` and `output`. When applied to an integer `n`, the `input` function returns a complex expression that contains the nth input value. (A complex expression is a pair of ASTs representing the real and imaginary parts of the input in symbolic form.) Similarly, the function `output` can be applied to two arguments `k` and `x` and returns an AST that stores the expression x into the kth output variable. Because of lack of space, we do not show the recursive part of `fftgen`, but we do show the base case of the recursion in Figure 7.

All factors N_1 and N_2 of N may seem equivalent, but some factors are more equivalent than others. One natural choice is to let N_1 be the smallest prime factor of N, but it turns out that if N_1 is a factor of N close to $\sqrt{N}$ the resulting codelet is faster. One reason is that the codelet performs fewer arithmetic operations (although we do not fully understand why). Another reason is that the codelet is more likely to take advantage of the large register sets of modern superscalar processors, as we shall now illustrate with an example. Suppose that a transform of size 16 is desired, but the processor can only compute a transform of size 4 using internal registers. If we choose $N_1 = N_2 = 4$, than the processor can load the input once from memory, compute 4 transforms of size 4 storing the result back into memory, and then do the same thing again. In total, the input is read twice. It is easy to see that if we let $N_1 = 2$, we force the processor to read the input array three times. Within our codelet generator, this trick could be implemented in just a few minutes.

We now describe the optimizer. The goal of the optimizer is to transform 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 an AST. Most of the rules are pretty obvious, such as "$a + 0 \Rightarrow a$" and the like, but some rules are far more subtle. We now give an example of how the rules are implemented in the actual codelet generator, and then we discuss some of the more subtle rules that we found useful.

Figure 8 shows a fragment of the actual implementation of the optimizer. The pattern-matching features of Caml Light turned out to be particularly useful for this purpose. By looking at the example, the reader can convince herself that a sufficiently powerful optimizer can be implemented quite easily [23, page 108].

By playing with the optimizer we found some interesting rules to make the codelets faster. Consider for example the two fragments of code in Figure 9. At first glance, it appears that the two fragments should perform comparably. After all, both contain the same number and type of arithmetic operations, and in the same order (subtractions and additions are performed by the same hardware, and are thus equivalent when talking about performance). The fragment on the right executes faster on all processors we have tried, however. The reason is that floating-point constants are not created out of thin air, but must be stored somewhere in memory. The fragment on the right loads the constant 0.5 only once, while the code on the left must load both 0.5 and -0.5 from memory. As a rule, the optimizer makes all constants positive and propagates the minus sign to the rest of the AST. We found that this rule typically yielded a speed improvement of about 10–15%.

Another interesting result that arose from our investigations is shown in Figure 10. Conventional wisdom [24, page 84] dictates that the common subexpression `c + d` be precomputed and stored into a temporary variable, as shown in the right part of the figure. On the contrary, we found that this transformation does not produce faster code on present-day

```
let simplify_times = fun
  (Real a) (Real b) -> (Real (a *. b))
| a (Real b) ->
     simplify_times (Real b) a
| (Uminus a) b ->
     simplify (Uminus (Times (a,b)))
| a (Uminus b) ->
     simplify (Uminus (Times (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)
| a b -> Times (a, b)
```

Figure 8: 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. Minus signs are propagated up, so that another set of rules (not shown) can collapse two consecutive minus signs. Multiplications by constants can be simplified when the constant is 0, 1 or -1.

compilers. Indeed, in some cases we found that the elimination of the common subexpression produced *slower* code. The reason for this behavior is not clear. From our understanding at this point, a C compiler may unnecessarily waste registers when temporary variables are declared explicitly.

4 Conclusions

This paper described the design and the performance of FFTW, a self-optimizing library for computing the one- and multidimensional complex discrete Fourier transform.

The current version of FFTW extends the program described in this paper in several directions. We have written three parallel versions, using Cilk [5], Posix threads [25] and MPI [26]. We also support multidimensional real-complex transforms. FFTW has now a mechanism to save plans to disk, and can use fragments of plans in order to reduce the planning time.

```
a = 0.5 * b;                    a = 0.5 * b;
c = -0.5 * d;                   c = 0.5 * d;
e = 1.0 + a;                    e = 1.0 + a;
f = 1.0 + c;                    f = 1.0 - c;
```

Figure 9: Two fragments of C code containing the same number and type of arithmetic operations, in the same order. Nonetheless, the fragment on the right is faster. See the text for an explanation.

```
                                {
                                  double tmp =
a = b + (c + d);                     c + d;
e = f + (c + d);                  a = b + tmp;
                                  e = f + tmp;
                                }
```

Figure 10: Two equivalent fragments of C code; the fragment on the right explicitly stores the common subexpression into a temporary variable. We found that, on modern compilers, the fragment on the left is not slower than the one in the right, and in some cases it is faster.

FFTW has enjoyed excellent acceptance in the Internet community. It was downloaded by more than 600 users in the first month after its release, many of whom have reported significant speedups in their applications. It has continued to gain users, and is now part of the `netlib` repository of scientific software. FFTW has been adopted in the FFT component of the Ptolemy project [27], a software environment for signal processing and simulation. In addition, the VSIP (Vector/Signal/Image Processing Forum) committee is discussing the possibility of incorporating FFTW into the VSIP reference implementation as an example of how to use FFTs that have an optimize/initialization phase before first use. Their goal is to define an industry-standard API for vector, signal, and image processing primitives for embedded real-time signal processing systems.

5 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. Charles also proposed the name 'codelets' and is responsible for $\Omega(n \log n)$ of the commas, that appear in this paper.

References

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

[2] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*. New York, NY: Cambridge University Press, 2nd ed., 1992.

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

[5] 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.

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

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

[8] I. J. Good, "The interaction algorithm and practical Fourier analysis," *J. Roy. Statist. Soc.*, vol. B 20, pp. 361–372, 1958.

[9] 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.

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

[11] 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.

[12] 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.

[13] J.-W. Hong and H. T. Kung, "I/O complexity: the red-blue pebbling game," in *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, (Milwaukee), pp. 326–333, 1981.

[14] J. E. Savage, "Space-time tradeoffs in memory hierarchies," Tech. Rep. CS 93-08, Brown University, CS Dept., Providence, RI 02912, October 1993.

[15] 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.

[16] J. W. Cooley, P. A. W. Lewis, and P. D. Welch, "The Fast Fourier Transform algorithm and its applications," *IBM Research*, 1967.

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

[18] C. Temperton, "Implementation of a self-sorting in-place prime factor FFT algorithm," *Journal of Computational Physics*, vol. 58, pp. 283–299, May 1985.

[19] C. Temperton, "A new set of minimum-add small-n rotated DFT modules," *Journal of Computational Physics*, vol. 75, pp. 190–198, 1988.

[20] 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.

[21] 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.

[22] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, "A Fortran to C converter," Tech. Rep. 149, AT&T Bell Laboratories, 1995.

[23] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press, 1985.

[24] J. L. Bentley, *Writing Efficient Programs*. Englewood Cliffs, NJ 07632: Prentice-Hall Software Series, 1982.

[25] D. R. Butenhof, *Programming with POSIX threads*. Addison-Wesley, 1997.

[26] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference*. MIT Press, 1995.

[27] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *Int. Journal of Computer Simulation*, vol. 4, pp. 155–182, Apr. 1994.