A High-Performance Fast Fourier Transform Algorithm for the Cray-2

David H. Bailey

July 10, 1986

Ref: *Journal of Supercomputing*, vol. 1, no. 1 (July 1987), pg. 43–60

Abstract

Most implementations of a radix-2 fast Fourier transform on large scientific computers use algorithms that involve memory accesses whose strides are powers of two. (The term *stride* means the memory increment between successive elements stored or fetched). Such strides are unacceptable for recently developed supercomputers, particularly the Cray-2, because of serious difficulties with memory bank conflicts.

This paper describes an algorithm for evaluating the fast Fourier transform that avoids this difficulty and thus could provide the basis for implementations that more fully utilize the power of the Cray-2. A Fortran program implementing this algorithm is included, and timing comparisons with the Cray assembly-coded library subroutine are shown.

The author is an employee of Sterling Software, Inc., under contract to the Numerical Aerodynamic Simulation Projects Office at NASA Ames Research Center. This work was performed under contract NAS2-11555.

Introduction

A "fast Fourier transform" (FFT) is merely a computationally efficient technique to evaluate the discrete Fourier transform (DFT). The DFT and the inverse DFT of the n-long sequence $x = (x_0, x_1, x_2, \cdots, x_{n-1})$ are mathematically defined as

$$\begin{aligned} F_j(x) &= \sum_{k=0}^{n-1} x_k e^{-2\pi ijk/n}, \qquad 0 \le j < n \\ F_j^{-1}(x) &= \frac{1}{n}\sum_{k=0}^{n-1} x_k e^{2\pi ijk/n}, \qquad 0 \le j < n \end{aligned}$$

The second formula is said to be the inverse DFT because it can be easily shown that performing the first operation followed by the second operation recovers the original sequence. Suppose n can be factored as $n = lm$. Write $j = pm + q$ and $k = rl + s$, where p and s range from 0 to $l-1$ and where q and r range from 0 to $m-1$. Then the formula for the DFT may be written as

$$\begin{aligned} F_{pm+q}(x) &= \sum_{s=0}^{l-1}\sum_{r=0}^{m-1} x_{rl+s} e^{-2\pi i(pm+q)(rl+s)/n} \\ &= \sum_{s=0}^{l-1}\sum_{r=0}^{m-1} x_{rl+s} e^{-2\pi ipms/n} e^{-2\pi iqrl/n} e^{-2\pi iqs/n} \\ &= \sum_{s=0}^{l-1}\sum_{r=0}^{m-1} x_{rl+s} e^{-2\pi ips/l} e^{-2\pi iqr/m} e^{-2\pi iqs/n} \\ &= \sum_{s=0}^{l-1}\left[\sum_{r=0}^{m-1} (x_{rl+s} e^{-2\pi iqs/n}) e^{-2\pi iqr/m}\right] e^{-2\pi ips/l}, \qquad 0 \le p < l, \quad 0 \le q < m \end{aligned}$$

Thus the computation of an n-point DFT has been reduced to performing a series of complex multiplications, l-point DFTs and m-point DFTs. By repeating this process, an n-point DFT can be reduced to a series of complex multiplications and p-point DFTs for various prime numbers p.

The Radix-2 FFT

The most frequently used application of the FFT technique is for values of n that are powers of two. Therefore in the following four sections of this paper it will be assumed that $n = 2^m$. In this case the above technique yields the following algorithm for evaluating the radix-2 DFT. Set $X_0(j) = x_j, \quad 0 \le j < n$. Then iterate the following calculations for t from 1 to m:

$$\begin{aligned} X_t(j2^t + k) &= X_{t-1}(j2^{t-1} + k) + \alpha_t^k X_{t-1}(j2^{t-1} + n/2 + k) \\ X_t(j2^t + 2^{t-1} + k) &= X_{t-1}(j2^{t-1} + k) - \alpha_t^k X_{t-1}(j2^{t-1} + n/2 + k) \\ \text{for} \quad 0 \le j < 2^{m-t}, &\qquad 0 \le k < 2^{t-1} \end{aligned}$$

where $\alpha_t = e^{-2\pi i/2^t}$. The final results $X_m(k),\ 0 \le k < n$ are then equal to the $F_k(x)$ in the original formula above for the DFT. This algorithm is known as the Stockham formulation of the FFT.

Note that the straightforward evaluation of the original formula for the DFT requires $8n^2$ operations, even if it is assumed that all powers of $\alpha = e^{-2\pi i/n}$ have been precalculated. The Stockham FFT algorithm produces the same result array in only $5n \log_2 n = 5m2^m$ arithmetic operations. For large n this work factor represents a huge savings over the straightforward algorithm. Thus it is not surprising that this and similar FFT techniques have been applied in a wide variety of numerically intensive fields, such as digital signal processing and computational fluid dynamics.

In order to conserve memory, many computer programs implementing the fast Fourier transform modify the above algorithm by overwriting the results of the right hand sides of the FFT equations into the same array. If this is done with the proper array indices, the result data values are correct, but a "bit reversal" permutation is necessary. This means that element $F_k(x)$ is found in $X_m(k')$, where k' is the index whose binary expansion is reversed from that of k (i.e., if $n = 64$ and $k = 19 = 010011_2$, then $k' = 50 = 110010_2$).

Many implementations of the FFT also precalculate the powers of α in a one-time initialization before the FFT is called with any data. Typically these powers are placed in an array U such that $U(k) = \alpha^k,\ 0 \le k < n/2$. Then the expression α_t^k is evaluated by merely recalling $U(k2^{m-t})$ from memory.

The most common procedure to perform one iteration of the FFT above is to increment j in the inner loop and to increment k in the outer loop. For most computers this ordering of the loops is faster because the fetch of the power of α from U may be done in the outer loop, since it does not depend on j.

Implementation of the FFT Algorithm on the Cray-2

The Cray-2 is the first of a new generation of vector supercomputers that feature very large main memories, in addition to speeds that equal or exceed previous supercomputers. The Cray-2 that has now been installed at NASA Ames Research Center features four central processing units (CPUs), together with over 268 million 64-bit words of main memory. This amount of main memory is greater than the combined main memories of all previously delivered Cray supercomputers. Each CPU in the Cray-2 has eight 64-word vector registers plus an even richer set of vector instructions than the Cray X-MP computers. The clock period of a Cray-2 CPU is only 4.1 nanoseconds, as compared to 9.5 nanoseconds for the X-MP line, so each Cray-2 CPU is potentially twice as fast as on the Cray X-MP. However, in practice such speedups are difficult to attain due to memory speed limitations.

For large memory computers like the Cray-2, conserving memory is of much lower priority than making sure that the full computational speed of the computer is being utilized. Thus it appears pointless to attempt to perform an FFT in place (i.e., by overwriting the input array as described above). In fact, the Cray library FFT subroutine available for the Cray X-MP computers, which have much more limited main memory than the Cray-2, has for years utilized a separate array for the left-hand-side at each iteration to avoid the need for a bit reversal permutation. It is certainly clear that any truly high-performance FFT for the Cray-2 needs to use a separate array to avoid the cost of a bit reversal permutation.

The most significant performance issue for any program to be run on the Cray-2 is to insure that memory bank conflicts are avoided. The main memory of the Cray-2 is organized into 128 interleaved memory banks. In this way a vector memory fetch with unit stride can return one word of memory every CPU clock period after the initial startup delay (provided there is no conflict with memory accesses by other CPUs). Although the Cray-2 has a large number of memory banks, the speed of the main memory circuitry is significantly slower than its CPU. As a result, the fetch of a single word of memory requires over 50 CPU clock periods, and the memory bank containing the word is reserved for most of this period of time. Thus, if a vector memory fetch has a stride that is a multiple of 128, then every element fetched resides on the same bank, and the overall speed of the vector fetch drops by a factor of nearly 50. Even if the stride is only two or four, the speed of the vector fetch is sharply reduced due to section conflicts, a consequence of the organization of memory into four sections. Thus it is imperative that power-of-two memory strides be avoided on the Cray-2, especially in a heavily used routine such as an FFT. The same conclusions hold for strides that are divisible by powers of two.

Power-of-two memory strides result in performance degradations on other systems as well. For instance, a number of systems are designed for primarily unit stride memory operations, and any nonunit stride results in a performance reduction. This is true, for instance, on the Cyber 205, the Fujitsu VP-400, and the NEC SX-2. In addition, all of these systems suffer additional performance reductions due to bank conflicts for memory strides divisible by large powers of two, although the reductions are not as dramatic as on the Cray-2.

Another impetus for finding an algorithm that avoids nonunit memory strides is that analyses of the Cray-2 indicate that it may be necessary to utilize local memory to fully tap the high levels of performance (over 250 million floating-point operations per second on a single CPU) that the machine is capable of attaining. Local memory is a register cache of 16,384 words contained within each CPU. Because of its fast access time, algorithms that effectively utilize it avoid the main memory speed limitations and may run significantly faster. Unfortunately, the hardware design of the Cray-2 CPU only allows for stride one vector access to local memory. Thus it is likely that a very high performance FFT on the Cray-2 will need to employ a stride one algorithm.

An FFT Without Power-of-Two Memory Strides

Unfortunately, the radix-2 FFT algorithm described above is riddled with power-of-two memory strides. In fact, in the usual implementation of this FFT, in which the inner loop increments j and the root of unity α_t^k is a loop constant, the strides of the array stores and fetches in the inner loop are 2^t and 2^{t-1}, respectively, where t is the iteration number. Thus the stride of *every* array store is a power of two, and except for the first iteration, the stride of *every* array fetch is a power of two. Furthermore, for FFTs with m greater than 12, *most* memory accesses have a stride that is a multiple of 128, the worst possible case for bank contention. Other formulations of the radix-2 FFT have similar difficulties with power-of-two strides. Clearly this situation is unacceptable for computers such as the Cray-2.

One way to avoid this power-of-two stride problem is to employ a radix-3 or radix-5 FFT algorithm. In this way all strides are odd numbers, and bank conflicts are completely avoided. However, this solution requires a drastic revision of many existing computer programs, the vast majority of which utilize data arrays whose dimensions are powers of two. Hockney and Jesshope [4, p. 309] suggest that the stride problem be handled by inserting "spacers" in the data array. While such a technique may be acceptable for a special-purpose program, it is not a practical alternative for a general-purpose library FFT, since every Fortran calling program has to employ the same complicated indexing scheme. Fornberg [3] describes a unit stride algorithm for the CDC 205, but this algorithm is not suitable for the Cray-2 since it relies on long vector compress-expand operations.

The stride problem may be solved in many applications by performing simultaneous FFT operations on a two-dimensional array of data. In this manner, the stride of vector accesses is either unity or the first dimension of the array, depending on which dimension the FFTs act upon. Unfortunately, not all applications permit simultaneous FFTs — some require very large single FFTs. Furthermore, many application codes that do permit simultaneous FFTs have power-of-two array dimensions, and simultaneous FFTs along the second dimension result in bank conflicts just as serious as in a one-dimensional FFT.

A preferable method of dealing with the stride difficulties mentioned above is to perform the FFT as described above, except with the inner loops reversed, so that k is incremented in the innermost loop. However, even in this case a power-of-two memory stride remains. This is because the array U containing roots of unity must now be accessed in the inner loop, and the stride of this access is 2^{m-t}, which is very large in the initial iterations of the FFT. However, this ordering of the computation does have the advantage that the strides of the four other array accesses are all unity (provided that complex data is stored so that the real and imaginary parts are separated and not interleaved as is the usual custom).

This analysis suggests that the second ordering of the inner loops is superior for the Cray-2 and similar computers provided the problem of the stride in the array U of roots of unity can be solved. To that end, it is proposed that these powers of α be stored in a different manner than is the usual custom. Instead of storing all the powers of α from 0 up to $n/2 - 1$ in U and indexing U with power-of-two strides, it is proposed that the powers of α needed at each iteration be stored in a separate section of the array, as follows:

$$\begin{array}{lcl}
U(1) & = & 1 \\
U(2),\ U(3) & = & 1,\ i \\
U(4),\ U(5),\ U(6),\ U(7) & = & 1,\ \sqrt{2}+i\sqrt{2},\ i,\ -\sqrt{2}+i\sqrt{2} \\
\cdot & & \\
\cdot & & \\
\cdot & & \\
U(2^{t-1}),\ U(2^{t-1}+1),\ \cdots,\ U(2^t-1) & = & 1,\ \alpha_t,\ \alpha_t^2,\ \cdots,\ \alpha_t^{2^{t-1}-1} \\
\cdot & & \\
\cdot & & \\
\cdot & & \\
U(2^{m-1}),\ U(2^{m-1}+1),\ \cdots,\ U(2^m-1) & = & 1,\ \alpha_m,\ \alpha_m^2,\ \cdots,\ \alpha_m^{2^{m-1}-1}
\end{array}$$

Clearly such a storage scheme wastes some memory, since each section includes as a subset the elements of all previous sections. However, the total storage required for the U array with this scheme is only $2^m - 1$ complex cells, which is only twice the amount normally required. Since the Cray-2 has ample memory this is not a significant problem.

Once the array U is filled in this manner (with real and imaginary parts separated), then all vector stores and fetches in the innermost loop have unit stride, and the operations suffer no delays due to bank or section conflicts. The only factor impeding the performance of this scheme is that the vector length of the inner loop is 2^t, which is very short for the first few iterations. For an implementation using local memory, this is not a serious issue, because startup times for local memory fetches are quite short. However, for an implementation using main memory this scheme is not optimal because the very short vector lengths in the first few iterations are just as deleterious to performance as are power-of-two memory strides.

A Mixed Strategy Algorithm

In the main memory case a slight modification of this scheme is somewhat faster. Recall that in the first (and most common) ordering of the inner loops, the strides of memory accesses during the first iteration are one and two, respectively, and the vector length is large ($n/2$). In the second iteration, the strides of memory accesses are two and four, which results is some slowdown on the Cray-2, but the vector length is still fairly large ($n/4$). These facts suggest an alternative procedure for evaluating the FFT — use the first ordering scheme for the first few iterations, and then use the second ordering scheme for all additional iterations. In this way, catastrophically short vector lengths and catastrophically bad strides are both completely avoided.

A Radix-4 Enhancement

The performance of a power-of-two FFT can be further enhanced on most state-of-the-art supercomputers by extending the basic FFT algorithm to the "radix-4" version. This algorithm is as follows. Let $n = 4^m$. Set $X_0(j) = x_j, \quad 0 \le j < n$. Then iterate the following calculations for t from 1 to m:

$$\begin{aligned}
c_0 &= X_{t-1}(j4^{t-1} + k) \\
c_1 &= \beta_t^k X_{t-1}(j4^{t-1} + n/4 + k) \\
c_2 &= \beta_t^{2k} X_{t-1}(j4^{t-1} + 2n/4 + k) \\
c_3 &= \beta_t^{3k} X_{t-1}(j4^{t-1} + 3n/4 + k) \\
d_0 &= c_0 + c_2 \\
d_1 &= c_0 - c_2 \\
d_2 &= c_1 + c_3 \\
d_3 &= i(c_1 - c_3) \\
X_t(j4^t + k) &= d_0 + d_2 \\
X_t(j4^t + 4^{t-1} + k) &= d_1 + d_3 \\
X_t(j4^t + 2 \cdot 4^{t-1} + k) &= d_0 - d_2 \\
X_t(j4^t + 3 \cdot 4^{t-1} + k) &= d_1 - d_3 \\
\text{for} \quad 0 \le j < 4^{m-t}, & \quad\quad 0 \le k < 4^{t-1}
\end{aligned}$$

where $\beta_t = e^{-2\pi i/4^t}$. The intermediate calculations have been spelled out in detail to make completely explicit some time-savings factorizations that permit efficient computation.

The main advantage in utilizing a radix-4 algorithm for a power-of-two FFT is that many supercomputers, including the Cray-2, function with vector registers within the CPU that are significantly faster in operation than the main memory circuitry. Thus it follows that maximum performance is obtained by minimizing the amount main memory fetches and stores for a given amount of computation. This radix-4 algorithm in effect performs a pair of radix-2 iterations with the same number of main memory operations, and thus the performance is improved. In the case of the Cray-2, which has largest ratio of CPU speed to memory speed of any current supercomputer, this improvement is significant.

If the radix-4 algorithm is performed by incrementing k in the inner loop, then all strides are unity as above. It only remains to precalculate the array U of roots of unity in a manner entirely analogous to the radix-2 case:

$$
\begin{array}{lcl}
U[1] & = & 1 \\
U[2],\ U[3],\ U[4],\ U[5] & = & 1,\ \sqrt{2}+i\sqrt{2},\ i,\ -\sqrt{2}+i\sqrt{2} \\
\quad\cdot & & \\
\quad\cdot & & \\
\quad\cdot & & \\
U[(4^{t-1}-1)/3+1],\ U[(4^{t-1}-1)/3+2], & & \\
\qquad\qquad \cdots,\ U[(4^t-1)/3] & = & 1,\ \beta_t,\ \beta_t^2,\ \cdots,\ \beta_t^{4^{t-1}-1} \\
\quad\cdot & & \\
\quad\cdot & & \\
\quad\cdot & & \\
U[(4^{m-1}-1)/3+1],\ U[(4^{m-1}-1)/3+2], & & \\
\qquad\qquad \cdots,\ U[(4^m-1)/3] & = & 1,\ \beta_m,\ \beta_m^2,\ \cdots,\ \beta_m^{4^{m-1}-1}
\end{array}
$$

Note that the storage requirement for the array U is even less than for the radix-2 algorithm – only $(4^m - 1)/3$ complex cells are required, which is only one third of the radix-2 requirement for a transform of comparable size.

Fortran Implementation

The appendix contains the listing of a Fortran subroutine that implements the FFT using a radix-4 mixed strategy algorithm as described above. Table 1 compares the performance of this Fortran subroutine with the assembly-coded, optimized library subroutine provided by Cray Research, Inc. The column headed "Order" gives the value of m, where $n = 2^m$ is the length of the transform. The next four columns contain the average CPU time in seconds and the average root-mean-square (RMS) error for ten trials of the two FFT programs. The errors are measured by performing ordinary FFTs followed by inverse FFTs and comparing the results with the original data. The last two columns give the ratios of the CPU times and the RMS errors, with ratios greater than one indicating faster performance and smaller errors by the new algorithm.

Conclusions

The table results indicate that the Fortran implementation of this algorithm is as much as 31% faster than the assembly-coded Cray library subroutine. Presumably an assembly-coded version of the new algorithm would compare even more favorably, especially for smaller FFT sizes, where the Fortran subroutine linkage significantly increases the run time. Further improvement might be possible by employing a radix-8 algorithm. The RMS error statistics in the table reveal another, completely unanticipated advantage of the new program: it is significantly more accurate than the library version. In one case it is 100 times more precise.

A side benefit of this new algorithm is that once the array U has been initialized with a certain value of the order m, then the subroutine may be called to perform an FFT for any order less than or equal to this value. In contrast, Cray's library subroutine requires

	New Algorithm		Cray Library		Ratio	
Order	CPU Time	RMS Error	CPU Time	RMS Error	Time	Error
6	0.000085	3.9705×10^{-15}	0.000060	2.7308×10^{-15}	0.7090	0.6878
7	0.000122	4.8336×10^{-15}	0.000114	3.8524×10^{-15}	0.9365	0.7970
8	0.000202	5.8956×10^{-15}	0.000240	5.2191×10^{-15}	1.1895	0.8853
9	0.000418	6.1882×10^{-15}	0.000504	5.4665×10^{-15}	1.2049	0.8834
10	0.000828	6.8713×10^{-15}	0.001041	6.5606×10^{-15}	1.2571	0.9548
11	0.001633	7.0026×10^{-15}	0.002142	1.0791×10^{-14}	1.3118	1.5410
12	0.003400	7.6461×10^{-15}	0.004400	1.2639×10^{-14}	1.2942	1.6530
13	0.007523	7.8237×10^{-15}	0.009011	3.5257×10^{-14}	1.1978	4.5064
14	0.015750	8.4909×10^{-15}	0.018667	5.4208×10^{-14}	1.1852	6.3842
15	0.031960	8.5684×10^{-15}	0.037816	2.0667×10^{-13}	1.1832	24.120
16	0.066081	9.1314×10^{-15}	0.078508	2.4869×10^{-13}	1.1881	27.235
17	0.143396	9.2125×10^{-15}	0.159578	6.3953×10^{-13}	1.1128	69.420
18	0.300444	9.7404×10^{-15}	0.335957	9.9882×10^{-13}	1.1182	102.54
19	0.616269	9.7929×10^{-15}	0.671813	7.3610×10^{-13}	1.0901	75.167
20	1.245387	1.0335×10^{-14}	1.435486	4.4930×10^{-13}	1.1526	43.474

Table 1: Comparative Timings and Errors

its working array to be re-initialized whenever the order m changes. Initialization times are not included in the timings in Table 1. If they were, the new algorithm would of course compare even more favorably.

References

1. E. Oran Brigham, *The Fast Fourier Transform*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974.

2. J. W. Cooley, and J. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series", *Mathematics of Computation*, 19 (1965), pp. 297-301.

3. B. Fornberg, "A Vector Implementation of the Fast Fourier Transform Algorithm", *Mathematics of Computation*, 36 (1981), pp. 189-191.

4. R. W. Hockney, and C. R. Jesshope, *Parallel Computers*, Adam Hilger Ltd., Bristol, England, 1981.

5. M. C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing", *Journal of the Association for Computing Machinery*, 15 (1968), pp. 252-64.

6. P. N. Swarztrauber, "FFT Algorithms for Vector Computers", *Parallel Computing*, 1 (1984), pp. 45-63.

7. P. N. Swarztrauber, "Vectorizing the FFTs", in *Parallel Computations*, G. Rodrigue, ed., Academic Press, N.Y., 1982.

8. C. Temperton, "Mixed Radix Fast Fourier Transforms Without Reordering", *Report No. 3*, European Centre for Medium-Range Weather Forecasting, Shinfield Park, Reading, UK.

APPENDIX

FORTRAN LISTING