A High-Performance FFT Algorithm for Vector Supercomputers

David H. Bailey

November 23, 1987

Ref: *Intl. Journal of Supercomputer Applications*, vol.2, no. 1 (1988), pg. 82–87

Abstract

Many traditional algorithms for computing the fast Fourier transform (FFT) on conventional computers are unacceptable for advanced vector and parallel computers because they involve nonunit, power-of-two memory strides. This paper presents a practical technique for computing the fast Fourier transform that completely avoids all such strides and appears to be near-optimal for a variety of current vector and parallel computers. Performance results of a program based on this technique are presented. Notable among these results is that a Fortran implementation of this algorithm on the Cray-2 runs up to 77% faster than Cray's assembly-coded library routine.

The author is with the Numerical Aerodynamic Simulation (NAS) Systems Division at NASA Ames Research Center, Moffett Field, CA 94035.

Introduction

As a result of the proliferation of advanced vector and parallel scientific computers, many traditionally used numeric computation algorithms are being reconsidered. Issues that previously were very important, such as conserving memory, are no longer significant, and other issues have come to the fore. One issue that is increasingly important for a wide range of new systems is memory access patterns, especially the memory stride (i.e., the increment in memory between successive elements in a vector fetch or store operation).

For a highly parallel, distributed memory system such as a hypercube, data locality is a very important issue. Even though such systems are designed for efficient communication of data between processors, it is still highly advantageous to be able to perform as much computation as possible locally within one processor-memory node. This means that unit stride memory accesses are optimal — larger strides increase the frequency that data must be accessed that resides on other processors. Uniprocessor systems with cached memories, such as the Convex C1 and the IBM 3090 VF, have a similar problem. On these systems, large strides increase the frequency of cache hits, which result in significantly decreased performance. Again, the optimal stride for memory accesses is one.

In vector supercomputers with large interleaved shared memories (such as Crays), large strides are not disadvantageous *per se*, but strides divisible by large powers of two result in extremely serious performance reductions due to memory bank conflicts. Because of the interleaved design of the memory banks, a vector fetch or store with unit stride operates at full speed, since each word resides on a different bank. However, vector fetches or stores with strides divisible by large powers of two suffer serious slowdowns, since each word resides on the same bank. On the Cray-2, strides divisible by 256 can result in slowdowns of more than a factor of 40.

In other systems, such as the CDC Cyber 205 and the recently introduced ETA-10, most hardware instructions are designed for unit stride memory access. Access of data with strides other than one requires the usage of gather-scatter instructions and as a result does not fully tap the potential performance of the machine.

This paper describes a technique for evaluating one-dimensional power-of-two FFT that completely avoids power-of-two memory strides. In addition, this algorithm features the following:

- All fetches and stores of the main data arrays in the inner loop are performed with unit stride.
- In the first few iterations, roots of unity are fetched in inner loops, but these fetches have unit stride.
- After the first few iterations, all roots of unity are fetched in the outer loop.
- The minimum length of inner loops can be adjusted to be at least the natural vector length for the machine being used, or up to the square root of the size of the transform.

- A power-of-two matrix transposition is required after the first few iterations, but this step can be performed without any power-of-two memory strides.
- An ordered transform is produced; no bit-reversal permutation is necessary.

Strides in FFT Algorithms

Unfortunately, by far the most popular FFT is the power-of-two FFT, and virtually all machine implementations of this FFT require numerous memory accesses with large power-of-two memory strides. For example, a commonly used algorithm for evaluating power-of-two FFTs is the following, which is known as the standard Stockham formulation of the FFT. Let $n = 2^m$, and let $\alpha_t = e^{\pm 2\pi i/p}$, where $p = 2^t$. Then iterate the following calculations for t from 1 to m:

$$\begin{aligned}
Y(j2^t + k) &= X(j2^{t-1} + k) + \alpha_t^k X(j2^{t-1} + n/2 + k) \\
Y(j2^t + 2^{t-1} + k) &= X(j2^{t-1} + k) - \alpha_t^k X(j2^{t-1} + n/2 + k) \\
0 \le j < 2^{m-t} & \qquad 0 \le k < 2^{t-1}
\end{aligned}$$

In the above formulas, X is the input array and Y is the output array for each iteration, and these formulas imply that data be copied from Y to X before the next iteration. However, in practice iterations are performed from X to Y and then from Y to X so that copying is not necessary. Actually, in older implementations of FFTs, memory space is usually saved by overwriting the results into the same array, rather than using a separate array for the results on the left-hand side. If this is done, the resulting data will not be stored in the proper locations, and a "bit-reversal" permutation of the final results will be necessary.

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 $U(k) = \alpha_m^k,\ 0 \le k < n/2$. 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 many 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.

On most new systems, however, each of the above-mentioned techniques for evaluating the FFT is inappropriate. Saving memory by not using a scratch array, at the cost of a bit-reversal permutation, is ridiculous on systems such as the Cray-2, which has over 268 million 64-bit words of main memory. Likewise, evaluating the above formulas with j incremented in the inner loop is extremely unwise on many new vector and parallel computers, because virtually all accesses of the X and Y arrays will employ large power-of-two memory strides, especially in the later iterations. And even the practice of fetching powers of α_t as described above results in power-of-two memory strides.

Previous Solutions to the Power-of-Two Stride Problem

A number of solutions have been proposed to the problem of power-of-two memory strides in FFTs. One solution is to simply avoid using power-of-two FFTs, in favor of radix-3 or radix-5 FFTs. Unfortunately, most applications of the FFT are not easily converted to using mixed radix FFTs. In some cases, the usage of a power-of-two FFT is inherent in the nature of the application and cannot be changed at any cost. And even where it is possible to convert an application to use mixed-radix FFTs, the programming cost of such a conversion is often prohibitive. Hockney and Jesshope [3] suggest using spacers in data to avoid power-of-two strides, but again this is not a practical solution to the problem for most applications.

Many applications of the FFT require two-dimensional or three-dimensional FFTs. Such FFTs can easily be performed with stride one memory accesses by merely coding a one-dimensional algorithm to operate on a vector of data simultaneously. Unfortunately, not all applications can utilize simultaneous FFT computations, and even where it is possible it is often not convenient to do so.

Some FFT algorithms, notably the Pease algorithm [4], permit stride one, long vector computation for virtually all operations. Fornberg [2] has reported success in implementing this technique on the CDC 205, for instance. Unfortunately, the Pease algorithm requires a bit-reversal permutation to be performed on the output data, and most vector computers are not efficient performing this permutation.

In a previous paper by the author [1], a technique was presented that avoids the power-of-two memory strides inherent in the Stockham FFT. The first "trick" is to perform the Stockham iterations incrementing k in the inner loop instead of j. In this way, the data arrays X and Y may be accessed with unit stride (provided that complex data is stored with real and imaginary parts separated instead of interleaved, as is the usual custom). The second idea mentioned in [1] is to store the powers of α_t that are needed for a single iteration as contiguous data in a separate section of the array U. In this way, the fetch of roots of unity from U always has stride one. The memory storage required for storing all roots of unity in this manner is only twice the usual amount, and using some extra memory in this fashion to increase performance is in fact an efficient usage of a large memory machine.

Unfortunately, in practice it is not efficient to perform all iterations with k incremented in the inner loop. This is because for the first few iterations the inner loop vector lengths are very short, which sharply reduces performance on any vector machine. Thus, in the implementation described in [1], it is necessary to perform the first few iterations with j incremented in the inner loop, which results in some power-of-two memory strides.

A technique reported in [1] that significantly improves the performance of the Cray-2 implementation is the usage of radix-4 versions of the above Stockham formulas. Using radix-4 FFT formulas in effect allows two radix-2 iterations to be performed within the fast CPU registers before results must be stored back to the main memory. This technique thus lessens the dependence of the algorithm on the relatively slow Cray-2 main memory, and a significant improvement in performance is obtained. The resulting Fortran program is as much as 30% faster than Cray's assembly-coded library routine on the Cray-2.

A New Technique for Performing Power-of-Two FFTs

In [4] and [5] Swarztrauber surveys the best algorithms currently available for performing FFTs on parallel and vector computers. Included in [5] are two variations of the Stockham FFT, each different than the algorithm listed above. These variant FFTs each have the property that all inner loop calculations can be performed with stride one accesses of the main data arrays. It is necessary, however, to switch from one variant to the other to avoid very short inner loop vector lengths, and this switch from one variant to the other requires a transposition permutation.

The formulas for Swarztrauber's variations of the Stockham FFT are as follows. Let $n = 2lm$. It may be assumed that the factors l and m are powers of two if desired, but the formulas are valid for any factorization. In the usual power-of-two case, l is initially set to be $n/2$ and decreases by a factor of two with each iteration. Similarly, m is initially set to be 1 and increases by a factor of two with each iteration. As before, X denotes the input data and Y denotes the output data for each iteration, but in an efficient implementation, the iterations are performed from X to Y and from Y to X. In the following, ω_p denotes $e^{\pm 2\pi i/p}$.

Radix-2 Variant 1

$$\begin{aligned}
c_0 &= X(j + 2kl) \\
c_1 &= X(j + 2kl + l) \\
Y(j + kl) &= c_0 + c_1 \\
Y(j + kl + lm) &= \omega_{2l}^{j}(c_0 - c_1) \\
0 \le j < l & \qquad 0 \le k < m
\end{aligned}$$

Radix-2 Variant 2

$$\begin{aligned}
c_0 &= X(k + jm) \\
c_1 &= X(k + jm + lm) \\
Y(k + 2jm) &= c_0 + c_1 \\
Y(k + 2jm + m) &= \omega_{2l}^{j}(c_0 - c_1) \\
0 \le j < l & \qquad 0 \le k < m
\end{aligned}$$

These formulas suggest that in the inner computational loops, j should be incremented in Variant 1, and k should be incremented in Variant 2. In this manner, all fetches and stores of data in the X and Y arrays are with unit stride. The manner in which roots of unity are fetched is not an issue in Variant 2, since this may be done in the outer loop. In Variant 1, on the other hand, roots of unity must be fetched in the inner loop, and these fetches ordinarily have power-of-two memory strides. Such strides can be avoided, however, by employing the technique mentioned above of precalculating the roots of unity needed for each iteration in a separate, contiguous section of the U array. Once this is done, all iterations of a power-of-two FFT using the above formulas may be performed using exclusively unit stride memory accesses.

The choice of the crossover point between the two variants depends on the computer system being used. Normally only the first few iterations are performed using Variant 1, because roots of unity must be fetched in the inner loop. Once the value of m is sufficiently large that the harmful effect of very short vector lengths is avoided, then Variant 2 is used for all succeeding iterations.

As mentioned above, the switch between the two variants requires that the data (considered as a $2l \times m$ matrix) be transposed. Unfortunately, the usual method for performing a matrix transposition with power-of-two dimensions involves numerous power-of-two memory strides. While it is not possible to transpose such a matrix with exclusively unit stride operations, the memory bank conflicts that occur on most vector computer systems can be avoided as follows. Assume for simplicity that $n = lm$ and $l \geq m$. Then set $p = l + 1$ and $q = m + 1$. Then the data array X may be transposed as follows:

$$\begin{aligned}
Y(iq + j) &= X(ip + lj) \qquad 0 \leq i < m - j, \ 0 \leq j < m \\
Y(iq + mj) &= X(ip + j) \qquad 0 \leq i < m, \ 1 \leq j < l - m \\
Y(iq + mj) &= X(ip + j) \qquad 0 \leq i < l - j, \ l - m \leq j < l
\end{aligned}$$

These formulas in effect perform a matrix transposition along diagonals. In this manner, all memory accesses are with constant strides that are one larger than a power of two. On many systems, including Cray X-MP and Cray-2 systems, memory accesses with such strides are performed virtually as fast as with stride one. On other systems, such as the CDC 205, any nonunit stride will result in reduced performance, but the reduction will be greatly moderated by the absence of bank conflicts.

Radix-4 Formulas

In the author's previous paper [1], it was found that radix-4 formulas resulted in a significant improvement in the performance on the Cray-2. This is due to the fact, as was mentioned above, that radix-4 iterations perform the work of two radix-2 iterations with only one access of the data arrays in main memory. The same principle applies to the algorithm described in this paper. The corresponding radix-4 formulas are as follows:

Radix-4 Variant 1

$$\begin{aligned}
c_0 &= X(j + 4kl) \\
c_1 &= X(j + 4kl + l) \\
c_2 &= X(j + 4kl + 2l) \\
c_3 &= X(j + 4kl + 3l) \\
d_0 &= c_0 + c_2 \\
d_1 &= c_0 - c_2 \\
d_2 &= c_1 + c_3
\end{aligned}$$

$$\begin{aligned}
d_3 &= \pm i(c_1 - c_3) \\
Y(j+kl) &= d_0 + d_2 \\
Y(j+kl+lm) &= \omega_{4l}^{j}(d_1 + d_3) \\
Y(j+kl+2lm) &= \omega_{4l}^{2j}(d_0 - d_2) \\
Y(j+kl+3lm) &= \omega_{4l}^{3j}(d_1 - d_3) \\
0 \le j < l & \qquad 0 \le k < m
\end{aligned}$$

Radix-4 Variant 2

$$\begin{aligned}
c_0 &= X(k+jm) \\
c_1 &= X(k+jm+lm) \\
c_2 &= X(k+jm+2lm) \\
c_3 &= X(k+jm+3lm) \\
d_0 &= c_0 + c_2 \\
d_1 &= c_0 - c_2 \\
d_2 &= c_1 + c_3 \\
d_3 &= \pm i(c_1 - c_3) \\
Y(k+4jm) &= d_0 + d_2 \\
Y(k+4jm+m) &= \omega_{4l}^{j}(d_1 + d_3) \\
Y(k+4jm+2m) &= \omega_{4l}^{2j}(d_0 - d_2) \\
Y(k+4jm+3m) &= \omega_{4l}^{3j}(d_1 - d_3) \\
0 \le j < l & \qquad 0 \le k < m
\end{aligned}$$

It should be noted that in practice, both the radix-4 and radix-2 formulas must be used, in order that transforms of both even and odd power-of-two sizes may be processed.

Performance Results

The above technique has been implemented in a Fortran program and run on a number of advanced vector computers. In each case, excellent performance results have been obtained. On the CDC 205, a rate of 105 million 64-bit floating-point operations per second (MFLOPS) was obtained on a transform of size 2^{17}. On the Convex C1P, a rate of 7.9 MFLOPS was obtained on a transform of size 2^{15} using the radix-2 version.

The results on the Cray-2, which is very sensitive to power-of-two memory strides, are particularly encouraging. Table 1 compares CFFTZ, which is a Fortran program based on the radix-4 technique described in this paper, to CFFT2, which is the assembly-coded library routine provided by Cray Research, Inc. The size of the transform is 2^m, where m is given in the first column. The column headed MFLOPS gives the performance of the program CFFTZ, based on the formula $5m2^m$ for the number of floating-point operations for a transform of size 2^m. The error statistics listed in this table are the root-mean-square

	CFFT2		CFFTZ			Time	Error
m	CPU Time	Error	CPU Time	Error	MFLOPS	Ratio	Ratio
8	0.00023	5.111×10^{-15}	0.00027	6.078×10^{-15}	37.92	0.846	0.8410
9	0.00048	5.658×10^{-15}	0.00045	6.130×10^{-15}	51.00	1.061	0.9229
10	0.00099	6.784×10^{-15}	0.00074	6.913×10^{-15}	69.60	1.347	0.9813
11	0.00204	1.102×10^{-14}	0.00148	7.052×10^{-15}	76.33	1.380	1.562
12	0.00420	1.252×10^{-14}	0.00274	7.608×10^{-15}	89.62	1.531	1.646
13	0.00868	3.510×10^{-14}	0.00538	7.865×10^{-15}	98.99	1.614	4.462
14	0.01871	5.445×10^{-14}	0.01055	8.430×10^{-15}	108.70	1.773	6.459
15	0.03987	2.095×10^{-13}	0.02427	8.555×10^{-15}	101.28	1.643	24.49
16	0.07719	2.518×10^{-13}	0.04724	9.092×10^{-15}	110.99	1.634	27.69
17	0.16113	6.416×10^{-13}	0.09524	9.248×10^{-15}	116.98	1.692	69.38
18	0.32692	9.999×10^{-13}	0.19488	9.758×10^{-15}	121.07	1.678	102.5
19	0.67228	7.362×10^{-13}	0.43286	9.847×10^{-15}	115.06	1.553	74.76
20	1.39522	4.498×10^{-13}	0.89189	1.035×10^{-14}	117.57	1.564	43.47

Table 1: Cray-2 Performance Figures

(RMS) error after performing a forward and an inverse FFT on pseudorandom data. These runs were made on one processor of the Cray-2 system with a normal background of jobs running on the other three processors.

Conclusions

An implementation of Swarztrauber's variations of the Stockham FFT algorithm, combined with special techniques for eliminating power-of-two memory strides in the access of roots of unity and in performing a matrix transposition, has resulted in a high-performance one-dimensional power-of-two FFT suitable for many vector computers. A Fortran radix-4 implementation of this method runs as much as 77% faster than Cray's assembly-coded library routine on the Cray-2. A side benefit of this program is that its numeric precision is superior to Cray's library routine. The reason for the improved precision is not known.

While the author's implementation has focused on single processor vector computers, the unit stride feature of the above algorithm is highly appropriate for parallel systems. Indeed, Swarztrauber's paper [6] emphasizes the application of his technique to hypercubes and other multiprocessor systems. It thus appears that this basic technique is well suited for a variety of advanced systems.

References

1. Bailey, D.H., "A High-Performance Fast Fourier Transform Algorithm for the Cray-2", *Journal of Supercomputing*, to appear.

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

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

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

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

6. Swarztrauber, P.N., "Multiprocessor FFTs", *Parallel Computing*, to appear.