FFTs in External or Hierarchical Memory

David H. Bailey

December 30, 1989

Abstract

Conventional algorithms for computing large one-dimensional fast Fourier transforms (FFTs), even those algorithms recently developed for vector and parallel computers, are largely unsuitable for systems with external or hierarchical memory. The principal reason for this is the fact that most FFT algorithms require at least m complete passes through the data set to compute a 2^m-point FFT.

This paper describes some advanced techniques for computing an ordered FFT on a computer with external or hierarchical memory. These algorithms (1) require as few as two passes through the external data set, (2) employ strictly unit stride, long vector transfers between main memory and external storage, (3) require only a modest amount of scratch space in main memory, and (4) are well suited for vector and parallel computation.

Performance figures are included for implementations of some of these algorithms on Cray supercomputers. Of interest is the fact that a main memory version outperforms the current Cray library FFT routines on the Cray-2, the Cray X-MP, and the Cray Y-MP systems. Using all eight processors on the Cray Y-MP, this main memory routine runs at nearly two gigaflops.

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

Introduction

The development of numerous advanced architecture computers has posed a considerable challenge to computer scientists. Many numeric algorithms that were completely satisfactory for traditional serial computers are unsatisfactory for these advanced systems. This phenomenon is particularly pronounced in the case of algorithms for evaluating one dimensional fast Fourier transforms.

One reason for this difficulty is the fact that many modern computers, particularly those with interleaved main memories, do very poorly with data that is accessed with a memory stride that is a large power of two. By far the most popular sizes of data to be transformed using FFTs are powers of two, and traditional implementations of FFTs for such data sets involve heavy use of power of two memory strides. Fortunately, it is possible to devise alternative FFT algorithms that do not rely on power of two strides. Indeed, some FFT algorithms can be performed using exclusively unit stride data access in inner computational loops [3], [5], [6], [9], [10]. Even for systems with external or hierarchical memory systems, these unit stride algorithms are a definite improvement over conventional algorithms, since unit strides improve the locality of accesses to and from external memory.

However, many FFT algorithms, both traditional and modern, still require roughly m passes through the data set to compute a 2^m-point FFT. The number of required passes can be significantly reduced by using radix-4 or radix-8 variations of these algorithms, but the number of passes remains proportional to m. Since such external data access is usually a crucial bottleneck in such computations, it would be highly desirable to reduce this number to a bare minimum.

The Basic "Four Step" FFT Algorithm

There is one algorithm in the FFT literature that is quite effective in reducing the number of passes through the dataset. Recently variants of this algorithm were featured in papers by Agarwal and Cooley [1, p. 150], Ashworth and Lyne, [4, p. 219], and Swarztrauber [10, p. 202 - 203]. Swarztrauber used this technique as a starting point for a very efficient hypercube FFT, and both [1] and [10] noted the suitability of this algorithm for systems with nonlocal memory systems, including hierarchical and distributed memory designs. However, as it turns out, this algorithm was actually first presented over twenty years ago in a paper by Gentleman and Sande [8, p. 569]. This early paper even described the application of this algorithm to a system with hierarchical memory. Unfortunately, this algorithm appears to have been largely forgotten in the interim, as a number of more recent papers have suggested much less efficient methods.

This algorithm, which shall hereafter be referred to as the "four step" FFT algorithm, can be stated very succinctly. Let $n = n_1 n_2$ be the size of the transform. Note that n does not necessarily need to be a power of two. On many systems, the implementation of this algorithm is most efficient when n_1 and n_2 are as close as possible to $\sqrt{n}$. In the following and hereafter, matrices will be assumed to be stored in memory columnwise as in the Fortran language. The FFT of n complex input data values can then be obtained by performing the following four steps:

1. Perform n_1 simultaneous n_2-point FFTs on the input data considered as a $n_1 \times n_2$ complex matrix.

2. Multiply the resulting data, considered as a $n_1 \times n_2$ matrix A_{jk}, by $e^{\pm 2\pi ijk/n}$. The $\pm$ sign is the sign of the transform.

3. Transpose the resulting $n_1 \times n_2$ complex matrix into a $n_2 \times n_1$ matrix.

4. Perform n_2 simultaneous n_1-point FFTs on the resulting $n_2 \times n_1$ matrix.

Several important features of this algorithm should be noted: first of all, note that both of the simultaneous FFT steps can be performed using exclusively unit stride data access, which is optimal on virtually any computer system. Secondly, this algorithm produces an ordered transform (provided the simultaneous FFTs are ordered) — it is not necessary to perform a bit reversal permutation, which is inefficient on many advanced computer systems. Finally, note that only three passes through the external data set are required to perform this algorithm — the second step can be performed on a block of data after the first step, before it is returned to memory. This bounded number of passes is in accordance with the I/O complexity results in [2].

Main Memory Performance Results using the Four Step FFT

Depending on implementation, the four step FFT algorithm may actually require a slightly larger number of floating-point arithmetic operations than conventional FFT algorithms. In spite of this slight handicap, it is remarkably efficient even for a single processor vector computer transforming data in main memory. As can be seen in tables 1 and 2, a straightforward implementation of this scheme is up to 10% faster than Cray's library routine on the Cray-2 and up to 20% faster than Cray's library routine on the Cray Y-MP. The percentage results on the Cray X-MP are very close to those on the Cray Y-MP, which is to be expected since the CPU and memory designs of the X-MP and Y-MP systems are very similar, differing mainly in speed of operation. For these tests, the four step FFT algorithm was implemented using a simple Fortran program; assembly code was employed only within the Cray library simultaneous FFT routine (CFFTMLT), which is called by this Fortran program to perform steps 1 and 4. The transpose step (step 3) was performed without power of two strides by employing a diagonal technique, as mentioned in [6, p. 85]. The Cray-2 library 1-D FFT routine (CFFT2) used in table 1 is an assembly-coded implementation of an algorithm described by the author in a previous paper [6]. The Cray Y-MP library 1-D FFT routine (CFFT2) used in table 2 is essentially the same routine that has been available for some time on the Cray X-MP systems.

The CPU times shown in both tables 1 and 2 are for forward 2^m-point FFTs followed by inverse FFTs, averaged over ten trials, in seconds. All megaflops performance figures in these tables are computed based on $10m2^m$ floating-point operations, even though the four step routine may perform slightly more than this figure. These tests were run in a typical daytime environment, and so the results reflect a normal amount of memory bank contention. The computers used for these tests belong to the Numerical Aerodynamic

Simulation (NAS) Systems Division at NASA Ames Research Center. This particular Cray-2 system has a clock period of 4.1 nanoseconds (ns), and has 268 million words of 80 ns DRAM main memory. The Cray Y-MP system used for these tests has a clock period of 6.3 ns and 33 million words of bipolar main memory. This Y-MP system was the first Y-MP delivered by Cray. Newer Y-MP systems have a faster clock (6 ns), and thus these results would be correspondingly better on the newer systems.

The results listed in tables 1 and 2 are single processor results — no attempt was made to employ more than one processor. However, with the new "autotasking" feature now available on Cray systems, it is possible to study the performance of a program using all available processors, with only a minimum of changes to the source code. When autotasking was invoked on the Fortran program mentioned above, performance levels very nearly eight times the single processor levels were achieved on the eight processor Y-MP. These results are shown in table 3. This very high speedup underscores the suitability for the four step FFT algorithm for parallel processing.

FFTs on Data in External or Hierarchical Memory Systems

The Cray-2 is noted for its very large main memory. Most Cray-2 systems include 268 million 64 bit words of main memory, although recently Cray has shipped a 536 million word system. However, the performance of the Cray-2 on many codes in a normal production environment is not outstanding, due to severe memory bank contention, a direct result of the relatively slow operation speed of DRAM memory chips. Most Cray X-MP and Y-MP systems utilize a faster technology (bipolar) in main memory, so that memory bank contention is very much reduced. However, bipolar memory chips are not available in nearly the density of equivalent generation DRAM chips, and so as a result the largest main memory currently available for Y-MP systems is 33 million 64 bit words. Y-MP systems typically have eight CPUs, so this means an average of only four million words per processor. Systems that support interactive as well as batch users must be even more restrictive in the amount of main memory that can be allocated to a single job.

As a result, users of the Cray X-MP and Y-MP systems who wish to perform large one dimensional FFTs are led to consider utilizing the solid state disk (SSD) available on these systems. SSD systems with a capacity of up to 536 million words are now available on the Y-MP. Users of the ETA-10 or the IBM 3090/VF systems have an analogous choice in utilizing the virtual memory system, which is a large semiconductor memory similar to the Cray SSD, but which does not require explicit programmer input/output commands. Users on other systems can even consider utilizing disk drives, although the relative slowness of such devices compared to main memory is a bottleneck even with the best of algorithms.

In addition to minimizing the number of data accesses to an external memory device, an obvious consideration in designing an efficient algorithm for such systems is to minimize the amount of scratch space required in main memory. Clearly if an external memory algorithm requires a substantial scratch array in main memory, then the largest transform size will again be limited by the available main memory. In addition, it will be assumed in the following that the amount of external memory is also limited and must be conserved.

Size	Four Step FFT		Cray Library FFT	
m	Time	MFLOPS	Time	MFLOPS
8	0.0005	42.5	0.0004	57.2
9	0.0008	60.9	0.0006	81.8
10	0.0013	76.4	0.0010	106.0
11	0.0021	106.6	0.0021	109.4
12	0.0036	137.8	0.0038	130.6
13	0.0074	143.8	0.0073	145.2
14	0.0145	158.5	0.0138	165.7
15	0.0300	163.9	0.0327	150.2
16	0.0559	187.5	0.0660	159.0
17	0.1248	178.6	0.1260	176.8
18	0.2426	194.5	0.2555	184.7
19	0.4971	200.4	0.5763	172.9
20	1.0260	204.4	1.1863	176.8

Table 1: The Four Step FFT vs. Cray's Library Routine on the Cray-2

Size	Four Step FFT		Cray Library FFT	
m	Time	MFLOPS	Time	MFLOPS
8	0.0003	68.68	0.0001	137.85
9	0.0005	102.24	0.0003	151.88
10	0.0008	128.16	0.0006	161.27
11	0.0013	168.91	0.0013	168.08
12	0.0024	201.90	0.0028	173.51
13	0.0049	215.88	0.0060	178.09
14	0.0103	222.57	0.0126	181.86
15	0.0212	231.67	0.0265	185.24
16	0.0443	236.78	0.0557	188.26
17	0.0935	238.39	0.1167	190.97
18	0.1976	238.81	0.2439	193.44
19	0.4117	241.96	0.5090	195.70
20	0.8587	244.23	1.0635	197.20

Table 2: The Four Step FFT vs. Cray's Library Routine on the Cray Y-MP

Size	Time	MFLOPS	Speedup
12	0.00079	625.09	3.096
13	0.00138	771.49	3.574
14	0.00218	1053.61	4.734
15	0.00376	1308.07	5.646
16	0.00667	1571.73	6.638
17	0.01318	1690.03	7.089
18	0.02566	1838.72	7.700
19	0.05275	1888.53	7.805
20	0.10882	1927.12	7.891

Table 3: Cray Y-MP Performance of the Four Step FFT Using Eight Processors

It will also be assumed for the time being that the final result in external memory must be physically ordered — index schemes or "virtual" orderings of external blocks will not be allowed.

Reducing the Scratch Space Requirement in the Four Step FFT

As presented above, a straightforward implementation of the four step FFT algorithm requires scratch space for several different purposes. These are as follows:

- $2n$ cells for the precomputed root of unity table.
- $2n$ cells of scratch space for the simultaneous FFT steps.
- $2n$ cells of scratch space for the transpose step.

The scratch space requirement for the simultaneous FFT steps can easily be reduced by noting that the n_1 simultaneous n_2-point FFTs (i.e. in step 1 of the four step FFT) may be performed in batches of v rows, where v is the natural vector length of the system being used. If the simultaneous FFTs employ an algorithm, such as the Stockham FFT, which requires a scratch array the same size as the input data array, then only $4vn_2$ scratch cells are required. This figure may be reduced by one half if an in-place algorithm can be efficiently used for the simultaneous FFTs. Note that if the individual processors do not rely on vector processing, then only one row need be fetched at a time, and these scratch space figures drop to only $4n_2$ cells and $2n_2$ cells, respectively. For step 4 of the four step FFT, the corresponding scratch space figures may be obtained by replacing n_2 by n_1 in the above discussion.

However, the scratch space requirements for the simultaneous FFT steps in reality are dependent more on the block size b of an efficient input/output (I/O) transfer between main and external memory. In other words, if the natural I/O block length is 128, then 128 rows of the $n_1 \times n_2$ complex matrix should be fetched into main memory, or else the

I/O operations will be highly inefficient. Thus it follows that a main memory scratch space of size $2bn_2 + 2vn_2$ is needed for the first step of the four step algorithm. In an similar manner, the last step of the four step FFT requires $2bn_1 + 2vn_1$ scratch cells. The second term of each of these expressions may be omitted if an in-place algorithm can be efficiently used for simultaneous FFTs in main memory.

The scratch space for the two FFT steps could be reduced to virtually zero if an FFT algorithm somewhat more complicated than the four step FFT were used. This algorithm is as follows:

1. Transpose the input data set, considered as a $n_1 \times n_2$ complex matrix, into a $n_2 \times n_1$ matrix.
2. Perform n_1 individual n_2-point one dimensional FFTs on the resulting $n_2 \times n_1$ matrix.
3. Multiply the resulting $n_2 \times n_1$ complex matrix A_{ij} by $e^{\pm 2\pi ijk/n}$.
4. Transpose the resulting $n_2 \times n_1$ matrix into a $n_1 \times n_2$ matrix.
5. Perform n_2 individual n_1-point one dimensional FFTs on the resulting $n_1 \times n_2$ matrix.
6. Transpose the resulting $n_1 \times n_2$ complex matrix into a $n_2 \times n_1$ matrix.

This algorithm, which could be termed by analogy the "six step" FFT algorithm, is very well suited for distributed memory systems, as the individual one dimensional FFTs can be performed in individual processors. Its main memory scratch requirement is only $4n_2$ cells for step 2 and $4n_1$ cells for step 5 (per processor). As before, if an in-place FFT algorithm can be efficiently used in main memory, then these figures can be reduced by one half. However, there are other ways of performing FFTs on systems such as MIMD hypercubes [10], and the six step FFT has the serious disadvantage of requiring an additional two transpose steps, which typically are the chief bottlenecks on any system with a distributed or external memory.

Reducing the size of the precalculated root of unity table used in step 2 of the four step FFT algorithm is somewhat trickier. Nonetheless, it can be reduced in size to virtually zero with only a slight increase in overall run time, by using what may be termed the dynamic block scheme for roots of unity. The full size $n_1 \times n_2$ root of unity table can be written as $U(j,k) = \alpha^{jk}$ where $\alpha = e^{\pm 2\pi i/n}$. Let $B(r,s)$ denote a block of dimensions $a \times b$ within the matrix U. Note that

$$\begin{aligned} U(j+ra, k+sb) &= \alpha^{(j+ra)(k+sb)} \\ &= \alpha^{jk}\alpha^{jsb}\alpha^{rak}\alpha^{rasb} \\ &= U(j,k)U(j,sb)U(ra,k)U(ra,sb) \end{aligned}$$

Thus an $a \times b$ block $B(r,s)$ in the interior of U can be dynamically computed as follows:

$$\begin{aligned} B(r,s) \quad = \quad & \text{top left block [i.e. } B(0,0)] \\ & \times \text{ "spike" from top edge} \\ & \times \text{ "spike" from left edge} \\ & \times \text{ upper left corner element of } B(r,s) \end{aligned}$$

This scheme can be visualized as in figure 1. It can be implemented with only one additional complex multiplication in the innermost loop. The storage requirements for those subsets of U array that must be precomputed are as follows:

Description	Size
Upper left basic block	$2ab$
"Spikes" from top edge	$2an_2/b$
"Spikes" from left edge	$2bn_1/a$
Block corners	$2n/ab$

If we assume that $n_1 = n_2 = a^2 = b^2$ (which is an optimal choice), then the total space is only $8\sqrt{n}$ cells, a sufficiently small amount that this data can be kept in main memory. In tests of this scheme on Cray systems, the author merely selected a and b to be 64, the natural vector length. With this choice, only a few thousand cells of main memory are required even for multimillion point transforms. Performance tests of FFTs using this scheme indicates that it adds only about five percent to the total run time (for larger transforms), and the accuracy of the dynamically calculated roots is excellent.

Transposing Arrays in External Memory

The transpose step (step 3 of the four step FFT) is perhaps the most challenging to perform efficiently on a data set residing in external memory. Before discussing this matter in detail, it should be recalled that the array to be transposed consists of complex data. In the following discussion it will be assumed that the real and imaginary parts of this data are stored in completely separate memory locations, not interleaved as is the Fortran convention. In this way the problem of transposing a complex array reduces to transposing two real arrays. In fact, separate storage of real and imaginary data avoids a significant performance degradation in computing with complex data on a number of systems, including the ETA-10 and the Cray-2, since the standard Fortran COMPLEX data format requires stride two access.

Probably the most efficient algorithm currently known to transpose data in external storage is due to Fraser [7]. A particularly attractive aspect of this algorithm is that it can easily be tuned for maximum efficiency on a given system. It is easier to exhibit an example of Fraser's algorithm than to precisely state it. Suppose one wishes to transpose a $2^8 \times 2^7$ matrix, which resides on an external random access dataset, into a $2^7 \times 2^8$ matrix. Suppose also that the size of an efficient I/O block is $64 = 2^6$, that two main memory buffers of size $512 = 2^9$ are available, and that an external scratch dataset of size 2^{15} is available. Let the

Figure 1: The Dynamic Block Scheme for Roots of Unity

notation `(0 1 2 ... 12 13 14)` denote the binary digit positions in the reverse binary expansion of an index in the 2^{15}-long input array. Then the steps required to transpose this array can be compactly presented as follows:

```
E1     0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
M1     0  1  2  3  4  5  8  9 10 11 12 13 14  6  7
M2     8  9 10  0  1  2  3  4  5 11 12 13 14  6  7
E2     8  9 10  0  1  2 11 12 13 14  3  4  5  6  7
M1     8  9 10  0  1  2 11 12 13 14  3  4  5  6  7
M2     8  9 10 11 12 13  0  1  2 14  3  4  5  6  7
E1     8  9 10 11 12 13 14  0  1  2  3  4  5  6  7
```

The notation at the beginning of each line indicates the source of the data in each operation: E1 denotes external dataset number 1, M2 denotes main memory buffer number 2, etc. Note that the transfers between external memory and main memory only alter locations 6 through 14, and leave locations 0 through 5 unchanged (i.e., 64-long contiguous blocks are preserved), and that transfers between two main memory buffers only alter locations 0 through 8 (i.e. only affect data within a single 512-long main memory buffer).

The first step, from external to main, involves fetching contiguous blocks of size 64 from disk with a block stride of four (i.e. fetch the first 64-long block, skip three blocks, fetch the fifth 64-long block, etc.). The first step is done in batches of 8 blocks, so that 512 words are fetched to one of the main memory buffers before proceeding. The second step, which is performed between the two main memory buffers, is to transpose the resulting 512-long array, considered as a 64×8 matrix, into a 8×64 matrix. In the third step, the eight 64-long blocks in the main memory buffer are stored out to external memory, this time with a block stride of eight. This completes one pass through the external data set. In the next pass, eight contiguous 64-long blocks are fetched into main memory, and the resulting 512-long array is transposed in a block fashion that preserves 8-long contiguous sections. Finally, the resulting 64-long blocks are stored back to external memory, again in a manner that achieves a certain block permutation. The array has now been transposed in just two passes. With an adjustment of the parameters (for example, with a block size of 32 and a memory buffer size of 1024), the transposition could be achieved in a single pass.

Even from the above example, the power and generality of Fraser's technique can be appreciated. Unfortunately, Fraser's algorithm cannot in general be performed in place (i.e. using only one external dataset), unless one relaxes the requirement for a physically transposed array (by utilizing pointers to index the external data blocks instead). However, in special cases typical of common FFT sizes, there are other methods that can be done in place and still produce a physically transposed array.

Consider first the case where $n_1 = n_2$, so that the matrix is square. In that case a block interchange technique can be used to transpose the array in a single pass, in place. This can be done by simply considering the external $n_1 \times n_2$ matrix to be decomposed into square blocks of size b on a side, where b is the block size of an efficient I/O operation.

The square blocks down the diagonal can be transposed simply by fetching the blocks one at a time into main memory, transposing them using any efficient main memory scheme, and storing the resulting matrices back in the same locations. The off-diagonal square blocks can be fetched in opposing pairs, transposed in main memory, and then stored back in opposite locations. One difficulty in applying this scheme is when the main memory block size b is a power of two (which it almost always is). Transposing matrices whose dimensions are powers of two in main memory, using the straightforward scheme, results in severe memory bank conflicts on many vector supercomputers. However, such arrays can be transposed completely without bank conflicts by fetching and storing opposite diagonals, as is described in [6, p. 85]. The main memory scratch space requirement for the entire scheme is $2b^2$ cells.

For the common case of power of two FFTs, it can be assumed that either $n_1 = n_2$ or else $n_1 = 2n_2$. In the second case, it does not appear possible to transpose the array in one pass, in place, using only full block I/O transfers. However, such arrays can be transposed in just two passes, in place, using only full block transfers, as follows. First, consider the $n_1 \times n_2$ external array as two blocks of size $n_2 \times n_2$, and transpose each of these two square blocks in place, as described in the previous paragraph. This completes the first pass. Now consider the resulting data array in external memory to be a $n_2 \times n_1$ matrix. Inspection of an example shows that the columns of the resulting array need to be de-interleaved — column $2j$, $0 \le j < n_2$ needs to be moved to column j, and column $2j+1$ needs to be moved to column $j+n_2$ (here the columns are numbered beginning with zero). This de-interleaving could be done b rows at a time using a main memory scratch array of size bn_1, but this task can be done more efficiently and without need of substantial scratch space by moving the columns in permutation cycles. For example, suppose $n_1 = 8$ and $n_2 = 4$, so that after the block transpose operations we have a 4×8 matrix. Then the first cycle would consist of storing column 1 in main memory, moving column 2 to column 1, column 4 to column 2, and column 1 (from main memory) to column 4. The second cycle would consist of storing column 3 in main memory, moving column 6 to column 3, column 5 to column 6, and column 3 (from main memory) to column 5. Columns 0 and 7 do not need to be moved. Note that this column movement procedure requires only $2n_2$ cells of main memory scratch space. The dominant scratch space requirement for this case is thus $2b^2$ (for each of the two square block transpositions), the same as the case $n_1 = n_2$.

Performance Results Using the Minimal Scratch Space FFT

The above procedure has been implemented and tested on the Cray Y-MP, using one processor and the SSD external memory device. The SSD I/O primitives SSREAD and SSWRITE were called directly from the Fortran program. As before, the Cray library simultaneous FFT routine (CFFTMLT) was used in steps one and four of the four step algorithm. This routine is not an in-place FFT, so that a scratch array in addition to the space for the data is required. Since the SSD is a rather limited resource, Fraser's algorithm was not employed for the transpose steps — the in-place schemes described in the previous section were employed instead. The block length b for efficient I/O transfers

Size	Scratch	Using Memory		Using SSD	
m	Space	Time	MFLOPS	Time	MFLOPS
16	338250	0.0704	149.00	0.1169	89.68
17	668266	0.1574	141.59	0.2529	88.10
18	668330	0.2897	162.90	0.3094	152.52
19	1328426	0.6391	155.86	0.6908	144.21
20	1328682	1.2263	171.02	1.3065	160.52
21	2649130	2.7007	163.07	2.8179	156.29
22	2650154	5.2996	174.12	5.6132	164.39

Table 4: Minimal Scratch Space FFT Performance Results

between main memory and SSD (or between main memory and disk) on the Cray Y-MP system is 512.

Table 4 includes results not only for an actual external memory (SSD) implementation of the above scheme on the Cray Y-MP, but also for a modified version of the program where the Fortran routines handling I/O actually just transfer data to a block of main memory, insteading of referencing the Cray SSD primitives. With the latter figures one can actually see how much of the performance degradation is due to the algorithm and how much is due to inefficiencies in the Cray I/O system routines. The total amount of main memory scratch space for this algorithm, including space for precalculated roots of unity, is also included in this table.

Performing an FFT with Only Two Passes

The schemes that have been described so far produce a physically ordered FFT on an external dataset in three or four passes. If one is willing to relax the requirement that the final result be physically ordered, or if one is willing to allow a scratch dataset in external memory of the same size as the input dataset, then the entire FFT operation can be performed in only two passes (subject to certain conditions). The author is indebted to Paul N. Swarztrauber for this observation.

As in the four step FFT above, it will be assumed in the following that $n = n_1 n_2$ and that b is the block size for efficient I/O operation. Also, all references to matrices will, as before, assume columnwise storage. For simplicity, it will be assumed for the time being that two buffers of size $2bn_1$ cells each are available in main memory, although it will later be seen that only one buffer this large is necessary. Similarly, it will be assumed for the time being that a scratch dataset equal in size to the input dataset is available in the external memory device, although it will be seen later than this scratch dataset is not necessary if one does not mind using pointers. This algorithm can then be stated as follows.

1. Consider the data in external memory as a $n_1 \times n_2$ complex matrix. Fetch the data b rows at a time into one of the main memory buffers. For each batch of b rows,

perform b simultaneous n_2-point FFTs on the $b \times n_2$ array in main memory, using the second main memory buffer as a scratch array.

2. Multiply the resulting data in each batch by appropriate roots of unity as in the four step algorithm.

3. Transpose each of the resulting $b \times n_2$ complex matrices into a $n_2 \times b$ matrix, using the second main memory buffer as a scratch array, and store the resulting data on the scratch dataset in contiguous order. Store successive batches of data in successive contiguous sections on the scratch dataset.

4. Consider the resulting data in the scratch dataset as a $n_2 \times n_1$ complex matrix. Fetch the data b rows at a time into one of the main memory buffers. For each batch of b rows, perform b simultaneous n_1-point FFTs on the $b \times n_1$ array in main memory, using the second main memory buffer as a scratch array, and return the resulting b rows to the same locations on external storage from which they were fetched.

As before, this FFT is an ordered transform — no bit reversal transposition is necessary. The reduction of the number of passes from three to two is accomplished by combining the four step FFT with Fraser's transposition algorithm.

Let $r = \max(n_1, n_2)$. Then at least one main memory buffer of size $2br$ is required in the above to hold b rows of the fetched data. However, the second main memory buffer can be sharply reduced in size in many cases of interest. The additional scratch requirement for performing the simultaneous FFTs in steps 1 and 4 can be reduced to only $2vr$ by performing the FFTs in batches of v rows, where v is the natural vector length of the system. If an in-place algorithm is used for the simultaneous FFTs, then this scratch requirement can be completely eliminated.

Also, in the most common case of power of two transforms, the additional scratch space needed for performing the main memory transpose in step 3 above can be reduced to only $2r$ cells by applying techniques similar to those mentioned above for transposing power of two arrays in external memory. One difference in this case is that the second dimension n_2 can be much larger than the first dimension b. Nonetheless, the basic scheme of transposing the square sub-blocks in place and then moving columns in permutation cycles can also be applied for this application.

Main memory space to hold precomputed roots of unity can be reduced from $2n$ to only $8r$ by using the dynamic block method described above. Thus the total main memory storage requirement for power of two transforms can be reduced to only $2(b+5)r$ cells using this algorithm.

The requirement for a separate scratch dataset in external memory can be eliminated by utilizing a block indexing scheme. At the end of step 3 above, the blocks of data then in main memory can be returned to the same set of blocks in external memory from which they were fetched, provided a table is maintained of where they are kept. Actually, a table is not even necessary — the permutation involved here is a simple index digit

Size	Scratch	Using SSD	
m	Space	Time	MFLOPS
16	207178	0.0824	127.26
17	668266	0.1659	134.34
18	668330	0.2705	174.44
19	1328426	0.5607	177.67
20	1328682	1.1960	175.35
21	2649130	2.4966	176.40
22	2650154	5.1872	177.89

Table 5: Two Pass FFT Performance Results

permutation. However, the ultimate user of the transformed data would also need to use the same indexing mechanism to access the data.

This "two pass" FFT algorithm has been implemented on the Cray Y-MP using SSD. A separate SSD scratch array was used instead of the virtual block scheme just mentioned. This implementation also employed many of the same procedures discussed above to conserve main memory scratch space. The resulting performance figures are shown in table 5. As expected, these results are even higher than the SSD figures in table 4. In fact, the performance figures listed in figure 5 are almost as high as those for Cray's main memory FFT, which are listed in table 2.

Conclusion

The excellent data locality of the four step FFT algorithm and its derivatives clearly is a significant advantage for a number of advanced computer systems. In addition, the fact that most of the computation in these schemes reduces to simultaneous FFTs permits some rather high performance implementations. It has also been demonstrated that some apparent weaknesses of the basic algorithm, such as its large root of unity and scratch space requirements and its reliance on array transpositions, can be largely eliminated by employing some advanced techniques.

The performance figures in tables 4 and 5 show that very large FFTs can be efficiently computed using a Cray Y-MP with SSD. In fact, with 33 million words of main memory and 268 million words of SSD, it should be possible to perform a FFT as large as $2^{27} = 134,217,728$ complex points, provided the SSD device can hold precisely 2^{28} data elements and no fewer. Such favorable results might not be possible on other systems with slower I/O to external memory, but the techniques that have been presented should greatly improve the performance reduction that otherwise occurs.

Another important limiting factor in performing very large FFTs in external memory, which has not been mentioned yet, is the fact that there is often a significant wall clock delay in performing I/O of any sort, even if the CPU time performance is acceptable. Wall clock performance is particularly important when one is using almost all of main memory, so that other jobs cannot be utilizing CPU resources when one's own job is waiting for I/O. Such wall clock delays can be mitigated by overlapping computation and I/O where possible, and by performing several I/O operations concurrently, provided the overall system I/O bandwidth is not a limiting factor. Also, some systems have I/O primitives that allow data to be accessed in external memory in sequences of contiguous blocks with a constant skip distance between blocks. This is exactly the situation in all of the algorithms mentioned above, and thus such an I/O function could be expected to substantially improve the wall clock performance of this FFT scheme, and perhaps the CPU time performance as well.

References

1. Agarwal, R. C., and Cooley, J. W., "Fourier Transform and Convolution Subroutines for the IBM 3090 Vector Facility", *IBM Journal of Research and Development*, vol. 30 (1986), p. 145 - 162.

2. Aggarwal, A., and Vitter, J. S., "The Input/Output Complexity of Sorting and Related Problems", *Communications of the ACM*, vol. 31 (1988), p. 1116 - 1127.

3. Armstrong, J., "A Multi-Algorithm Approach to Very High Performance 1D FFTs", *Journal of Supercomputing*, vol. 2 (1988), p. 415 - 433.

4. Ashworth, M., and Lyne, A. G., "A Segmented FFT Algorithm for Vector Computers", *Parallel Computing*, vol. 6 (1988), p. 217 - 224.

5. Bailey, D. H., "A High-Performance Fast Fourier Transform Algorithm for the Cray-2", *Journal of Supercomputing*, vol. 1 (1987), p. 43 - 60.

6. Bailey, D. H., "A High-Performance FFT Algorithm for Vector Supercomputers", *International Journal of Supercomputer Applications* vol. 2 (1988), p. 82 - 87.

7. Fraser, D., "Array Permutation by Index-Digit Permutation", *Journal of the Association for Computing Machinery*, vol. 23 (1976), p. 298 - 309.

8. Gentleman, W. M., and Sande, G., "Fast Fourier Transforms – For Fun and Profit", *AFIPS Proceedings*, vol. 29 (1966), p. 563 - 578.

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

10. Swarztrauber, P. N., "Multiprocessor FFTs", *Parallel Computing*, vol. 5 (1987), p. 197 - 210.