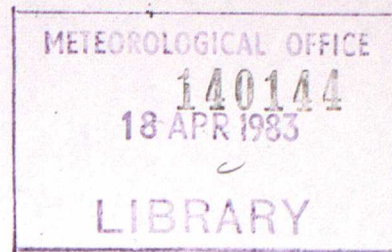


Library



MET. O. 11 TECHNICAL NOTE NO. 170

FAST FOURIER TRANSFORMS ON THE CYBER 205

by

C. Temperton

Met. O. 11
Meteorological Office
London Road
Bracknell
Berkshire, U.K.

March 1983

N.B. This paper has not been published. Permission to quote from it must be obtained from the Assistant Director of the above Meteorological Office Branch.

FH2A

FAST FOURIER TRANSFORMS ON THE CYBER 205

Clive Temperton
Meteorological Office
Bracknell, Berkshire, U.K.

1. Introduction

The Fast Fourier Transform (FFT) is one of the most widely used algorithms in computational physics. In a recent paper [11], the author presented a unified derivation of the algorithm and its many variants, and outlined some of its uses in the field of numerical weather prediction. The specialization of the algorithm to the real/half-complex case was described in [12].

The increasing use of vector machines for large-scale scientific computation has had considerable impact on the development of numerical algorithms, including the FFT. A detailed study of the implementation of FFT's on the Cray-1 was presented in [10]; Swarztrauber [9] has described the problem of vectorizing the FFT in more general terms, but with particular reference to the Cray-1. Wang [13] also considered the problem in a rather general way. Implementation on the CDC STAR-100 has been described by Korn and Lambiotte [7] and Fornberg [3].

In this paper we consider the vectorization of the FFT algorithm on the CDC Cyber 205, a machine which is a direct descendant of the STAR-100. Section 2 outlines some relevant features of the 205, and in particular some important ways in which it differs from the Cray-1. In Sections 3 and 4, various ways of vectorizing the complex radix-2 algorithm are studied, and in Section 5 these are extended to the radix-4 case. Section 6 summarizes the relationship between the vectorization schemes developed here for the Cyber 205 and those developed in [10] for the Cray-1. Section 7 describes a package for multiple mixed-radix real/half-complex transforms developed for a numerical weather prediction model on the Cyber 205 at the U.K. Meteorological Office [3], and compares its performance with that of a similar package developed for use on the Cray-1 at ECMWF. Conclusions are presented in Section 8.

2. The Cyber 205

For a detailed description of the architecture and technology of the Cyber 205 (and other vector machines), a very useful reference is the book by Hockney and Jesshope [6]. From the point of view of implementing the FFT algorithm, there are several important differences between the Cyber 205 and the Cray-1.

First, the definition of a vector is different. On the Cyber 205 the elements of a vector must be stored contiguously in memory, while on the Cray-1 they can be separated by any constant increment.

Second, all arithmetic on the Cyber 205 is memory-to-memory, in contrast to the register-to-register arithmetic on the Cray-1. (Operands in Cray-1 arithmetic must first be loaded from memory, and results stored; temporary results can be held in the vector registers, and the memory transfers can take place in parallel with the arithmetic).

For 64-bit arithmetic, the maximum possible computation rate (expressed in millions of floating-point operations per second, or megaflops) is similar on the two machines: up to 100 megaflops on the (two-pipe) Cyber 205 and 75 megaflops on the Cray-1 for additions or multiplications, or 200 and 150 megaflops respectively for triadic operations of the form $\underline{a} = \underline{b} * (\underline{c} + \underline{d})$ or $\underline{a} = \underline{b} * \underline{c} + \underline{d}$. Here one of the operands must be a scalar on the Cyber 205, but all three may be vectors on the Cray-1. The Cyber 205 also has provision for 32-bit floating-point arithmetic, which can proceed at up to twice these rates, i.e. up to 400 megaflops for triadic operations.

On any vector machine, the time taken for a vector instruction can be expressed as $T = a + bN$, where a is a start-up time, b is the arithmetic rate and N is the length of the vector. An important difference between the Cray-1 and the Cyber 205 is in the start-up time a ; it is less than 10 machine cycles on the Cray-1, but more than 50 cycles on the Cyber 205. (A machine cycle is 12.5 ns on the Cray-1, 20 ns on the Cyber 205). An instructive way of looking at the effect of

the vector start-up time is to consider Hockney's parameter $n_{\frac{1}{2}} [6]$, the vector length required to achieve half the maximum performance. The value of $n_{\frac{1}{2}}$ is around 10 for the Cray-1, while on the two-pipe Cyber 205 it is approximately 100 for 64-bit arithmetic (¹⁵⁰~~200~~ for triadic operations) and 200 for 32-bit arithmetic (³⁰⁰~~400~~ for triadic operations). On the Cray-1 the maximum speed is reached for a vector length of 64, while on the Cyber 205 the efficiency continues to increase up to a maximum allowable vector length of 64K-1.

To compensate for the restriction to contiguously stored vectors on the Cyber 205, several means are provided for the efficient rearrangement of data in memory. Two such operations which will be used in this paper are the merge and gather/scatter instructions. If two vectors A and B are merged using a bit string consisting of a pattern of n 1's followed by n 0's, repeated as often as required, then the result vector contains the first n elements of A, followed by the first n elements of B, then the next n elements of A, and so on. (In fact, this regularity is not necessary; the bit string can be completely random). The merge instruction has a start-up time of about 50 cycles, and then delivers 2 64-bit results or 4 32-bit results per cycle (i.e., the same result rate as for addition or multiplication).

The gather and scatter instructions are equivalent to the Fortran statements $A(I)=B(\text{INDEX}(I))$ and $A(\text{INDEX}(I))=B(I)$. Two refinements provided on the Cyber 205 are (a) group gather/scatter, in which whole vectors of contiguous elements are transferred, and (b) gather/scatter using a constant increment, in which case the index array can be dispensed with. These two refinements can be combined; the resulting instruction has a start-up time of 26 cycles, plus 48 cycles for each new group, with a delivery rate the same as that for the merge instruction.

3. Vectorization of the radix-2 algorithm

Let us assume that we need to compute M simultaneous transforms, each of length N. On the Cray-1, provided that M is quite modest (roughly $M > N/10$), the most effective solution is simply to do the transforms in parallel, with a constant

vector length of $M/10$. The details of the FFT indexing structure then become irrelevant from the point of view of vectorization. The data to be transformed can be stored as rows or columns of a two-dimensional array; provided that memory bank conflicts are avoided, either choice is equally efficient. (This is particularly convenient for a two-dimensional transform).

On the Cyber 205, because of the restriction to contiguously stored vectors, the data to be transformed must be stored as columns of a (row-wise indexed) two-dimensional array; in other words, the transforms must be interleaved. A constant vector length of M can then be achieved in exactly the same way as on the Cray-1. Unless M is very large (of order at least several hundred), this will not be very efficient, because of the large value of $n_1/2$. Fortunately, interleaving the transforms allows us to take advantage of the degree to which a single transform is vectorizable. If an operation in the single transform can be done with a vector length n , then the same operation in the case of M interleaved transforms can be done with a vector length Mn . For notational convenience we can therefore consider the vectorization of a single transform.

Suppose that the factors of N have been stored in an array IFAX(1) to IFAX(NFAX), and that a complex array of trigonometric function values has been defined by

$$\text{TRIGS}(K+1) = \exp(2iK\pi/N), \quad 0 \leq K \leq N-1$$

The data to be transformed is in an array A , and a work array C is provided. Each array acts alternately as input or output for successive stages of the algorithm. The decimation-in-frequency form can be expressed by the following simple program:

```

C   DECIMATION IN FREQUENCY
      COMPLEX A(N),C(N),TRIGS(N)
      INTEGER IFAX(NFAX)

      LA=1
      DO 10 I=1,NFAX

```



```

      IFAC=IFAX(I)
      CALL PASS(A,C,TRIGS,IFAC,LA,N)
C    [now reverse roles of A and C]
      LA=LA*IFAC
10 CONTINUE
      STOP
      END

```

Use of the decimation-in-time variant will change the details of the following discussion, but will not affect the conclusions.

At the risk of reinforcing the myth that only the radix-2 version of the FFT is worth considering, we will use this simple case as an example of the indexing structure in the subroutine PASS. The following corresponds to the "self-sorting" [11] or "Stockham" version of the algorithm:

```

C    SELF-SORTING, DECIMATION IN FREQUENCY FOR IFAC=2
      SUBROUTINE PASS(A,C,TRIGS,IFAC,LA,N)
      COMPLEX A(N),C(N),TRIGS(N)
C
      MU=N/IFAC
      IA=1; IB=MU+1; JA=1; JB=LA+1
      I=0; J=0; JUMP=(IFAC-1)*LA
      DO 20 K=1,MU,LA
      DO 10 L=1,LA
      C(JA+J)=A(IA+I)+A(IB+I)
      C(JB+J)=TRIGS(K)*(A(IA+I)-A(IB+I))
      I=I+1
      J=J+1
10 CONTINUE

```



```

      J=J+JUMP
20 CONTINUE
      RETURN
      END

```

Clearly the innermost (DO 10) loop is vectorizable, with an increment of 1 between the elements of each vector. There are four vectors, with starting addresses A(IA), A(IB), C(JA), C(JB) (each of these is of course a complex vector, corresponding to two real vectors). The trigonometric function value TRIGS(K) is a scalar within the loop. The vector length is LA, which is 1 during the first pass, and (in the radix-2 case) doubles at each successive pass. In the multiple case the vector length is M during the first pass, increasing to MN/2 by the last pass. Apart from the fact that it uses decimation in frequency rather than in time, this is the multiple Stockham algorithm as implemented by Korn and Lambiotte [7].

Pease [8] developed a version of the FFT algorithm with a different indexing structure; unlike the self-sorting algorithm, it delivers the results in scrambled (bit-reversed) order. The outer structure is the same as for the algorithm described above, but the subroutine PASS becomes simpler:

```

C   PEASE, DECIMATION IN FREQUENCY FOR IFAC=2
      SUBROUTINE PASS(A,C,TRIGS,IFAC,LA,N)
      COMPLEX A(N),C(N),TRIGS(N)
      MU=N/IFAC
      IA=1; IB=MU+1; JA=1; JB=2
      I=0; J=0
      DO 10 K=1,MU
      C(JA+J)=A(IA+I)+A(IB+I)
      C(JB+J)=TRIGS(K)*(A(IA+I)-A(IB+I))
      I=I+1
      J=J+2

```


1Ø CONTINUE

RETURN

END

In this case a different TRIGS array is required for each pass. The two vectors $A(IA+I)$, $A(IB+I)$ still have increments of 1, but $C(JA+J)$ and $C(JB+J)$ now have increments of 2. On the Cray-1 they are nevertheless valid vectors, and as shown in [10] this is the best way to vectorize a single transform on the Cray-1 if the final unscrambling step is not counted as part of the cost of the transform. (In some applications it can be dispensed with). However, on the Cyber 205 $C(JA+J)$ and $C(JB+J)$ are not valid vectors. The remedy is to modify the inner loop as follows:

DO 1Ø K=1,MU

$C(IA+I) = A(IA+I) + A(IB+I)$

$C(IB+I) = \text{TRIGS}(K) * (A(IA+I) - A(IB+I))$

$I = I + 1$

1Ø CONTINUE

and to follow it by a permutation step. It is easy to see that the required vector of length N can be assembled by alternately picking successive elements from the vectors of length $N/2$ starting at $C(IA)$ and $C(IB)$, so the merge instruction can be used. In the multiple case, the Pease algorithm can be carried out with vector lengths of $MN/2$ for the arithmetic loop and MN for the merge instruction. This corresponds to the implementation described by Korn and Lambiotte [7] and Wang [13].

As mentioned above, the disadvantage of the Pease algorithm is that the results appear in scrambled order; considerable effort is devoted in [7] and [13] to the vectorization of the unscrambling step. A little thought shows that the modified code given above can equally well be used to implement the Stockham algorithm. As in the case of the Pease algorithm vectorized for the Cyber 205,

a different TRIGS array is required for each pass. In this case we also need a different bit string in each pass for the merge step; the required result vector consists of LA elements from C(IA), then LA elements from C(IB), then the next LA elements from C(IA), and so on. Again the vector length in the multiple case is $MN/2$ for the arithmetic loop and MN for the merge instruction. This corresponds to the implementation of the Stockham algorithm described by Fornberg [4] and Wang [13].

We have just shown that the Stockham (self-sorting) algorithm can be vectorized as efficiently as the Pease algorithm; since the Stockham algorithm also eliminates the unscrambling step, there is no need to consider the Pease algorithm further. The question which remains to be resolved concerns the relative advantages of the two distinct ways of implementing the Stockham algorithm: the original scheme with shorter vectors, or the modified scheme which has longer vectors but requires a merge operation after each pass. In the next section, these two schemes will be analysed and compared.

4. Analysis of schemes for the radix-2 Stockham algorithm

Let us denote by Scheme C the first procedure described above, using a nested loop with shorter vectors, and by Scheme D the second procedure using a single loop followed by a merge operation. It is immediately clear that Scheme C is much more economical in terms of the storage of trigonometric function tables; since $TRIGS(K)$ is a scalar within the inner loop, only $N/2$ complex (N real) values are required for the whole algorithm. In Scheme D, $TRIGS(K)$ is a vector within the loop, and so an array of MN real values is required for each pass, where M is the number of transforms being performed together. Moreover, since a different TRIGS array is required for each pass (except the last, where $TRIGS(K)=1$ for all K and the multiplication can be omitted), a total of $MN(\log_2 N - 1)$ trigonometric values must be stored if none of them are to be recalculated and the array is to be reusable. In the following analysis, it is (perhaps rather generously) assumed that enough storage is available for Scheme D to be implemented in this way.

The fact that $\text{TRIGS}(K)$ is a scalar in Scheme C but a vector in Scheme D also has important consequences in terms of the amount of vector arithmetic required. The Scheme D loop requires 2 complex additions and 1 complex multiplication, a total of 6 real vector additions and 4 real vector multiplications. In Scheme C, each of these multiplications can be linked with an addition using triadic operations. In terms of real arithmetic, the complex statement

$$C(JB+J) = \text{TRIGS}(K) * (A(IA+I) - A(IB+I))$$

would normally be written in the form

$$\underline{c}' = \underline{a}_0 - \underline{a}_1$$

$$\underline{d}' = \underline{b}_0 - \underline{b}_1$$

$$\underline{c} = \cos\theta * \underline{c}' - \sin\theta * \underline{d}'$$

$$\underline{d} = \sin\theta * \underline{c}' + \cos\theta * \underline{d}'$$

where $\underline{c} + i\underline{d}$ corresponds to $C(JB+J)$, $\cos\theta + i\sin\theta$ to $\text{TRIGS}(K)$, $\underline{a}_0 + i\underline{b}_0$ to $A(IA+I)$, and $\underline{a}_1 + i\underline{b}_1$ to $A(IB+I)$.

This sequence can be rewritten as four triadic operations:

$$\underline{c}'' = \cos\theta * (\underline{a}_0 - \underline{a}_1)$$

$$\underline{d}'' = \cos\theta * (\underline{b}_0 - \underline{b}_1)$$

$$\underline{c} = \underline{c}'' - \tan\theta * \underline{d}''$$

$$\underline{d} = \tan\theta * \underline{c}'' + \underline{d}''$$

The only snag here is that $\tan\theta$ may be infinite; to minimize round-off error, if $|\tan\theta| > 1$ the sequence can be rewritten using $\sin\theta$ and $\cot\theta$. The inner loop of Scheme C can thus be implemented in 2 real vector additions and 4 triadic operations.

We will now examine the time required by each scheme for a single pass. We assume that the real and imaginary parts of the complex numbers are stored in separate arrays; the number of vector start-ups could be reduced slightly by rearranging the data, but this will not greatly affect the following analysis.

In Scheme C, the inner loop is performed $N/(2*IA)$ times. For $K=1$, $\text{TRIGS}(K)=1$ and the multiplication can be omitted; the loop simply contains 4 real vector

additions. For $K > 1$, the loop contains 2 real vector additions and 4 triads. Assuming start-up times of 50 cycles for an addition and 75 cycles for a triad, the total time spent in vector start-ups is

$$200(N/LA - 1) \text{ cycles}$$

The additional time spent on arithmetic (for M simultaneous transforms) is

$$M(3N/2 - LA) \text{ cycles}$$

including the saving during the first pass through the inner loop.

In Scheme D, there are 10 real arithmetic operations of vector length $MN/2$, and two merges of length MN (one each for the real and imaginary parts). We can in fact omit the redundant complex multiplications as in Scheme C, so the vector length is reduced to $M(N/2 - LA)$ for 6 of the arithmetic operations. The time spent in vector start-ups is 600 cycles, and the time spent on arithmetic and merges is

$$M(7N/2 - 3*LA) \text{ cycles}$$

Thus Scheme C will be faster than Scheme D if

$$200N/LA < 800 + 2M(N - LA)$$

i.e. if

$$M > 100\left(\frac{1}{LA} - \frac{3}{N-LA}\right)$$

Scheme C is at the greatest disadvantage during the first pass, when $LA=1$ and the vector length is only M ; the above analysis shows that Scheme C will still be faster for $M > 100$. During the penultimate pass, when $LA=N/4$, Scheme C will always be faster. During the last pass, both schemes are equivalent since the complex multiplications are all redundant, and no merge operation is required in Scheme D.

In Table 1, we present measured times for $N=64$ and $M=16, 64$ and 256 , for each stage of the transform. The time taken by Scheme C decreases rapidly with successive passes (especially for small M), as the vector length increases and the number of start-ups is reduced. The corresponding time taken by Scheme D decreases slowly, the only saving being the smaller number of complex multiplications required at each stage. Eventually, Scheme C becomes faster than Scheme D; for

M=256 this is true throughout. The results of the above analysis are thus confirmed.

Table 1: Times in μs for each stage of a set of M complex transforms of length $N = 64 = 2^6$

LA	M=16		M=64		M=256	
	C	D	C	D	C	D
1	414	99	492	311	858	1160
2	222	97	314	307	672	1144
4	132	96	220	299	572	1114
8	86	92	172	284	505	1052
16	59	84	136	253	441	929
32	38	38	100	100	346	346

The analysis can be extended to compare the use of Schemes C and D to compute the whole transform. We use the relationships

$$\sum \frac{1}{LA} = 2 - \frac{2}{N}, \quad \sum LA = N-1$$

where the sums are taken over the $\log_2 N$ passes. The savings during the last pass are also taken into account.

Scheme C takes a total of $400N - 200(\log_2 N + 2)$ cycles for vector start-ups, and $1.5MN \log_2 N - M(N-1)$ cycles for arithmetic. Scheme D takes $600 \log_2 N - 400$ cycles for vector start-ups, and $3.5MN \log_2 N - M(4N-3)$ cycles for arithmetic and merges. Overall, Scheme C is faster if

$$M > (400N - 800 \log_2 N) / (2N \log_2 N - 3N + 2)$$

For example if $N=64$, Scheme C is faster for $M \geq 36$; if $N=256$, Scheme C is faster for $M \geq 29$.

Notice also that if M is large enough, the contribution of the vector start-ups can be ignored. In the limit, Scheme C is more than twice as fast as Scheme D.

Table 2: Total times in μs for M complex transforms of length $N=64=2^6$

M	C	D	D+C
16	951	506	475
32	1102	854	755
84	1434	1554	1246
128	2084	2950	2061
256	3394	5745	3394

On the Cray-1, the fastest way to compute a single transform (with input and output correctly ordered) is to use one vectorization scheme initially, and to switch to a different scheme during the later stages of the transform [10]. The same is true here; we can use Scheme D initially, and switch to Scheme C as LA increases. Table 2 presents the total time for M transforms of length $N=64$ using Schemes C, D and the combined scheme in which the fastest method is used at each stage. For $M=16$ the switch is made halfway through the process (i.e. after 3 passes), while for $M=256$ the fastest method is to use Scheme C throughout.

The analysis presented in this Section can easily be extended to the case of 32-bit arithmetic; the time spent on vector start-ups is the same as for 64-bit arithmetic, while the time spent on arithmetic is halved. For a single pass, Scheme C will be faster than Scheme D if

$$M > 200 \left(\frac{1}{LA} - \frac{3}{N-LA} \right)$$

Thus for the first pass ($LA=1$), Scheme C will be faster for $M > 200$. For the penultimate pass ($LA=N/4$) Scheme C will always be faster.

For the whole transform, Scheme C will be faster if

$$M > (400N - 800 \log_2 N) / (N \log_2 N - 1.5N + 1)$$

i.e. the break-even value of M is doubled over that for 64-bit arithmetic. It remains true in 32-bit arithmetic that Scheme C is asymptotically more than twice as fast as Scheme D.

5. The radix-4 algorithm

The subroutine PASS given in Section 3 can be generalized [11] to factors other than 2. In this section we consider the extension of the two vectorization schemes described in Section 4 to the radix-4 case. For IFAC=4, subroutine PASS is as follows :

C SELF-SORTING, DECIMATION IN FREQUENCY FOR IFAC=4

SUBROUTINE PASS (A,C,TRIGS,IFAC,LA,N)

COMPLEX A(N),C(N),TRIGS(N)

MU=N/IFAC

IA=1; IB=IA+MU; IC=IB+MU; ID=IC+MU;

JA=1; JB=JA+LA; JC=JB+LA; JD=JC+LA;

I=0; J=0; JUMP=(IFAC-1)*LA

DO 20 K=0,MU-LA,LA

DO 10 L=1,LA

$$\begin{bmatrix} C(JA+J) \\ C(JB+J) \\ C(JC+J) \\ C(JD+J) \end{bmatrix} = \Omega(K) * W_4 * \begin{bmatrix} A(IA+I) \\ A(IB+I) \\ A(IC+I) \\ A(ID+I) \end{bmatrix}$$

I=I+1

J=J+1

10 CONTINUE

J=J+JUMP

20 CONTINUE

RETURN

END

Here W_4 is the DFT matrix of order 4, and $\Omega(K)$ is a diagonal matrix defined by

$$\Omega(K) = \text{diag}(\text{TRIGS}(1), \text{TRIGS}(K+1), \text{TRIGS}(2K+1), \text{TRIGS}(3K+1))$$

Again the innermost loop is vectorizable, with an increment of 1. The vector length in successive passes is 1, 4, 16, ...; in the multiple case this becomes $M, 4M, 16M, \dots$. This version corresponds to vectorization Scheme C; however, some of the computation can be done with longer vectors. If we expand the code given above to use explicitly the algorithm for multiplication by W_4 , the inner loop becomes

```

DO 10 L=1, LA
  T1(I)=A(IA+I)+A(IC+I)
  T2(I)=A(IB+I)+A(ID+I)
  T3(I)=A(IA+I)-A(IC+I)
  T4(I)=A(IB+I)-A(ID+I)

  [ C(JA+J) ]   [ T1(I)+T2(I) ]
  [ C(JB+J) ]   [ T3(I)+iT4(I) ]
  [ C(JC+J) ]   [ T1(I)-T2(I) ]
  [ C(JD+J) ]   [ T3(I)-iT4(I) ]
           = Ω(K)*

  I=I+1
  J=J+1
10 CONTINUE

```

The statements which evaluate the temporary vectors T1 to T4 can be moved outside the nested loop structure, and implemented as four complex vector additions of length $N/4$; in fact by locating T2 and T4 immediately after T1 and T3 respectively, they can be implemented as two complex vector additions of length $N/2$.

The radix-4 version of Scheme C requires a table of only $3N/4$ complex trigonometric function values, and the complex multiplications by $\Omega(K)$ can again be implemented using triadic operations.

Computing the temporary vectors $T1/T2$ and $T3/T4$ takes 200 cycles for vector start-ups and MN cycles for arithmetic. The inner loop for the remaining computation is performed $N/(4*LA)$ times. For $K=0$, $\Omega(K)$ is the identity matrix, and the multiplication can be omitted; the loop simply contains 8 real vector additions. For $K > 0$, the loop contains 2 real vector additions and 12 triads. The total time spent in vector start-ups in this part of the computation is

$$250N/LA - 600 \text{ cycles}$$

while the additional time spent on arithmetic is

$$M(7N/4 - 3*LA) \text{ cycles}$$

Thus the total time for one pass using Scheme C is

$$250N/LA - 400 \text{ cycles}$$

for start-ups, and

$$M(2.75N - 3*LA) \text{ cycles}$$

for arithmetic.

As in the radix-2 case, Scheme D is obtained by storing the results in locations $C(IA)$, $C(IB)$, $C(IC)$, $C(ID)$ and collapsing the nested loop to a single loop with vector length $N/4$, though as for Scheme C some of the computation can be done with vector length $N/2$. An array of $1.5MN$ real trigonometric function values is required for each pass, and the complex multiplications cannot be implemented by triadic operations. Multiplication by W_4 takes four real additions of length $MN/2$ and 8 real additions of length $MN/4$. Multiplication by $\Omega(K)$ takes 4 real multiplications and 2 real additions, all of length $3M(N-LA)/4$ if multiplications of $C(IB)$ by 1 are avoided.

This part of the computation thus takes 300 cycles for vector start-ups, plus $M(4.25N - 2.25*LA)$ cycles for arithmetic.

The second part of each pass of Scheme D consists of a permutation of the results obtained in the first part. The required result vector consists of groups of $M*LA$ elements taken in rotation from $C(IA)$, $C(IB)$, $C(IC)$ and $C(ID)$. There are

three ways of performing this permutation. Merging the first and second halves of the array C using a bit string consisting of 1's and 0's in alternating groups of length $M \cdot LA$, and then repeating this operation a second time, will give the required result. Applying this technique to both the real and imaginary parts takes 4 merges of length MN , requiring 200 cycles for start-ups and $2MN$ cycles for delivery of the results.

The permutation can also be achieved using the "group scatter" instruction; altogether there are $2N/LA$ groups, each of length $M \cdot LA$. However, it turns out to be slightly faster simply to use a loop of "move" instructions, taking $100N/LA$ cycles for start-ups and MN cycles for delivery. "Multiple move" is faster than "double merge" if

$$100N/LA + MN < 200 + 2MN$$

i.e. if

$$M > 100 \left(\frac{1}{LA} - \frac{2}{N} \right)$$

If $M > 100$, then the multiple move technique is faster even in the worst case $LA=1$.

As in the radix-2 case, a detailed comparison can be made of Schemes C and D for each pass. To summarize the results, even in the case $LA=1$ Scheme C will be faster than Scheme D if $M > 70$. For the last pass, Schemes C and D are equivalent; no complex multiplications are required, and in Scheme D no permutation of the results is needed.

In Table 3 we present measured times for $N=64$ and $M=16, 64$ and 256 , for each stage of the transform. Schemes D_1 and D_2 use "double merge" and "multiple move" respectively for the permutation steps. The results confirm that as either LA or M increases, D_2 becomes faster than D_1 , and C becomes faster than either.

For the whole transform, Scheme C takes approximately $330N - 400 \log_4 N$ cycles for start-ups and $2.75MN \log_4 N - MN$ cycles for arithmetic, while Scheme D (with the

Table 3: Times in μs for each stage of a set of M complex transforms of length $N=64=4^3$

LA	M=16			M=64			M=256		
	C	D ₁	D ₂	C	D ₁	D ₂	C	D ₁	D ₂
1	515	167	257	680	545	578	1350	2062	1856
4	170	161	166	327	525	478	956	1985	1722
16	68	68	68	191	191	191	683	683	683

permutations implemented by multiple moves) takes approximately $900 \log_4 N + 130N$ cycles for start-ups and $5.25MN \log_4 N - 3.5MN$ cycles for arithmetic. Overall, Scheme C is faster if

$$M > (230N - 1300 \log_4 N) / 2.5N(\log_4 N - 1)$$

For example, if $N=64$, Scheme C is faster for $M \geq 34$; if $N=256$, Scheme C is faster for $M \geq 28$. These break-even points are almost the same as for the radix-2 algorithm. Asymptotically, Scheme C is almost twice as fast as Scheme D.

As in the radix-2 case we can use Scheme D initially, and switch to Scheme C as LA increases. Table 4 presents the total time for M transforms of length $N=64$ using Schemes C, D₁ and D₂ and the combined scheme in which the fastest method is used at each stage. For $M=128$ and $M=256$ the fastest method is to use Scheme C throughout.

Table 4: Total times in μs for M complex transforms of length $N=64=4^3$

M	C	D ₁	D ₂	D+C
16	753	396	491	396
32	903	682	746	624
64	1198	1261	1247	1064
128	1801	2417	2252	1801
256	2989	4730	4261	2989

Comparing Table 4 with Table 2, notice finally that the radix-4 algorithm, however implemented, is always significantly faster than its radix-2 counterpart, in contrast to the experience of Korn and Lambiotte [7].

6. Relationship between Cray-1 and Cyber 205 vectorization schemes

In [10], two distinct vectorization schemes, A and B, were developed for computing a single transform on Cray-1. The best approach was found to consist of a combination in which one or two passes were carried out using Scheme B, and the remainder using Scheme A. For M simultaneous transforms, a simple vectorization scheme was developed in which the transforms were done in parallel. It was found that only a very modest value of M was needed to make this the fastest approach.

On the Cyber 205, with its much larger value of $n_1/2$ [6], computing a single transform of length N via the FFT would necessarily be slow and inefficient unless N was very large indeed. In this paper we have developed two schemes, C and D, for the vectorization of M simultaneous transforms. Both of these schemes may be regarded as extensions to the multiple case of the Cray-1 Scheme A.

Unfortunately there is no corresponding multiple analogue on the Cyber 205 of the Cray-1 Scheme B, which might be used to increase the vector length during the early stages of the transform. Scheme B is developed by turning the nested loop structure of subroutine PASS inside-out, and it works on the Cray-1 despite the fact that the elements of the resulting vectors are no longer contiguously stored. The only way to implement this scheme on the Cyber 205 would be to shuffle the data before and after each pass, and the results of the previous sections suggest that this would not be worthwhile.

One disadvantage of using the self-sorting variant of the FFT to compute M simultaneous transforms of length N is that a work space of length MN is required. On the Cray-1 this should rarely be a problem, since there is nothing to be gained by increasing M beyond 64; nevertheless, as pointed out in [10], it is possible to eliminate the work space (at the expense of extra memory references) by using the Cooley-Tukey [2] or Gentleman-Sande [5] variants of the FFT algorithm.

On the Cyber 205 we really want to use as large a value of M as possible in order to maximize the vector length, and the availability of work space is more likely to be a problem. As in the self-sorting case, the FFT variants of [2] and [5] could be implemented with a vector length $M*LA$, but unfortunately on the Cyber 205 this would still require a similar amount of work space. The reason lies in the memory-to-memory arithmetic; while on the Cray-1 temporary results can be held in the vector registers, on the Cyber 205 extra memory must be provided for them, and it is not difficult to see that if a vector length of $M*LA$ is to be achieved then work space of order MN must be provided, and we might as well have used the self-sorting variant in the first place. Thus it seems that eliminating the work space on the Cray-1 is possible but rarely necessary, while on the Cyber 205 it is highly desirable but impossible.

7. A real transform package

In this section we report on a multiple real FFT package implemented on the Cyber 205 for the operational numerical weather prediction model at the U.K. Meteorological Office [3]. From a mathematical point of view, the package is identical to that described in [12], which was implemented on a Cray-1. The transform length is restricted to values of the form $N=2^p 3^q 5^r$. In order to decrease the operation count, there is scope for grouping the factors together, so that in addition to the factors 2, 3 and 5 there are sections of coding for factors 4, 6 and 8. As in [12], the algorithm for a real/half-complex transform of the form

$$x_j = \sum_{k=0}^{N-1} c_k \exp(2ijk\pi/N) \quad (1)$$

or its inverse

$$c_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j \exp(-2ijk\pi/N) \quad (2)$$

where the x_j 's are real and the c_k 's satisfy $c_{N-k} = c_k^*$, is derived by pruning out redundant operations from the corresponding full complex transform of length N .

A saving of 20% or so in the operation count is thereby obtained over the more conventional techniques described in [1].

The approach to vectorizing the complex transform denoted by Scheme C in preceding sections translates directly to the specialized real-half-complex transform algorithm. The entire transform can be carried out using an array of trigonometric function values of length N . All complex multiplications can be implemented using triadic operations. The vector length is in general $LA \cdot M$, though in fact since it is convenient in this algorithm to interleave real and imaginary parts of complex numbers in blocks of length $LA \cdot M$ [12], it is frequently possible to handle real and imaginary parts together, thus doubling the vector length. Because of the more complicated indexing, the trick used in the radix-4 algorithm of Section 5, whereby some of the computation can be done using longer vectors, cannot readily be translated to the real/half-complex case.

Scheme D could also be translated to this case, though the more complicated structure of the real/half-complex transform algorithm would cause considerable difficulty and loss of efficiency. As in the complex case, enormous arrays of trigonometric function values and bit strings would be required. It is clear from the analysis of the previous sections that Scheme D would only be competitive for a rather small number of simultaneous transforms, and this approach was not pursued.

The real FFT package for the Cyber 205 was first written in straightforward vector Fortran, and subsequently translated into "special call" format using Q8 calls for all vector instructions and descriptor manipulations. This gave some speed advantage at short vector lengths, but the main reason was to permit the use of 32-bit arithmetic before a suitable compiler became available. The timing information presented here refers to the special call version, and to the forward transform, Eq. (1). The inverse transform, Eq. (2), runs slightly slower because of some multiplications which could not readily be linked with additions in this case.

Comparative figures are given for the FFT package described in [12] and run on the Cray-1, using the simple multiple vectorization scheme with a vector length of M. It should be noted that the Cray-1 package was written in CAL (Cray Assembly Language), and runs about twice as fast as a corresponding vectorized Fortran version. 32-bit arithmetic is not available on Cray-1.

Table 5 presents the time per real transform (in microseconds) for three values of N, and for four values of M, the number of transforms being performed together. Results for N=180, 192 and 200 are shown to demonstrate that the choice of N is not as critical as is sometimes thought for efficiency of the FFT. Values of M=16, 64, 256 and 1024 are shown to demonstrate the increasing efficiency of long vectors on the Cyber 205. in contrast to the Cray-1 where there is no advantage in increasing the vector length beyond 64.

Table 5: Time per real transform in μs

N	M	Cyber 205 (64-bit)	Cyber 205 (32-bit)	Cray-1 (64-bit)
180 (5×6^2)	16	87	78	36
	64	35	26	26
	256	22	13	26
	1024	18	10	26
192 ($4 \times 6 \times 8$)	16	81	72	36
	64	33	24	25
	256	21	12	25
	1024	18	9	25
200 ($5^2 \times 8$)	16	98	88	41
	64	40	30	30
	256	25	15	30
	1024	21	11	30

Notice that 32-bit arithmetic is asymptotically twice as fast as 64-bit arithmetic on the Cyber 205, but because the start-up times for each vector operation (and other overheads) remain the same, many transforms must be performed together before this ratio is approached.

Some timing results (not shown) were also obtained on the Cyber 205 with the vector link instruction suppressed. If many transforms are performed together, the times are then about 50% longer, as expected from the ratio of multiplications to additions in the algorithm. For fewer simultaneous transforms, the effects of suppressing the link instruction are again less marked.

Table 6: Megaflop rates for real FFT packages

N	M	Cyber 205 (64-bit)	Cyber 205 (32-bit)	Cray-1 (64-bit)
180 (5×6^2)	16	32	35	74
	64	80	106	104
	256	129	214	104
	1024	152	277	104
192 ($4 \times 6 \times 8$)	16	32	35	71
	64	78	105	100
	256	124	207	100
	1024	145	274	100
200 ($5^2 \times 8$)	16	34	37	79
	64	83	111	109
	256	132	220	109
	1024	155	293	109

Comparing times on the Cyber 205 with those on the Cray-1, at $M=16$ the Cray-1 is markedly faster. At $M=64$, 32-bit arithmetic on the Cyber matches 64-bit arithmetic on the Cray. At $M=256$, 64-bit arithmetic on the Cyber beats that on the Cray, while at $M=1024$ 32-bit arithmetic on the Cyber is almost three times as fast as 64-bit arithmetic on the Cray.

It is also of interest to compare the performance achieved in terms of megaflops. As detailed in Section 2, for a computation in which there are as many multiplications as additions (and assuming on the Cyber 205 that they can all be linked), the machine architectures impose maximum limits on the Cray-1 of about 150 megaflops, and on the (two-pipe) Cyber 205 of 200 megaflops for 64-bit arithmetic, or 400 megaflops for 32-bit arithmetic. For computations in which the mix of additions and multiplications is in the ratio 2:1, the corresponding limits are about 110, 150 and 300 megaflops respectively. Table 6 shows that these limits are very nearly achieved in the FFT packages, but that many transforms must be performed together on the Cyber 205 before maximum performance is approached.

8. Conclusions

In this paper we have shown that the multiple Stockham (self-sorting) FFT algorithm can be implemented on the Cyber 205 as efficiently as the multiple Pease algorithm, and is preferable since it eliminates the need for reordering the data. There are two ways to implement the Stockham algorithm, denoted here by Schemes C and D. Scheme D is faster for a small number of transforms and during the early stages of the algorithm, but requires a large amount of storage for trigonometric function values. Scheme C requires only N such values, and is asymptotically about twice as fast as Scheme D. It was also shown that radix-4 transforms were significantly faster than radix-2.

Timing results were presented for a multiple real transform package based on Scheme C, and compared with corresponding results for the Cray-1. In 64-bit arithmetic, the Cyber 205 is faster than the Cray-1 if more than about 200 transforms

can be performed in parallel. 32-bit arithmetic on the Cyber 205 is faster than 64-bit arithmetic on the Cray-1 for more than 64 simultaneous transforms.

From a theoretical point of view, the Cray-1 and Cyber 205 may be considered equivalent since both are pipelined vector computers. However, comparing the results of this paper with those of [10], and considering the different approaches found to be appropriate in each case, it is clear that in some respects the two machines are quite dissimilar in practice. Important differences arise not just because of the different value of the parameter $n_{\frac{1}{2}}$, as discussed by Hockney and Jesshope [6], but also because of the difference between memory-to-memory and register-to-register arithmetic, and because of the different definitions of a vector on the two machines.

References

- [1] J.W. COOLEY, P.A.W. LEWIS & P.D. WELCH, The Fast Fourier Transform algorithm : programming considerations in the calculation of sine, cosine and Laplace transforms, J. Sound Vib. 12 (1970), 315-337.
- [2] J.W. COOLEY & J.W. TUKEY, An algorithm for the machine calculation of complex Fourier series, Math. Comp. 19 (1965), 297-301.
- [3] M.J.P. CULLEN, Current progress and prospects in numerical techniques for weather prediction models, to appear in J. Comp. Phys.
- [4] B. FORNBERG, A vector implementation of the Fast Fourier Transform algorithm, Math. Comp. 36 (1981), 189-191.
- [5] W.M. GENTLEMAN & G. SANDE, Fast Fourier Transforms - for fun and profit, Proc. AFIPS Joint Computer Conference 29 (1966), 563-578.
- [6] R.W. HOCKNEY & C.R. JESSHOPE, "Parallel Computers", Adam Hilger Ltd., Bristol, U.K., 1981.
- [7] D.G. KORN & J.J. LAMBIOTTE, Computing the Fast Fourier Transform on a vector computer, Math. Comp. 33 (1979), 977-992.
- [8] M.C. PEASE, An adaptation of the Fast Fourier Transform for parallel processing, J.ACM 15 (1968), 252-264.
- [9] P.N. SWARZTRAUBER, Vectorizing the FFT's, in "Parallel Computations" (ed. G. Rodrigue), Academic Press, 1982.
- [10] C. TEMPERTON, Fast Fourier Transforms and Poisson-solvers on Cray-1, in "Supercomputers", Infotech State of the Art Report, Infotech International Ltd., Maidenhead, U.K., 1979.
- [11] C. TEMPERTON, Self-sorting mixed-radix Fast Fourier Transforms, to appear in J. Comp. Phys.
- [12] C. TEMPERTON, Fast mixed-radix Real Fourier Transforms, submitted to J. Comp. Phys.
- [13] H.H. WANG, On vectorizing the Fast Fourier Transform, BIT 20 (1980), 233-243.