Chapter 10

Fast Fourier Transforms

This chapter describes the CM Fortran interface to the CMSSL Fast Fourier Transforms (FFTs). CMSSL currently provides three types of FFTs:

- complex-to-complex FFT (CCFFT)
- real-to-complex FFT (RCFFT)
- complex-to-real FFT (CRFFT)

In addition, CMSSL provides array conversion utilities that convert a real array to a complex array suitable for input to the RCFFT, and convert the complex array produced by the CRFFT back to a real array.

One section of this chapter is devoted to each of the following topics:

- introduction
- complex-to-complex FFT
- real-to-complex and complex-to-real FFTs
- array conversion utilities
- references

This chapter assumes a basic understanding of Fourier Transforms.

Copyright © 1994 Thinking Machines Corporation

10.1 Introduction

Each FFT provided by CMSSL can be used for forward and inverse transforms, and can perform a transform on any selected set of array axes. The following table compares the three types of FFTs.

Table 6. Comparison of the CCFFT, RCFFT, and CRFFT

	CCFFT	RCFFT	CRFFT
Number of array axes	Any	Any	Any
Number of transform axes	Any	Any*	Any*
Length of transform axes	Power of 2	Power of 2	Power of 2
Length of non-transform axes	Any	Any	Any
Input data type	Complex	Real embedded in complex	Complex
Output data type	Complex	Complex	Real embedded in complex
Input ordering	Any	Any	Any
Output ordering	Any	Any	Any
Forward	Twiddles with negative exponent	Twiddles with negative exponent	Twiddles with negative exponent
Inverse	Twiddles with positive exponent	Twiddles with positive exponent	Twiddles with positive exponent
Layout directives	Any	Any	Any

* The RCFFT and CRFFT are performed along one axis only. The CCFFT can be performed along up to $n-1$ axes, if desired.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

10.1.1 The CMSSL FFT Library Calls

CMSSL provides two FFT user interfaces:

- The *Simple FFT* (available for the CCFFT only). This interface is used to transform a data set in the same direction along all axes. You can use the simple FFT to perform multidimensional FFTs, but not to perform multiple independent FFTs concurrently. For multiple instances, you must be able to specify instance axes (axes along which no transform is performed).
- The *Detailed FFT* (available for all FFT types). This interface provides (at the cost of added interface complexity) a great deal of flexibility, including support for multiple instances. For each axis of a multidimensional array, the Detailed FFT allows you to choose the following:
  - Whether a transform is performed along the axis, and if so, in which direction (forward or inverse).
  - The address ordering (normal or bit-reversed) used to store input and output data values along the axis.
  - The scaling factor (none, axis length, or square root of axis length) applied to the transform results. At the end of the FFT, each element of the array is divided by the product of the scaling factors. (The axes for which you specified "none" as the scaling factor do not contribute to this product.)

These features are described in the man pages later in this chapter.

To perform an FFT, you must follow these steps:

1. Call the **fft_setup** routine.

   This routine computes internal values (including twiddle factors), allocates an internal FFT setup object, and computes an *FFT setup ID*, which you must then use as an argument in all subsequent Simple FFT, Detailed FFT, and deallocation calls associated with this setup call.

2. Call the **fft** or **fft_detailed** routine.

   These routines execute the Simple and Detailed FFTs, respectively. You may follow one call to the setup routine with multiple calls to these routines. However, the result of a call to the setup routine can

Copyright © 1994 Thinking Machines Corporation

only be used for arrays with the same rank, axis extents, layout directives, and data type as the array supplied in the setup routine. If any of these parameters change, you must call **fft_setup** again to create another setup structure.

3. Call the **deallocate_fft_setup** routine.

   This routine deallocates a specified FFT setup strucure and frees the CM storage space it required.

You may have more than one setup ID active at a time; that is, you may call the setup routine more than once before deallocating any setup IDs. When you call **fft**, **fft_detailed**, or **deallocate_fft_setup**, you must be sure to provide the correct setup ID.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

10.2 Complex-to-Complex FFT

The CMSSL complex-to-complex FFT is a Discrete Fourier Transform in which the same call is used to specify a one-dimensional or multidimensional transform on any legal array of floating-point data. For example, in Fourier Analysis Cyclic Reduction (FACR) for a three-dimensional problem (see references 1, 6, 16, and 18 in Section 10.5), the data array has at least three axes. You may choose to perform Fourier transformation on planes represented by two of the axes, while tridiagonal systems of equations are solved along the third axis.

10.2.1 Butterfly Computations, Twiddle Factors, and the CCFFT Setup Phase

The CMSSL FFT is of the Cooley-Tukey variety (see references 3 and 4). The Discrete Fourier Transform relies on a set of coefficients known as *twiddle factors*. The twiddle factors are roots of unity that depend only upon the size of the data set and the radix. In the CMSSL CCFFT, the twiddle computations are performed during a setup phase and used during the subsequent evaluation phase. This organization allows you to amortize the expense of twiddle factor computation over several transformations.

The sign of the twiddle factor exponents determines the direction of an FFT. In the CMSSL FFT, the *forward* transform is defined as one that uses twiddle factors with a *negative* exponent, and the *inverse* transform is defined as one that uses twiddle factors with a *positive* exponent.

10.2.2 Bit Ordering and Bit Reversal

The Cooley-Tukey FFT bit-reverses the addresses in which it stores results. For example, if an 8-point FFT is performed, then for normal-ordered input, the indices of the output are

0, 4, 2, 6, 1, 5, 3, 7.

In binary, this sequence is

000, 100, 010, 110, 001, 101, 011, 111

that is, the normal sequence with bits reversed.

Copyright © 1994 Thinking Machines Corporation

The Detailed FFT allows you to specify whether the address ordering of the input is normal or bit-reversed, and whether the desired address ordering of the result is normal or bit-reversed.

Optimizing CCFFT Address Ordering

For an axis that is being transformed in a CCFFT,

- If you specify *opposite* address orderings for input and output, the Detailed FFT performs the single bit-reversal that is inherent in the Cooley-Tukey algorithm, and produces output that is bit-reversed relative to the input.
- If you specify the *same* address ordering for input and output, the Detailed FFT performs an *extra* bit reversal, and produces output with the same ordering as the input. The extra bit reversal exacts a performance cost.

For an axis that is not being transformed,

- If you specify *opposite* address orderings for input and output, the Detailed FFT performs a single bit-reversal and produces output that is bit-reversed relative to the input.
- If you specify the *same* address ordering for input and output, the Detailed FFT performs *no* bit reversal.

Thus, for optimal CCFFT performance with the Detailed FFT, specify

- Opposite address orderings for input and output, for each axis that is being transformed.
- The same address ordering for input and output, for each axis that is not being transformed.

Note that violating these recommendations along any axis decreases performance.

The Simple FFT does not provide bit-reversal options; it always produces output data with the same bit ordering as the input data. It does this by automatically performing a bit-reversal on the transformed data, thus annihilating the bit-reversal inherent in the FFT.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

To bit-reverse the address ordering of a dataset without transforming it, call the Detailed FFT specifying no transform for all axes and specifying *opposite* values for the input and output address ordering.

These points are independent of the directions of the transforms.

10.2.3 Multidimensional and Multiple-Instance FFTs

The CMSSL FFT allows you to specify whether you want to perform a forward transform, an inverse transform, or no transform along each axis of a multidimensional CM array. As described later in this chapter, it is also possible to perform bit-reversal on an axis without transforming it. The axes along which you perform a transform or a bit-reversal define the dataset; the axes along which you perform neither a transform nor bit-reversal are the instance axes, representing multiple independent data sets.

The CMSSL FFT routines perform multidimensional and multiple-instance FFTs by combining the following principles:

- If you supply a multidimensional data set, the routines perform a multidimensional FFT by computing a one-dimensional FFT along each axis of the cell in turn.
- If you specify one or more instance axes (possible only with the Detailed FFT, as described below), the routine computes multiple concurrent, independent one-dimensional FFTs along each cell axis in turn.

For example, if your array has rank 3 and extents ($p \times q \times r$), you can perform any of the FFTs indicated in Table 7.

Copyright © 1994 Thinking Machines Corporation

Table 7. Possible FFTs with a three-dimensional input array.

Data Axis Numbers	Instance Axis Numbers	Resulting FFTs
1	2, 3	*qr* concurrent one-dimensional FFTs along the first axis.
2	1, 3	*pr* concurrent one-dimensional FFTs along the second axis.
3	1, 2	*pq* concurrent one-dimensional FFTs along the third axis.
1, 2	3	*r* concurrent two-dimensional FFTs. The dataset for each two-dimensional FFT is defined by the first two axes. The third axis defines the instances.
1, 3	2	*q* concurrent two-dimensional FFTs. The dataset for each two-dimensional FFT is defined by the first and third axes. The second axis defines the instances.
2, 3	1	*p* concurrent two-dimensional FFTs. The dataset for each two-dimensional FFT is defined by the second and third axes. The first axis defines the instances.
1, 2, 3	none	One three-dimensional FFT on the entire array.

10.2.4 Current Restrictions

The current FFT algorithm requires that the length of each axis subject to transformation or bit-reversal is equal to some power of 2. All other array axes can have any length.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

10.2.5 Implementation and Performance

This section discusses the current implementation of the CCFFT and provides performance guidelines.

The CMSSL CCFFT accepts any layout of the data array. Axes may be serial or NEWS-ordered, and may have any weight. However, performance is strongly influenced by the data layout.

The performance characteristics of the CCFFT implementation on the CM are quite different from those observed on many other computers. The differences are a result of the CM memory system. The subsections that follow provide some insight into the implementations and their performance behavior. Several different algorithms are used for the current version of the CCFFT. The combination of algorithms used in a given call depends upon hardware configuration and data layout.

Data Allocation

The default layout for CM Fortran arrays is block layout, also called the consecutive data allocation scheme. In this scheme, a CM array axis is divided into groups of successive elements, and the elements in each group are assigned to the same processing element. (This scheme contrasts with the cyclic data allocation scheme, in which successive elements along a CM array axis are assigned to different processing elements.)

The address field of each array element consists of two groups of bits: those that identify the processing element address, and those that identify the local memory address. For example, in the case of a one-dimensional array of length $N = 2^n$ distributed across $P = 2^p$ processing elements, each array element's address field can be viewed as follows:

$$(i_{n-1} \quad \ldots \quad i_{n-p} \quad | \quad i_{n-p-1} \quad \ldots \quad i_0)$$

The higher-order bits on the left side specify the processing element; the lower-order bits on the right side specify the local memory address. (For multi- dimensional arrays, each axis is encoded separately.)

Communication Requirements for the FFT

In a radix-2 FFT, the data interaction for each stage is between elements that differ in only one bit in the encoding of their address indices. This bit is the

Copyright © 1994 Thinking Machines Corporation

most significant bit in the first stage, and progresses toward the least significant bit in the last stage. A complete FFT has n stages.

An FFT along an axis that is local to a processing element requires no communication between processing elements. An FFT along an axis that spans multiple processing elements requires communication for the butterfly operations whose data elements reside on different processing elements. With block layout, communication is required in the first p FFT stages.

Transpositions

In the current FFT implementation on the CM-5, when data elements on different processing elements are to be combined, the FFT routine transposes the array so that the butterfly operations for these data elements can be performed locally. This procedure is illustrated in the following two examples.

Example 1: One-Dimensional Array. Assume that a one-dimensional array of length 16 is allocated on four processing elements as shown on the left side of Figure 30. The address field for this array is

(3 2 | 1 0)

where address bits 2 and 3 specify the processing element and address bits 0 and 1 specify the local memory address.

The first two FFT stages operate on data elements whose addresses differ in bits 2 and 3, respectively. These stages therefore need to combine data elements residing on different processing elements. The FFT routine first makes these data elements local to a processing element by transposing the array as shown on the right side of Figure 30. This procedure effectively corresponds to exchanging processing element address bits with local memory address bits. After performing the first two FFT stages, the FFT routine transposes the array back to the original layout and carries out the remaining two local FFT stages.

Copyright © 1994 Thinking Machines Corporation

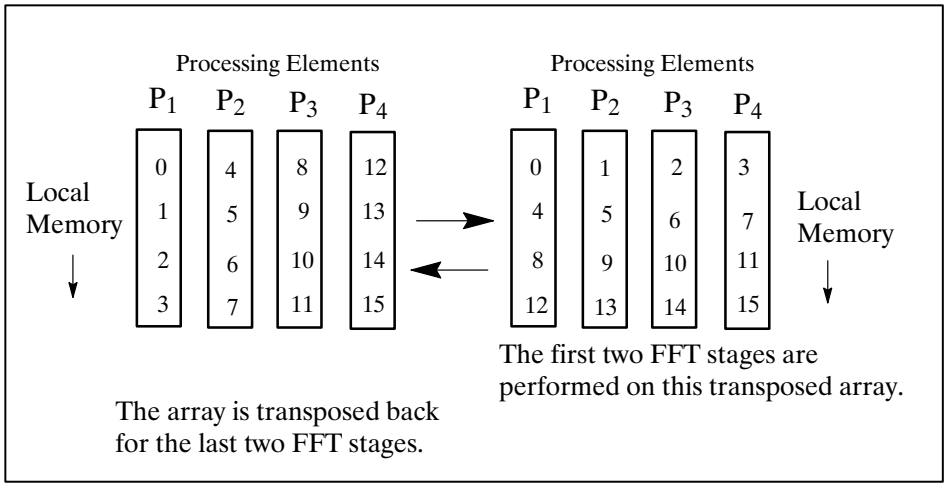

Figure 30. The original array is transposed for the first two FFT stages and then transposed back for the last two stages.

Example 2: Two-Dimensional Array. Assume we want to perform an FFT on an N x N array on P processing elements, where N is a multiple of P. If we lay out this array as (**:serial, :news**), then the FFT along the first axis can be executed purely locally. For the FFT along the second axis, a transposition is needed. If we view the first axis as an instance axis, the transpose operation places N/P instances of axis 2 on each processing element, where they can be combined locally. Subsequently, an inverse transposition restores the results into the original geometry.

Performance Considerations

An FFT of an array of size N involves $O(N\log N)$ arithmetic operations. For a non-local FFT, N data elements need to be transferred between processing elements during a transposition. Given the fact that on the CM-5 the communication time for transposes dominates the time for FFT arithmetic operations, you can optimize performance by minimizing the number of transpose operations. The following strategies are suggested:

- For a multidimensional array, specify a layout that makes as many of the FFT axes as possible local. From the discussion in Example 2 above, it follows that for an FFT along a non-local (**:news**) axis, the best performance is obtained if both the length of the FFT axis and

Copyright © 1994 Thinking Machines Corporation

the number of instances (the product of the other axis lengths) are multiples of the number of processing elements along the FFT axis.

- Performing an FFT along a non-local axis is equivalent to transposing the data explicitly using the **gen_matrix_transpose** routine, then calling the FFT routine to perform the local FFT, and then calling **gen_matrix_transpose** again to perform the inverse transposition, restoring the data to its original layout. Calling **gen_matrix_transpose** explicitly in this way can be advantageous if your application does not require the inverse transposition. In this case you can reduce the communication time significantly by omitting the second call to **gen_matrix_transpose**. The **gen_matrix_transpose** routine is described in Chapter 15. For an illustration of this principle, see Section 10.2.6.

Special Performance Note for Three-Dimensional and Higher-Rank FFTs

The **gen_matrix_transpose** routine yields superior performance when a local axis is exchanged with a non-local axis that is distributed across the vector units on the same processing node. By far the best performance is obtained when the non-local axis spans only two vector units (that is, only the lowest off-chip bit is set). The next best performance results when the non-local axis spans four vector units on the same processing node (the two lowest off-chip bits are set). For the general case of an axis that spans multiple processing nodes, the transpose performance does not depend significantly of the number of contiguous off-chip bits (except if the axis spans only two nodes, in which case the performance is slightly better).

You can use this fact to improve the performance of FFTs of arrays with three or more dimensions. With nodal CMSSL, you can also exploit this fact with two-dimensional FFTs. For example, consider an FFT of a three-dimensional array ***A*** of size 128 x 128 x 128 on a machine with 64 processing nodes (256 vector units). To optimize performance, choose the following array layout:

```
CMF$LAYOUT   A                                          &
CMF$            (:block=128     :pdesc=0                &
CMF$            ,:block=64      :pdesc=1                &
CMF$            ,:block=1       :pdesc=254)
```

With this layout, axis 1 is local to a vector unit. Axis 2 spans two vector units; the subgrid length along axis 2 is 64. Axis 3 spans 128 vector units; the sub-

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

grid length along axis 3 is 1. Thus, ***A*** is distributed among all 256 vector units. You can now solve the problem by following these steps:

1. Call **fft_detailed** to perform an FFT along axis 1, which is local.
2. Exchange axes 1 and 2 by calling **gen_matrix_transpose**. (Note that the transposed array must have the same processing element layout as ***A***.) Since axis 2 is entirely on-node, this transposition is performed very efficiently.
3. Call **fft_detailed** to perform an FFT along axis 2, which is now local.
4. Exchange axes 2 and 3 so that axis 3 becomes local.
5. Call **fft_detailed** to perform an FFT along axis 3.

10.2.6 Efficient Computation of Convolutions and Correlations

One of the main applications of the FFT is to compute convolutions and correlations. To compute the convolution R^*S of two arrays R and S, one can proceed as follows:

1. Use the FFT to compute the discrete Fourier transforms $F(R)$ and $F(S)$.
2. Multiply the two transforms together element by element to form the product $F(R)F(S)$.
3. Take the inverse Fourier transform F^{-1} of the product.

(A similar procedure can be used to compute correlations.)

As discussed in Section 10.2.5, an FFT along a non-local axis requires transpositions. The convolution procedure as outlined above would generally require six transpositions. To reduce the number of required transpositions, follow these steps:

1. Use **gen_matrix_transpose** to transpose arrays R and S into arrays R^{T} and S^{T} so that the FFT axis is local to a processing element.
2. Perform a forward FFT on R^{T} and S^{T}. These FFTs will only involve operations on data elements local to a processing element.
3. Form the product $F(R^{\mathrm{T}})F(S^{\mathrm{T}})$.

Copyright © 1994 Thinking Machines Corporation

4. Perform an inverse FFT on the product. This FFT will also only involve operations on data elements local to a processing element.

5. Optionally, use **gen_matrix_transpose** to transpose the convolution back to an array geometry that conforms with *R* or *S*.

This procedure contains only two or three transposes, depending on whether step 5 is included. Because the non-local FFT time is dominated by transpositions, the explicit use of **gen_matrix_transpose** in the above procedure can significantly enhance overall performance.

10.2.7 For Users Familiar with the CM-200 FFT

The complex-to-complex FFT on the CM-5 uses a different algorithm than the CM-200 FFT. This difference results in different performance behavior and memory requirements. The change in algorithm is motivated by a fundamental difference in the communication systems of the CM-200 and the CM-5. The FFT is a computationally very efficient algorithm; therefore, its performance is very sensitive to how well the communication system is used. On the CM-200, the available communications bandwidth per processing element increases with the number of processing elements over which the data set is distributed. This is not the case with the CM-5 communication system, for which the bandwidth per processing element is reduced or remains constant with the number of processing elements over which the data set is distributed.

On the CM-200, the communications for successive butterfly stages are pipelined until all channels on each processing element are utilized for the subset of processing elements over which the data set is distributed. The communication time for an FFT on N points distributed evenly over P processing elements is proportional to $N/P + \log_2 P \approx N/P$ for most cases. However, pipelining of the communications does not yield maximal use of the CM-5 communication system. In fact, using the CM-200 algorithm on the CM-5 would result in a communication time proportional to N/P $(\log_2 P)$. With a different algorithm, the communication time can be made proportional to

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

$$\frac{N}{2P}\left\{\left\lceil\frac{\log_2 P}{\log_2\frac{N}{2P}}\right\rceil + 1\right\}.$$

In the case $N \geq P^2$, the communication time is also proportional to N/P on the CM-5. This case corresponds to a transposition, a local FFT, a second matrix transposition, and a second local FFT. Two transpositions are required for a normal input order and consecutive (block) data allocation, as described in Section 10.2.5. With bit-reversed input order, one transposition suffices. One transposition would suffice also for normal input order and cyclic data allocation. For $N < P^2$, additional transpositions are needed.

For a somewhat more detailed discussion of the two algorithms, see for example reference 14 in Section 10.5.

Copyright © 1994 Thinking Machines Corporation

Complex-to-Complex Fast Fourier Transform

The routines described below use a complex-to-complex Fast Fourier Transform (CCFFT) algorithm to calculate the Discrete Fourier Transform of an *n*-dimensional CM array. The Simple FFT call, **fft**, performs the same operation (either a forward or an inverse FFT) along all axes of the array. With the Detailed FFT call, **fft_detailed**, you can specify different transform operations, bit orderings, and scaling factors for each axis of the array.

SYNTAX

setup_id = **fft_setup** (***A***, *type, ier*)

fft (***A***, *type, op, setup_id, ier*)

fft_detailed (***A***, *type, ops, in_bit_orders, out_bit_orders, scales, setup_id, ier*)

deallocate_fft_setup (*setup_id*)

ARGUMENTS

A Complex CM array. The ***A*** you supply in the **fft_setup** call must have the same rank, axis extents, layout directives, and precision as the ***A*** arguments you supply in all subsequent **fft** or **fft_detailed** calls associated with the setup call. All axes that are to be transformed must have power-of-2 extents.

When you call **fft_setup**, you may supply arbitrary values in ***A***; the setup routine neither examines nor modifies the contents of the array, but rather uses its size, shape, and data type to create the setup object.

When you call **fft** or **fft_detailed**, ***A*** is the array to be transformed. Upon completion, this array is overwritten with the results.

The elements of ***A*** may be complex or double- precision complex. The floating-point precision of the result data always matches that of the input.

type Front-end variable of type CHARACTER*4 that specifies the type of FFT to be performed. Specify **'CTOC'** for a complex-to-complex transform.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

op Front-end scalar integer variable indicating the direction in which the ***A*** axes are to be transformed in a Simple FFT. Must be one of the following symbolic constants (or the equivalent integer value):

CMSSL_f_xform (1) forward transformation
CMSSL_i_xform (2) inverse transformation

ops Front-end integer vector with length equal to the rank of ***A***. Each element specifies the direction (if any) in which the corresponding ***A*** axis is to be transformed in a Detailed FFT. Each element must be one of the following symbolic constants (or the equivalent integer value):

CMSSL_nop (0) no operation
CMSSL_f_xform (1) forward transformation
CMSSL_i_xform (2) inverse transformation

in_bit_orders Front-end integer vector with length equal to the rank of ***A***. Each element indicates the initial address ordering of the corresponding ***A*** axis in a Detailed FFT. Each element must be one of the following symbolic constants (or the equivalent integer value):

CMSSL_normal (0) normal address ordering
CMSSL_bit_reversed (1) bit-reversed address ordering

out_bit_orders Front-end integer vector with length equal to the rank of ***A***. Each element indicates the desired output address ordering of the corresponding ***A*** axis in a Detailed FFT. Each element must be one of the following symbolic constants (or the equivalent integer value):

CMSSL_normal (0) normal address ordering
CMSSL_bit_reversed (1) bit-reversed address ordering

scales Front-end integer vector with length equal to the rank of ***A***. Each element specifies the scaling factor (if any) for the corresponding ***A*** axis. Each element must be one of the following symbolic constants (or the equivalent integer value):

CMSSL_noscale (0) no scaling
CMSSL_scale_sqrt (1) scale by square root of the axis length
CMSSL_scale_n (2)scale by the axis length

Copyright © 1994 Thinking Machines Corporation

At the end of the **fft_detailed** call, each element of ***A*** is divided by the product of the scaling factors. (The axes for which you specified **CMSSL_noscale** do not contribute to this product.)

setup_id Scalar integer variable. The ID of a setup structure returned by the **fft_setup** routine.

ier Scalar integer variable. Upon return, contains 0 if the call succeeded. The **fft_setup** routine may return one of the following errors:

−1 You have supplied an invalid CM array.

−2 You have supplied a CM array with a data type other than single- or double-precision complex.

The **fft** routine may return the following error:

−1 You are attempting to transform axes whose lengths are not a power of 2.

DESCRIPTION

To perform a CCFFT, you must follow these steps:

1. Call **fft_setup**.

   This routine computes internal values (including the twiddle factors), allocates an internal FFT setup object, and computes an FFT setup ID, which you must supply in all subsequent Simple FFT, Detailed FFT, and deallocation calls associated with this setup call.

2. Call **fft** or **fft_detailed**.

   These routines execute the Simple and Detailed FFTs, respectively. You may follow one call to **fft_setup** with multiple calls to these routines. However, the result of a call to **fft_setup** can only be used for arrays with the same rank, axis extents, layout directives, and precision as the array supplied in the setup routine. If these parameters change, you must call **fft_setup** again to create another setup structure.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

3. Call **deallocate_fft_setup**.

   This routine removes an FFT setup object that is no longer needed. Because an FFT setup object occupies both partition manager and processing element memory, you should free this space by deallocating the setup object after the completion of all FFT invocations that use it.

The time required to compute the contents of an FFT setup structure is substantially longer than the time required to actually perform an FFT. For this reason, and because it is common to perform FFTs on many arrays with the same rank, axis lengths, layout directives, and precision, the setup phase is separated from the transform phase.

You may have more than one setup ID active at a time; that is, you may call the setup routine more than once before deallocating any setup IDs. When you call **fft**, **fft_detailed**, or **deallocate_fft_setup**, you must be sure to provide the correct setup ID.

Twiddle Factor Exponents. The sign of the twiddle factor exponents determines the direction of an FFT. In the CMSSL FFT, the *forward* transform is defined as one that uses twiddle factors with a *negative* exponent, and the *inverse* transform is defined as one that uses twiddle factors with a *positive* exponent.

Multidimensional and Multiple-Instance FFTs. The **fft** routine computes a multidimensional transform by performing a one-dimensional transform along each axis in turn. The **fft_detailed** routine supports multiple instances, and computes one or more concurrent, independent one-dimensional FFTs along each data axis in turn.

Front-End Scalar Argument for Simple FFT. In the **fft** call, the *op* argument determines whether the routine performs a forward or an inverse FFT transform along each axis. The transform direction is the same for all axes. The simple FFT uses the following conventions:

- Along each axis, input and output data are stored using the same bit ordering.
- For a forward FFT, no scaling factors are used. For an inverse FFT, the axis length is used as the scaling factor for each axis.

Front-End Vector Arguments for Detailed FFT. The **fft_detailed** call requires four front-end vector arguments, *ops, in_bit_orders*, *out_bit_orders*, and *scales*, whose

Copyright © 1994 Thinking Machines Corporation

lengths must equal the rank of ***A***. Within each of these vectors, each element specifies an option for the corresponding axis of ***A***, as follows:

- The *ops* vector specifies the direction in which the corresponding ***A*** axis is to be transformed, or specifies that the axis is not to be transformed. In a multiple-instance FFT, the axes to be transformed are the data axes, and the axes that are not to be transformed are the instance axes.

- The *in_bit_orders* and *out_bit_orders* vectors specify the order in which the original and the resulting ***A*** values are stored along each axis. In normal address ordering (**CMSSL_normal**), data values are stored consecutively. Consecutive order is the only compiler-supported data allocation scheme. In bit-reversed ordering, consecutive data values are stored according to sequential addresses whose high- and low-order bits have been swapped; that is, the consecutive allocation scheme is applied to the bit-reversed indices.

  Although **fft_detailed** can handle any combination of bit orderings, performance is best when the axes of an untransformed array and a transformed array have opposite bit-orderings — that is, input normal and output bit-reversed, or input bit-reversed and output normal.

- The *scales* vector specifies a scaling factor (none, square root of axis length, or axis length) for each axis. At the end of the **fft_detailed** call, each element of ***A*** is divided by the product of the scaling factors. (The axes for which you specified **CMSSL_noscale** do not contribute to this product.)

  For example, suppose ***A*** has dimensions (*m, n, p*) and you supply

  (**CMSSL_scale_n**, **CMSSL_scale_sqrt**, **CMSSL_noscale**)

  as the *scales* vector. At the end of the **fft_detailed** call, each element of ***A*** is divided by *m* * sqrt(*n*).

  One artifact of the Fourier Transform algorithm is that if no scaling is performed, then a forward FFT followed immediately by an inverse FFT results in the original array with all values multiplied by the FFT size — that is, the product of the lengths of the transformed axes. Scaling is typically used to correct this effect and prevent arithmetic overflow. Typically, a scaling factor of **CMSSL_noscale** is used for all axes in a forward transform, and a scaling factor of **CMSSL_scale_n** is used for all axes being transformed in an inverse transform. Alternatively, you may specify a scaling factor of **CMSSL_scale_sqrt** for all transformed axes in both the forward and the inverse FFTs. This alternative is less efficient, but can reduce the risk of arithmetic overflow for very large numbers.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Scaling is independent of the operations performed along the axes. For example, it is possible to specify a scaling factor of **CMSSL_scale_n** for an axis that is not being transformed.

Bit-Reversing the Addressing of a CM Array. You can use the Detailed FFT to bit-reverse the addressing of any *n*-dimensional CM array. To do this, specify **CMSSL_nop** for all elements of the *ops* argument, and opposite values for the input and output orderings; that is, either **CMSSL_normal** for the *in_bit_orders* values and **CMSSL_bit_reversed** for the *out_bit_orders* values, or **CMSSL_bit_reversed** for the *in_bit_orders* values and **CMSSL_normal** for the *out_bit_orders* values.

NOTES

Setup Uses Array Shape, Layout, and Data Type. The setup ID computed by the **fft_setup** call can be used for all arrays with the same shape, layout, and data type (including precision) as the *A* argument supplied in the **fft_setup** call — and *only* for such arrays. If a transform is to be performed on two arrays, *A* and *B*, of the same shape, layout, and data type, then *one* call to the setup routine suffices, even if transforms are performed on different axes of the two arrays. But if *A* and *B* have different shapes, layouts, or data types, then you must make one call for each array. Note that to be the same shape, two arrays must not only have the same number of axes, but also the same axis lengths. Thus, if two arrays with the same number of axes differ in even one axis length or in any layout directives, they require separate setup calls.

Setup is Private. The FFT setup data contains information private to the FFT. The format of this data is not documented and may change between CMSSL releases. For this reason, application code should never access or modify the contents of an FFT setup structure.

Header File. The FFT routines use predefined symbolic constants. Therefore, you must include the statement `INCLUDE '/usr/include/cm/cmssl-cmf.h'` at the top of the main file of any FFT program. This file defines symbolic constants and declares the type of the CMSSL functions.

Copyright © 1994 Thinking Machines Corporation

Bit Reversal and Performance. For an axis that is being transformed,

- If you specify *opposite* address orderings for input and output, the Detailed FFT performs the single bit-reversal that is inherent in the Cooley-Tukey algorithm, and produces output that is bit-reversed relative to the input.
- If you specify the *same* address ordering for input and output, the Detailed FFT performs an *extra* bit reversal, and produces output with the same ordering as the input. The extra bit reversal exacts a performance cost.

For an axis that is not being transformed,

- If you specify *opposite* address orderings for input and output, the Detailed FFT performs a single bit-reversal and produces output that is bit-reversed relative to the input.
- If you specify the *same* address ordering for input and output, the Detailed FFT performs *no* bit reversal.

Thus, for optimal FFT performance, specify

- Opposite address orderings for input and output, for each axis that is being transformed.
- The same address ordering for input and output, for each axis that is not being transformed.

To bit-reverse the address ordering of a dataset without transforming it, call the Detailed FFT specifying no transform for all axes and specifying *opposite* values for the input and output address ordering.

These points are independent of the directions of the transforms.

EXAMPLES

Sample code that uses the complex-to-complex FFT can be found on-line in the subdirectory

```
fft/cmf/
```

of a CMSSL examples directory whose location is site-specific.

The following code skeletons show the basic usage of the Simple FFT and the Detailed FFT.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Code Skeleton Showing Basic Usage of Simple FFT. The following code demonstrates the syntax of the **fft** routine:

```
      PROGRAM simple_manual_example
      INCLUDE '/usr/include/cm/cmssl-cmf.h'

      INTEGER setup_id, ier
      COMPLEX my_cm_array (256, 256)

C     initialize our cm array (this makes sure it's
C        on the cm)
      my_cm_array = (1.0, 0.0)

C     create an FFT setup id before taking the FFT
      setup_id = fft_setup (my_cm_array, 'CTOC', ier)

C     perform the forward transform on my_cm_array
      CALL fft(my_cm_array,'CTOC',CMSSL_f_xform,setup_id,ier)

C     use the transformed array here

C     perform the inverse transform on my_cm_array
      CALL fft(my_cm_array,'CTOC',CMSSL_i_xform,setup_id,ier)

C     finally, scrap the setup
      CALL deallocate_fft_setup( setup_id )
      STOP
      END
```

Code Skeleton Showing Basic Usage of Detailed FFT. The code below demonstrates the syntax of the **fft_detailed** routine. After allocating and initializing the needed variables, this code performs a forward FFT followed by an inverse FFT on the array **my_cm_array**. Notice that the output of the forward FFT call is in bit-reversed order. The transformed array is operated on with data in bit-reversed order, thereby avoiding two applications of the reordering function (one after the forward FFT, one after the inverse FFT).

```
      PROGRAM detailed_manual_example
      INCLUDE '/usr/include/cm/cmssl-cmf.h'

C     declare and initialize variables
      INTEGER forward_ops(2), inverse_ops(2), normal_bits(2),
     & reversed_bits(2), unscaled(2), n_scaled(2), setup, ier
      DATA forward_ops/CMSSL_f_xform, CMSSL_f_xform/,
     &  inverse_ops/CMSSL_i_xform, CMSSL_i_xform/,
```


Copyright © 1994 Thinking Machines Corporation

```
      &  normal_bits/CMSSL_normal, CMSSL_normal/,
      &  reversed_bits/CMSSL_bit_reversed,CMSSL_bit_reversed/,
      &  unscaled/CMSSL_noscale, CMSSL_noscale/,
      &  n_scaled/CMSSL_scale_n, CMSSL_scale_n/,
      &  ier/0/
       COMPLEX my_cm_array (256, 256)

C      initialize our cm array (this ensures it's on the cm)
       my_cm_array = (1.0, 0.0)

C      get the setup
       setup_id = fft_setup( my_cm_array. 'CTOC', ier )

C      call a forward detailed FFT
       CALL fft_detailed( my_cm_array, 'CTOC', forward_ops,
      & normal_bits, reversed_bits, unscaled, setup_id, ier )

C     Do something with the transformed array here

C      call an inverse detailed FFT
       CALL fft_detailed( my_cm_array, 'CTOC', inverse_ops,
      & reversed_bits, normal_bits, n_scaled, setup_id, ier )

C      scrap the setup
       CALL deallocate_fft_setup (setup_id)
       STOP
       END
```


Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

10.3 Real-to-Complex and Complex-to-Real FFTs

CMSSL provides a real-to-complex FFT (RCFFT) for computing the Fourier transform of real data, and a complex-to-real FFT (CRFFT) for the transformation of *conjugate symmetric* complex sequences. The transform X_k (for $k = 0, 1, \ldots, N-1$) of a real sequence x_n ($n = 0, 1, \ldots, N-1$), is conjugate symmetric, i.e., $X_k = \overline{X_{N-k}}$, and conversely, the transform of such a sequence is real. The Fourier Transform of a real or conjugate symmetric sequence can be computed using half the storage and half the arithmetic of a complex-to-complex FFT (CCFFT). The savings in arithmetic and storage are realized by only computing X_k for $k = 0, 1, \ldots, N/2$, and exploiting the fact that X_0 and $X_{N/2}$ are both real. These numbers are stored as one complex number. As a result, the conjugate symmetric sequence requires precisely the same amount of storage as the real sequence. We refer to the domain in which the real data exists as the *real domain*, and the domain in which the conjugate symmetric sequence exists as the *complex domain*.

The CMSSL RCFFT and CRFFT can be used to perform a transform along any axis of a multidimensional array. We refer to this axis as the *conjugate symmetric axis*, since after a real-to-complex transform along this axis, it represents half of a conjugate symmetric sequence. The CMSSL RCFFT and CRFFT compute the transform in place and fully realize the savings in storage.

The reduced need for storage also results in a performance gain. If the conjugate symmetric axis extends over several processing elements, then the bit-reversal operation needed to make the output ordering the same as the input ordering only needs to move half as much data as if the real array of size N had been mapped to a complex array of size N. Furthermore, for multidimensional arrays, subsequent Fourier transforms, if any, operate on a data set half the size, and a factor of two performance gain is realized.

The performance gain in the CMSSL RCFFT or CRFFT itself is best when the conjugate symmetric axis is local to a processing element. Some data reordering is required, and the efficiency in performing the arithmetic is not quite as high as in the CCFFT. Therefore, the performance gain is slightly less than the ideal factor of two. When the conjugate symmetric axis is distributed across several processing elements, then the reordering time is significant, and the total communication time for a real array of size N is comparable to the communication time for a complex array of size N. Hence, in this case the performance gain for the RCFFT or CRFFT proper is marginal. However, the aforementioned performance gains from the reduced array size can still be realized.

Copyright © 1994 Thinking Machines Corporation

10.3.1 Relationship of CRFFT, RCFFT, and CCFFT

The CMSSL RCFFT is based upon the CMSSL CCFFT followed by post-processing. Similarly, the CRFFT has a preprocessing stage followed by a CCFFT.

The input array to the RCFFT must be a complex array, with a pair of successive elements along the conjugate symmetric axis forming a complex number. For one-dimensional data arrays, this is a natural ordering that may be performed without any data motion. But for multidimensional arrays, a data reordering is required, in general, as discussed in Section 10.3.2 below. If there are at least two data elements per processing element along the axis to be transformed, then local memory reordering suffices; otherwise, communication is required. Similarly, the output of the CRFFT is a complex array, with complex numbers forming pairs of successive real elements along the conjugate symmetric axis. The array conversion utilities described in Section 10.4 convert a real array to a complex array suitable for transformation by the RCFFT, and convert the complex array resulting from the CRFFT into a real array.

In the CMSSL RCFFT and CRFFT, a forward transform uses twiddle factors with a negative exponent, and an inverse transform uses twiddle factors with a positive exponent, just as in the CCFFT. Therefore, to invert an RCFFT, it is necessary to change the operation from forward to inverse as well as changing the FFT type from real-to-complex to complex-to-real.

10.3.2 Data Orderings

For general information about normal and bit-reversed data orderings, refer to Section 10.2.2.

For the RCFFT and CRFFT, there are several important data ordering issues. In the complex domain, the ordering is normal when successive complex data elements along an axis are assigned to successive complex array locations, and the order is bit-reversed if the complex data are assigned to data array locations based on the bit-reversed values of their indices, as described for the CCFFT. Figure 31 illustrates the data allocation in the complex domain for a one-dimensional array. For the axis representing the conjugate symmetric sequence, location zero is special. It represents two real elements, namely the real components of frequency component 0

Copyright © 1994 Thinking Machines Corporation

and frequency component $N/2$, both of which have an imaginary component equal to zero.

Order	re	im	re	im	re	im	re	im	re	im	re	im	re	im	re	im
Normal	0_r	8_r	1_r	1_c	2_r	2_c	3_r	3_c	4_r	4_c	5_r	5_c	6_r	6_c	7_r	7_c
Bit-reversed	0_r	8_r	4_r	4_c	2_r	2_c	6_r	6_c	1_r	1_c	5_r	5_c	3_r	3_c	7_r	7_c

Figure 31. Normal and bit-reversed order of a conjugate symmetric sequence allocated to a complex one-dimensional array.

In the real domain, the order is normal if the lower-ordered element in a pair of successive elements along the conjugate symmetric axis is assigned to the real component, and the higher-ordered element is assigned to the imaginary component, of a complex number. Furthermore, successive pairs must be assigned to successive complex locations along the axes to which the RCFFT is applied. The order is bit-reversed in the real domain if successive pairs of elements are assigned to complex locations along the conjugate symmetric axis based on the bit-reversed value of the index of the pair. Figure 32 illustrates the normal and bit-reversed ordering in the real domain.

Order	re	im	re	im	re	im	re	im	re	im	re	im	re	im	re	im
Normal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bit-reversed	0	1	8	9	4	5	12	13	2	3	10	11	6	7	14	15

Figure 32. Normal and bit-reversed order of a real sequence allocated to a complex one-dimensional array.

The mapping of real data into a complex array of higher dimension depends upon which axis is to be transformed. Figure 33 shows an 8 x 8 real array. For an RCFFT along the vertical axis, you must map this array into a complex array as shown in Figure 34. For an RCFFT along the horizontal axis, you must map the array into a complex array as shown in Figure 35. The **complex_from_real** routine, described in Section 10.4, performs these mappings.

Copyright © 1994 Thinking Machines Corporation

$x(00)$	$x(01)$	$x(02)$	$x(03)$	$x(04)$	$x(05)$	$x(06)$	$x(07)$
$x(10)$	$x(11)$	$x(12)$	$x(13)$	$x(14)$	$x(15)$	$x(16)$	$x(17)$
$x(20)$	$x(21)$	$x(22)$	$x(23)$	$x(24)$	$x(25)$	$x(26)$	$x(27)$
$x(30)$	$x(31)$	$x(32)$	$x(33)$	$x(34)$	$x(35)$	$x(36)$	$x(37)$
$x(40)$	$x(41)$	$x(42)$	$x(43)$	$x(44)$	$x(45)$	$x(46)$	$x(47)$
$x(50)$	$x(51)$	$x(52)$	$x(53)$	$x(54)$	$x(55)$	$x(56)$	$x(57)$
$x(60)$	$x(61)$	$x(62)$	$x(63)$	$x(64)$	$x(65)$	$x(66)$	$x(67)$
$x(70)$	$x(71)$	$x(72)$	$x(73)$	$x(74)$	$x(75)$	$x(76)$	$x(77)$

Figure 33. A real 8 x 8 array.

$w_r(00)$	$w_r(01)$	$w_r(02)$	$w_r(03)$	$w_r(04)$	$w_r(05)$	$w_r(06)$	$w_r(07)$
$w_i(00)$	$w_i(01)$	$w_i(02)$	$w_i(03)$	$w_i(04)$	$w_i(05)$	$w_i(06)$	$w_i(07)$
$w_r(10)$	$w_r(11)$	$w_r(12)$	$w_r(13)$	$w_r(14)$	$w_r(15)$	$w_r(16)$	$w_r(17)$
$w_i(10)$	$w_i(11)$	$w_i(12)$	$w_i(13)$	$w_i(14)$	$w_i(15)$	$w_i(16)$	$w_i(17)$
$w_r(20)$	$w_r(21)$	$w_r(22)$	$w_r(23)$	$w_r(24)$	$w_r(25)$	$w_r(26)$	$w_r(27)$
$w_i(20)$	$w_i(21)$	$w_i(22)$	$w_i(23)$	$w_i(24)$	$w_i(25)$	$w_i(26)$	$w_i(27)$
$w_r(30)$	$w_r(31)$	$w_r(32)$	$w_r(33)$	$w_r(34)$	$w_r(35)$	$w_r(36)$	$w_r(37)$
$w_i(30)$	$w_i(31)$	$w_i(32)$	$w_i(33)$	$w_i(34)$	$w_i(35)$	$w_i(36)$	$w_i(37)$

Figure 34. The mapping of an 8 x 8 real array into a 4 x 8 complex array for transformation on the vertical axis.

$w_r(00)$	$w_i(00)$	$w_r(01)$	$w_i(01)$	$w_r(02)$	$w_i(02)$	$w_r(03)$	$w_i(03)$
$w_r(10)$	$w_i(10)$	$w_r(11)$	$w_i(11)$	$w_r(12)$	$w_i(12)$	$w_r(13)$	$w_i(13)$
$w_r(20)$	$w_i(20)$	$w_r(21)$	$w_i(21)$	$w_r(22)$	$w_i(22)$	$w_r(23)$	$w_i(23)$
$w_r(30)$	$w_i(30)$	$w_r(31)$	$w_i(31)$	$w_r(32)$	$w_i(32)$	$w_r(33)$	$w_i(33)$
$w_r(40)$	$w_i(40)$	$w_r(41)$	$w_i(41)$	$w_r(42)$	$w_i(42)$	$w_r(43)$	$w_i(43)$
$w_r(50)$	$w_i(50)$	$w_r(51)$	$w_i(51)$	$w_r(52)$	$w_i(52)$	$w_r(53)$	$w_i(53)$
$w_r(60)$	$w_i(60)$	$w_r(61)$	$w_i(61)$	$w_r(62)$	$w_i(62)$	$w_r(63)$	$w_i(63)$
$w_r(70)$	$w_i(70)$	$w_r(71)$	$w_i(71)$	$w_r(72)$	$w_i(72)$	$w_r(73)$	$w_i(73)$

Figure 35. The mapping of an 8 x 8 real array into an 8 x 4 complex array for transformation on the horizontal axis.

Copyright © 1994 Thinking Machines Corporation

10.3.3 Multidimensional and Multiple-Instance Real-to-Complex and Complex-to-Real FFTs

For general information about multidimensional and multiple-instance FFTs, refer to Section 10.2.3.

The RCFFT and CRFFT in CMSSL support multiple instances, but do not properly support multidimensional transforms. The discussion below explains how to use these routines to perform multidimensional transforms.

The fact that the real components of frequency components 0 and $N/2$ for the transformation of a real sequence of length N are stored in complex location 0 along the transformed axis implies that a "complex" array of conjugate symmetric sequences is not an array of complex numbers in a strict sense. This may be important if a CCFFT is desired along other axes of a multidimensional array. For instance, if the RCFFT is performed along the vertical axis of Figure 34, then the result is returned in the form shown in Figure 36. The first row of complex elements contains two real sequences: one in the real component, one in the imaginary component. Applying a CCFFT to the horizontal axis yields the correct Discrete Fourier Transform of all sequences along the horizontal axis except for the two real sequences stored in complex row location zero. These two real sequences are transformed as a pair, as described in Section 10.3.4. The transform of each sequence can be obtained by applying the necessary postprocessing *only to row zero*. Such postprocessing is currently not supported in CMSSL.

Storage location								
re(0,:)	$X_r(00)$	$X_r(01)$	$X_r(02)$	$X_r(03)$	$X_r(04)$	$X_r(05)$	$X_r(06)$	$X_r(07)$
im(0,:)	$X_r(40)$	$X_r(41)$	$X_r(42)$	$X_r(43)$	$X_r(44)$	$X_r(45)$	$X_r(46)$	$X_r(47)$
re(1,:)	$X_r(10)$	$X_r(11)$	$X_r(12)$	$X_r(13)$	$X_r(14)$	$X_r(15)$	$X_r(16)$	$X_r(17)$
im(1,:)	$X_i(10)$	$X_i(11)$	$X_i(12)$	$X_i(13)$	$X_i(14)$	$X_i(15)$	$X_i(16)$	$X_i(17)$
re(2,:)	$X_r(20)$	$X_r(21)$	$X_r(22)$	$X_r(23)$	$X_r(24)$	$X_r(25)$	$X_r(26)$	$X_r(27)$
im(2,:)	$X_i(20)$	$X_i(21)$	$X_i(22)$	$X_i(23)$	$X_i(24)$	$X_i(25)$	$X_i(26)$	$X_i(27)$
re(3,:)	$X_r(30)$	$X_r(31)$	$X_r(32)$	$X_r(33)$	$X_r(34)$	$X_r(35)$	$X_r(36)$	$X_r(37)$
im(3,:)	$X_i(30)$	$X_i(31)$	$X_i(32)$	$X_i(33)$	$X_i(34)$	$X_i(35)$	$X_i(36)$	$X_i(37)$

Figure 36. The conjugate symmetric sequences in normal order, resulting from the transformation along the vertical axis of the 4 x 8 complex array of real numbers. The subscripts *r* and *i* indicate the real and imaginary components of the data elements.

Copyright © 1994 Thinking Machines Corporation

The postprocessing of row zero is different from the postprocessing in the CMSSL RCFFT. The postprocessing of row zero should uncover *two* conjugate symmetric sequences, and a reordering is required. Let the transform of X in Figure 36 along the horizontal axis be χ, that is,

$$X(km) = \sum_{n=0}^{N-1} x(nm) e^{-ikn(2\pi/N)}$$

and

$$\chi(kl) = \sum_{m=0}^{M-1} X(km) e^{-ilm(2\pi/M)}$$

Then the transformation on row zero, $X(0, :) + iX(4, :)$ in Figure 37, by a CCFFT represents the transformation of a pair of real sequences embedded in a one-dimensional complex array. The postprocessing of a pair of real sequences embedded in a one-dimensional complex array requires postprocessing as described in Section 10.3.4. This postprocessing produces two conjugate symmetric sequences, each of half length. The elements of these two sequences can be stored as shown in Figure 37, in which the result is in normal order. The conjugate symmetric sequence resulting from the transformation of the real sequence $X(0, :)$ is stored in the first half of complex row zero, and the conjugate symmetric sequence resulting from the transformation of real sequence $X(4, :)$ is stored in the second half of complex row zero. Within each half, the storage is consistent with the storage for the RCFFT along the vertical axis. Whereas the convention is to store the two real components of the transformation of a real sequence — i.e., $X_r(0)$ and $X_r(N/2)$ — in complex location zero, there is no conventional storage scheme for the transform of a pair of real sequences. Since the postprocessing is required only for an array with rank one less than the original array, the current implementation of FFTs in CMSSL leaves the postprocessing of multidimensional transforms to the user.

Copyright © 1994 Thinking Machines Corporation

Storage location								
re(0,:)	$\chi_r(00)$	$\chi_r(01)$	$\chi_r(02)$	$\chi_r(03)$	$\chi_r(04)$	$\chi_r(05)$	$\chi_r(06)$	$\chi_r(07)$
im(0,:)	$\chi_r(00)$	$\chi_i(01)$	$\chi_i(02)$	$\chi_i(03)$	$\chi_r(04)$	$\chi_i(05)$	$\chi_i(06)$	$\chi_i(07)$
re(1,:)	$\chi_r(10)$	$\chi_r(11)$	$\chi_r(12)$	$\chi_r(13)$	$\chi_r(14)$	$\chi_r(15)$	$\chi_r(16)$	$\chi_r(17)$
im(1,:)	$\chi_i(10)$	$\chi_i(11)$	$\chi_i(12)$	$\chi_i(13)$	$\chi_i(14)$	$\chi_i(15)$	$\chi_i(16)$	$\chi_i(17)$
re(2,:)	$\chi_r(20)$	$\chi_r(21)$	$\chi_r(22)$	$\chi_r(23)$	$\chi_r(24)$	$\chi_r(25)$	$\chi_r(26)$	$\chi_r(27)$
im(2,:)	$\chi_i(20)$	$\chi_i(21)$	$\chi_i(22)$	$\chi_i(23)$	$\chi_i(24)$	$\chi_i(25)$	$\chi_i(26)$	$\chi_i(27)$
re(3,:)	$\chi_r(30)$	$\chi_r(31)$	$\chi_r(32)$	$\chi_r(33)$	$\chi_r(34)$	$\chi_r(35)$	$\chi_r(36)$	$\chi_r(37)$
im(3,:)	$\chi_i(30)$	$\chi_i(31)$	$\chi_i(32)$	$\chi_i(33)$	$\chi_i(34)$	$\chi_i(35)$	$\chi_i(36)$	$\chi_i(37)$

Figure 37. One potential data allocation of complex and conjugate symmetric sequences resulting from a two-dimensional transform of an 8 x 8 real array initially mapped into a 4 x 8 complex array. The subscripts *r* and *i* indicate the real and imaginary components of the data elements.

Note that in many applications the values $X(0, :)$ and $X(N/2, :)$ may not be needed, and can be set to zero. Note further that even if the postprocessing is omitted, a real-to-complex multidimensional transform followed by a complex-to-real multidimensional inverse transform always reproduces the original result.

10.3.4 Implementation

The Fourier transform

$$X_k = \sum_{n=0}^{N-1} x_n e^{-ikn(2\pi/N)}$$

and its inverse

$$x_n = 1/N \sum_{k=0}^{N-1} X_k e^{ikn(2\pi/N)}$$

are computed efficiently using the Fast Fourier Transform. But if x_n is a real sequence, $x_n = \overline{x_n}$, then X_k can be computed with half as many arithmetic operations as required for a complex sequence, and only half as much stor-

Copyright © 1994 Thinking Machines Corporation

age is needed. Many algorithms for real sequences are developed by modifying the Cooley-Tukey complex-to-complex FFT that follows from the splitting formula

$$X_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-ik2n(2\pi/N)} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-ik(2n+1)(2\pi/N)}$$

Ignoring the scale factor $1/N$ and defining

$$Y_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-ikn\frac{2\pi}{N/2}}$$

$$Z_k = \sum_{n=0}^{N/2-1} x_{2n+1} e^{-ikn\frac{2\pi}{N/2}}$$

we have

$$X_k = Y_k + e^{-ik(2\pi/N)} Z_k, \quad k = 0, \ldots, N/2 - 1$$

But since $Y_{k+N/2} = Y_k$, $Z_{k+N/2} = Z_k$, and $e^{-i(k+N/2)\,2\pi/N} = -e^{-ik\,2\pi/N}$, we have

$$X_{k+N/2} = Y_k - e^{-ik\,2\pi/N} Z_k, k = 0, \ldots, N/2 - 1.$$

This splitting algorithm for X_k is the basis for the Cooley-Tukey complex-to-complex FFT, which consists of repeated application of the splitting algorithm. In order to evaluate, say, only the first half, X_k ($k = 0, 1, \ldots, N/2$), for a conjugate symmetric sequence, it is necessary to modify the splitting formulas so that only the first half of Y_k and Z_k are referenced.

Transformation of Pairs of Sequences

A *pair* of real sequences, y_n and z_n, can be transformed using a single complex transform by defining the complex sequence $x_n = y_n + iz_n$ and computing its complex transform X_k. Since the Fourier transform is linear, we have

Copyright © 1994 Thinking Machines Corporation

$$X_k = Y_k + i\,Z_k$$

where Y_k and Z_k are the Fourier transforms of y_n and z_n, respectively. Since Y_k and Z_k are complex, they do not correspond to the real and imaginary parts of X_k. But since y_n and z_n are real, Y_k and Z_k are conjugate symmetric. Therefore

$$X_{N-k} = Y_{N-k} + i\,Z_{N-k} = \overline{Y_k} + i\,\overline{Z_k}$$

or

$$\overline{X_{N-k}} = Y_k - i\,Z_k$$

and

$$Y_k = 1/2\ (X_k + \overline{X_{N-k}}), \quad k = 0, 1, \ldots, N/2$$

$$Z_k = 1/2i\ (X_k - \overline{X_{N-k}}), \quad k = 0, 1, \ldots, N/2$$

Therefore, the transform of a pair of real sequences of length N can be computed from a single complex transform of length N: first, form $x_n = y_n + iz_n$; next, compute X_k; and finally, compute Y_k and Z_k. This technique is particularly useful for multidimensional transforms on real data, but is not used in CMSSL.

Note that since the Fourier Transform is a linear operation, if the same operation is applied to both transformed sequences, followed by an inverse transform, then explicit computation of Y_k and Z_k is not necessary. There is a large class of applications, including the solution of certain partial differential equations, to which this case applies. For example, an approximate derivative of the sequences y_n and z_n can be computed by multiplying X_k by ik and performing the inverse FFT.

Transformation of a Single Sequence

For the transformation of a *single* real sequence of length N by pre- and postprocessing around a complex transform of length $N/2$, let x_n be a real sequence of length N, where N is an even integer. Define $y_n = x_{2n}$, $z_n = x_{2n+1}$, and $w_n = y_n + iz_n$. The sequence w_n is of length $N/2$. Next, compute the Fourier transform W_k of w_n. From the previous derivations, we have

$$Y_k = 1/2\ (W_k + \overline{W_{N/2-k}}), \quad k = 0, 1, \ldots, N/4$$

$$Z_k = 1/2i\ (W_k - \overline{W_{N/2-k}}), \quad k = 0, 1, \ldots, N/4$$

Copyright © 1994 Thinking Machines Corporation

where Y_k and Z_k are the Discrete Fourier transforms of the sequences $y_n = x_{2n}$ and $z_n = x_{2n+1}$. We wish to compute X_k, $k = 0, 1, \ldots, N/2$, where

$$X_k = Y_k + e^{-ik(2\pi/N)} Z_k, \quad k = 0, \ldots, N/2 - 1$$

The sequence X_k for $k = 0, 1, \ldots, N/4$ can be computed from

$$X_k = Y_k + e^{-ik(2\pi/N)} Z_k, \quad k = 0, \ldots, N/4$$

For the indices $k = N/4 + 1, \ldots, N/2$, we have

$$X_{N/2-k} = Y_{N/2-k} + e^{-i(N/2-k)(2\pi/N)} Z_{N/2-k}, \quad k = 0, \ldots, N/4 - 1$$

But since the sequence x_n is real, both Y_k and Z_k are conjugate symmetric, and

$$\overline{X_{N/2-k}} = Y_k - e^{-ik(2\pi/N)} Z_k, \quad k = 0, \ldots, N/4 - 1$$

In summary, for the transformation of a single real sequence of length N, form the complex sequence $w_n = x_{2n} + ix_{2n+1}$ of length $N/2$, compute its transform using a complex-to-complex transform, and then obtain the transforms Y_k of x_{2n} and Z_k of x_{2n+1} through one postprocessing step, followed by another postprocessing step to obtain X_k. The two postprocessing steps can be combined into

$$X_k = 1/2\,[(W_k + \overline{W_{N/2-k}}) - ie^{-ik(2\pi/N)}\,(W_k - \overline{W_{N/2-k}})],\ k = 0, 1, ..., N/4$$

$$\overline{X_{N/2-k}} = 1/2\,[(W_k + \overline{W_{N/2-k}}) + ie^{-ik(2\pi/N)}\,(W_k - \overline{W_{N/2-k}})],$$
$$k = 0, 1, ..., N/4-1$$

Summary

The single real-sequence transform as outlined above is the algorithm used in the current CMSSL. The arithmetic requirements are the same as for the transform of pairs of sequences. The postprocessing requires data motion in

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

both algorithms. The communication is incurred because of the *index* reversal:

$$k \longleftrightarrow N/2 - k$$

For data motion between processing elements, the communication time between processing elements for a CCFFT on $N/2$ points distributed over P processing elements is proportional to $N/2P$. The communication time for the postprocessing is also proportional to $N/2P$; i.e., the total communication time is proportional to N/P. Hence, in the current release, the communication time for the RCFFT on a real sequence of length N is approximately the same as the communication time for a CCFFT on a complex sequence of length N.

10.3.5 Steps in Performing the Real-to-Complex and Complex-to-Real FFTs

To perform an RCFFT, follow these steps:

1. Use the **complex_from_real** routine (described in Section 10.4) to place the real data into a complex array, with pairs of real data elements along the axis to be transformed (the conjugate symmetric axis) forming complex numbers. Note that the extent of the conjugate symmetric axis must be a power of 2.

2. Call **fft_setup** and **fft_detailed** specifying an RCFFT. The **fft_detailed** routine computes the first half of the conjugate symmetric sequence in place. The real components of elements X_0 and $X_{N/2}$ are stored in array location zero along the transformed axis. This routine

    f. Internally calls the CCFFT with the proper twiddle factors.

    g. Performs the postprocessing by

    - Calling index-reversal routines if needed, performing the data communication in a close to optimal manner.

    - Performing the butterfly-like computations for the postprocessing.

Copyright © 1994 Thinking Machines Corporation

To perform a CRFFT, follow these steps:

1. Place the conjugate symmetric data into a complex array with the real components of elements X_0 and $X_{N/2}$ in the first location of the axis to be transformed (i.e., the conjugate symmetric axis).

2. Call **fft_setup** and **fft_detailed** specifying a CRFFT. The **fft_detailed** routine computes in place the real sequence resulting from the Fourier Transform of the conjugate symmetric sequence. Pairs of successive real elements are allocated into the real and imaginary components of complex numbers along the axis representing the conjugate symmetric sequence. This routine

    a. Internally performs the preprocessing by

        - Calling index-reversal routines if needed, performing the data communication in a close to optimal manner.

        - Performing the butterfly-like computations for the preprocessing.

    b. Internally calls the CCFFT with the proper twiddle factors.

3. Call the **CMSSL_real_from_complex** routine to extract the resulting real array.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Real-to-Complex and Complex-to-Real Fast Fourier Transform

The Fourier Transform routines described below use a real-to-complex Fast Fourier Transform (RCFFT) algorithm to compute the Discrete Fourier Transform of real data stored in a complex *n*-dimensional CM array. The real components of elements 0 and *N*/2 of each resulting conjugate symmetric sequence, together with components 1 through *N*/2 − 1, are stored in the complex array. The real components of elements 0 and *N*/2 are stored in location zero along the axis representing each conjugate symmetric sequence.

These routines also use a complex-to-real Fast Fourier Transform (CRFFT) to compute the Discrete Fourier Transform of an *n*-dimensional array of conjugate symmetric sequences stored as described above.

SYNTAX

setup_id = **fft_setup** (***A***, *type, conj_sym_axis, ier*)

fft_detailed (***A***, *type, ops, in_bit_orders, out_bit_orders, scales, setup_id, ier*)

deallocate_fft_setup (*setup_id*)

ARGUMENTS

A Complex CM array. The ***A*** you supply in the **fft_setup** call must have the same rank, axis extents, layout directives, and data type as the ***A*** arguments you supply in all subsequent **fft_detailed** calls associated with the setup call. All axes that are to be transformed must have power-of-2 extents.

When you call **fft_setup**, you may supply arbitrary values in ***A***; the setup routine neither examines nor modifies the contents of the array, but rather uses its size, shape, and data type to create the setup object.

When you call **fft_detailed**, ***A*** is the array to be transformed. Upon completion, this array is overwritten with the results.

Copyright © 1994 Thinking Machines Corporation

The elements of ***A*** may be complex or double- precision complex. The floating-point precision of the result data always matches that of the input.

conj_sym_axis Scalar integer variable. Identifies the *conjugate symmetric axis*: the axis along which real data is aligned for a real-to-complex transform, or along which each conjugate symmetric sequence is aligned for a complex-to-real transform. The ordering of array dimensions is consistent with CM Fortran, with the left-most array axis being axis 1. The extent of the conjugate symmetric axis must be a power of 2.

type Front-end variable of type CHARACTER*4 that specifies the type of FFT to be performed. Specify **'RTOC'** for a real-to-complex transform or **'CTOR'** for a complex-to-real transform. (Note: in the setup call, these two types are interchangeable, as the setup action is the same for real-to-complex and complex-to-real transforms.)

ops Front-end integer vector with length equal to the rank of ***A***. Each element specifies the direction (if any) in which the corresponding ***A*** axis is to be transformed. Each element must be one of the following symbolic constants (or the equivalent integer value):

CMSSL_nop (0) no operation
CMSSL_f_xform (1) forward transformation
CMSSL_i_xform (2) inverse transformation

in_bit_orders Front-end integer vector with length equal to the rank of **A**. Each element indicates the initial address ordering of the corresponding ***A*** axis. Each element must be one of the following symbolic constants (or the equivalent integer value):

CMSSL_normal (0) normal address ordering
CMSSL_bit_reversed (1) bit-reversed address ordering

out_bit_orders Front-end integer vector with length equal to the rank of ***A***. Each element indicates the desired output address ordering of the corresponding ***A*** axis. Each element must be one of the following symbolic constants (or the equivalent integer value):

CMSSL_normal (0) normal address ordering
CMSSL_bit_reversed (1) bit-reversed address ordering

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

scales Front-end integer vector with length equal to the rank of ***A***. Each element specifies the scaling factor (if any) for the corresponding ***A*** axis. Each element must be one of the following symbolic constants (or the equivalent integer value):

CMSSL_noscale (0) no scaling
CMSSL_scale_sqrt (1) scale by square root of the axis length
CMSSL_scale_n (2) scale by the axis length

Note that on the conjugate symmetric axis, the "axis length" is the declared length of the complex array axis, just as in the case of the CCFFT.

At the end of the **fft_detailed** call, each element of ***A*** is divided by the product of the scaling factors. (The axes for which you specified **CMSSL_noscale** do not contribute to this product.)

setup_id Scalar integer variable. The ID of a setup structure returned by the **fft_setup** routine.

ier Scalar integer variable. Upon return, contains 0 if the call succeeded. The **fft_setup** routine may return one of the following errors:

−1 You have supplied an invalid CM array.

−2 You have supplied a CM array with a data type other than single- or double-precision complex.

DESCRIPTION

The RCFFT computes the Fourier transform of real data, and the CRFFT transforms *conjugate symmetric* complex sequences. The transform X_k (for $k = 0, 1, \ldots, N-1$) of a real sequence x_n ($n = 0, 1, \ldots, N-1$), is conjugate symmetric, i.e., $X_k = \overline{X_{N-k}}$, and conversely, the transform of such a sequence is real. We refer to the domain in which the real data exists as the *real domain*, and the domain in which the conjugate symmetric sequence exists as the *complex domain*.

You can use the CMSSL RCFFT and CRFFT to perform a transform along any axis of a multidimensional array. The axis you select for the RCFFT or CRFFT is referred to as the *conjugate symmetric axis*, since after a real-to-complex transform along this axis, it represents half of a conjugate symmetric sequence. The CMSSL RCFFT and CRFFT compute the transform in place.

Copyright © 1994 Thinking Machines Corporation

To perform a RCFFT or CRFFT, you must follow these steps:

1. Call **fft_setup**.

   This routine computes internal values (including twiddle factors), allocates an internal FFT setup object, and computes an FFT setup ID, which you must supply in all subsequent Detailed FFT and deallocation calls associated with this setup call.

2. Call **fft_detailed**.

   This routine executes the FFT. You may follow one call to **fft_setup** with multiple calls to **fft_detailed**. However, the result of a call to **fft_setup** can only be used for **fft_detailed** calls using arrays with the same rank, axis extents, layout directives, and data type as the array supplied in the setup routine. If these parameters change, you must call **fft_setup** again to create another setup structure.

3. Call **deallocate_fft_setup**.

   This routine removes an FFT setup object that is no longer needed. Because an FFT setup object occupies both partition manager and processing element memory, you should free this space by deallocating the setup object after the completion of all FFT invocations that use it.

You may have more than one setup ID active at a time; that is, you may call the setup routine more than once before deallocating any setup IDs. When you call **fft_detailed** or **deallocate_fft_setup**, you must be sure to provide the correct setup ID.

Twiddle Factor Exponents. The sign of the twiddle factor exponents determines the direction of an FFT. In the CMSSL FFT, the *forward* transform is defined as one that uses twiddle factors with a *negative* exponent, and the *inverse* transform is defined as one that uses twiddle factors with a *positive* exponent.

Data Orderings. For the RCFFT and CRFFT, there are several important data ordering issues. In the complex domain, the ordering is normal when successive complex data elements along an axis are assigned to successive complex array locations, and the order is bit-reversed if the complex data are assigned to data array locations based on the bit-reversed values of their indices, as described for the CCFFT. The figure below illustrates the data allocation in the complex domain for a one-dimensional array. For the axis representing the conjugate symmetric sequence, location zero is special. It represents two real elements, namely the real components of frequency component 0 and frequency component $N/2$, both of which have an imaginary component equal to zero.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Order	**re**	**im**	**re**	**im**	**re**	**im**	**re**	**im**	**re**	**im**	**re**	**im**	**re**	**im**	**re**	**im**
Normal	0_r	8_r	1_r	1_c	2_r	2_c	3_r	3_c	4_r	4_c	5_r	5_c	6_r	6_c	7_r	7_c
Bit-reversed	0_r	8_r	4_r	4_c	2_r	2_c	6_r	6_c	1_r	1_c	5_r	5_c	3_r	3_c	7_r	7_c

Normal and bit-reversed order of a conjugate symmetric sequence allocated to a complex one-dimensional array.

In the real domain, the order is normal if the lower-ordered element in a pair of successive elements along the conjugate symmetric axis is assigned to the real component, and the higher-ordered element is assigned to the imaginary component, of a complex number. Furthermore, successive pairs must be assigned to successive complex locations along the axes to which the RCFFT is applied. The order is bit-reversed in the real domain if successive pairs of elements are assigned to complex locations along the conjugate symmetric axis based on the bit-reversed value of the index of the pair. The next figure illustrates the normal and bit-reversed ordering in the real domain.

Order	**re**	**im**	**re**	**im**	**re**	**im**	**re**	**im**	**re**	**im**	**re**	**im**	**re**	**im**	**re**	**im**
Normal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bit-reversed	0	1	8	9	4	5	12	13	2	3	10	11	6	7	14	15

Normal and bit-reversed order of a real sequence allocated to a complex one-dimensional array.

The mapping of real data into a complex array of higher dimension depends upon which axis is to be transformed. For example, consider the following 8 x 8 real array:

$$
\begin{array}{cccccccc}
x(00) & x(01) & x(02) & x(03) & x(04) & x(05) & x(06) & x(07) \\
x(10) & x(11) & x(12) & x(13) & x(14) & x(15) & x(16) & x(17) \\
x(20) & x(21) & x(22) & x(23) & x(24) & x(25) & x(26) & x(27) \\
x(30) & x(31) & x(32) & x(33) & x(34) & x(35) & x(36) & x(37) \\
x(40) & x(41) & x(42) & x(43) & x(44) & x(45) & x(46) & x(47) \\
x(50) & x(51) & x(52) & x(53) & x(54) & x(55) & x(56) & x(57) \\
x(60) & x(61) & x(62) & x(63) & x(64) & x(65) & x(66) & x(67) \\
x(70) & x(71) & x(72) & x(73) & x(74) & x(75) & x(76) & x(77)
\end{array}
$$

A real 8 x 8 array.

For an RCFFT along the vertical axis, you must map this array into a complex array as follows:

Copyright © 1994 Thinking Machines Corporation

$w_r(00)$	$w_r(01)$	$w_r(02)$	$w_r(03)$	$w_r(04)$	$w_r(05)$	$w_r(06)$	$w_r(07)$
$w_i(00)$	$w_i(01)$	$w_i(02)$	$w_i(03)$	$w_i(04)$	$w_i(05)$	$w_i(06)$	$w_i(07)$
$w_r(10)$	$w_r(11)$	$w_r(12)$	$w_r(13)$	$w_r(14)$	$w_r(15)$	$w_r(16)$	$w_r(17)$
$w_i(10)$	$w_i(11)$	$w_i(12)$	$w_i(13)$	$w_i(14)$	$w_i(15)$	$w_i(16)$	$w_i(17)$
$w_r(20)$	$w_r(21)$	$w_r(22)$	$w_r(23)$	$w_r(24)$	$w_r(25)$	$w_r(26)$	$w_r(27)$
$w_i(20)$	$w_i(21)$	$w_i(22)$	$w_i(23)$	$w_i(24)$	$w_i(25)$	$w_i(26)$	$w_i(27)$
$w_r(30)$	$w_r(31)$	$w_r(32)$	$w_r(33)$	$w_r(34)$	$w_r(35)$	$w_r(36)$	$w_r(37)$
$w_i(30)$	$w_i(31)$	$w_i(32)$	$w_i(33)$	$w_i(34)$	$w_i(35)$	$w_i(36)$	$w_i(37)$

The mapping of an 8 x 8 real array into a 4 x 8 complex array for transformation on the vertical axis.

On the other hand, for an RCFFT along the horizontal axis, you must map the array into a complex array as follows:

$w_r(00)$	$w_i(00)$	$w_r(01)$	$w_i(01)$	$w_r(02)$	$w_i(02)$	$w_r(03)$	$w_i(03)$
$w_r(10)$	$w_i(10)$	$w_r(11)$	$w_i(11)$	$w_r(12)$	$w_i(12)$	$w_r(13)$	$w_i(13)$
$w_r(20)$	$w_i(20)$	$w_r(21)$	$w_i(21)$	$w_r(22)$	$w_i(22)$	$w_r(23)$	$w_i(23)$
$w_r(30)$	$w_i(30)$	$w_r(31)$	$w_i(31)$	$w_r(32)$	$w_i(32)$	$w_r(33)$	$w_i(33)$
$w_r(40)$	$w_i(40)$	$w_r(41)$	$w_i(41)$	$w_r(42)$	$w_i(42)$	$w_r(43)$	$w_i(43)$
$w_r(50)$	$w_i(50)$	$w_r(51)$	$w_i(51)$	$w_r(52)$	$w_i(52)$	$w_r(53)$	$w_i(53)$
$w_r(60)$	$w_i(60)$	$w_r(61)$	$w_i(61)$	$w_r(62)$	$w_i(62)$	$w_r(63)$	$w_i(63)$
$w_r(70)$	$w_i(70)$	$w_r(71)$	$w_i(71)$	$w_r(72)$	$w_i(72)$	$w_r(73)$	$w_i(73)$

The mapping of an 8 x 8 real array into an 8 x 4 complex array for transformation on the horizontal axis.

The **complex_from_real** routine performs these mappings.

Multidimensional and Multiple-Instance FFTs. An RCFFT can only be performed on the conjugate symmetric axis, and must be performed *prior* to any CCFFTs you may perform on the array. Moreover, if you perform a CCFFT on the other axes of the array following an RCFFT, the result is a Fourier transform for all components of the array *except* the subarray of rank one less than the array, formed by excluding the conjugate symmetric axis and defined by location 0 along the conjugate symmetric axis.

Similarly, if the array is formatted for a CRFFT, and you perform a CCFFT prior to a CRFFT, the result is not a Fourier transform for all components of the array.

In an RCFFT, axis *conj_sym_axis* is transformed first. The CRFFT operates in the reverse order, and therefore transforms axis *conj_sym_axis* last.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

Front-end Vector Arguments. The **fft_detailed** call requires four front-end vector arguments, *ops, in_bit_orders*, *out_bit_orders*, and *scales*, whose lengths must equal the rank of ***A***. Within each of these vectors, each element specifies an option for the corresponding axis of ***A***, as follows:

- The *ops* vector specifies the direction in which the corresponding ***A*** axis is to be transformed, or specifies that the axis is not to be transformed. In a multiple-instance FFT, the axes to be transformed are the data axes, and the axes that are not to be transformed are the instance axes.

- The *in_bit_orders* and *out_bit_orders* vectors specify the order in which the original and the resulting ***A*** values are stored along each axis. In normal address ordering (**CMSSL_normal**), data values are stored consecutively. Consecutive order is the only compiler-supported data allocation scheme. In bit-reversed ordering, consecutive data values are stored according to sequential addresses whose high- and low-order bits have been swapped; that is, the consecutive allocation scheme is applied to the bit-reversed indices. Note that bit reversal is always done with respect to the *complex* array, as described above.

  Although **fft_detailed** can handle any combination of bit orderings, performance is best when the axes of an untransformed array and a transformed array have opposite bit-orderings — that is, input normal and output bit-reversed, or input bit-reversed and output normal.

- The *scales* vector specifies a scaling factor (none, square root of axis length, or axis length) for each axis. (Note that on the conjugate symmetric axis, the "axis length" is the declared length of the complex array axis, just as in the case of the CCFFT.) At the end of the **fft_detailed** call, each element of ***A*** is divided by the product of the scaling factors. (The axes for which you specified **CMSSL_noscale** do not contribute to this product.)

  For example, suppose ***A*** has dimensions (*m, n, p*), and you choose the first axis as the conjugate symmetric axis and supply

  (**CMSSL_scale_n**, **CMSSL_scale_sqrt**, **CMSSL_noscale**)

  as the *scales* vector. At the end of the **fft_detailed** call, each element of ***A*** is divided by m * sqrt(n).

  One artifact of the Fourier Transform algorithm is that if no scaling is performed, then a forward FFT followed immediately by an inverse FFT results in the original array with all values multiplied by the product of the lengths of the transformed axes. Scaling is typically used to correct this effect and prevent arithmetic overflow. Typically, a scaling factor of **CMSSL_noscale** is

Copyright © 1994 Thinking Machines Corporation

used for all axes in a forward transform, and a scaling factor of **CMSSL_scale_n** is used for all axes being transformed in an inverse transform. Alternatively, you may specify a scaling factor of **CMSSL_scale_sqrt** for all transformed axes in both the forward and the inverse FFTs. This alternative is less efficient, but can reduce the risk of arithmetic overflow for very large numbers.

Scaling is independent of the operations performed along the axes. For example, it is possible to specify a scaling factor of **CMSSL_scale_n** for an axis that is not being transformed.

Bit-Reversing the Addressing of a CM Array. You can use the Detailed FFT to bit-reverse the addressing of any *n*-dimensional CM array. To do this, specify **CMSSL_nop** for all elements of the *ops* argument, and opposite values for the input and output orderings; that is, either **CMSSL_normal** for the *in_bit_orders* values and **CMSSL_bit_reversed** for the *out_bit_orders* values, or **CMSSL_bit_reversed** for the *in_bit_orders* values and **CMSSL_normal** for the *out_bit_orders* values.

NOTES

Setup Uses Array Shape, Layout, and Data Type. The setup ID computed by the **fft_setup** call can be used for all arrays with the same shape, layout, and data type (including precision) as the *A* argument supplied in the **fft_setup** call — and *only* for such arrays. If a transform is to be performed on two arrays, *A* and *B*, of the same shape, layout, and data type, then *one* call to the setup routine suffices, even if transforms are performed on different axes of the two arrays. But if *A* and *B* have different shapes, layouts, or data types, then you must make one call for each array. Note that to be the same shape, two arrays must not only have the same number of axes, but also the same axis lengths. Thus, if two arrays with the same number of axes differ in even one axis length or in any layout directives, they require separate setup calls.

Setup is Private. The FFT setup data contains information private to the FFT. The format of this data is not documented and may change between CMSSL releases. For this reason, application code should never access or modify the contents of an FFT setup structure.

Header File. The FFT routines use predefined symbolic constants. Therefore, you must include the statement `INCLUDE '/usr/include/cm/cmssl-cmf.h'` at the

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

top of the main file of any FFT program. This file defines symbolic constants and declares the type of the CMSSL functions.

Bit Reversal and Performance. For an axis that is being transformed and that has a non-local component,

- If you specify *opposite* address orderings for input and output, the Detailed FFT performs the single bit-reversal that is inherent in the Cooley-Tukey algorithm, and produces output that is bit-reversed relative to the input.
- If you specify the *same* address ordering for input and output, the Detailed FFT performs an *extra* bit reversal, and produces output with the same ordering as the input. The extra bit reversal exacts a performance cost.

For an axis that is being transformed and that is local to a processing element,

- If you specify normal input and bit-reversed output on the conjugate symmetric axis for an RCFFT, *or* bit-reversed input and normal output on the conjugate symmetric axis for a CRFFT, the Detailed FFT performs *two extra* bit reversals.
- All other cases behave as in the non-local case, above.

For an axis that is not being transformed,

- If you specify *opposite* address orderings for input and output, the Detailed FFT performs a single bit-reversal and produces output that is bit-reversed relative to the input.
- If you specify the *same* address ordering for input and output, the Detailed FFT performs *no* bit reversal.

To bit-reverse the address ordering of a dataset without transforming it, call the Detailed FFT specifying no transform for all axes and specifying *opposite* values for the input and output address ordering.

These points are independent of the directions of the transforms.

Axis Transformation Order. In an RCFFT, axis *conj_sym_axis* is transformed first. The CRFFT operates in the reverse order, and therefore transforms axis *conj_sym_axis* last. See the section called "Multidimensional and Multiple-Instance FFTs" in the description above for cautions related to multidimensional FFTs.

Copyright © 1994 Thinking Machines Corporation

FFT Inversion. To invert an RCFFT, you must change the operation from forward to inverse as well as changing the FFT type from real-to-complex to complex-to-real. The **'RTOC'** and **'CTOR'** keywords for the *type* argument select the algorithm (FFT type); the *ops* argument chooses the sign of the exponents of the twiddles, and thus the direction of the transform.

EXAMPLES

Sample CM Fortran code that uses the RCFFT and CRFFT can be found on-line in the subdirectory

```
fft/cmf/
```

of a CMSSL examples directory whose location is site-specific.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

10.4 Array Conversion Utilities for the Real-to-Complex and Complex-to-Real FFTs

CMSSL offers two array conversion utilities:

- The **complex_from_real** routine converts a real array to a complex array suitable for input to the RCFFT.
- The **real_from_complex** routine converts a complex array to a real array. The conversion assumes that the elements of the complex array are ordered as in an array transformed by the CRFFT.

These routines are intended primarily for use with the RCFFT and CRFFT, but can be used with any real and complex arrays whose ranks and axis extents satisfy the necessary conditions. Details are provided in the man page that follows.

Copyright © 1994 Thinking Machines Corporation

Array Conversion Utilities for the Real-to-Complex and Complex-to-Real FFTs

The **complex_from_real** routine converts a real array to a complex array suitable for input to the real-to-complex Fast Fourier Transform (RCFFT). The **real_from_complex** routine converts a complex array, whose elements are ordered as in an array transformed by the complex-to-real FFT (CRFFT), to a real array.

SYNTAX

complex_from_real (***A, B,*** *axis*)

real_from_complex (***B, A,*** *axis*)

ARGUMENTS

A Complex CM array of the same rank as ***B***. Corresponding axes of ***A*** and ***B*** must have equal extents, with the exception of the axis identified by the *axis* argument. Axis *axis* of ***A*** must have half the length of axis *axis* of ***B***.

When you call **complex_from_real**, ***A*** is the complex array resulting from the conversion, and is an output argument.

When you call **real_from_complex**, ***A*** is the complex CM array you wish to convert to a real array, and is an input argument.

B Real CM array of the same rank as ***A***. Corresponding axes of ***A*** and ***B*** must have equal extents, with the exception of the axis identified by the *axis* argument. Axis *axis* of ***B*** must have twice the length of axis *axis* of ***A***.

When you call **complex_from_real**, ***B*** is the real CM array you wish to convert to a complex array, and is an input argument.

When you call **real_from_complex**, ***B*** is the real array resulting from the conversion, and is an output argument.

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

axis The axis that determines the mapping of real to complex or complex to real data elements, as explained in the Description below. When **complex_from_real** and **real_from_complex** are used in conjunction with the RCFFT and CRFFT, *axis* is the axis along which the transform is performed (the conjugate symmetric axis).

DESCRIPTION

Real-to-Complex Conversion. The **complex_from_real** routine converts ***B*** to a complex array and places the results in ***A***. Each element of ***B*** is mapped to the real or imaginary part of an element of ***A***. (This storage scheme is more efficient than a scheme that maps real elements exclusively to the real parts of a complex array.) If your arrays are one-dimensional, and if ***A*** has extent n and ***B*** has extent $2n$, then **complex_from_real** performs the equivalent of the following CM Fortran statements:

```
forall (i=1:n)
    A(i) = cmplx(B(2*i-1),B(2*i))
```

If your arrays have rank greater than 1, the mapping of real to complex data elements depends on the axis you specify in the *axis* argument; the mapping defined above occurs along axis *axis*. If you define

$\boldsymbol{B}(n_1, n_2, ..., 2n_{axis}, ..., n_p)$
$\boldsymbol{A}(n_1, n_2, ..., n_{axis}, ..., n_p)$

where $2n_{axis}$ is the extent of axis *axis* of ***B*** and n_{axis} is the extent of axis *axis* of ***A***, then **complex_from_real** performs the equivalent of the following CM Fortran statements:

```
forall (i=1:n1, j=1:n2, ..., m=1:naxis, ..., t=1:np)
    A(i,j, ..., m, ..., t) =
&       cmplx(B(i,j,...,2*m-1,...,t),
&           B(i,j,...,2*m,...,t))
```

For example, if ***B*** is

$x(00)$	$x(01)$	$x(02)$	$x(03)$	$x(04)$	$x(05)$	$x(06)$	$x(07)$
$x(10)$	$x(11)$	$x(12)$	$x(13)$	$x(14)$	$x(15)$	$x(16)$	$x(17)$
$x(20)$	$x(21)$	$x(22)$	$x(23)$	$x(24)$	$x(25)$	$x(26)$	$x(27)$
$x(30)$	$x(31)$	$x(32)$	$x(33)$	$x(34)$	$x(35)$	$x(36)$	$x(37)$

Copyright © 1994 Thinking Machines Corporation

and *axis* = 2, then each element is mapped to the corresponding location in the following depiction of ***A***:

$w(00)_r$	$w(00)_i$	$w(01)_r$	$w(01)_i$	$w(02)_r$	$w(02)_i$	$w(03)_r$	$w(03)_i$
$w(10)_r$	$w(10)_i$	$w(11)_r$	$w(11)_i$	$w(12)_r$	$w(12)_i$	$w(13)_r$	$w(13)_i$
$w(20)_r$	$w(20)_i$	$w(21)_r$	$w(21)_i$	$w(22)_r$	$w(22)_i$	$w(23)_r$	$w(23)_i$
$w(30)_r$	$w(30)_i$	$w(31)_r$	$w(31)_i$	$w(32)_r$	$w(32)_i$	$w(33)_r$	$w(33)_i$

In this depiction, *r* and *i* indicate the real and imaginary parts of a data element, respectively. Thus, elements $x(00)$ and $x(01)$ of ***B*** are mapped to the real and imaginary parts, respectively, of element $w(00)$ of ***A***.

Complex-to-Real Conversion. The **real_from_complex** routine converts ***A*** to a real array and places the results in ***B***. Each real or imaginary part of an element of ***A*** is mapped to an element of ***B***. If your arrays are one-dimensional, and if ***A*** has extent *n* and ***B*** has extent $2n$, then **real_from_complex** performs the equivalent of the following CM Fortran statements:

```
forall (i=1:n)
    B(2*i-1) = real(A(i), mold = 0.0)
forall (i=1:n)
    B(2*i) = aimag(A(i))
```

If your arrays have rank greater than 1, the mapping of complex to real data elements depends on the axis you specify in the *axis* argument; the mapping defined above occurs along axis *axis*. If you define

$\boldsymbol{B}(n_1, n_2, ..., 2n_{axis}, ..., n_p)$
$\boldsymbol{A}(n_1, n_2, ..., n_{axis}, ..., n_p)$

where $2n_{axis}$ is the extent of axis *axis* of ***B*** and n_{axis} is the extent of axis *axis* of ***A***, then **real_from_complex** performs the equivalent of the following CM Fortran statements:

```
 forall (i=1:n1, j=1:n2, ..., m=1:naxis, ..., t=1:np)
     B(i,j, ..., 2*m-1, ..., t) =
&         real(A(i,j,...,m,...,t),mold=0.0)
 forall (i=1:n1, j=1:n2, ..., m=1:naxis, ..., t=1:np)
     B(i,j,...,2*m,...,t)=
&         aimag(A(i,j,...,m,...,t))
```

This mapping is the inverse of that performed by **complex_from_real**.

Use with FFT. The complex CM array produced by the **complex_from_real** routine is set up correctly for input to the RCFFT. The axis identified by the *axis* argument is

Version 3.2, April 1994
Copyright © 1994 Thinking Machines Corporation

the axis to be transformed (called the *conjugate symmetric axis* in the FFT documentation). Similarly, the conversion performed by the **real_from_complex** routine assumes that the data elements of the input *A* are ordered as in an array transformed by the CRFFT. A typical program structure is as follows:

```
setup_id = fft_setup (A, 'RTOC', axis, ier)
B = ...
                              .
                              .
                              .
call complex_from_real (A, B, axis)
call fft_detailed (A, 'RTOC',...)
                              .
                              .
                              .
call fft_detailed (A, 'CTOR',...)
                              .
                              .
                              .
call real_from_complex (B, A, axis)
```

EXAMPLES

Sample CM Fortran code that uses the array conversion utilities can be found online in the subdirectory

fft/cmf/

of a CMSSL examples directory whose location is site-specific.

Copyright © 1994 Thinking Machines Corporation

10.5 References

For further information about the CCFFT, see the following references:

1. Buzbee, B. L., G. H. Golub, and C. W. Nielson. On Direct Methods for Solving Poisson's Equations. *SIAM J. Numer. Anal.* **7**, no. 4 (1970): 627–56.

2. Chu, C. *The Fast Fourier Transform on Hypercube Parallel Computers*. Ph.D. Thesis, Center for Applied Mathematics, Cornell University, Ithaca NY 14585.

3. Cooley, J. C., P. Lewis, and P. D. Welch. The Fast Fourier Transform Algorithm: Programming Considerations in the Calculation of the Sine, Cosine, and Laplace Transforms. *J. Sound Vibrations* **12**, no. 3 (1970): 315–37.

4. Cooley, J. C., and J. W. Tukey. An Algorithm for the Machine Computation of Complex Fourier Series. *Math. Comp.* **19** (1965): 291–301.

5. Edelman, A. *Optimal Matrix Transposition and Bit-Reversal on Hypercubes: Node Address–Memory Address Exchanges.* Thinking Machines Corporation Technical Report, 1989.

6. Hockney, R. W. The Potential Calculation and Some Applications. *Methods Comput. Phys.* **9** (1970): 135–211.

7. Hong, J. W., and H. T. Kung. I/O Complexity: The Red-Blue Pebble Game. In *Proc. of the 13th ACM Symposium on the Theory of Computation*. ACM, 1981. Pp. 326–33.

8. Johnsson, S. L. Communication Efficient Basic Linear Algebra Computations on Hypercube Architectures. *J. Parallel Distributed Comput.***4**, no. 2 (1987): 133–72.

9. Johnsson, S. L. and C-T. Ho. Matrix Transposition on Boolean n-Cube Configured Ensemble Architectures. *SIAM J. Matrix Anal. Appl.* **9**, no. 3 (1988): 419–54.

10. Johnsson, S. L. and C-T. Ho. *Optimal Communication Channel Utilitzation for Matrix Transposition and Related Permutations on Boolean Cubes.* Technical Report TR–03–91, Harvard University, Division of Applied Sciences, January 1991.

Copyright © 1994 Thinking Machines Corporation

11. Johnsson, S. L., C-T. Ho, M. Jacquemin, and A. Ruttenberg. Computing Fast Fourier Transforms on Boolean Cubes and Related Networks. In *Advanced Algorithms and Architectures for Signal Processing II,* **826**. Society of Photo-Optical Instrumentation Engineers, 1987. Pp. 223–31.

12. Johnsson, S. L., Jacquemin, M., and C-T. Ho. *High Radix FFT on Boolean Cube Networks*. Harvard University, Division of Applied Sciences, Technical Report TR–25–91. To appear in the *Journal of Computational Physics*.

13. Johnsson, S. L. and R. L. Krawitz. Cooley-Tukey FFT on the Connection Machine. *Parallel Computing* **18**, no. 11 (1992): 1201–21.

14. Johnsson, S. L., M. Jacquemin, and R. L. Krawitz. Communication Efficient Multi-Processor FFT. *J. Comp. Phys* **102**, no. 2 (1992): 381–97.

15. Johnsson, S. L and C.–T. Ho. Boolean Cube Emulation of Butterfly Networks Encoded by Gray Code. Yale University Department of Computer Science, Technical Report YALEU/DCS/RR–764, 1990. Also Thinking Machines Corporation Technical Report BA90–1, TMC–5. To appear in *Journal of Parallel and Distributed Computing*.

16. Swarztrauber, P. N. The Methods of Cyclic Reduction, Fourier Analysis, and the FACR Algorithm for the Discrete Solution of Poisson's Equation on a Rectangle. *SIAM Review* **19** (1977): 490–501.

17. Swarztrauber, P. N. Multiprocessor FFTs. *Parallel Computing* **5** (1987): 197–210.

18. Temperton, C. On the FACR(1) Algorithm for the Discrete Poisson Equation. *J. of Computational Physics* **34** (1980): 314–29.

19. Van Loan, C. *Computational Frameworks for the Fast Fourier Transform*. SIAM, 1992.

In addition, for information about the CRFFT and RCFFT, see the following references:

20. Bergland, G.D. A fast Fourier transform for real-valued series. *Comm. ACM* **11** (1968): 703–10.

21. Briggs, W.L. Further symmetries of in-place FFTs. *SIAM J. Sci. Stat. Comput*. **8** (1987): 644–54.

Copyright © 1994 Thinking Machines Corporation

22. Pelz, R.B. Parallel compact FFTs for real sequences. Submitted to *SIAM J. Sci. Stat. Comput.*

23. Swarztrauber, P.N. Symmetric FFTs. *Math. Comp*. **47** (1986): 323−46.

24. Swarztrauber, P.N. and S. L. Johnsson. Multiprocessor FFTs for real sequences. Technical Report, National Center for Atmospheric Research, August 1991.

25. Sweet, R.A., V.E. Henson, and J. Porsche. A fast Fourier transform for real data on the hypercube. In preparation.

Copyright © 1994 Thinking Machines Corporation