The Inverse Shuffle Transpose and Multidimensional Fast Fourier Transforms

Alan H. Karp
Hewlett-Packard Laboratories
alan.karp@hp.com

October 6, 2003

Abstract

Combining the steps of a multi-pass transpose algorithm with the butterflies of a Fast Fourier Transform (FFT) algorithm improves the performance of the calculation of multi-dimensional FFTs on machines with hierarchical memories. This paper introduces the Inverse Shuffle Transpose and shows how it can be combined with an extension of the Pease FFT to produce a multidimensional FFT that avoids explicit transposes yet accesses memory with good spatial locality. This approach has the additional advantages of being independent of the number of dimensions and having uniform memory access.

1 Introduction

Discrete Fourier transforms are important in a wide variety of applications, from modeling underground formations in oil exploration to the search for extraterrestrial intelligence. Nevertheless, the Fourier tranform would be merely a curiosity were it not for the Fast Fourier Transform (FFT) [**?**], which reduces the complexity of a Fourier transform of length N from $O(N^2)$ to $O(N \log_2 N)$ when N is a power of 2. The FFT relies on the composite nature of N and symmetries in the complex roots of unity; the computational complexity is still $O(N^2)$ when N is prime. (The Fractional FFT [**?**] has better scaling in this case.) A two dimensional FFT applies the algorithm separately to each row and then to each column, with the obvious extension to more dimensions.

The performance of algorithms on modern computers often depends more on the memory access patterns than on the time to do the arithmetic. Some machines have memory banks that can only deliver data every few cycles; accessing the a bank before it is ready results in a delay. Others have set associative caches that have only a small number of locations that can hold a particular piece of data [**?**]. FFT algorithms that involve power of two strides perform poorly on such machines. Other machines have hierarchical memories that deliver blocks of data to successively smaller, but faster, cache storage units. FFT algorithms with little spatial

or temporal locality in the data references perform poorly on these machines. Still other machines have all these problems.

This paper is concerned with multidimensional FFTs on machines with hierarchical memories. Such machines perform well in either of two modes, namely

- algorithms that move data to cache and use it many times, and
- algorithms that stream the data through the processor, using every word tranferred at least once.

Algorithms with neither spatial nor temporal locality perform poorly.

There are one dimensional FFT algorithms, *e.g.* Stockham [**?**], that access the data at small stride. However, after computing the FFT of the rows at low stride, the next step of the multidimensional calculation computes the FFT over the columns, a large stride access. Most implementations avoid this problem by explicitly transposing the array before the second step, and efficient transpose algorithms have been developed [**?**]. Others have developed block algorithms that do portions of the 2D FFT while blocks of data are in cache. Unfortunately, the code for such implementations is complicated by the data blocking, and a different version is needed for each number of dimensions.

Section 2 introduces the *Inverse Shuffle Transpose*. This transpose is combined with a variant of the Pease FFT in Section 3. In Section 4 the performance of this approach is compared with several old and existing implementations provided by computer vendors as part of their math libraries.

2 Inverse Shuffle Transpose

Transposing a 2D array is easy. An NxM array can be transposed by

```
for i = 1 to N
  for j = 1 to M
     B(i,j) = A(j,i)
```

The problem with this approach is that array A is accessed at large stride.[1] Even worse, if the dimensions of A are powers of two, as is common for FFTs, then we have added problems with memory banks and cache access. Some machines with hierarchical memories allow bypassing the cache on stores, so it might be better to access array B at large stride. However, even on these machines the main memory often has banks.

Since the simplest approach performs poorly on many machines, we'll need to do something more complicated. One approach is to divide arrays A and B into blocks that fit into the cache. We can then apply the simple algorithm for each block. Using this approach also allows us to to a transpose in place at the cost of a work array the size of the block.

[1] Throughout this paper we'll assume that the arrays are stored in row major order.

Although block transposes are quite efficient, Section 3 shows that it is worth looking at multi-pass algorithms. Here we'll introduce the *inverse shuffle transpose*. Recall that the perfect shuffle of an array X of even length N is the mapping

```
for i = 0 to N/2-1
  Y(2*i  ) = X(i)
  Y(2*i+1) = X(i+N/2)
```

The name of this mapping comes from the analogy with cards – cutting the deck into equal parts and interleaving the resulting halves. The equivalent inverse shuffle is

```
for i = 0 to N/2-1
  Y(i)     = X(2*i)
  Y(i+N/2) = X(2*i+1)
```

which undoes the perfect shuffle. The inverse shuffle is sometimes referred to as the *even-odd sort permutation*.

2.1 Radix 2 Transpose

If array A has dimensions N_1 and N_2, where N_1 and N_2 are powers of 2, we can think of A as a linear array X of $N = N_1N_2$ elements. The (i_1,i_2) element of A is at location $i_2 + i_1N_2$ in X, where $0 \le i_1 < N_1$, and $0 \le i_2 < N_2$. Letting $N_2 = 2^{n_2}$, we can transpose A in n_2 inverse shuffle steps.

The proof uses integer (truncating) arithmetic in which $7/2 = 3$, and the notation "//" for modulo arithmetic, *e.g.* $7//2 = 1$. Let $k = i_2 + i_1N_2$, $0 \le k < N$, be the index of $A(i_1, i_2)$ in the linear representation. After $p < n_2$ inverse shuffle steps element k is at location

$$k_p = k/2^p + (N/2^p)\sum_{r=0}^{p-1} 2^r[(k/2^r)//2]. \tag{1}$$

The proof follows by induction. The first inverse shuffle step, $p = 1$, moves element k to location $k_1 = k/2$ if k is even and to $k_1 = k/2 + N/2$ if k is odd. We can represent this mapping by

$$k_1 = k/2 + (N/2)(k//2). \tag{2}$$

Note that $k//2$ is the low order bit of k, which we'll denote d_0. Because N_2 is a multiple of 2, $k//2 = i_2//2$ making d_0 the low order bit of i_2 as well. Equation 1 also produces this result.

The second step, $p = 2$, moves element k_1 to position $k_2 = k_1/2$ if k_1 is even and to $k_1/2 + N/2$ if k_1 is odd. As before, we have

$$k_2 = k/4 + (N/4)d_0 + (N/2)[(k/2)//2]. \tag{3}$$

We note that $d_1 = (k/2)//2 = (i_2/2)//2$ is the second least significant bit of i_2 because N_2 is a multiple of 4. Again, we see that Equation 1 satisfies this equation.

For step $p+1$, we have

$$k_{p+1} = k/2^{p+1} + (N/2^{p+1}) \sum_{r=0}^{p-1} d_r 2^r + (N/2)[(i_2/2^p)//2], \tag{4}$$

which is

$$k_{p+1} = k/2^{p+1} + (N/2^{p+1}) \sum_{r=0}^{p} d_r 2^r, \tag{5}$$

completing the proof.

Applying this equation to find the location of element k after n_2 steps we see that

$$k_{n_2} = k/2^{n_2} + N_1(N_2/2^{n_2}) \sum_{r=0}^{n_2-1} d_r 2^r. \tag{6}$$

Using the facts that $N_2 = 2^{n_2}$, $k = i_2 + i_1 N_2$ and $i_2 < N_2$ gives

$$k_{n_2} = i_1 + N_1 \sum_{r=0}^{n_2-1} d_r 2^r. \tag{7}$$

We see that $k_{n_2} = i_1 + i_2 N_1$ because the summation in Equation 7 is just the binary representation of i_2. Hence, k_{n_2} is at the location of $A(i_2, i_1)$ when that array is treated as stored in row major order.

2.2 Mixed Radix

The inverse shuffle generalizes to mixed radix problems. For a factor f, we take f elements and move them to every N/fth position. In other words, element (i_1, i_2), which is at location $k = i_2 + i_1 N_2$ in the linear representation, goes to location $k_1 = k/f + (N/f)(k//f)$.

Say that $N_2 = \prod_{j=1}^{F} f_j$, where the factors f_j need not be prime. Then for $1 \le p \le F$, the p'th inverse shuffle step is

$$k_p = k/F_p + N/F_p \sum_{r=0}^{p-1} d_r F_r, \tag{8}$$

where $F_p = \prod_{j=1}^{p} f_j$, $F_0 = 1$, and $0 \le d_r < f_r$ is the rth digit of both k and i_2. Hence, $k_F = i_1 + i_2 N_1$, as in the radix-2 case. When $f_1 = N_2$, the inverse shuffle is exactly the simple transpose presented in Section 2.

2.3 Higher Dimensionality

An important property of the inverse shuffle transpose is that it affects only the last dimension of the mapping. That means that the first dimension in the representation can be a composite of all the remaining dimensions of a multidimensional array. Hence, if the array has dimensions N_q, $1 \leq q \leq Q$, and we apply the inverse shuffle transpose, the result is a cyclic permutation of the dimensions. In other words, we end up with an array of dimension $N_Q, N_1, N_2, \ldots, N_{Q-1}$. Repeating the process Q times returns the data to its original order.

We can see this behavior by applying the inverse shuffle to an array with Q dimensions. Element $A(i_1, \ldots, i_Q)$ is at location

$$k = N_Q \sum_{j=1}^{Q-2} i_j \prod_{m=j+1}^{Q-1} N_m + N_Q i_{Q-1} + i_Q \tag{9}$$

in the linear representation. For example, a three dimensional array element at $A(i_1, i_2, i_3)$ is at location $k = i_1 N_3 N_2 + i_2 N_3 + i_3$ in the linear representation.

If $N_Q = \prod_{j=1}^{F} f_j$, the F'th inverse shuffle step moves element k to position

$$k_F = \sum_{j=1}^{Q-2} i_j \prod_{m=j+1}^{Q-1} N_m + i_{Q-1} + (N/N_Q) \sum_{r=0}^{p-1} d_r F_r. \tag{10}$$

The second summation is the decomposition of i_Q, so we see that this index has moved to the first position in the representation, and al the others are shifted one position.

3 Multidimensional FFT

There are many ways to compute the FFT. Some can be done without requiring additional storage; some require a bit reversal; others operate at low stride. Here we'll use a variant due to Pease that has the same memory access pattern as the inverse shuffle transpose.

3.1 Radix-2 Pease Formulation

The Pease formulation of the FFT of a 1D array x of length $N = 2^n$ starts with a bit reversal permutation, denoted by the permutation P_N. It then performs n *butterfly* stages.

```
for f = 1 to n
  for j = 0 to N-1
    if f == 1
      y(j) = x(P(j))
    else
      y(j) = x(j)
  for j = 0 to N/2-1
```

```
x(j)     = y(2*j) + W(f,j)*y(2*j+1)
x(j+N/2) = y(2*j) - W(f,j)*y(2*j+1)
```

where $W(f, j)$, the *twiddle factors*, are defined in Section 3.2. At the end, the result is stored in array x. This pseudo-code is not the most efficient implementation. In particular, the first loop over j is not needed. The exchance can be done by pointer swapping, and the bit reversal can be combined with the first butterfly.

There are two disadvantages to the Pease formulation. First, it can not be done in place, which limits the size of arrays that can be tranformed using an in-memory algorithm. This problem is somewhat mitigated by the fact that only enough extra space is needed for half the array because the first half can be reused.

The second disadvantage of the Pease formulation is that it requires a bit reversal. However, bit reversals for multidimensional FFTs should not be a problem. Consider a 3D FFT that fills the 1 GB memory of a machine. If the dimensions are of equal size, then each bit reversal involves only 1 KB of data, an amount small enough to fit in cache. Even a 2D FFT of this total size has rows of 32 KB, which also fit in cache. Efficient bit reversals can be done, even when the row doesn't fit in cache [**?**].

3.2 Radix-2 Twiddle Factors

The array of twiddle factors, W, is a particular arrangement of the roots of unity, $\omega_N^j = \exp(2\pi ij/N)$, where $i = \sqrt{-1}$. The first thing to note is that each row of W has length $N/2$ because $W(f, j + N/2) = -W(f, j)$ for $0 \leq j < N/2$. With this observation, a convenient representation is to write $W(f, j) = (\omega_N^j)^{N/2^f}$, with $1 \leq f \leq n$. Hence, the first row of W consists of all ones. The first half of the second row is ones and all the elements in the second half are i, the positive fourth root of unity. The last row of the array consists of the roots of unity with positive imaginary part.

Although W doesn't consume too much space, we can access the same numeric values by properly computing the index into a list of the roots of unity. In particular, replacing $W(f, j)$ in the pseudo-code with $w(j/(N/2^f))$, where $w(j) = \omega_N^j$, gives us the desired coefficients. We need not worry about the strided access to w because the array will fit in cache for the multidimensional problems we're considering.

3.3 Radix-2 Multidimensional FFT

The data access pattern in the radix-2 Pease formulation is nearly identical to that of the inverse shuffle transpose. The only change is to store the odd numbered terms, those computed from the difference, in the second half of the output array instead of the second half of the row being computed.

```
for q = 1 to Q
   w = roots(n(q))
   for f = 1 to n(q)
      do j = 0, N - 1
```

```
   if q == 1
      k = mod(j,N(q))
      y(j) = x(P(k))
   else
      y(j) = x(j)
do j = 0 to N/2 - 1
   k = (N(q)/2**f)*(j/(N/2**f))
   x(j    ) = y(2*j)+w(k)*y(2*j+1)
   x(j+N/2) = y(2*j)-w(k)*y(2*j+1)
```

There are only a few differences from this code and that in Section 3.1. First, there is a loop over the number of dimensions. Also, the bit reversals are done separately for each row by computing the index modulo the length of a row. Finally, the indexing of the twiddle factor array is more complicated.

This last point warrants discussion. In Section 3.2 we used an index of $j/(N/2^f)$, where N in that case corresponds to N_q in the multidimensional case. In the pseudo-code, though, we use an index of $(N_q/2^f)(j/(N/2^f)$ because the elements move relative to the array as a whole instead of relative to the row.

3.4 Mixed Radix Pease

It shouldn't come as a surprise to find out that the data access pattern for the mixed radix Pease forumulation is the same as that of the mixed radix inverse shuffle transpose. There are two issues that we need to deal with, the generalization of the bit reversal and the handling of the twiddle factors. In both cases there is an ambiguity in what factorization is being used. For example, 96 can be factored as $(32, 3), (3, 32), (8, 4, 3), (3, 8, 4), \ldots$. Rather than pick a factorization at random, we require the programmer to specify each dimension as an array of factors to be used. In terms of the notation used earlier, we write $N_q = \prod_{f=1}^{F} N_{qf}$.

We need to implement a digit reversal given a factorization of N_q. Since we're interested in bit reversing sequences that fit in cache, we'll take the direct approach of computing an index vector that can be used for a gather operation. We also take a brute force approach to computing this index vector. The jth element in the row is at

$$j = j_1 + \sum_{f=2}^{F} j_f \prod_{k=2}^{f} N_{qk}. \tag{11}$$

After reordering that element will be at position

$$j_r = j_F + \sum_{f=1}^{F-1} j_{F-f} \prod_{k=F-f}^{F-1} N_{qk} \tag{12}$$

in the digit reversed array. The permutation becomes $P_{N_q}(j) = j_r$.

We can use a very simple modification of the way we get the twiddle factors for radix-2 problems. Instead of using a factor of 2^f we use $M_{qf} = \prod_{k=1}^{f} N_{qk}$

when computing an index at the fth butterfly. Hence, the twiddle factor we use at butterfly f is $w((N_q/M_{qf})(j/(N_q/M_{qf})))$.

4 Performance

References