IMPLEMENTING FFT'S IN SISAL*

Dorothy Bollman, Flor Sanmiguel, and Jaime Seguel
Department of Mathematics
University of Puerto Rico at Mayaguez, PR 00681-5000

Abstract. Tensor notation is a powerful tool for improvising fast Fourier transform (FFT) algorithms in a functional language. In this work we apply this idea to the development of FFT's in the functional language Sisal. As an example, we develop a high performance Sisal implementation of a variant of Pease's FFT on a Cray Y-MP.

1. Introduction. Although the utility of tensor notation for FFT algorithms has long been recognized [3], it has been only recently [4], [8], [9], that this notation has gained wide spread acceptance by FFT practitioners. Tensor notation aids not only in algorithm expression and comprehension, but also in algorithm development and is an ideal tool for developing parallel-vector algorithms for functional programs. In this work we describe work in progress that exploits this idea to develop a family of high performance FFT's in the functional language Sisal. As an example, we present a high performance Sisal implementation of a variant of the radix 4 Pease [6] algorithm.

In this section, we establish definitions and notation. In Section 2, we review a well-known family of FFT's that are variants of the original Cooley-Tukey [1] algorithm and discuss their implementations in Sisal. In Sections 3 and 4 we compare Sisal implementations of variants of the radix 4 Korn-Lambiote and Pease algorithms. The Pease variant outperforms Korn-Lambiote as well as Cann's [1] implementation of Stockham's algorithm.

The n-point DFT of a complex sequence $x = x(k)$ of period n is the complex sequence of period n determined by the set of equations

$$y(j) = \sum_{k=0}^{n-1} \omega_n^{jk} x(j), \quad 0 \le j \le n-1, \tag{1}$$

where $\omega_n = e^{-2\pi i/n}$ and $i = \sqrt{-1}$. In matrix notation, the DFT is given by

$$y = F_n x, \quad F_n = (\omega_n^{ij}), \quad 0 \le i,j \le n-1. \tag{2}$$

Thus the DFT of a vector of length n can be computed in time $O(n^2)$. However in 1965, Cooley and Tukey [3] showed that the DFT can be computed in time $O(n\ logn)$. Subsequently, variations of the Cooley-Tukey algorithm, which vary in vector size and data flow, have appeared in the literature. These algorithms together with the original Cooley-Tukey algorithm are generically known as fast Fourier transforms (FFT's). In the next section we shall describe a family of FFT's in terms of their tensor formulations.

The tensor product of an $m \times m$ matrix $A = (a_{ij})$, $i,j = 0,...,m-1$, by an $n \times n$ matrix B is the $mn \times mn$ matrix defined by

$$A \otimes B = (a_{ij}B). \tag{3}$$

If x is a vector of length mn and if we partition x into m blocks of length n, $X_0, X_1, \ldots, X_{m-1}$, then

$$(I_m \otimes B)x = (BX_0, BX_1, \ldots, BX_{m-1}). \tag{4}$$

Thus, the computation of $(I_m \otimes B)x$ can be thought of as the *parallel* application of B to the m vectors $X_0, X_1, \ldots, X_{m-1}$. On the other hand, it is easily shown that

$$(B \otimes I_m)x = (b_{0,0}X_0 + \ldots + b_{0,n-1}X_{n-1}, \ldots, b_{n-1,0}X_0 + \ldots + b_{n-1,n-1}X_{n-1}), \tag{5}$$

where $B = (B_{ij})$, $\quad 0 \le i,j \le n-1$, and $X_0, X_1, \ldots, X_{n-1}$ represents a partitioning of x into n blocks of length m. In view of (5), $B \otimes I_m$ has been termed a *vector* operation. Although these parallel/vector

* This work was supported by NSF grant RII-8905080, the Computational Mathematics Group of Puerto Rico EPSCoR II grant, and the NSF Cornell NSI Army Research Office grant.

interpretations of the operations $(I_m \otimes B)x$ and $(B \otimes I_m)x$ are convenient for conceptional purposes, the actual machine vectorization/parallelizations depend on their use in programs and decisions made by the compiler.

An important aspect of FFT implementations is data communication, which is manifested in the tensor formulations by permutation matrices, or stride permutations. A *stride* permutation $P(mn, n)$ acting on a vector of length mn is defined by

$$P(mn,n)x = ((x_0, x_n, \ldots, x_{(m-1)n}), (x_1, x_{n+1}, \ldots, x_{(m-1)n+1}), \ldots (x_{n-1}, x_{2n-1}, \ldots, x_{mn-1})), \tag{6}$$

where $x = (x_0, x_1, \ldots, x_{mn-1})$.

A useful property of stride permutations is the so called Commutation Theorem:

$$P(mn,n)(A \otimes B) = (B \otimes A)P(mn,n), \tag{7}$$

where A and B are $m \times m$ and $n \times n$, respectively.

2. A family of FFT's. We define

$$D_s^{rs} = diag(1, \omega_{rs}, \omega_{rs}^2, \ldots, \omega_{rs}^{s-1}) \tag{8}$$

and

$$T_s^{rs} = \bigoplus_{i=0}^{r-1} (D_s^{rs})^i, \tag{9}$$

where

$$(D_s^{rs})^i = diag(1, \omega_{rs}^i, \ldots, {\omega_{rs}}^{i(s-1)}) \tag{10}$$

and where $\oplus$ respresents direct sum, i.e., T_s^{rs} is a matrix whose nonzero elements consist of $(D_s^{rs})^i$, $i = 0, 1, \ldots, r-1$, along the main diagonal. In what follows we denote T_{ri-1}^{ri} simply by T^{r^i}.

An FFT for computing $F_n x$ can be described in terms of a factorization of the matrix F_n. The FFT's discussed here, the proofs of which can be found in, e.g., [8], all follow from the following identity:

$$F_{rs} = (F_r \otimes I_s)T_s^{rs}(I_r \otimes F_s)P(sr, r). \tag{11}$$

The original Cooley-Tukey algorithm in tensor notation is:

$$F_{r^k} = \{\prod_{i=1}^{k} (I_{r^{k-i}} \otimes F_r \otimes I_{r^{i-1}})(I_{r^{k-i}} \otimes T^{r^i})\} R(r^k), \tag{12}$$

where $R(r^k)$ represents the digit-reversal permutation. Note the succint way in which the tensor notation bares the idea of the algorithm. In view of the above definitions, formula (12) says that in order to compute the FFT of a vector x of length r^k, we perform a digit-reversal permuation on x. Then for each stage $i = 1, \ldots, k$, apply the "twiddle" factor T^{r^i} to each of r^{k-i} blocks of length r^i, followed by r^{k-i} applications of the "butterfly" operations $F_r \otimes I_{r^{i-1}}$ to each resulting block.

One of the problems with implementing the Cooley-Tukey algorithm on parallel/vector machines is that vector lengths become small and/or that overparallelzation takes place for small values of i. More suitable algorithms for parallel/vector machines are the "parallel" algorithm

$$F_{r^k} = \{\prod_{i=1}^{k} P(r^k, r)(I_{r^{k-1}} \otimes F_r)P(r^k, r^{k-1})(T^{r^i} \otimes I_{r^{k-i}})P(r^k, r)\} R(r^k), \tag{14}$$

originally developed by Pease [6] and its "vector" variant

$$F_{r^k} = \{\prod_{i=1}^{k} (F_r \otimes I_{r^{k-1}})(T^{r^i} \otimes I_{r^{k-i}})P(r^k, r)\} R(r^k), \tag{13}$$

first described by Korn-Lambiote [5].

Each of the above algorithms is given in its "decimation in time" form, i.e., the "combine phase" is applied to the result of digit-reversal permuting the input vector. In the "decimation in frequency" (DF) versions digit-reversal is applied to the output of the combine phase. The DF forms are easily obtained from the DT forms by taking the transpose of each side of the given formula and making use of following properties:

$$(A \otimes B)^t = A^t \otimes B^t, \tag{15}$$

where A^t denotes the transpose of A and

$$P^t(mn, n) = P^{-1}(mn, n) = P(mn, m). \tag{16}$$

Cooley-Tukey DF (Gentleman-Sande):

$$F_{r^k} = R(r^k)\{\prod_{i=1}^{k}(I_{r^{i-1}} \otimes T^{r^{k-i+1}})(I_{r^{i-1}} \otimes F_r \otimes I_{r^{k-i}})\}. \tag{17}$$

Pease DF:

$$F_{r^k} = R(r^k)\{\prod_{i=1}^{k} P(r^k, r^{k-1})(T^{r^{k-i+1}} \otimes I_{r^{i-1}})P(r^k, r)(I_{r^{k-1}} \otimes F_r)P(r^k, r^{k-1})\}. \tag{18}$$

Korn-Lambiote DF:

$$F_{r^k} = R(r^k)\{\prod_{i=1}^{k} P(r^k, r^{k-1})(T^{r^{k-i+1}} \otimes I_{r^{i-1}})(F_r \otimes I_{r^{k-1}})\}. \tag{19}$$

The above tensor formulas completely functionalize the given algorithms. One only has to program the functions, $R(r^k)x$, $(I_s \otimes B)x$, $(B \otimes I_s)x$, $(T^{n/s} \otimes I_s)x$, $(I_s \otimes T^{n/s})x$, $F_r x$, $P(n, n/s)x$, and Dx, where D is a diagonal matrix, to obtain the entire family of algorithms. However, the resulting algorithms are not necessarily optimal. In the first place, replacing B by F_r in $B \otimes I_s$ or $I_s \otimes B$ does not take advantage of the symmetries in F_r. Also, the compiler might not fuse loops across functions optimally. In particular, for implementations on a Cray, special care must be taken in fusing a power of 2 stride permutation in order to avoid memory bank conflicts. Nevertheless, the tensor notation provides a powerful tool for deriving the necessary functions for programming optimal FFT's.

3. A Sisal program for the radix 4 DF Korn-Lambiote algorithm. We now turn to the problem of developing Sisal programs for FFT algorithms defined by their tensor representations. We shall present only key functions and we shall do this in terms of a psuedo code rather than detailed, less readable, Sisal code. In particular, we shall represent complex vectors by the data type **array**[complex]. "Complex" is not actually a data type in Sisal 1.8. In our programs we implement complex arrays by pairs of real arrays.

All of the FFT's of the previous section involve a separate ordering phase, i.e., application of digit-reversal either before or after the combine phase. The Stockham algorithm [2], which has received considerable attention by implementers of FFT's on Crays, avoids the ordering phase by distributing digit-reversal throughout the combine phase. Cann [1] has developed, in an imperative style, a Sisal implementation of the radix 4 Stockham algorithm which he claims outperforms, on a Cray Y-MP, the FFT of the signal processing library provided by Cray Research, Inc..

The virtue of the Stockham algorithm is that it avoids the cost of digit- reversal. However, in some applications of the FFT, such as convolution, it is desirable to separate the combine and ordering phases. In this section and the next we compare functional versions of the DF Korn-Lambiote and the DF Pease algorithms and their implementations on a Cray Y-MP. In order to reduce the number of serial steps and the number of flops, as well as the number of memory stores and fetches, which is important for a Cray, we consider the case $r = 4$.

Let us first examine the DF version of the Korn-Lambiote algorithm which is traditionally regarded as a vector algorithm. For $r = 4$, (19) becomes

$$F_{4^k} = R(4^k)\{\prod_{i=1}^{k} P(4^k, 4^{k-1})(T^{4^{k-i+1}} \otimes I_{4^{i-1}})(F_4 \otimes I_{4^{k-1}})\} \tag{20}$$

The implementation of this algorithm clearly requires a sequential loop of length k followed by an application of digit-reversal. Our goal is to simplify the structure of each serial step by combining each application of $P(4^k, 4^{k-1})$ with the twiddle factor $T^{4^{k-i+1}} \otimes I_{4^{i-1}}$ and the butterfly operation $F_4 \otimes I_{4^{k-1}}$ in such a way as to eliminate power of 2 strides. To this end, let V_i be the diagonal of $T^{4^{k-i+1}} \otimes I_{4^{i-1}}$. Then for any vector y of length n and any i,

$$P(4^k, 4^{k-1})(T^{4^{k-i+1}} \otimes I_{4^{i-1}})(F_4 \otimes I_{4^{k-1}})y = P(4^k, 4^{k-1})(V_i \circ (F_4 \otimes I_{4^{k-1}}))y \tag{21}$$

where $\circ$ denotes component-wise vector multiplication. Now we would like to fuse $P(4^k, 4^{k-1})$ with the loop for $(V_i \circ (F_4 \otimes I_{4^{k-1}}))y$. However, this necessitates a scatter operation, which is not readily implementable in Sisal 1.8. Consequently, we take advantage of the ability of Sisal's forall expression to return multiple values. We compute, not $(V_i \circ (F_4 \otimes I_{4^{k-1}}))$ as a single vector, but rather the four block components $Y_0, E_{i,1} \circ Y_1, E_{i,2} \circ Y_2$, and $E_{i,3} \circ Y_3$, where $V_i = (I, E_{i,1}, E_{i,2}, E_{i,3})$ and $(F_4 \otimes I_{4^{k-1}})y = (Y_0, Y_1, Y_2, Y_3)$. Now $P(4^k, 4^{k-1})$ can be applied by interleaving the four blocks $Y_0, E_{i,1} \circ Y_1, E_{i,2} \circ Y_2$, and $E_{i,3} \circ Y_3$). If we precompute all values of $E_{i,1}, E_{i,2}, E_{i,3}$, then each serial step of our Korn-Lambiote variant will consist of an invocation of the function B followed by an *interleave*. The function B is defined by the following pseudo Sisal code:

```
function B(k:integer;W1,W2,W3:array[complex];x:array[complex]
          returns array[complex],array[complex],array[complex],array[complex])
let
 ek1 := exp(4, k - 1);
 vector1,
 vector2,
 vector3,
 vector4 := for j in 0,ek1-1
             comp1,comp2,comp3,comp4 :=
               f4(x[j], x[j + ek1, x[j + 2 * ek1], x[j + 3 * ek1], W1[j], W2[j], W3[j])
            returns array of comp1
                    array of comp2
                    array of comp3
                    array of comp4
            end for
in
 array_setl(vector1,0),
 array_setl(vector2,0),
 array_setl(vector3,0),
 array_setl(vector4,0)
end let
end function
```

Here $f4$ is an optimized function which computes the component-wise product of a vector of the form $(1, w_1, w_2, w_3)$ by the Fourier transform of a vector x of length 4. The function *interleave* can be implemented by a loop of length 4^{k-1} and stride 1 as follows:

```
function interleave(length:integer;x1,x2,x3,x4:array[complex] returns array[complex])
 array_setl(for i in 0,length-1
             returns value of catenate array[0:x1[i],x2[i],x3[i],x4[i]]
            end for,0)
end function
```

Now fft can be implemented as follows

```
function fft(k:integer;index:array[integer];E1,E2,E3:array[array[complex]];x:array[complex]
          returns array[complex])
```

```
let
 ek1 := exp(4, k − 1);
 n := 4 * ek1;
 z :=
          for initial
             i := 1;
             y := x
             while i <= k repeat
               y := interleave(ek1, B(k, E1[oldi], E2[oldi], E3[oldi], oldy));
               i := oldi + 1 ;
             returns value of y
          end for
in
 permute(z, n, index)
end let
end function
```

3. A Sisal program for the radix 4 DF Pease algorithm. Although the implementation of $P(4^k, 4^{k-1})$ in terms of *interleave* eliminates power of 2 strides, it does not come without cost, namely, it must be invoked at each serial step. In the case of the Pease algorithm, the cost of performing the first two stride permutations in

$$F_{4^k} = R(4^k)\{\prod_{i=1}^{k} P(4^k, 4^{k-1})(T^{4^{k-i+1}} \otimes I_{4^{i-1}})P(4^k, 4)(I_{4^{k-1}} \otimes F_4)P(4^k, 4^{k-1})\} \tag{22}$$

can be transferred to the precomputation of the twiddle factors. To see this, define V_i as before and observe that for any vector y

$$\begin{aligned} &P(4^k, 4^{k-1})(T^{4^{k-i+1}} \otimes I_{4^{i-1}})P(4^k, 4)(I_{4^{k-1}} \otimes F_4)P(4^k, 4^{k-1})y \\ &= P(4^k, 4^{k-1})V_i \circ P(4^k, 4^{k-1})P(4^k, 4)(I_{4^{k-1}} \otimes F_r)P(4^k, 4^{k-1})y \\ &= P(4^k, 4^{k-1})V_i \circ (I_{4^{k-1}} \otimes F_4)P(4^k, 4^{k-1})y \end{aligned} \tag{23}$$

In this case, we define $P(4^k, 4^{k-1})V_i = (I, E_{i,1}, E_{i,2}, E_{i,3})$ and we precompute all values of $E_{i,1}, E_{i,2}, E_{i,3}$. Now each serial step can be performed by a simple invocation of the following function

```
function B(k:integer;W1,W2,W3:array[complex];x:array[complex] returns array[complex])
let
 ek1 := exp(4, k − 1);
in
array_setl(
 for j in 0,ek1-1
  quad := f4(x[j], x[j + ek1], x[j + 2 * ek1], x[j + 3 * ek1], W1[j], W2[j], W3[j])
 returns value of catenate quad
 end for,0)
end let
end function
```

The *fft* function in this case is the same as the one given for the Korn- Lambiote variant except that *interleave* is omitted. The savings obtained by the elimination of the interleave operation is considerable. For example, execution time on the Cray Y-MP for $n = 4^9$ using all 8 processors is only 55% that of the Korn-Lambiote variant.

Each of the above algorithms is valid only for vector lengths that are powers of 4, i.e., even powers of 2. When $n = 2^k$ where k is odd, $F_n(x)$ can be computed making use of the following formula which is obtained from (11) by transposing each side and taking $n = rs$ where $r = 2$ and $s = 2^{k-1}$:

$$F_n = P(2^k, 2^{k-1})(I_2 \otimes F_{2^{k-1}})T^{2^k}(F_2 \otimes I_{2^{k-1}}) \tag{24}$$

The Sisal code for implementing (24) is straightforward. We compute $F_n(x)$ by computing $T^{2^k}(F_2 \otimes I_{2^{k-1}})x$ in two halves in a way similar to the computation of the B function for the Korn-Lambiote variant. Then an "even" FFT is applied to the two halves and the two results are interleaved.

Tables 1 and 2 compare running times on the NERSC CRAY Y-MP for our Sisal implementation of the Pease variant (PFFT) with those of Cann's (CFFT) after having changed the single precision real arithmetic of CFFT to double precision as used in PFFT. PFFT was compiled with the option *-maxconcur* while CFFT was compiled with the options recommended by Cann, i.e., *-hybrid -concur -vector.*

Table 1. Execution times in seconds for PFFT, $n = 2^k$.

k	1 CPU	4 CPU's	8 CPU's
16	0.042011	0.012496	0.006037
17	0.087329	0.024462	0.013726
18	0.175957	0.046390	0.024890
19	0.377403	0.099038	0.058168
20	0.825157	0.201309	0.107798

Table 2. Execution times in seconds for CFFT, $n = 2^k$.

k	1 CPU	4 CPU's	8 CPU's
16	0.034853	0.014313	0.013358
17	0.061205	0.023366	0.018410
18	0.123053	0.046848	0.038326
19	0.259286	0.094445	0.071659
20	0.523968	0.204599	0.159223

The above times do not include initialization, which for CFFT includes the computation of a table of sines and cosines and which for PFFT includes the computation of a table of roots of unity as well as the computation of a table of indices used for digit-reversal. Times for PFFT do, however, include the digit-reversal permutation, which constitute, no matter the number of processors, about 11% of the above running times.

The next two tables campare MFLOP rates for PFFT and CFFT.

Table 3. MFLOP rates for PFFT, $n = 2^k$.

k	1 CPU	4 CPU's	8 CPU's
16	106	356	738
17	109	391	697
18	114	432	805
19	114	431	735
20	108	443	961

Table 4. MFLOP rates for CFFT, $n = 2^k$.

k	1 CPU	4 CPU's	8 CPU's
16	112	272	292
17	127	334	424
18	145	381	465
19	137	377	498
20	153	392	504

In general, the performance of PFFT in comparison to CFFT will be best for $n = 2^k$ when k is even, as can be expected from (24).

The simple loop structure of PFFT is obtained in large part by trading code for the memory space needed to store, in a redundant way, various roots of unity. For example, for $n = 2^k$, k even, the precomputed tables in each of the programs requires the computation and storage of $\frac{3}{4}kn$ complex numbers. Computation of these tables is from three to ten times slower than computation of the table in CFFT. On the other hand, using a highly efficient algorithm [7], the computation of the table of digit-reversal indices is very fast, e.g., .001 seconds for $n = 4^9$ and eight processors. But in any case, the time for initialization is insignificant if the program is used to compute a large number of FFT's of the same size.

The same ideas used in PFFT could be applied to obtain fast programs for F_n where $n = r^k$ and r is any positive integer. This would require some slight changes in the precomputation of the twiddle factors as well as an optimal function for the computation of F_r for a given r. PFFT could also be modified, at the cost of additional loop structure, to reduce the amount of memory needed to store the precomputed powers of ω.

Conclusions. We have presented a mathematical framework for functionalizing and optimizing FFT algorithms. Starting with an FFT algorithm expressed in tensor notation, one can use the laws of tensor algebra to obtain tensor formulas that take into account a given architecture and are easily translated into Sisal. Work is in progress to develop Sisal implementations for other FFT algorithms using this same methodology. An interesting problem is to investigate the possibility of developing a high performance "universal" FFT Sisal program, in the sense that given an FFT algorithm A in terms of a tensor formula, the program optimally computes $F_n(x)$ according to A.

Acknowledgement

The authors are grateful to Lawrence Livermore National Laboratory for the Cray computer time made available through the Sisal Scientific Computing Initiative (SSCI).

References

[1] D. Cann, "Retire Fortran? A Debate Rekindled," Comm. ACM, vol. 30(8) (1992), 81-89.

[2] W. T. Cochran, et al. "What is the Fast Fourier Transform?," IEEE Trans. Audio Electroacoust., AU-15(2) (1967), 45-55.

[3] J.W. Cooley and J.W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," Math. Comp., vol. 19 (1965), 297-301.

[4] J. Johnson, R. Johnson, D. Rodriguez, and R. Tolimieri, "A Methodology for Designing, Modifying and Implementing Fourier Transform Algorithms on Various Architectures," Circuits, Systems and Signal Processing vol. 9 (1990), 449-500.

[5] D.G. Korn and J.J. Lambiotte, "Computing the Fast Fourier Transform on a Vector Computer," Math. Comp., vol. 33 (1979), 977-992.

[6] M.C. Pease, "An Adaptation of the Fast Fourier Transform for Parallel Processing," J. ACM, vol. 15 (1968), 252-265.

[7] J. Seguel, D. Bollman, and V. Celis, "A family of parallel algorithms for digit-reversal," submitted.

[8] R. Tolimieri, M. An, and C. Lu, Algorithms for Discrete Fourier Transform and Convolution, Springer-Verlag, New York, 1989.

[9] C. Van Loan, Computational Frameworks for the Fast Fourier Transform, S.I.A.M., 1992.