FFT IMPLEMENTATION STRATEGIES

David Sepiashvili, Markus Püschel,
Gavin Haentjens, José Moura

Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA

Jeremy Johnson

Mathematics and Computer Science
Drexel University
Philadelphia, PA

ABSTRACT

In this paper we systematically investigate the performance of a family of FFT algorithms and implementation strategies. We consider a large class of recursive algorithms based on the Cooley-Tukey decomposition and a series of implementation strategies for memory allocation and data access. This investigation provides justification for some of the design decisions made by FFTW, a highly optimized portable package for computing the FFT. This study analyzes implementation techniques incorporated into the SPIRAL system (see http://www.ece.cmu.edu/~spiral). SPIRAL is an ongoing project whose goal is to automate the efficient implementation and platform adaptation of signal processing algorithms.

1. INTRODUCTION

Beginning in 1965 with the paper by Cooley and Tukey [1], there has been continuous effort to obtain fast implementations of the FFT. Over the years many variants of the original algorithm have been proposed and many techniques have been developed to obtain efficient implementations of these algorithms. In the late eighties there was an effort to present the design alternatives in a uniform manner using the framework of matrix factorizations of the discrete Fourier transform (DFT) matrix (see [2, 3] for a summary of these efforts). In [4] the different factorizations were described by mathematical formulas that served as the guide for obtaining alternative implementations of the FFT. Moreover, it was suggested that these formulas could be directly translated into programs with different performance characteristics and consequently a search could be performed to find the optimal implementation of the FFT. This search program was outlined in [5] and has been carried out on a subset of formulas [6, 7, 8]. A compelling case for the benefits of search to find fast implementations of the FFT has been made by the success of FFTW [7], where a limited search, using dynamic programming, was used to optimize the code on alternative computing platforms.

In this paper, we investigate the performance of a class of FFT algorithms and implementation strategies. We consider only recursive algorithms based on the Cooley-Tukey factorization; however, even with this single factorization rule, there are many possible algorithms arising from different breakdown strategies, i.e., application of the Cooley-Tukey rule. A given breakdown strategy, in turn, can be naturally implemented in several different ways. These alternative implementation strategies need different amounts of temporary storage and a different number of passes through the data. These differences have a substantial impact on runtime performance.

Each possible algorithm that is considered can be represented by an ordered binary tree. It has been shown previously that different trees can lead to different performance (a different breakdown strategy will access the data in a different pattern). In fact, a subset of the trees correspond to the algorithmic choices considered in FFTW. Interestingly when we searched for the trees, and implementation strategy, with optimal performance we were led to design choices made in FFTW.

2. MATHEMATICAL FRAMEWORK

The discrete Fourier transform (DFT) of size n is represented by the $n \times n$ matrix

$$\mathrm{F}_n = [\omega_n^{k\ell} \mid k, \ell = 0 \ldots n-1],$$

where $\omega_n = e^{2j\pi/n}$. The Cooley-Tukey FFT (fast Fourier transform, [1, 2]) is written as a factorization of F_n into a product of structured sparse matrices. Let $n = rs$, then, by Cooley-Tukey,

$$\mathrm{F}_n = (\mathrm{F}_r \otimes \mathrm{I}_s) \cdot \mathrm{T}_s^n \cdot (\mathrm{I}_r \otimes \mathrm{F}_s) \cdot \mathrm{L}_r^n, \qquad (1)$$

where T is the twiddle matrix, L is the stride permutation matrix, and $\otimes$ denotes the tensor (or Kronecker) product of

This work was supported by DARPA through research grant DABT63-98-1-0004 administered by the Army Directorate of Contracting.

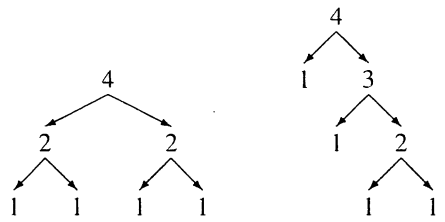

Fig. 1. (a) a balanced, and (b) a right-most break-down tree

matrices, [4]. The twiddle matrix is diagonal,

$$\mathrm{T}_r^n = \bigoplus_{i=0}^{r-1} \mathrm{diag}\left(w_n^0, w_n^1, \cdots, w_n^{s-1}\right)^i,$$

and the stride permutation L_r^n is the permutation that maps $ir + j \longrightarrow js + i$ for $0 \leq i < s$ and $0 \leq j < r$. As an example, consider the case $n = 4 = 2 \cdot 2$:

$$\mathrm{F}_4 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & j \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Formula (1) represents a recursive algorithm and different factorizations $n = rs$ give rise to different recursions. However, even if r, and hence s, are fixed, there are different ways to interpret (1), yielding different algorithms with different characteristics and performance. We will elaborate on that in Section 3.

In this paper we will restrict ourselves to the most important case $n = 2^k$. With k given, the particular instance of (1) corresponds to a partition $k = k_1 + k_2$, $k_1, k_2 > 0$. We represent this choice by the labeled tree with k at the root, k_1 at the left child, and k_2 at the right child. Recursive application of (1) to the children gives rise to ordered (right and left children are distinguishable) binary partition trees. Different trees define different algorithms. It can be shown that there are asymptotically $O(5^k/k^{3/2})$ many trees with root k. Ideally, an implementation of the FFT should support all possible trees and allows searching for the optimal one on a given computing platform. This search enables adaptation. All implementations considered in this paper are of this kind. Figure 1 shows two possible trees for $k = 4$. In Section 3, we will establish a relationship between the topological structure of a break-down tree and the properties of the corresponding algorithm and implementation.

3. ALGORITHM CHOICES

We present three algorithms corresponding to the Cooley-Tukey FFT. All approaches use the same breakdown strategy, but differ in how temporary storage is allocated and accessed.

To obtain an algorithm for computing the DFT from the Cooley-Tukey factorization, we need to interpret the symbols and mathematical operations in Equation (1) as programming constructs, [4]. The matrix-vector product $y = \mathrm{F}_n\, x$ is obtained using the Cooley-Tukey factorization by computing, $u = \mathrm{L}_r^n\, x$, followed by $v = (\mathrm{I}_r \otimes \mathrm{F}_s)u$, $w = T_s^n x$, and finally $y = (\mathrm{F}_r \otimes \mathrm{I}_s)w$. The factors involving smaller DFTs are computed by looping over the input vector and recursively calling a function to compute the smaller DFTs. The computation $(\mathrm{F}_r \otimes \mathrm{I}_s) = \mathrm{L}_r^n(\mathrm{I}_s \otimes \mathrm{F}_r)\,\mathrm{L}_s^n$ requires that the input and output vectors be accessed at stride s, or that two extra permutations be inserted. Many optimizations to this basic approach are possible: the permutation L_r^n can be combined with the following factor as re-addressing rather than performing a runtime permutation; the twiddle factor can be combined with either the previous or following factor; and the last three factors can easily be computed in-place thereby avoiding extra temporary storage. In-place computation of the stride permutation L_r^n is not straight-forward (see [9]), and requires additional computations which may negate the benefits of reduced temporary storage. In our experiments, we considered three possible approaches, each with varying amounts of temporary storage.

The first approach is in-place, but rather than performing an in-place stride permutation, as is done in [6], we combine all of the stride permutations, and perform the combined bit-reversal permutation outside of the recursion.

Algorithm 1 (FFT-BR) *In-place bit-reversed output algorithm.*

1. $(\mathrm{L}_r^n \cdot x) = (\mathrm{I}_r \otimes \tilde{\mathrm{F}}_s) \cdot (\mathrm{L}_r^n \cdot x)$ *[Compute $\tilde{\mathrm{F}}_s$, r times at stride r]*
2. $x = \tilde{\mathrm{T}}_r^n \cdot x$
3. $x = (\mathrm{I}_s \otimes \mathrm{F}_r) \cdot x$
4. $(\mathrm{B}_n \cdot x) = x$ *[done outside of the recursion]*

The second approach allocates temporary storage in each recursive call, but rather than simply using the extra temporary space for computing the initial stride permutation, it combines the stride permutation as re-addressing in the following factor. The amount of temporary storage required is $S(n) \leq n + S(n/2) \leq 2n$, and it is maximized for left-most breakdown trees.

Algorithm 2 (FFT-TS) *Out-of-place algorithm with extra temporary storage requirements.*

1. $t = (\mathrm{I}_r \otimes \mathrm{F}_s) \cdot (\mathrm{L}^n_r \cdot x)$ *[Compute* F_s*, r times with input accessed at stride r and output at stride* 1.*]*
2. $t = \mathrm{T}^n_s \cdot t$
3. $(\mathrm{L}^n_s \cdot y) = (\mathrm{I}_s \otimes \mathrm{F}_r) \cdot (\mathrm{L}^n_s \cdot t)$

The third approach, which is the one used by FFTW, is identical to the second except that the extra temporary storage is not required. The reduction in temporary storage is obtained by restricting the set of FFTs that it supports to those corresponding to right-most breakdown trees. The crucial property of a right-most breakdown tree is that the second recursive call is removed and hence the computation can proceed in-place. Algorithm 2 is written so that only as much temporary space as is required is used. Thus, Algorithms 2 and 3 are identical for rightmost trees.

Let B_n denote the bit-reversal permutation and let $\tilde{\mathrm{F}}_n = \mathrm{B}_n \mathrm{F}_n$, then it can be shown that

$$\tilde{\mathrm{F}}_n = (\tilde{\mathrm{F}}_r \otimes I_s)\, \tilde{\mathrm{T}}^n_r\, L^n_s (I_r \otimes \tilde{\mathrm{F}}_s) L^n_r,$$

where $\tilde{\mathrm{T}}^n_r$ is a permuted twiddle factor. This can be computed in-place, and the DFT can be obtained by applying B_n to $\tilde{\mathrm{F}}_n\, x$. This modification is used in the first algorithm.

Algorithm 3 (FFT-RT) *Out-of-place algorithm for rightmost trees only.*

1. $y = (\mathrm{I}_r \otimes \mathrm{F}_s) \cdot (\mathrm{L}^n_r \cdot x)$ *[Compute* F_s*, r times with input accessed at stride r and output at stride* 1.*]*
2. $y = \mathrm{T}^n_s \cdot y$
3. $(\mathrm{L}^n_s \cdot y) = (\mathrm{I}_s \otimes \mathrm{F}_r) \cdot (\mathrm{L}^n_s \cdot y)$ *[Compute* F_r*, s times at stride s.* F_r *must be computed in-place.]*

4. IMPLEMENTATION

To make an authoritative comparison of different algorithms it is mandatory to use careful tuning strategies and to produce excellent, high quality code so that unexpected overheads of suboptimal code do not mask the timing results. For this reason we have developed highly optimized assembly code implementations of Algorithms 1-3. In addition, the implementations share the following features.

1. For small sizes $n = 2^k$, highly optimized code modules are provided. For these sizes, the DFT can be computed either by recursion or using the code module. In our tree notation, a leaf means using a code module allowing for leaves larger than 1.
2. The twiddle factors T are precomputed and stored.
3. The interface to the recursive programs supports strided access of the inputs and outputs.
4. All trees within the scope of the algorithm are supported, and a search module, using dynamic programming, is provided to find the trees that lead to highest performance. See Section 5.

For details on the code optimizations performed we refer the reader to [10].

5. SEARCH FOR FAST TREES

As mentioned, the number of trees for size $n = 2^k$ is asymptotically $O(5^k/k^{3/2})$, which makes an exhaustive search infeasible. We use a dynamic programming (DP) approach to reduce the search space. Assume we want to find a fast tree for size k, and fast trees for sizes $1 \ldots k-1$ are already computed. The number k is split in all possible ways $k = i + (k-i)$, $i = 1 \ldots k-1$ and the corresponding tree is built by using the previously found trees as subtrees for i and $k-i$. Among the $k-1$ trees obtained this way, the best one is chosen for size k. Using DP, only $O(k^2)$ trees have to be timed to find a fast tree for size k. DP is not guaranteed to find the optimal tree due to that fact that the cost of a tree may depend on the context in which it is called. Nonetheless, DP has been shown [10] to produce very good results.

6. EXPERIMENTAL RESULTS

In this section we show the experimental results with our implementations of Algorithms 1-3. The testing platform was a Pentium II MMX, 450 MHz, with 384 MB of SDRAM, running WinNT 4.0 Workstation. The compiler was Microsoft Visual C++ 6.0. All our implementations are in assembly and have the features briefly described in Section 4 The trees for the respective sizes have been found with the dynamic programming (DP) search as described in Section 5. In our first implementations we provided code modules only for sizes $k = 1, 2$. Figure 2 shows the runtimes of the best trees found for $k = 1 \ldots 20$. The runtime is given in cycles, divided by $k \cdot 2^k$ (higher is slower). We observe that FFT-BR is slower, and that FFT-RT and FFT-TS coincide. The reason for this is, that DP found only right-most trees when applied to FFT-TS, in which case both implementations are equal. We performed additional optimization on FFT-RT and FFT-TS, in particular we provided code modules up to size $k = 4$. Again, FFT-RT and FFT-TS found the same trees. The result is the lowest curve in Figure 2. We conclude that the extra cost incurred by extra temporary storage outweighs the potential gain from considering arbitrary trees. This coincides with the design of FFTW. Note, however, that if a full temporary vector is always used, then balanced trees can outperform rightmost trees [11]. The actual trees found are given in Table 1. For a

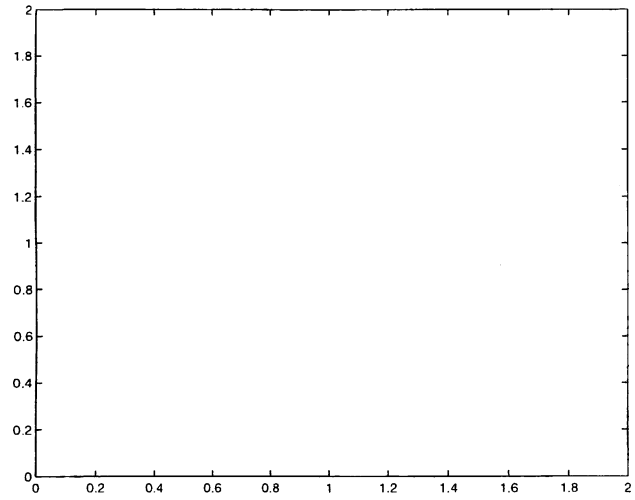

Fig. 2. Runtime comparison of our FFT implementations

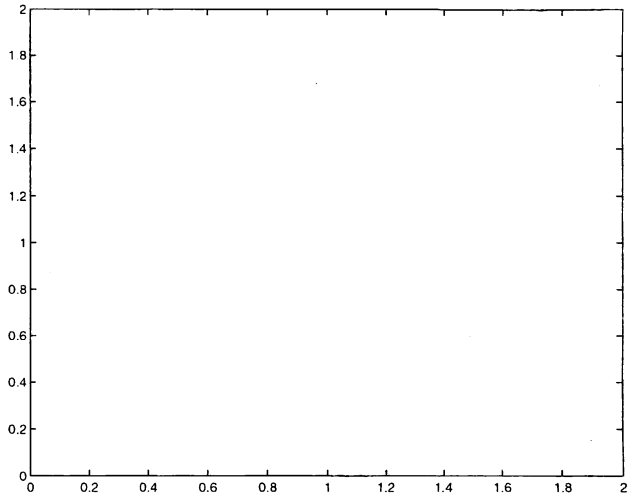

Fig. 3. FFT-RT (optimized) versus FFTW

k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
	0	0	0	0	2	2	4	4	3	4	2	2	2	2	2	2	2	2	2	2

Table 1. The best trees for FFT-RT found with DP

given k, a value of 0 means the code module is used, a number i signifies the left leaf, and the right subtree for $k-i$ has to be looked up recursively.

To giver further validity to our conclusions, we compared our best implementation against FFTW, [7] (see Figure 3), one of the fastest FFT implementations available. The runtimes are normalized w.r.t. FFTW, i.e., FFTW corresponds to the straight line. The FFTW trees where found using the option FFTW_MEASURE, which triggers a DP search similar to ours. We observe that for sizes up to 2^{12}, our implementation is faster, most likely due to the additional optimizations provided by the assembly coding. For larger sizes, FFTW outperforms our code. We hypothesize that this is due to the larger code modules used for small transforms (up to size 64) used by FFTW.

7. REFERENCES

[1] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Math. of Computation*, vol. 19, pp. 297–301, 1965.

[2] R. Tolimieri, M. An, and C. Lu, *Algorithms for discrete Fourier transforms and convolution*, Springer, 2nd edition, 1997.

[3] C. Van Loan, *Computational Framework of the Fast Fourier Transform*, Siam, 1992.

[4] J. Johnson, R. W. 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, no. 4, pp. 449–500, 1990.

[5] L. Auslander, J. Johnson, and R. W. Johnson, "Automatic implementation of FFT algorithms," Tech. Rep. DU-MCS-96-01, Dept. of Math. and Computer Science, Drexel University, Philadelphia, PA, June 1996, Presented at the DARPA ACMP PI meeting.

[6] S. Egner, *Zur Algorithmischen Zerlegungstheorie Linearer Transformationen mit Symmetrie*, Ph.D. thesis, Univ. Karlsruhe, Informatik, 1997.

[7] Matteo Frigo and Steven G. Johnson, "FFTW: An adaptive software architecture for the FFT," in *ICASSP 98*, 1998, vol. 3, pp. 1381–1384, `http://www.fftw.org`.

[8] J. Johnson and M. Püschel, "In Search for the Optimal Walsh-Hadamard Transform," in *Proceedings ICASSP*, 2000, vol. IV, pp. 3347–3350.

[9] Norman Brenner, "Algorithm 467: Matrix transposition in place," in *Collected Algorithms of the ACM*, number 467. 1972.

[10] D. Sepiashvili, " Performance Models and Search Methods for Optimal FFT Implementations ," M.S. thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, 2000.

[11] G. Haentjens, "An Investigation of Recursive FFT Implementations," M.S. thesis, Dept. of Electrical and Computer Engineering, Carnegie Mellon University, 2000.