A Recursive Implementation of the Dimensionless FFT

Xu Xu and Jeremy R. Johnson

Technical Report DU-CS-05-01 Department of Computer Science Drexel University Philadelphia, PA 19104 December 2004

A Recursive Implementation of the Dimensionless FFT

A Thesis

Submitted to the Faculty

of

Drexel University

by

Xu Xu

In partitial fulfillment of the

requirements for the degree of

Master of Science in Computer Science

December 2003

© Copyright 2004 Xu Xu. All Rights Reserved.

Table of Contents

List of 7	lables .		iv
List of H	Figures		v
Abstract	t		viii
Chapter	1: Intr	oduction	1
Chapter	2: The	e Dimensionless Fast Fourier Transform (FFT)	4
2.1	The M	ulti-Dimensional Discrete Fourier Transform (DFT) \ldots .	4
	2.1.1	Matrix/Vector Notation	4
	2.1.2	Matrix Operations	5
	2.1.3	The Discrete Fourier Transform	6
	2.1.4	The Multi-Dimensional DFT	7
2.2	The Fa	ast Fourier Transform (FFT)	9
	2.2.1	The Cooley-Tukey Factorization	9
	2.2.2	A Divide and Conquer Algorithm for the FFT	11
	2.2.3	An FFT Implementation	14
2.3	The Di	imensionless FFT	16
	2.3.1	The Dimensionless Cooley-Tukey Factorization	16
	2.3.2	A Divide and Conquer Algorithm for the Dimensionless Cooley-	
		Tukey Factorization	18
Chapter	3: FF1	Γ₩	24
3.1	FFTW	V Overview	24
3.2	FFTW	Plans and the Recursive Planner	27

	3.2.1	FFTW Plan Data Structure	27
	3.2.2	Recursive Planner	33
3.3	3.3 Computing the One-dimensional DFT		
	3.3.1	Executor	35
	3.3.2	Codelets	36
3.4	Multi-	dimensional Fourier Transform in FFTW	38
	3.4.1	Multi-dimensional Plans	38
	3.4.2	Creating a Plan for Multi-dimensional DFTs	38
	3.4.3	Computing the Multi-dimensional DFT	41
Chapter	r 4: Ext	tending FFTW to Compute the Dimensionless FFT	42
4.1	The R	ecursive Evaluation of the Dimensionless FFT	43
	4.1.1	Generalized Stride Permutations	43
	4.1.2	Sample Computation of the Recursive Dimensionless FFT $$	49
	4.1.3	Dimensionless Executor	54
4.2	Modif	ications to FFTW to Support the Dimensionless FFT	57
	4.2.1	Modified Plan	57
	4.2.2	Modified Recursive Planner	58
	4.2.3	Modified Recursive Executor	62
Chapter	r 5: Co	delets for the Dimensionless FFT	63
5.1	Overv	iew of the SPIRAL System	64
	5.1.1	Formula Generator	64
	5.1.2	Formula Translator	67
	5.1.3	Search Engine	70
5.2	5.2 Building a Multi-dimensional Codelet Library		
	5.2.1	Generating Multi-Dimensional DFTs	71

	5.2.2	Finding Fast Implementations of Small Multi-dimensional		
		DFTs 77	3	
	5.2.3	Building the Codelet Library	3	
Chapte	r 6: Pei	formance	0	
6.1	Perfor	mance of the Row-Column Algorithm	0	
6.2	Perfor	mance of the Dimensionless FFT	0	
6.3	Perfor	mance Evaluation	2	
	6.3.1	Comparison of Codelets	3	
	6.3.2	A Performance Model	7	
Chapter 7: Conclusion			9	
Bibliography				

List of Tables

3.1	Number of factorization and FFTW plan	28
4.1	The precision between the simulator (Figure 4.1) and FFTW $\ . \ . \ .$.	55
5.1	Runtime for multi-dimensional DFTs of size 64, unit: nanosecond \ldots	75
6.1	Multi-dimensional DFTs that were tested in FFTW	81
6.2	The Runtime of FFTW's no-twiddle codelet in seconds	84
6.3	The Runtime of dimensionless no-twiddle codelet in seconds $\ . \ . \ . \ .$	84
6.4	The Runtime of FFTW's twiddle codelet in seconds	85
6.5	The Runtime of dimensionless twiddle codelet in seconds	85

List of Figures

2.1	The possible Cooley-Tukey factorizations of $N = 16$	13
2.2	A divide and conquer algorithm for FFT	15
2.3	A divide and conquer algorithm for dimensionless FFT \ldots	20
2.4	Function index, find the dimension to split	21
2.5	Permute vector with $I_a \otimes L_m^n \otimes I_b$	22
2.6	Multiply twiddle factor $I_a \otimes T_m^n \otimes I_b$	23
3.1	A possible plan for $N = 128$	26
3.2	All possible factorizations of $N = 64$	27
3.3	FFTW plan data structure	28
3.4	fftw_plan_node data structure	30
3.5	fftw_codelet_desc data structure	31
3.6	fftw_twiddle data structure	31
3.7	wisdom data structure	32
3.8	Exporting wisdom	32
3.9	Importing wisdom	33
3.10	Computing One-dimensional FFT	35
3.11	The no-twiddle codelet of size 4	37
3.12	The twiddle codelet of size 4	39
3.13	Multi-dimensional FFTW plan	40
4.1	The Executor Simulator	56
4.2	Updated <i>fftw_plan</i> data structure	58

4.3	Updated <i>fftw_plan_node</i> data structure	59
4.4	Updated <i>fftw_codelet_desc</i> data structure	60
4.5	Updated <i>fftw_twiddle</i> data structure	60
5.1	The Process of SPIRAL	65
5.2	Rule trees of formulas in Equation 5.3 and Equation 5.4 \ldots	67
5.3	Components of SPL	68
5.4	SPL program for the algorithm in Equation 5.3 \ldots \ldots \ldots	69
5.5	The AST corresponding for the SPL program in Figure 5.4 \ldots .	69
5.6	Partitions of N=16	72
5.7	Function Int2Part and Part2MDFT	74
5.8	SPL file, $F_2 \otimes F_2 \otimes F_2 \otimes F_2$	75
5.9	Computing $F_2 \otimes F_2 \otimes F_2 \otimes F_2$ with stride permutation	76
5.10	Twiddle Codelets of $F_2 \otimes F_2 \otimes F_2 \otimes F_2$	78
5.11	No-twiddle Codelets of $F_2 \otimes F_2 \otimes F_2 \otimes F_2$	79
6.1	The runtime ratio of multi-dimensional DFT vs. one-dimensional DFT in FFTW	81
6.2	The runtime ratio of dimensionless FFT to FFTW	82
6.3	The runtime ratio of multi-dimensional DFT to one-dimensional DFT in the dimensionless FFT	83
6.4	The ratio of the runtime of the dimensionless no-twiddle codelets to FFTW's no-twiddle codelets	86
6.5	The ratio of the runtime of the dimensionless twiddle codelets to FFTW's twiddle codelets	86
6.6	A FFTW plan of input size 2^{20}	88

vii

Abstract A Recursive Implementation of the Dimensionless FFT Xu Xu Jeremy R. Johnson

The discrete Fourier transform (DFT) is an important tool in many branches of science and engineering, and has been studied extensively. For many applications, it is important to have an implementation of the DFT that is as fast as possible.

Recently high performance packages for computing the fast Fourier transform (FFT) have been developed using automatic code generation and an adaptable framework for optimizing the implementation to different computing platforms. FFTW is a well known package that follows this approach and is currently one of the fastest implementations of the FFT available.

FFTW utilizes a recursive formulation of the FFT that employs different breakdown strategies and a collection of highly tuned base cases called codelets. In this work, we extend FFTW to compute arbitrary multidimensional DFTs using a modification of the one-dimensional code provided by FFTW. The modification is based on the concept of a dimensionless FFT which allows a multi-dimensional DFT to be obtained from a one-dimensional FFT simply be reordering the inputs and modifying the multiplicative constants (twiddle factors) used in the program.

Chapter 1: Introduction

The divide and conquer construction used by the fast Fourier transform (FFT) allows a discrete Fourier transform (DFT) of size mn to be computed using n transforms of size m followed by m transforms of size n [1]. This construction requires that the input data be accessed at stride and the intermediate data obtained after computing the n transforms of size m be scaled by the so called "twiddle factors". Multidimensional DFTs are normally computed using one-dimensional FFTs along each of the dimensions. For example, let X(a, b) with $0 \le a < m$ and $0 \le b < n$ be a function of two variable stored in an $m \times n$ array. The two-dimensional $m \times n$ DFT of X can be calculated by applying m one-dimensional n-point DFTs to the rows of X followed by n one-dimensional m-point DFTs to the columns. The one-dimensional DFTs are computed using the FFT. This approach, called the row-column algorithm, can be generalized to DFTs with arbitrarily many dimensions; however, it has the shortcoming that the divide and conquer construction used by the FFT can only be applied separately to the number of points in each dimension. It does not allow the use of smaller transforms of size equal to an arbitrary factor of the number of data points. The dimensionless FFT [2] allows a multi-dimensional DFT of total size N = RSto be computed using R multi-dimensional DFTs of size S followed by S multidimensional DFTs of size R independent of dimension. This is identical to the onedimensional construction except that a slightly different input permutation is required and the values of the twiddle factors are different. The permutation and twiddle factors depend on the dimension. The original presentation of the dimensionless FFT was based on an iterative algorithm for computing the FFT and was motivated by the desire to produce FFT hardware that could be used for one, two, and three dimensional transforms [2, 7].

Recently there has been efforts to automatically optimize the performance of im-

portant signal processing routines such as the FFT [6, 8, 11]. These approaches search for a good decomposition (breakdown strategy) of the FFT in an effort to best utilize the number of registers, cache, and other features of the underlying hardware. A good breakdown can be far more important than saving a few arithmetic operations. The use of the dimensionless FFT allows decomposition sizes that are not available in the row-column algorithm and can provide improved performance for many multi-dimensional DFTs.

This thesis shows that FFTW [6], one of the fastest public domain FFT packages, can be modified to support a recursive implementation of the dimensionless FFT. FFTW can use many different recursive divide and conquer strategies, and dynamic programming is used to empirically determine the "best" strategy. The desired strategy is stored in a tree data structure called a plan. The plan also stores the necessary twiddle factors. An executor uses the plan to compute the FFT from a collection of small FFTs, called codelets, implemented using straight-line code. To support the dimensionless FFT, extra information must be stored in the plan to keep track of the dimension of the various DFTs that arise, and a set of multi-dimensional FFT codelets must be provided. The plan generator must be extended to produce the twiddle factors required by the dimensionless FFT, and the executor must support one additional parameter needed for the generalized permutations that can arise. These changes were incorporated into FFTW and empirical data is presented showing the potential for improved performance as compared to the row-column algorithm currently provided in FFTW.

The remainder of this thesis discusses these ideas in detail. Chapter 2 derives a recursive formulation of the dimensionless FFT, Chapter 3 introduces the structure of FFTW. The modifications to FFTW to implement the dimensionless FFT are described in Chapter 4. Chapter 5 introduces the SPIRAL system and shows how to use it to build the dimensionless FFT codelet library. The performance data is

provided in Chapter 6 and the conclusion is given in Chapter 7.

Chapter 2: The Dimensionless Fast Fourier Transform (FFT)

This chapter reviews the DFT and the FFT and presents a generalization of the FFT, called the dimensionless FFT, which applies to multi-dimensional DFTs. The FFT is presented using matrix notation because the dimensionless FFT is most easily stated and defined using this formulation (compare Theorem 1 and 2).

2.1 The Multi-Dimensional Discrete Fourier Transform (DFT)

This section defines the multi-dimensional DFT as a matrix-vector product. Matrix notation and operators needed later are reviewed and defined.

2.1.1 Matrix/Vector Notation

The (k, j) entry of a matrix A is denoted by either a_{kj} or A(k, j). It is convenient to index vectors and matrices starting with zero. For example,

$$A = \left(\begin{array}{ccc} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{array}\right).$$

The transpose of a matrix A is denoted by A^T . The colon notation is used to access elements with a given stride.

Definition 1 (Colon notation) Let x be a column vector with n elements, and $u \le v \le n$, then

$$x(u:k:v) = (x_u, x_{u+k}, \cdots, x_{u+pk})^T$$
(2.1)

where $u + pk \le v < u + (p+1)k$.

The diagonal matrix whose entries are a_1, a_2, \dots, a_n is denoted as $diag(a_1, a_2, \dots, a_n)$. For example,

$$diag(1,2,3,4) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

The inverse of a nonsingular matrix A is denoted by A^{-1} , and the $n \times n$ identity matrix is denoted by I_n .

2.1.2 Matrix Operations

The direct sum and tensor product of matrices are defined for later use.

Definition 2 (The Direct Sum) Let A_0, \dots, A_{R-1} be matrices. The direct sum of A_0, \dots, A_{R-1} is the block diagonal matrix

$$\bigoplus_{i=0}^{R-1} A_i = diag(A_0, A_1, \cdots, A_{R-1}).$$

Definition 3 (Tensor product) Let A be an $m \times n$ matrix and B be a $p \times q$ matrix. The tensor product $A \otimes B$ is the $mp \times nq$ block matrix

$$A \otimes B = \begin{pmatrix} a_{0,0}B & \cdots & a_{0,q-1}B \\ \vdots & \ddots & \vdots \\ a_{p-1,0}B & \cdots & a_{p-1,q-1}B \end{pmatrix}$$
(2.2)

The tensor product has the following properties:

- 1. If A, B, C and D are matrices with compatible dimensions, then $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$.
- 2. If A, B and C are matrices, then $(A \otimes B) \otimes C = A \otimes (B \otimes C)$.
- 3. If P and Q are permutation matrices, then so is $P \otimes Q$.

4. If I_a and I_b are identical matrices with size a and b, respectively, then $I_a \otimes I_b = I_{ab}$.

A proof of these properties is given in [2].

2.1.3 The Discrete Fourier Transform

Definition 4 (Discrete Fourier Transform) The Discrete Fourier Transform (DFT) of the discrete function X(a), $0 \le a < N$, is defined by

$$Y(b) = \sum_{a=0}^{N-1} \omega_N^{ab} X(a),$$
(2.3)

where $\omega_N = e^{2\pi i/N}$.

If the function X(a) and Y(b) are represented by column vectors x and y of size N, Equation 2.3 can be rewritten as

$$y = F_N x, \tag{2.4}$$

where F_N is the $N \times N$ DFT matrix, whose (i, j) entry $F_{i,j} = \omega_N^{ij}, 0 \le i, j < N$. For example

$$F_{2} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, F_{4} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix},$$

$$F_{8} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_{8} & i & \omega_{8}^{3} & -1 & \omega_{8}^{5} & -i & \omega_{8}^{7} \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & \omega_{8}^{3} & -i & \omega_{8} & -1 & \omega_{8}^{7} & i & \omega_{8}^{5} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & \omega_{8}^{5} & i & \omega_{8}^{7} & -1 & \omega_{8} & -i & \omega_{8}^{3} \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & \omega_{8}^{7} & -i & \omega_{8}^{5} & -1 & \omega_{8}^{3} & i & \omega_{8} \end{pmatrix}.$$

2.1.4 The Multi-Dimensional DFT

Definition 5 (Multi-dimensional DFT) Let $X(a_1, \dots, a_t)$ to be a function of t variables, where $0 \le a_1 < n_1, \dots, 0 \le a_t < n_t$. The t-dimensional DFT of X is defined by

$$Y(b_{1}, \cdots, b_{t}) = \sum_{\substack{0 \le a_{1} < n_{1}, \cdots \\ 0 \le a_{t} < n_{t}}} e^{2\pi i a_{1} b_{1}/n_{1}} \cdots e^{2\pi i a_{t} b_{t}/n_{t}} X(a_{1}, \cdots, a_{t})$$

$$= \sum_{\substack{0 \le a_{1} < n_{1}}} e^{2\pi i a_{1} b_{1}/n_{1}} \cdots \sum_{\substack{0 \le a_{t-1} < n_{t-1}}} e^{2\pi i a_{t-1} b_{t-1}/n_{t-1}}$$

$$(\sum_{\substack{0 \le a_{t} < n_{t}}} e^{2\pi i a_{t} b_{t}/n_{t}} X(a_{1}, \cdots, a_{t})) \qquad (2.5)$$

for $0 \le b_1 < n_1, \cdots, 0 \le b_t < n_t$.

Equation 2.5 implies that the multi-dimensional DFT can be computed by applying one-dimensional DFTs along each dimension. A sequence of n_t -point DFTs are applied to the function obtained by fixing the first t - 1 inputs to X for each value of (a_1, \dots, a_{t-1}) . Then a sequence of n_{t-1} -point DFT's are applied to the function obtained by fixing all but the (t - 2)-nd input to the result of the application of the n_t -point DFT. This process continues until finally a sequence of n_1 -points DFT is applied.

The multi-dimensional DFT can be interpreted as a matrix vector product. If the functions $X(a_1, a_2, \dots, a_t)$ and $Y(b_1, b_2, \dots, b_t)$ are stored lexicographically in vectors x and y, then

$$y = (F_{n_1} \otimes F_{n_2} \otimes \dots \otimes F_{n_t})x.$$
(2.6)

For example, when $n_1 = 2, n_2 = 4$,

$$x = (X(0,0), X(0,1), X(0,2), X(0,3), X(1,0), X(1,1), X(1,2), X(1,3))^T,$$

$$y = (Y(0,0), Y(0,1), Y(0,2), Y(0,3), Y(1,0), Y(1,1), Y(1,2), Y(1,3))^T,$$

and

$$Y(b_1, b_2) = \sum_{0 \le a_1 < 2} (-1)^{a_1 b_1} \sum_{0 \le a_2 < 4} i^{a_2 b_2} X(a_1, a_2),$$

$$y = \begin{pmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{pmatrix} \begin{pmatrix} F_4 \\ F_4 \end{pmatrix} \begin{pmatrix} X(0,0) \\ X(0,1) \\ X(0,2) \\ X(0,3) \\ X(1,0) \\ X(1,1) \\ X(1,2) \\ X(1,3) \end{pmatrix}$$
$$= (F_2 \otimes I_4)(I_2 \otimes F_4)x$$
$$= (F_2 \otimes F_4)x.$$
(2.7)

In general, for a two-dimensional DFT, Equation 2.5 implies

$$y = (F_{n_1} \otimes F_{n_2})x$$
$$= (F_{n_1} \otimes I_{n_2})(I_{n_1} \otimes F_{n_2})x.$$

Computing the two-dimensional DFT by first computing $t = (I_{n_1} \otimes F_{n_2})x$ and then $y = (F_{n_1} \otimes I_{n_2})t$ is called the **row-column** algorithm. If the function $X(a_1, a_2)$ is stored in an $n_1 \times n_2$ matrix, the computation proceeds by applying n_2 -point DFTs to the rows of X followed by n_1 -point DFTs to the columns of the partially transformed matrix.

The generalization of the row-column algorithm applied to a t-dimensional DFT corresponds to the factorization

$$F_{n_1} \otimes \cdots \otimes F_{n_t} = \prod_{i=1}^t (I_{N(i-1)} \otimes F_{n_i} \otimes I_{N/N(i)}),$$

where $N = n_1 n_2 \cdots n_t$, and $N(i) = n_1 n_2 \cdots n_i$.

2.2 The Fast Fourier Transform (FFT)

This section presents a factorization of the DFT matrix that corresponds to the divide and conquer step of the FFT. [1, 3, 4]

2.2.1 The Cooley-Tukey Factorization

The DFT matrix F_{mn} can be factorized into $F_{mn} = (F_m \otimes I_n)T(I_m \otimes F_n)P$, where T is a diagonal matrix called the twiddle factor matrix and L is a permutation matrix called a stride permutation.

Definition 6 (Twiddle factor) The twiddle factor matrix T_n^{mn} is defined by

$$T_n^{mn} = \bigoplus_{i=0}^{m-1} (W_n(\omega_{mn})^i))$$
(2.8)

where $W_n(\omega_{mn}) = diag(1, \omega_{mn}, \cdots, \omega_{mn}^{n-1})$. For example,

•

$$T_2^4 = diag(1, 1, 1, i)$$
, and $T_4^8 = diag(1, 1, 1, \omega_8, 1, \omega_8^2, 1, \omega_8^3)$.

Definition 7 (Stride permutation) Assume x is an mn-dimensional column vector. The stride permutation matrix of size mn with stride n is defined by

$$L_n^{mn} x = \begin{pmatrix} x(0:n:mn-1)^T \\ x(1:n:mn-1)^T \\ \vdots \\ x(n-1:n:mn-1)^T \end{pmatrix}.$$
 (2.9)

For example,

The stride permutation matrix has the following properties.

- 1. Assume N = RS, then $(L_S^N)^{-1} = L_R^N$.
- 2. Assume N = RS, and A_R and A_S are any arbitrary $R \times R$ and $S \times S$ matrices, then $A_R \otimes A_S = L_R^N (A_S \otimes A_R) L_S^N$.

The Cooley-Tukey factorization is stated precisely in the following theorem.

Theorem 1 (Cooley-Tukey Factorization) Assume N = RS, then

$$F_N = (F_R \otimes I_S) T_S^N (I_R \otimes F_S) L_R^N$$

Proof: Since $(L_R^N)^{-1} = L_S^N$, it is equivalent to show that the (k, l)-th element of the (i, j) block of $F_N L_S^N$ is equal to the (k, l)-th element of the (i, j) block of $(F_R \otimes I_S)T_S^N(I_R \otimes F_S)$. The row and column indices of the (k, l)-th element of the (i, j)block of $F_N L_S^N$ are (iS + k) and (j + lR). Therefore the (k, l)-th element of the (i, j)block of $F_N L_S^N$ is equal to $\omega_N^{(iS+k)(j+lR)}$.

Since N = RS and $\omega_N^N = 1$, $\omega_N^R = \omega_S$, and $\omega_N^S = \omega_R$, $\omega_N^{(iS+k)(j+lR)} = \omega_R^{ij} \omega_N^{kj} \omega_S^{kl}$, the (i, j) block of $F_N L_S^N$ is equal to $\omega_R^{ij} W_S^j F_S$.

Observe that

$$T_S^N(I_R \otimes F_S) = \bigoplus_{j=0}^{R-1} W_S^j F_S.$$

Consequently, the (i, j) block of $(F_R \otimes I_S)T_S^N(I_R \otimes F_S)$ is equal to $\omega_R^{ij}W_S^jF_S$. Therefore,

$$F_N L_S^N = (F_R \otimes I_S) T_S^N (I_R \otimes F_S)$$

Since $(L_S^N)^{-1} = L_R^N$,

$$F_N = (F_R \otimes I_S) T_S^N (I_R \otimes F_S) L_R^N.$$

For example,

$$F_{8} = (F_{2} \otimes I_{4})T_{4}^{8}(I_{2} \otimes F_{4})L_{2}^{8}$$
$$= \begin{pmatrix} I_{4} & I_{4} \\ I_{4} & -I_{4} \end{pmatrix}T_{4}^{8}\begin{pmatrix} F_{4} & 0 \\ 0 & F_{4} \end{pmatrix}L_{2}^{8}$$

2.2.2 A Divide and Conquer Algorithm for the FFT

An algorithm for computing the DFT can be obtained from the Cooley-Tukey factorization by applying the factors one after another to the input vector, using temporary vectors as needed. To compute $y = (F_R \otimes I_S)T_S^N(I_R \otimes F_S)L_R^N x$,

- 1. $t_1 = L_R^N x$; permute the input vector.
- 2. $t_2 = (I_R \otimes F_S)t_1$; compute R copies of F_S .
- 3. $t_3 = T_S^N t_2$; multiply by twiddle factors.
- 4. $y = L_R^N (I_S \otimes F_R) L_S^N t_3$; compute S copies of F_R at stride S.

Note: $I_R \otimes F_S$ is easily implemented with a loop that applies F_S to consecutive segments of the input. The computation of F_S and F_R can be performed by applying the Cooley-Tukey factorization recursively. An algorithm derived in this way is commonly called the fast Fourier transform.

Computing a given DFT F_N with Equation 2.4 directly takes computing time $\theta(N^2)$. In the Cooley-Tukey factorization, computation of the twiddle factor and the

stride permutation can be implemented in linear time, therefore the computing time for Cooley-Tukey factorization is

$$T(N) = RT(S) + ST(R) + \theta(N).$$
(2.10)

If $N = 2^K$, Equation 2.10 becomes $T(N) = 2T(N/2) + \theta(N)$, which can easily be shown to be $\theta(N \log N)$. [5]

There are many different ways to apply the Cooley-Tukey factorization recursively. For example, the Cooley-Tukey factorization can be applied in the following different ways to obtain

$$F_{16} = (F_2 \otimes I_8) T_8^{16} (I_2 \otimes F_8) L_2^{16}$$
$$= (F_4 \otimes I_4) T_4^{16} (I_4 \otimes F_4) L_4^{16}$$
$$= (F_8 \otimes I_2) T_2^{16} (I_8 \otimes F_2) L_8^{16}$$

In each of these factorizations, the Cooley-Tukey factorization can be applied to the smaller DFTs until the base case, F_2 , is obtained. At each stage of the computation the factorization can be applied for each of different factorization of N = RS. The sequence of recursive application of Cooley-Tukey factorization can be recorded in a tree, called factorization tree, where nodes are labeled by the corresponding transform size, and each distinct tree corresponds to a different FFT algorithm. A fully expanded tree is a tree whose leaf nodes are prime numbers. Figure 2.1 shows all 5 factorization trees for $N = 2^4$. Tree (a) corresponds to recursive factorization to the right, and tree (d) corresponds to recursive factorization to the left. These two trees correspond to the standard recursive FFT algorithms.



Figure 2.1: The possible Cooley-Tukey factorizations of N = 16

For $N = 2^{K}$, there are K - 1 nontrivial factorizations of F_{N} , corresponding to

$$K = 1 + (K - 1)$$

= 2 + (K - 2)
= :
= (K - 1) + 1.

Therefore the number of fully expanded factorization trees of $N = 2^K$ is equal to the recurrence

$$b_K = \sum_{i=1}^{K-1} b_i b_{K-i},$$
(2.11)

with the base case $b_1 = 1$. It is well-known [5] that $b_k = \theta(\frac{4^K}{K^{3/2}})$.

If the trees are not necessarily fully expanded, and leaf nodes are allowed to be of composite size, the number of factorization trees satisfies the recurrence

$$b_K = \sum_{i=1}^{K-1} b_i b_{K-i} + 1, \qquad (2.12)$$

and it can be shown that $b_k = \theta(\frac{5^K}{K^{3/2}})$.

2.2.3 An FFT Implementation

This section contains a Matlab implementation of the FFT described in the previous section. The purpose of the implementation is to precisely state and verify the algorithm; it is not intended to be an efficient implementation. An efficient implementation will be discussed in Chapter 3.

In order to implement an arbitrary FFT algorithm as discussed in the previous section, it is necessary to pass a factorization tree that indicates the factorizations that occur in the algorithm. The implementation here uses a split table to indicate how a given node is factored. If D[k]=j, then

$$F_{2^{k}} = (F_{2^{j}} \otimes I_{2^{k-j}}) T_{2^{k-j}}^{2^{k}} (I_{2^{j}} \otimes F_{2^{k-j}}) L_{2^{j}}^{2^{k}}$$

is used whenever a node of size 2^k is encountered. This implies that the same factorization is used for all computations of F_{2^k} occurring in the algorithm, and consequently this implementation does not allow all possible algorithms.

Figure 2.2 shows the function $FFT_2(N, n, D, x)$, which implements the computation of $y = F_N x$ with Cooley-Tukey factorization. It is named FFT_2 because there is already a function named FFT in Matlab. FFT_2 takes the following arguments as its input.

- N: a positive integer, which is supposed to be a power of 2
- n: a positive integer, with $n = \log_2 N$
- D: a column vector with at least n entries, representing the split table.
- x: a column vector with size N

FFT2 computes the given FFT with the following steps.

1. Split N

```
function y=FFT2(N,n,D,x)
if D(n) == n
    % The base case
    y = Fourier(N)*x;
else
    % factor n=r+s, N=RS
    r = D(n,1); R = 2^r; s = n-r; S = 2^s;
    % Permute input vector
    t1 = ones(N,1); %initialize temporary vector
    for k=1:S
        for j=1:R
            t1((j-1)*S+k) = x((k-1)*R+j);
        end
    end
    % Compute kron(I_R,F_S)
    t2 = ones(N,1);%initialize temporary vector
    for k=1:R
        t2((k-1)*S+1:1:k*S,1) = FFT2(S,s,D,t1((k-1)*S+1:1:k*S,1));
    end
    % Multiply by twiddle factors
    t3 = zeros(N,1); %initialize temporary vector
    w = ones(S,1);
    w_n = exp(-2*pi*i/N*[0:S-1])';
    for k=1:R
        t3((k-1)*S+1:1:k*S,1) = w.*t2((k-1)*S+1:1:k*S,1);
        w = w.*w_n; % product
    end
    % Compute kron(F_S,I_R)
    y = ones(N,1); %initialize output vector
    for k=1:S
        y(k:S:k+(R-1)*S,1) = FFT2(R,r,D,t3(1:S:k+(R-1)*S,1));
    end
```

Figure 2.2: A divide and conquer algorithm for FFT

This step checks the input DFT size. If D[n] = n, then it computes the DFT directly using a matrix-vector product. Otherwise it factors N = RS with $R = 2^{D[k]}$ and S = N/R.

- 2. Permute the input vector with $t_1 = L_R^N x$.
- 3. Compute $t_2 = (I_R \otimes F_S)t_1$
- 4. Multiply by the twiddle factors $t_3 = T_S^N t_2$.
- 5. Compute $y = (F_R \otimes I_S)t_3$ using $(F_R \otimes I_S) = L_R^N(I_S \otimes F_R)L_S^N$.

2.3 The Dimensionless FFT

The dimensionless FFT of size N computes all possible multi-dimensional DFTs of size N. The structure of the algorithm depends only on the size N and is independent of dimension. For a given $N = 2^K$, there are N/2 multi-dimensional DFTs of size N. For example, let \mathcal{F}_{16} denotes the set of multi-dimensional DFTs of size 16,

$$\mathcal{F}_{16}: \begin{cases} F_{16} \\ F_2 \otimes F_8, F_4 \otimes F_4, F_8 \otimes F_2 \\ F_2 \otimes F_2 \otimes F_4, F_2 \otimes F_4 \otimes F_2, F_4 \otimes F_2 \otimes F_2 \\ F_2 \otimes F_2 \otimes F_2 \otimes F_2 \otimes F_2 \end{cases}$$

The elements of \mathcal{F}_{16} are organized in rows by dimension.

This section provides a generalization of the Cooley-Tukey factorization that applies simultaneously to all multi-dimensional DFTs of size $N = 2^{K}$. This factorization serves as the basis of the dimensionless FFT.

2.3.1 The Dimensionless Cooley-Tukey Factorization

The dimensionless Cooley-Tukey factorization provides an analog of the Cooley-Tukey factorization (Theorem 1) for one-dimensional DFTs that applies to all multidimensional DFTs of a fixed size. The dimensionless Cooley-Tukey factorization provides a generic factorization that simultaneously applies to all multi-dimensional DFTs of size N. When the dimension changes, the only change in the factorization is the value of the twiddle factor and the permutation.

Let \mathcal{F}_N be an arbitrary multi-dimensional DFT of size N. Then $\mathcal{F}_N = (\mathcal{F}_R \otimes I_S)D(I_R \otimes \mathcal{F}_S)P$, where \mathcal{F}_R and \mathcal{F}_S are multi-dimensional DFTs of size R and S respectively, D is a diagonal matrix and P is a permutation matrix. Theorem 2 states this result more precisely.

Theorem 2 (Dimensionless Cooley-Tukey Factorization) Assume $N = 2^K$ with $N = n_1 \times n_2 \times \cdots \times n_t$, $n_i = 2^{k_i}$. Let $\mathcal{F}_N = F_{n_1} \otimes F_{n_2} \otimes \cdots \otimes F_{n_t}$, and let N = RS to be a non-trivial factorization of N with R = N(l-1)a, $S = b\overline{N(l)}$ and $n_l = ab$, and $N(l) = n_1 n_2 \cdots n_l$, $1 \le l \le t$ and N(0) = 1, $\overline{N(l)} = N/N(l)$. Then

$$\mathcal{F}_N = (\mathcal{F}_R \otimes I_S) D(I_R \otimes \mathcal{F}_S) P \tag{2.13}$$

where $\mathcal{F}_R = F_{n_1} \otimes F_{n_2} \otimes \cdots \otimes F_{n_{l-1}} \otimes F_a$ $\mathcal{F}_S = F_b \otimes F_{n_{l+1}} \otimes \cdots \otimes F_{n_t}$ $D = I_{N(l-1)} \otimes T_b^{n_l} \otimes I_{\overline{N(l)}}$ $P = I_{N(l-1)} \otimes L_a^{n_l} \otimes I_{\overline{N(l)}}$

Proof: Given $\mathcal{F}_N = F_{n_1} \otimes F_{n_2} \otimes \cdots \otimes F_{n_t}$ and N = RS, let l be the smallest index such that $n_1 \times n_2 \times \cdots \times n_l \ge R$. Because $n_i = 2^{k_i}$, it is always possible to factor $n_l = a \times b$ such that $n_1 \times n_2 \times \cdots \times n_{l-1} \times a = R$ and $b \times n_{l+1} \times \cdots \times n_t = S$. By the Cooley-Turkey theorem, $F_{n_l} = (F_a \otimes I_b)T_b^{n_l}(I_a \otimes F_b)L_a^{n_l}$, and

$$\mathcal{F}_N = F_{n_1} \otimes F_{n_2} \otimes \cdots \otimes F_{n_{l-1}} \otimes (F_a \otimes I_b) T_b^{n_l} (I_a \otimes F_b) L_a^{n_l}) \otimes F_{n_{l+1}} \otimes \cdots \otimes F_{n_t}$$
(2.14)

Using the notation N(i) and $\overline{N(i)}$ and Property 1 of the tensor product, Equation

2.14 can be rewritten as

$$\mathcal{F}_{N} = (\mathcal{F}_{N(l-1)} \otimes F_{a} \otimes I_{b} \otimes I_{\overline{N(l)}})(I_{N(l-1)} \otimes T_{b}^{n_{l}} \otimes I_{\overline{N(l)}})$$
$$(I_{N(l-1)} \otimes I_{a} \otimes F_{b} \otimes \mathcal{F}_{\overline{N(l)}})(I_{N(l-1)} \otimes L_{a}^{n_{l}} \otimes I_{\overline{N(l)}})$$
$$= (\mathcal{F}_{R} \otimes I_{S})(I_{N(l-1)} \otimes T_{b}^{n_{l}} \otimes I_{\overline{N(l)}})(I_{R} \otimes \mathcal{F}_{S})(I_{N(l-1)} \otimes L_{a}^{n_{l}} \otimes I_{\overline{N(l)}}).$$

2.3.2 A Divide and Conquer Algorithm for the Dimensionless Cooley-Tukey Factorization

With Theorem 2, the dimensionless FFT of a column vector x can be computed as

$$y = \mathcal{F}_N x = (\mathcal{F}_R \otimes I_s) D(I_R \otimes \mathcal{F}_s) P x.$$
(2.15)

Because the dimensionless Cooley-Turkey factorization also splits one large FFT matrix \mathcal{F}_N into two smaller FFT matrices \mathcal{F}_R and \mathcal{F}_S , applying it recursively leads to a divide and conquer algorithm. This section uses Matlab code to explain the algorithm.

The function that computes $y = \mathcal{F}_N x$, called *FFTMD*, takes the following input arguments.

- N: a positive integer, which is supposed to be a power of 2
- n: a t-dimensional column vector, such that $\prod_{i=1}^{t} n_i = N, n_i = 2^{k_i}$
- D: a column vector with at least $\log_2 N$ dimensions, corresponding to the split table
- x: a column vector with N entries

The matlab code is shown in Figure 2.3, 2.4, 2.5 and 2.6. Similar to FFT2, FFTMD implements a divide and conquer algorithm with the following steps.

1. Split N

If the dimension is 1, use *FFT2*, otherwise use the split table to factor N = RSand determine the index l such that $R = \prod_{i=1}^{l-1} n(i, 1) * a$ and n(l) = ab. The arguments n1 and n2 in Figure 2.3 correspond to the arguments N(l-1) and $\overline{N(l)}$ in theorem 2, respectively.

- 2. Permute the input vector with $t_1 = Px$, where $P = (I_{N(l-1)} \otimes L_a^{n_l} \otimes I_{\overline{N(l)}})x$.
- 3. Compute $t_2 = (I_R \otimes \mathcal{F}_S)t_1$
- 4. Multiply by the twiddle factors $t_3 = (I_{N(l-1)} \otimes T_b^{n_l} \otimes I_{\overline{N(l)}})t_2$
- 5. Compute $y = (\mathcal{F}_R \otimes I_S)t_3$

```
function y=FFTMD(N,n,D,x)
t = size(n); t = t(1,1); y = ones(N,1);
if t == 1
    \% base case, one dimensional FFT
    y = FFT2(N, log2(N), D, x);
else
    % split N=RS
    r = D(log2(N), 1); R = 2^r; S = N/R;
    \% Find the n(l) to be factored
    l = index(N,R,n);
    n1 = 1;
    for i=1:1-1
        n1 = n1*n(i,1);
    end
    n2 = N/n1/n(1,1);
    a = R/n1; b=S/n2;
    % Permutes input vector
    t1 = VectorPerm(n1,n(1,1),a,n2,x);
    % Compute kron(I_R,F_S)
    T = ones(t-l+1,1);
    T(1,1) = b;
    for (j=2:t-i+1)
        T(j,1) = n(i+j-1,1);
    end
    for j=1:R
        t2((j-1)*S+1:1:j*S) = FFTMD(S,T,D,t1((j-1)*S+1:1:j*S));
    end
    % Multiply by twiddle factors
    t3 = VectorD(n1, n(i, 1), b, n2, t2);
    % Compute kron(F_R,I_S)
    if a==1
        T = ones(1-1,1);
        for j=1:1-1
            T(j,1) = n(j,1);
        end
    else
        T = ones(1,1);
        for j=1:1-1
            T(j,1) = n(j,1);
        {\tt end}
        T(1,1) = a;
    end
    for j=1:S
        y(j:S:j+(R-1)*S,1) = FFTMD(R,T,D,t3(j:S:j+(R-1)*S));
    end
end
```



```
function y=index(n,R,t)
m = 1;
for i=1:t
    m = m*n(i,1);
    if m>n
        break;
    end
end
i;
```

Figure 2.4: Function index, find the dimension to split

```
% Vector permutation implement function
% Inputs:
%
     a: a positive integer, which is supposed to be a power of 2
%
     n: a positive integer, which is supposed to be a power of 2
%
     m: a positive integer, which is supposed to be a power of 2, and m<=n
%
    b: a positive integer, which is supposed to be a power of 2
     x: an a*n*b dimensional column vector
%
% Output:
%
     y: an a*n*b dimensional column vector,
%
        y=kron(eye(a),kron(stride(n,m),eye(b)))*x
function y = VectorPerm(a,n,m,b,x)
k = a*b*n;
s = size(x);
s = s(1);
if (s~=k)
    error('Input matrix and vector dimensions do not match');
end
y = ones(s, 1);
for i=1:a
    t = zeros(n*b, 1);
    for j=1:n/m
        for k=1:m
            t(((k-1)*n/m+j-1)*b+1:1:((k-1)*n/m+j)*b,1) =
            x(((j-1)*m+k-1)*b+1+(i-1)*n*b:1:((j-1)*m+k)*b+(i-1)*n*b,1);
        end
    end
    %y((i-1)*n*b+1:1:i*n*b,1) = t;
    for j=1:n*b
        y((i-1)*n*b+j,1) = t(j,1);
    end
end
y;
```



```
% Vector twiddle function
% Inputs:
%
     a: a positive integer, which is supposed to be a power of 2
%
     n: a positive integer, which is supposed to be a power of 2
%
     m: a positive integer, which is supposed to be a power of 2, and m<=n
%
     b: a positive integer, which is supposed to be a power of 2
%
     x: an a*n*b dimensional column vector
% Output:
%
     y: an a*n*b dimensional column vector,
%
        y=kron(eye(a),kron(twiddle(n,m),eye(b)))*x
function y = VectorD(a,n,m,b,x)
k = a*b*n;
s = size(x);
s = s(1);
if (s~=k)
    error('Input matrix and vector dimension do not match');
end
I = sqrt(-1);
y = ones(s, 1);
for i=1:a
    t = zeros(n*b, 1);
    w = ones(m*b, 1);
    w_n = \exp(-2*pi*I/n*[0:m-1])';
    w_n = kron(w_n, ones(b,1)); % tensor product
    for j=1:n/m
        t((j-1)*m*b+1:1:j*m*b,1) =
        w.*x((i-1)*n*b+(j-1)*m*b+1:1:(i-1)*n*b+j*m*b,1);
        w = w.*w_n;
    end
    y((i-1)*n*b+1:1:i*n*b,1) = t;
end
y;
```

Figure 2.6: Multiply twiddle factor $I_a \otimes T_m^n \otimes I_b$

Chapter 3: FFTW

This chapter presents an overview of FFTW, an efficient package for computing oneand multi-dimensional DFTs. FFTW has significantly better performance than standard approaches such as that found in Numerical Recipes [12] and outperforms many industrial grade packages (see www.fftw.org/benchft). FFTW self adapts to tune its performance to a given hardware platform. The self adaptation is done by generating and timing alternative decompositions and selecting the best through a dynamic programming approach.

In the remainder of this chapter, sufficient details of FFTW (version 2.1.3) are provided to understand the modifications made to support the dimensionless FFT. Additional details, including the documentation and the source code are available at http://www.fftw.org.

3.1 FFTW Overview

In FFTW, the computation of the DFT is performed by an **executor** using highly optimized, composable blocks of C code called **codelets**. A codelet is a specialized piece of code that computes part of the transform. The combination of codelets applied by the executor is specified by a data structure called a **plan**. The **planner** determines an appropriate plan for a given transform size and hardware platform by searching for the fastest possible execute time. Once a good plan has been determined, it can be saved and used for future computation.

The **executor** implements the Cooley-Tukey factorization. The algorithm recursively computes R DFTs of size S, multiplies the *twiddle factors*, and finally computes S DFTs of size R. FFTW has a library of *codelets* that implement small DFTs. There are two kinds of codelets: 1) *no-twiddle* codelets, which compute the DFT of a fixed size, and are used as the base case of the recursion, 2) *twiddle* codelets, which multiply their input by the twiddle factors before computing the DFT of a fixed size, and are used for the internal levels of the recursion.

FFTW uses a modified version of the Cooley-Tukey factorization given in the following equation.

$$y = L_R^N (I_S \otimes F_R) T_R^N L_S^N (I_R \otimes F_S) L_R^N x$$
(3.1)

The computation is performed in two steps.

1.
$$y = (I_R \otimes F_S) L_R^N x$$

2.
$$y = L_R^N (I_S \otimes F_R) T_R^N L_S^N y$$

The first step is an out-of-place computation with the result stored in the output vector y, whereas the second step is performed in-place without any need for extra storage. Step one may be computed recursively where the strides from the recursive calls are combined and the base case is computed by a no-twiddle codelet. Step two is always performed by a twiddle codelet which is written so that the computation can be done in-place.

The **plan** is a recursive data structure that specifies the recursive computation used for a DFT of a given size. The plan corresponds to the tree notation used in Chapter 2 to represent the recursive computation of the DFT. However, the only trees that arise in FFTW are rightmost trees, i.e. trees whose left children are all leaf nodes. This is due to the fact that in FFTW the left factor of the Cooley-Tukey factorization is always computed by a twiddle codelet. In addition to the structure of the recursive computation, the plan data structure (more precisely, the plan_node data structure) also includes pointers to the codelets used and stores the values of the twiddle factors used by the twiddle codelets.

For example, Figure 3.1 shows the plan corresponding to the factorization

$$F_{128} = (F_4 \otimes I_{32}) T_{32}^{128} (I_4 \otimes F_{32}) L_4^{128}$$

= $(F_4 \otimes I_{32}) T_{32}^{128} (I_4 \otimes ((F_8 \otimes I_4) T_4^{32} (I_8 \otimes F_4) L_8^{32})) L_4^{128}$



Figure 3.1: A possible plan for N = 128

The twiddle factors are stored in the two internal nodes.

$$T_{32}^{128} = \bigoplus_{i=0}^{3} W_{32}^{i} = \bigoplus_{i=0}^{3} diag(1, \omega_{128}, \omega_{128}^{2}, \cdots, \omega_{128}^{31})^{i}$$
$$T_{4}^{32} = \bigoplus_{i=0}^{7} W_{4}^{i} = \bigoplus_{i=0}^{7} diag(1, \omega_{32}, \omega_{32}^{2}, \omega_{32}^{3})^{i}$$

In order to construct an efficient plan, FFTW measures the execution time of different plans and selects the plan which gives the best performance. Ideally, FFTW should try all possible plans, but in practice this is not feasible, since there are exponentially many possible plans. While the number of plans (rightmost trees) is considerably less than the total number of recursive FFT algorithms (see Chapter 2), there are still too many for exhaustive search. Therefore, FFTW uses a dynamic programming algorithm to do the search. The dynamic programming algorithm assumes that the execution time of the best plan is fixed for a given input size, and is not related to the calling context. A table of plans is created bottom up, starting with N = 2. For the next larger entry in the table, all possible splits, with the left child a twiddle codelet, are timed using the plans in the table for the recursive calls and the split with the fastest runtime is selected and added to the table.

For example, assume the planner is searching for the best plan for N = 64, and the best plans for N = 2, 4, 8, 16 and 32 are known. The five splits shown in Figure 3.2, using the best known plans, are generated and timed. If exhaustive search is


Figure 3.2: All possible factorizations of N = 64

used, 32 plans must be timed instead of 5.

Table 3.1 shows the number of plans that can be used by FFTW to compute DFT_N , $N = 2^K$, for $K = 1, 2, \dots, 20$, using codelets up to size 64. The total number of FFT algorithms is also listed for comparison purposes. The number of plans satisfies the recurrence relation

$$T_6(K) = \begin{cases} \sum_{i=1}^6 T_6(K-i) & K > 6\\ 2^{K-1} & K \le 6 \end{cases}$$

It can be shown that $T_6(K) = \theta(\alpha^K)$, where $\alpha \doteq 1.98$ is the largest real root of the characteristic equation

$$x^6 - x^5 - x^4 - x^3 - x^2 - x - 1 = 0$$

Recall that the total number of FFT algorithms is $\theta(5^K/K^{3/2})$. Despite the reduced number that FFTW considers, there are still exponentially many possibilities.

3.2 FFTW Plans and the Recursive Planner

This section discusses the plan data structure and the recursive planner in detail.

3.2.1 FFTW Plan Data Structure

The FFTW plan, $fftw_plan$, is a data structure containing all necessary information to compute a one-dimensional DFT: the recursive factorization used, codelets and twiddle factors. The $fftw_plan$, defined in Figure 3.3, is a data structure describing a factorization tree. Details of some components of $fftw_plan$ are listed below.

K	Number of factorizations of $N = 2^K$	Number of possible FFTW
		plan of $N = 2^K$
1	1	1
2	2	2
3	5	4
4	15	8
5	51	16
6	189	32
7	732	63
8	2223	125
9	7107	248
10	23484	492
11	79071	976
12	267972	1936
13	901860	3840
14	2968023	7617
15	9838575	15109
16	32774598	29970
17	1.09×10^{8}	59448
18	3.65×10^{8}	117920
19	1.22×10^{9}	233904
20	4.05×10^{9}	463968

Table 3.1: Number of factorization and FFTW plan

```
struct fftw_plan_struct {
    int n;
    int refcnt;
    fftw_direction dir;
    int flags;
    int wisdom_signature;
    enum fftw_node_type wisdom_type;
    struct fftw_plan_struct *next;
    fftw_plan_node *root;
    double cost;
    fftw_recurse_kind recurse_kind;
    int vector_size;
};
```

typedef struct fftw_plan_struct *fftw_plan;

Figure 3.3: FFTW plan data structure

- n is the size of the transform. While n can be any positive integer, FFTW is best at handling sizes of the form 2^a, 3^b, 5^c, 7^d, 11^e, 13^f, where e + f is either 0 or 1.
- dir can be -1 or 1, and is the sign of the exponent in the formula that defines the DFT. The aliases FFTW_FORWARD (-1) and FFTW_BACKWARD (+1) are provided.
- 3. **flags** is a boolean OR of zero or more of the following:
 - FFTW_MEASURE: this flag tells FFTW to find the optimal plan by actually *computing* several FFTs and measuring their execution time.
 - FFTW_ESTIMATE: this flag tells FFTW to provide a "reasonable" plan instead of computing FFTs. It is the default case.
 - FFTW_OUT_OF_PLACE: this flag produces a plan assuming that the input and output arrays are distinct. It is the default case.
 - FFTW_IN_PLACE: this flag produces a plan assuming that the output is the same as the input array.
 - FFTW_USE_WISDOM: this flag uses any *wisdom* that is available to help in the creation of the plan.
- 4. **refcnt** is the reference count of a given *fftw_plan*.
- 5. **next** is a pointer to another *fftw_plan*, that points to the right child of the current node.
- 6. **root** is a pointer to an *fftw_plan_node*, which is defined in Figure 3.4. It contains a pointer to the codelet and twiddle factors used by the left child, or just the codelet if the node corresponds to the base of the recursion.

```
typedef struct fftw_plan_node_struct {
     enum fftw_node_type type;
     union {
          /* nodes of type FFTW_NOTW */
          struct {
               int size;
               fftw_notw_codelet *codelet;
               const fftw_codelet_desc *codelet_desc;
          } notw;
          /* nodes of type FFTW_TWIDDLE */
          struct {
               int size;
               fftw_twiddle_codelet *codelet;
               fftw_twiddle *tw;
               struct fftw_plan_node_struct *recurse;
               const fftw_codelet_desc *codelet_desc;
          } twiddle:
          /* other cases are not listed here */
     } nodeu;
```

```
int refcnt;
} fftw_plan_node;
```

Figure 3.4: *fftw_plan_node* data structure

7. **cost** is the evaluation of the specified plan. If it is known, the cost is the execution time.

In addition to *fftw_plan*, other data structures are used.

1. *fftw_plan_node* data structure

Figure 3.4 shows the definition of *fftw_plan_node*, which contains three components.

- (a) type refers to the type of the current *fftw_plan_node*. In FFTW, there are a total of seven types of *fftw_plan_nodes*.
- (b) **nodeu** is the union of seven structures, corresponding to the seven types

Figure 3.5: *fftw_codelet_desc* data structure

```
typedef struct fftw_twiddle_struct {
    int n;
    const fftw_codelet_desc *cdesc;
    fftw_complex *twarray;
    struct fftw_twiddle_struct *next;
    int refcnt;
} fftw_twiddle;
```

Figure 3.6: *fftw_twiddle* data structure

of *fftw_plan_nodes*. Figure 3.4 only shows the structures for the types FFTW_NOTW and FFTW_TWIDDLE, corresponding to the no-twiddle and twiddle codelets. Each structure contains pointers to the codelet, the codelet description, the twiddle factors and the recursive *fftw_plan_node*.

- (c) **refcnt** is the reference count to *fftw_plan_node*.
- 2. *fftw_codelet_desc* data structure

Figure 3.5 shows the definition of $fftw_codelet_desc$, which contains a description of a codelet.

3. *fftw_twiddle* data structure

```
struct wisdom {
    int n;
    int flags;
    fftw_direction dir;
    enum fftw_wisdom_category category;
    int istride;
    int ostride;
    int ostride;
    int vector_size;
    enum fftw_node_type type; /* this is the wisdom */
    int signature; /* this is the wisdom */
    fftw_recurse_kind recurse_kind; /* this is the wisdom */
    struct wisdom *next;
};
```

Figure 3.7: *wisdom* data structure

```
void fftw_export_wisdom(void(* emitter)(char c, void *), void *data);
void fftw_export_wisdom_to_file(FILE *output_file);
char *fftw_export_wisdom_to_string(void);
```

Figure 3.8: Exporting wisdom

The definition of $fftw_twiddle$ is shown in Figure 3.6. This structure is used to represent a twiddle factor T_s^n , where n is given directly, and s is given in **cdesc**. The values of the twiddle factors are stored in the array **twarray**.

4. wisdom data structure

FFTW implements a method for saving plans to disk and restoring them. This mechanism is called *wisdom*, whose definition is given in Figure 3.7.

When a plan is created with FFTW_MEASURE and FFTW_USE_WISDOM flags, a copy of this plan is stored. Thereafter, if the FFTW_USE_WISDOM is set and the plan was previously stored it is simply reused. FFTW has mechanisms to save and read wisdom to disk, which are shown in Figure 3.8 and 3.9.

```
fftw_status fftw_import_wisdom(int(* get_input)(void *),void *data);
fftw_status fftw_import_wisdom_from_file(FILE *input_file);
fftw_status fftw_import_wisdom_from_string(const char *input_string);
```

Figure 3.9: Importing wisdom

3.2.2 Recursive Planner

In FFTW, the planner creates a plan for an FFT of a given size using dynamic programming. It also provides *wisdom* to retrieve previously computed plans in order to save time since the dynamic programming procedure involves executing many alternative FFTs.

The following two functions are responsible for creating a plan.

The function $fftw_create_plan$ returns either a plan to compute a DFT of size n or NULL if it fails. The function $fftw_create_plan_specific$ takes additional arguments that specify the input and output arrays and strides used to execute different FFTs. The resulting plans will be optimized using the given arrays and strides. If the user calls the function $fftw_create_plan$, it will allocate temporary arrays for input and output, set the stride to 1, and call the function $fftw_create_plan_specific$.

The function *fftw_create_plan_specific* first creates an empty table, which is implemented using a linked list. The table stores the "optimal" plans (recall that since the sub-optimality principle is not guaranteed to be satisfied, the plans computed by dynamic programming (DP) may not be optimal and are only approximations to the optimal plans) of different sizes, and used for the search. After the table is created, the function *fftw_create_plan_specific* calls the function *planner* to search for a good plan of the given size. The function *planner* first checks the table to see whether there is already an optimal plan for the given size. If there is already such a plan in the table, then the function *planner* returns this plan, otherwise it checks the wisdom by calling the function *planner_wisdom*. If such a plan is found in wisdom, then it returns the plan. If no plan is found in the table or wisdom, the function *planner* calls the function *planner_normal* to search for a good plan using DP, and adds the returned plan to the table and wisdom.

34

The function planner_normal tries all possible factorizations $F_N = L_R^N(I_S \otimes F_R)T_R^N L_S^N(I_R \otimes F_S)L_R^N$ such that there is a twiddle codelet of size R. It also considers the computation of F_N with a no-twiddle codelet of size N if it exists. For each factorization, planner_normal calls the function planner recursively to determine the optimal plan of size S. Once the plan of size S is returned, a plan for F_N is constructed using this plan and the twiddle codelet of size R. The FFT using the constructed plan is then executed and timed. The function planner_normal returns the plan corresponding to the factorization with the smallest runtime, or the plan with the no-twiddle codelet if it takes less time than all the factored FFTs.

3.3 Computing the One-dimensional DFT

The arguments are listed and discussed below.

After an $fftw_plan$ is created, it can be used to compute the one-dimensional DFT for one or multiple input vectors. In FFTW, the computation of one-dimensional DFT starts from function fftw, which could compute multiple one-dimensional DFTs, or $fftw_one$, which only computes one DFT.



Figure 3.10: Computing One-dimensional FFT

- **plan** is a valid *fftw_plan* created by the recursive planner. (See Section 3.2).
- howmany is the number of DFTs to be computed. It is faster to use *fftw* to compute many DFTs than calling *fftw_one* many times, computing one DFT each time.
- in, istride and idist describe the input array(s). The input array is segmented into howmany subarrays with base addresses in, in+idist, ..., in+(howmany-1)*dist. Each subarray is accessed at stride istride.
- **out**, **ostride** and **odist** describe the output array(s) in the same way as input arrays. If the plan specifies an in-place transform, **ostride** and **odist** are always ignored.

3.3.1 Executor

Figure 3.10 shows the process of computing a one-dimensional DFT. The computation starts from the function $fftw_one$, then calls the function $fftw_executor_simple$, which calls either a codelet to compute the base case of the FFT or executor_many to recursively compute the FFT using the Cooley-Tukey factorization (Theorem 1). The factorizations used are determined by the specified plan. The combination of $fftw_executor_simple$ and executor_many is called the executor. Suppose an input transform of size N is factorized into N = RS by a specific $fftw_plan$, the executor recursively computes R transforms of size S by calling the executor recursively or a no-twiddle codelet. Then it calls the twiddle_codelet, which multiplies the results by twiddle factors, and computes S transforms of size R at a fixed stride.

3.3.2 Codelets

Codelets are optimized fragments of C code specialized to compute a DFT of a fixed size. In FFTW, there are seven kinds of codelets, the most important are no-twiddle codelet, *fftw_notw_codelet*, and twiddle codelet, *fftw_twiddle_codelet*. A no-twiddle codelet computes the DFT of a fixed size, and is used as the base case in the recursion. The twiddle codelet is used for the internal levels of the recursion, and includes computation of the twiddle factors. Twiddle codelets compute multiple DFTs of a fixed size and consequently use a loop to iterate over the straight-line code for a given DFT.

• The no-twiddle codelet

A no-twiddle codelet takes four input arguments, which represent the initial address (a pointer) of the input and output array, the input stride and output stride, respectively. The no-twiddle codelets perform an out-of-place computation, it computes the DFT of the input array, and puts the result into the output array. Figure 3.11 shows the source code of *fftw_no_twiddle_4*, the no-twiddle codelet of size 4.

• The twiddle codelet

A twiddle codelet takes five input arguments: a pointer to the input/output array, a pointer to the twiddle factor, the input/output stride, the number of transforms to compute, and the initial address distance between the input/output vectors of different transforms. The twiddle codelets perform in-place computation, with the input array modified to contain the result when the call completes. Figure 3.12 shows the source code of $fftw_twiddle_4$, the twiddle codelet of size

```
void fftw_no_twiddle_4(const fftw_complex *input, fftw_complex *output,
                      int istride, int ostride)
     fftw_real tmp3; fftw_real tmp11; fftw_real tmp9;
     fftw_real tmp15; fftw_real tmp6; fftw_real tmp10;
     fftw_real tmp14; fftw_real tmp16;
```

```
ASSERT_ALIGNED_DOUBLE;
{
     fftw_real tmp1; fftw_real tmp2;
     fftw_real tmp7; fftw_real tmp8;
     ASSERT_ALIGNED_DOUBLE;
     tmp1 = c_re(input[0]);
     tmp2 = c_re(input[2 * istride]);
     tmp3 = tmp1 + tmp2;
     tmp11 = tmp1 - tmp2;
     tmp7 = c_im(input[0]);
     tmp8 = c_im(input[2 * istride]);
     tmp9 = tmp7 - tmp8;
     tmp15 = tmp7 + tmp8;
}
{
     fftw_real tmp4; fftw_real tmp5;
     fftw_real tmp12; fftw_real tmp13;
     ASSERT_ALIGNED_DOUBLE;
     tmp4 = c_re(input[istride]);
     tmp5 = c_re(input[3 * istride]);
     tmp6 = tmp4 + tmp5;
     tmp10 = tmp4 - tmp5;
     tmp12 = c_im(input[istride]);
     tmp13 = c_im(input[3 * istride]);
     tmp14 = tmp12 - tmp13;
     tmp16 = tmp12 + tmp13;
}
c_re(output[2 * ostride]) = tmp3 - tmp6;
c_re(output[0]) = tmp3 + tmp6;
c_im(output[ostride]) = tmp9 - tmp10;
c_im(output[3 * ostride]) = tmp10 + tmp9;
c_re(output[3 * ostride]) = tmp11 - tmp14;
c_re(output[ostride]) = tmp11 + tmp14;
c_im(output[2 * ostride]) = tmp15 - tmp16;
c_im(output[0]) = tmp15 + tmp16;
```

{

}

3.4 Multi-dimensional Fourier Transform in FFTW

FFTW provides routines to compute multi-dimensional DFT using the row-column algorithm.

3.4.1 Multi-dimensional Plans

The data structure for a plan for a multi-dimensional DFT, *fftwnd_plan*, is defined in Figure 3.13. Most of the components are similar to those in *fftw_plan*, except for the following.

- rank is a nonnegative integer equal to the dimensions of the transform.
- **n** is a pointer to an integer array of size **rank**, containing the size of the transform in each dimension.
- plans holds the one-dimensional FFTW plans for each dimension.

3.4.2 Creating a Plan for Multi-dimensional DFTs

The following two functions create a valid *fftwnd_plan*.

The function *fftwnd_create_plan* returns a valid *fftwnd_plan*, or NULL if for some reason, the plan cannot be created. This can happen if memory runs out or if the arguments are invalid. The function *fftwnd_create_plan_specific* takes additional arguments specific input/output arrays and their strides. The additional arguments

```
void fftw_twiddle_4(fftw_complex *A, const fftw_complex *W,
                    int iostride, int m, int dist)
{
     int i;
    fftw_complex *inout;
     inout = A;
    for (i = m; i > 0; i = i - 1, inout = inout + dist, W = W + 3) {
          fftw_real tmp1, tmp25, tmp6, tmp24;
          fftw_real tmp12, tmp20, tmp17, tmp21;
          ASSERT_ALIGNED_DOUBLE;
          tmp1 = c_re(inout[0]); tmp25 = c_im(inout[0]);
     { fftw_real tmp3, tmp5, tmp2, tmp4;
       ASSERT_ALIGNED_DOUBLE;
       tmp3 = c_re(inout[2 * iostride]); tmp5 = c_im(inout[2 * iostride]);
       tmp2 = c_re(W[1]); tmp4 = c_im(W[1]);
       tmp6 = (tmp2 * tmp3) - (tmp4 * tmp5);
       tmp24 = (tmp4 * tmp3) + (tmp2 * tmp5); }
     { fftw_real tmp9, tmp11, tmp8, tmp10;
       ASSERT_ALIGNED_DOUBLE;
       tmp9 = c_re(inout[iostride]); tmp11 = c_im(inout[iostride]);
       tmp8 = c_re(W[0]); tmp10 = c_im(W[0]);
       tmp12 = (tmp8 * tmp9) - (tmp10 * tmp11);
       tmp20 = (tmp10 * tmp9) + (tmp8 * tmp11); }
     { fftw_real tmp14, tmp16, tmp13, tmp15;
       ASSERT_ALIGNED_DOUBLE;
       tmp14 = c_re(inout[3 * iostride]); tmp16 = c_im(inout[3 * iostride]);
       tmp13 = c_re(W[2]); tmp15 = c_im(W[2]);
       tmp17 = (tmp13 * tmp14) - (tmp15 * tmp16);
       tmp21 = (tmp15 * tmp14) + (tmp13 * tmp16); }
     { fftw_real tmp7, tmp18, tmp27, tmp28;
       ASSERT_ALIGNED_DOUBLE;
       tmp7 = tmp1 + tmp6; tmp18 = tmp12 + tmp17;
       c_re(inout[2 * iostride]) = tmp7 - tmp18;
       c_re(inout[0]) = tmp7 + tmp18;
       tmp27 = tmp25 - tmp24; tmp28 = tmp12 - tmp17;
       c_im(inout[iostride]) = tmp27 - tmp28;
       c_im(inout[3 * iostride]) = tmp28 + tmp27; }
     { fftw_real tmp23, tmp26, tmp19, tmp22;
       ASSERT_ALIGNED_DOUBLE;
       tmp23 = tmp20 + tmp21; tmp26 = tmp24 + tmp25;
       c_im(inout[0]) = tmp23 + tmp26;
       c_im(inout[2 * iostride]) = tmp26 - tmp23;
       tmp19 = tmp1 - tmp6; tmp22 = tmp20 - tmp21;
       c_re(inout[3 * iostride]) = tmp19 - tmp22;
       c_re(inout[iostride]) = tmp19 + tmp22; }
     }
}
```

```
typedef struct {
     int is_in_place; /* 1 if for in-place FFTs, 0 otherwise */
     int rank; /*
                * the rank (number of dimensions) of the
                * array to be FFTed
                */
     int *n; /*
              * the dimensions of the array to the
              * FFTed
              */
     fftw_direction dir;
     int *n_before; /*
                     * n_before[i] = product of n[j] for j < i</pre>
                     */
     int *n_after; /* n_after[i] = product of n[j] for j > i */
     fftw_plan *plans; /* 1d fftw plans for each dimension */
     int nbuffers, nwork;
     fftw_complex *work; /*
                          * work array big enough to hold
                          * nbuffers+1 of the largest dimension
                          * (has nwork elements)
                          */
} fftwnd_data;
```

typedef fftwnd_data *fftwnd_plan;

Figure 3.13: Multi-dimensional FFTW plan

are similar to those in function *fftw_create_plan_specific*. Multi-dimensional plans are created by calling the one-dimensional planner for each dimension. The resulting one-dimensional plans are inserted into the multi-dimensional plan.

3.4.3 Computing the Multi-dimensional DFT

The function *fftwnd* computes one or more multi-dimensional DFTs. The function is defined as following. The arguments are similar to those in function *fftw*.

FFTW applies the row-column algorithm to compute multi-dimensional DFTs (Section 2.1.4). More precisely, it takes the one-dimensional plan of each dimension and uses fftw to compute the DFT in each dimension. The one-dimensional DFTs are computed multiple times at a stride.

Chapter 4: Extending FFTW to Compute the Dimensionless FFT

FFTW uses the row-column algorithm to compute multi-dimensional DFTs. This chapter explains the changes that were made to FFTW so that it can use the dimensionless FFT to compute multi-dimensional DFTs.

In order to support the dimensionless FFT theorem (Theorem 2) in FFTW, the following modifications were made.

1. FFTW plan data structures

In order to extend the plan to multi-dimensional DFTs, the dimension, transform sizes in each dimension, and generalized twiddle factors must be incorporated.

2. Recursive planner

For a given multi-dimensional DFT, the recursive calls used by the planner involve multi-dimensional DFTs. The planner was modified to determine the necessary transforms for each possible split. Since there are many possible multi-dimensional DFTs (2^{n-1} possibilities of size 2^n) and only a small subset is used for a particular transform, a hash table was used to store "optimal" plans. In the one-dimensional case this is not necessary since there is only one transform for a given size.

3. Executor

The executor was modified to support the generalized stride permutation that occurs in the dimensionless FFT.

4. Codelet library

A library of multi-dimensional DFTs are required to support the base cases used by the dimensionless FFT. In the remainder of this chapter, each of these changes is elaborated except the generation of the multi-dimensional codelet library, which is discussed in the next chapter.

4.1 The Recursive Evaluation of the Dimensionless FFT

FFTW incorporates the initial stride permutation in the Cooley-Tukey factorization into the addressing, used to access the input vectors. In recursive calls to the executor, the necessary re-addressing is obtained simply by multiplying the stride and the stride information is passed to recursive calls as an extra parameters to the executor. In order to incorporate the initial permutation used by the dimensionless FFT into addressing calculations, a block parameter in addition to the stride parameter is required. This section shows how to recursively update the block and stride information used by the dimensionless FFT. After deriving the necessary addressing operations, a program is presented to illustrate how this is implemented when computing the dimensionless FFT. In the following section this is incorporated into FFTW.

4.1.1 Generalized Stride Permutations

The following example illustrates how strides are combined in recursive calls in the one-dimensional DFT. Assume

$$N = R_1 S_1 = R_1 (R_2 S_2), (4.1)$$

and the Cooley-Tukey factorization is repeatedly applied to obtain

$$y = F_N x$$

= $(F_{R_1} \otimes I_{S_1}) T_{S_1}^N (I_{R_1} \otimes F_{S_1}) L_{R_1}^N x$
= $(F_{R_1} \otimes I_{S_1}) T_{S_1}^N (I_{R_1} \otimes ((F_{R_2} \otimes I_{S_2}) T_{S_2}^{S_1} (I_{R_2} \otimes F_{S_2}) L_{R_2}^{S_1})) L_{R_1}^N x.$ (4.2)

Equation 4.2 first permutes the input vector x with the stride permutation matrix $L_{R_1}^N$ to produce a temporary vector

$$t_1 = L_{R_1}^N x = \begin{pmatrix} x(0:R_1:N-1)^T \\ x(1:R_1:N-1)^T \\ \vdots \\ x(R_1-1:R_1:N-1)^T \end{pmatrix}$$

Thereafter, t_1 contains R_1 blocks of size S_1 , each accessed at stride R_1 . The *i*th block starts at address *i*. The recursive application of F_{S_1} is applied to each of these blocks. In the recursive call, the input block is permuted with the stride permutation $L_{R_2}^{S_1}$ to produce

$$t_{2} = L_{R_{2}}^{S_{1}} x(0:R_{1}:N-1)^{T}$$

$$= \begin{pmatrix} x(0:R_{1}R_{2}:N-1)^{T} \\ x(R_{1}:R_{1}R_{2}:N-1)^{T} \\ \vdots \\ x(R_{1}(R_{2}-1):R_{1}R_{2}:N-1)^{T} \end{pmatrix}$$

The elements of t_2 are accessed with a combined stride equal to R_1R_2 . Additional recursive calls access its data at stride equal to the product of all of the combined strides. FFTW uses this fact in the executor by passing the current stride to each recursive call and multiplying the input stride by the stride required by the stride permutation at the current level in the factorization tree.

If the same recursive factorization of N is used to compute a multi-dimensional DFT \mathcal{F}_N of size N, using the dimensionless FFT, a similar recursive address computation is possible with three parameters: a **base address**, a **stride** and a **block**. Let x be the input array and $t = Px = (I \otimes L \otimes I)x$, with the three parameters, the index correspondence is

$$t(base + i) = x(base + \lfloor i/block \rfloor \times stride + i \mod block).$$

$$(4.3)$$

The derivation of the parameters is shown in the following theorem.

Theorem 3 (Generalized Stride Argument) Let

 $exec(x,y,N,[n_1,\cdots,n_t],base,stride,block)$ be a recursive function, based on the dimensionless FFT, to compute $y(base) = \mathcal{F}_N x(base,stride,block)$, where

$$\begin{aligned} \mathcal{F}_N &= F_{n_1} \otimes \dots \otimes F_{n_t}, \\ N &= n_1 \times \dots \times n_t = n_1 \times block, \\ & \left(\begin{array}{c} x(base:1:base+block-1) \\ x(base+stride:1:base+stride+block-1) \\ & \vdots \\ x(base+(n_1-1)*stride:1: \\ base+(n_1-1)*stride+block-1) \end{array} \right) \\ y(base) &= y(base:1:base+N-1). \end{aligned}$$

Let $1 \leq l \leq t$ with $n_l = ab$, $N_1 = n_1 \cdots n_{l-1}$, $\overline{N_1} = n_{l+1} \cdots n_t$, and $R = N_1 a$ and $S = b\overline{N_1}$, and assume the dimensionless factorization

$$\mathcal{F}_N = (\mathcal{F}_R \otimes I_S)(I_{N_1} \otimes T_b^{n_l} \otimes I_{\overline{N_1}})(I_R \otimes \mathcal{F}_S)(I_{N_1} \otimes L_a^{n_l} \otimes I_{\overline{N_1}})$$

is used to compute $y(base) = \mathcal{F}_N x(base, stride, block)$, where

$$\mathcal{F}_R = F_{n_1} \otimes \cdots \otimes F_{n_{l-1}} \otimes F_a,$$
$$\mathcal{F}_S = F_b \otimes F_{n_{l+1}} \otimes \cdots \otimes F_{n_t}.$$

Then the recursive calls to compute the R copies of \mathcal{F}_S using the dimensionless FFT accesses sub-vectors of x determined by base', stride' and block'. The values base', stride' and block' are determined according to two cases.

1. l > 1

In this case, the *i*-th, $0 \le i < R$, call to $exec(x, y, S, [b, n_{l+1}, \cdots, n_t], base', stride', block')$ is used to compute

$$y(base') = \mathcal{F}_S x(base', stride', block')$$

,

with the values

$$stride' = a\overline{N_1},$$

$$block' = S/b = \overline{N_1},$$

$$base' = base + i_0 \times stride + i_1 \times n_l \times \overline{N_1} + i_2 \times \overline{N_1},$$
(4.4)

where $0 \le i_0 < n_1, \ 0 \le i_1 < N_1/n_1, \ 0 \le i_2 < a, \ and \ i = i_0 R/n_1 + i_1 a + i_2.$

2. l = 1

In this case $N_1 = 1$, $R = N_1 a = a$, and the *i*-th call to $exec(x, y, S, [b, n_{l+1}, \dots, n_t], base', stride', block')$ is used to compute

$$y(base') = \mathcal{F}_S x(base', stride', block')$$

with the values

$$stride' = a \times stride,$$

$$block' = S/b = \overline{N_1} = block,$$

$$base' = base + i \times stride,$$
(4.5)

where $0 \leq i < a$.

Proof: The input vector x(base, stride, block) = x, where $block = n_2 \cdots n_t$. Before applying $I_R \otimes F_S$, the input vector is permuted with $P = I_{N_1} \otimes L_a^{n_l} \otimes I_{\overline{N_1}}$. The result of applying the permutation P can be related to the *base*, *stride* and *block* parameters by considering two cases separately.

1. l > 1

In this case, the permutation P can be blocked into n_1 identical permutations

that act independently on the blocks of x of size determined by *stride* and *block*.

$$Px = (I_{N_{1}} \otimes L_{a}^{n_{l}} \otimes I_{\overline{N_{1}}})x$$

$$= ((I_{n_{1}} \otimes I_{N_{1}/n_{1}}) \otimes L_{a}^{n_{l}} \otimes I_{\overline{N_{1}}})x$$

$$= (I_{n_{1}} \otimes (I_{N_{1}/n_{1}} \otimes L_{a}^{n_{l}} \otimes I_{\overline{N_{1}}}))x$$

$$= \begin{pmatrix} I_{N_{1}/n_{1}} \otimes L_{a}^{n_{l}} \otimes I_{\overline{N_{1}}} & \\ & \ddots & \\ & & I_{N_{1}/n_{1}} \otimes L_{a}^{n_{l}} \otimes I_{\overline{N_{1}}} \end{pmatrix}$$

$$\begin{pmatrix} x(base : 1 : base + block - 1) \\ & \vdots \\ x(base + (n_{1} - 1)stride : 1 : base + (n_{1} - 1)stride + block - 1) \end{pmatrix}$$

The permutation $I_{N_1/n_1} \otimes L_a^{n_l} \otimes I_{\overline{N_1}}$ naturally segments x into chunks of size S for which the parameters *base'*, *stride'* and *block'* are easily determined.

$$Px = \begin{pmatrix} x(base : 1 : base + \overline{N_1} - 1) \\ x(base + a\overline{N_1} : 1 : base + a\overline{N_1} + \overline{N_1} - 1) \\ \vdots \\ x(base + (b - 1)a\overline{N_1} : 1 : base + (b - 1)a\overline{N_1} + \overline{N_1} - 1) \\ \begin{pmatrix} x(base + (b - 1)a\overline{N_1} : 1 : base + (b - 1)a\overline{N_1} + \overline{N_1} - 1) \\ \vdots \\ x(base + n_1\overline{N_1} : 1 : base + 2\overline{N_1} + 1) \\ \vdots \\ x(base + n_1\overline{N_1} : 1 : base + n_1\overline{N_1} + \overline{N_1} - 1) \\ \vdots \\ x(base + stride : 1 : base + stride + \overline{N_1} - 1) \\ \vdots \\ x(base + stride : 1 : base + stride + \overline{N_1} - 1) \\ \vdots \\ x(base + stride : 1 : base + stride + \overline{N_1} - 1) \\ \vdots \\ x(base + stride : 1 : base + stride + \overline{N_1} - 1) \\ \vdots \\ x(base + stride : 1 : base + stride + \overline{N_1} - 1) \\ \vdots \\ y(base + stride = 1 : base + stride + \overline{N_1} - 1) \\ \vdots \\ y(base + stride = 1 : base + stride + \overline{N_1} - 1) \\ \vdots \\ y(base + stride = 1 : base + stride + \overline{N_1} - 1) \\ \vdots \\ y(base + \overline{N_1} - 1) \\ y(base + \overline{N_1} - 1) \\ \vdots \\ y(base + \overline{N_1} - 1) \\ y(ba$$

Inside each chunk of size $S = b\overline{N_1}$, $stride' = a\overline{N_1}$ and $block' = \overline{N_1}$. The starting index for each of the R chunks of size S is indicated by R values of base'. Within one group of size block there are aN_1/n_1 chunks of size S. These chunks start at index $i_1n_l\overline{N_1} + i_2\overline{N_1}$, where $0 \le i_1 < N_1/n_1$ and $0 \le i_2 < a$. The base address increases by stride when going from one group of size block to the next. Since there are n_1 groups the base address are given by the equation in case 1 of the theorem.

2. l = 1

In this case $N_1 = 1$, $\overline{N_1} = block$, and $P = (L_a^{n_1} \otimes I_{\overline{N_1}})$. The permutation P permutes blocks of size $\overline{N_1}$.

$$Px = (L_a^{n_1} \otimes I_{\overline{N_1}})x$$

$$= (L_a^{n_1} \otimes I_{block})$$

$$\begin{pmatrix} x(base:1:base+block-1) \\ \vdots \\ x(base+(n_1-1)stride:1:base+(n_1-1)stride+block-1) \end{pmatrix}$$

$$= \begin{pmatrix} x(base+(n_1-1)stride:1:base+block-1) \\ x(base+a \times stride:1:base+block-1) \\ \vdots \\ x(base+(b-1)a \times stride:1:base+(b-1)a \times stride+block-1) \end{pmatrix}$$

$$\begin{pmatrix} x(base+(b-1)a \times stride:1:base+(b-1)a \times stride+block-1) \\ \vdots \\ \vdots \end{pmatrix}$$

The parameters for the R recursive calls to \mathcal{F}_S are determined by grouping the permuted blocks into chunks of size S. Inside each chunk of size S elements are obtained from blocks of size *block* accessed at *stride'* = $a \times stride$. The parameter *block'* = *block*. The R chunks of size S are accessed beginning at $base' = base + i \times stride$.

4.1.2 Sample Computation of the Recursive Dimensionless FFT

In this section, two sample factorizations are used to illustrate the input data access pattern described in Theorem 3.

The dimensionless FFT algorithm is applied, using the factorization $4096 = 8 \times 16 \times 32$, to two different multi-dimensional DFTs: $F_4 \otimes F_{16} \otimes F_{32} \otimes F_2$ and $F_4 \otimes F_{128} \otimes F_8$. In each example we illustrate how the permutations are combined and how the recursion parameters in Theorem 3 are set. The first example illustrate case 1 of Theorem 3 and the second example illustrates case 2.

1. $\mathcal{F}_{4096} = F_4 \otimes F_{16} \otimes F_{32} \otimes F_2$

In this case, the dimensionless Cooley-Tukey factorization is

$$y = \mathcal{F}_{4096}x$$

= $(\mathcal{F}_8 \otimes I_{512})D_1(I_8 \otimes \mathcal{F}_{512})P_1x$
= $(\mathcal{F}_8 \otimes I_{512})D_1(I_8 \otimes ((\mathcal{F}_{16} \otimes I_{32})D_2(I_{16} \otimes \mathcal{F}_{32})P_2))P_1x$
= $((F_4 \otimes F_2) \otimes I_{512})D_1(I_8 \otimes (((F_8 \otimes F_2) \otimes I_{32})D_2(I_{16} \otimes (F_{16} \otimes F_2))P_2))P_1x,$

where

$$P_1 = I_4 \otimes L_2^{16} \otimes I_{64},$$

$$D_1 = I_4 \otimes T_8^{16} \otimes I_{64},$$

$$P_2 = I_8 \otimes L_2^{32} \otimes I_2,$$

$$D_2 = I_8 \otimes T_{16}^{32} \otimes I_2,$$

and the corresponding parameters (see Theorem 3) are

$$l_1 = 2, N_1 = 4, \overline{N_1} = 64, n_{l_1} = 16, a_1 = 2, b_1 = 8,$$

 $l_2 = 2, N_2 = 8, \overline{N_2} = 2, n_{l_2} = 32, a_2 = 2, b_2 = 16.$

When applying this factorization to an input vector x, x is first permuted by P_1 , producing a temporary vector

$$t_{1} = P_{1}x = (I_{4} \otimes L_{2}^{16} \otimes I_{64})x$$

$$= \begin{pmatrix} L_{2}^{16} \otimes I_{64} & & \\ & L_{2}^{16} \otimes I_{64} & \\ & & L_{2}^{16} \otimes I_{64} & \\ & & & L_{2}^{16} \otimes I_{64} \end{pmatrix} \begin{pmatrix} x(0:1:1023) \\ x(1024:1:2047) \\ x(2048:1:3071) \\ x(3072:1:4095) \end{pmatrix}.$$
(4.6)

Each block $x(base_b : 1 : base_b + 1023)$, where $base_b = i_0 \times 1024, 0 \le i_0 < 4$, is permuted with $L_2^{16} \otimes I_{64}$. Therefore

$$t_{1}(base_{b}:1:base_{b}+1023) = (L_{2}^{16} \otimes I_{64})x(base_{b}:1:base_{b}+1023)$$

$$= (L_{2}^{16} \otimes I_{64}) \begin{pmatrix} x(base_{b}+64:1:base_{b}+127) \\ x(base_{b}+128:1:base_{b}+191) \\ x(base_{b}+192:1:base_{b}+255) \\ \vdots \\ x(base_{b}+896:1:base_{b}+959) \\ x(base_{b}+960:1:base_{b}+1023) \end{pmatrix}$$

$$= \begin{pmatrix} \begin{pmatrix} x(base_{b}:1:base_{b}+63) \\ x(base_{b}+896:1:base_{b}+191) \\ \vdots \\ x(base_{b}+896:1:base_{b}+191) \\ \vdots \\ x(base_{b}+64:1:base_{b}+192) \\ \vdots \\ x(base_{b}+960:1:base_{b}+127) \\ x(base_{b}+960:1:base_{b}+1023) \end{pmatrix}. (4.7)$$

The two halves in Equation 4.7 are two chunks of size $S_1 = 512$. Within each chunk of size 512, the stride argument is $a_1\overline{N_1} = 2 \times 64 = 128$. Since there are

four blocks of size 1024 in Equation 4.6, there are a total of $4 \times 2 = 8$ chunks of size $S_1 = 512$ in t_1 , and each chunk can be represented by x(base, 128, 64), where $base = i_0 \times 1024 + i_1 \times 64$, $0 \le i_0 < 4$, $0 \le i_1 < 2$. The executor uses a recursive call to apply \mathcal{F}_{512} to these chunks. The *i*-th recursive call, where $i = 2i_0 + i_1$, takes the chunk starting at $base = i_0 \times 1024 + i_1 \times 64$.

In each recursive call, the input vector is x(base, 128, 64). When applying the given factorization, in the recursive call, the executor permutes the input vector with P_2 , which produces a temporary vector

$$t_{2} = P_{2}x(base, 128, 64)$$

$$= (I_{8} \otimes L_{2}^{32} \otimes I_{2}) \begin{pmatrix} x(base : 1 : base + 63) \\ x(base + 128 : 1 : base + 191) \\ \vdots \\ x(base + 896 : 1 : base + 959) \end{pmatrix}.$$
(4.8)

In Equation 4.8, each block starting at $base'_b = base + i_0 \times 128, 0 \le i_0 < 8$ is permuted with $L_2^{32} \otimes I_2$. The permuted result for each block is

$$(L_{2}^{32} \otimes I_{2})x(base'_{b}:1:base'_{b}+63) = \begin{pmatrix} x(base'_{b}:1:base'_{b}+1) \\ x(base'_{b}+4:1:base'_{b}+5) \\ \vdots \\ x(base'_{b}+60:1:base'_{b}+61) \end{pmatrix} \\ \begin{pmatrix} x(base'_{b}+60:1:base'_{b}+61) \\ x(base'_{b}+6:1:base'_{b}+3) \\ x(base'_{b}+6:1:base'_{b}+7) \\ \vdots \\ x(base'_{b}+62:1:base'_{b}+63) \end{pmatrix} \end{pmatrix}$$

The two halves in the above equation are chunks of size $S_2 = 32$. Within each chunk, the stride argument is $a_2\overline{N_2} = 2 \times 2 = 4$. There are $8 \times 2 = 16$ such chunks in t_2 , each of which starts at $base + i_0 \times 128 + i_2 \times 2$. The executor uses a no-twiddle codelet to apply \mathcal{F}_{32} to each of the resulting chunks of size 32.

2. $\mathcal{F}_{4096} = F_4 \otimes F_{128} \otimes F_8$

In this case, the dimensionless Cooley-Tukey factorization is

$$y = \mathcal{F}_{4096}x$$

= $(\mathcal{F}_8 \otimes I_{512})D_1(I_8 \otimes \mathcal{F}_{512})P_1x$
= $(\mathcal{F}_8 \otimes I_{512})D_1(I_8 \otimes ((\mathcal{F}_{16} \otimes I_{32})D_2(I_{16} \otimes \mathcal{F}_{32})P_2))P_1x$
= $((F_4 \otimes F_2) \otimes I_{512})D_1(I_8 \otimes ((F_{16} \otimes I_{32})D_2(I_{16} \otimes (F_4 \otimes F_8))P_2))P_1x$

where

$$P_{1} = I_{4} \otimes L_{2}^{128} \otimes I_{8},$$

$$D_{1} = I_{4} \otimes T_{64}^{128} \otimes I_{8},$$

$$P_{2} = L_{16}^{64} \otimes I_{32},$$

$$D_{2} = T_{4}^{64} \otimes I_{32}.$$

The corresponding parameters (in Theorem 3) are

$$l_1 = 2, N_1 = 4, a_1 = 2, b_1 = 64, \overline{N_1} = 8$$

 $l_2 = 1, N_2 = 1, a_2 = 16, b_2 = 4, \overline{N_2} = 8$

The computation starts from permuting input vector x with P_1 .

$$t_{1} = P_{1}x = (I_{4} \otimes L_{2}^{128} \otimes I_{8})x$$

$$= \begin{pmatrix} L_{2}^{128} \otimes I_{8} & & \\ & L_{2}^{128} \otimes I_{8} & \\ & & L_{2}^{128} \otimes I_{8} & \\ & & & L_{2}^{128} \otimes I_{8} & \\ & & & & L_{2}^{128} \otimes I_{8} \end{pmatrix} \begin{pmatrix} x(0:1:1023) \\ x(1024:1:2047) \\ x(2048:1:3071) \\ x(3072:1:4095) \end{pmatrix}$$

$$(4.9)$$

Each block $x(base_b : 1 : base_b + 1023)$, where $base_b = i_0 \times 1024, 0 \le i_0 < 4$, is

permuted with $L_2^{128} \otimes I_8$.

$$t_{1}(base_{b}: 1 : base_{b} + 1023) = (L_{2}^{128} \otimes I_{8})x(base_{b}: 1 : base_{b} + 1023)$$

$$= (L_{2}^{128} \otimes I_{8}) \begin{pmatrix} x(base_{b}: 1 : base_{b} + 7) \\ x(base_{b} + 8 : 1 : base_{b} + 15) \\ \vdots \\ x(base_{b} + 1016 : 1 : base_{b} + 1023) \end{pmatrix}$$

$$= \begin{pmatrix} x(base_{b}: 1 : base_{b} + 7) \\ x(base_{b} + 16 : 1 : base_{b} + 23) \\ \vdots \\ x(base_{b} + 1008 : 1 : base_{b} + 1015) \\ \vdots \\ x(base_{b} + 8 : 1 : base_{b} + 15) \\ x(base_{b} + 24 : 1 : base_{b} + 31) \\ \vdots \\ x(base_{b} + 1016 : 1 : base_{b} + 1023) \end{pmatrix}.$$

$$(4.10)$$

The two halves in Equation 4.10 are chunks of size $S_1 = 512$. Within each chunk, the stride argument is $a_1\overline{N_1} = 2 \times 8 = 16$. There are a total of $4 \times 2 = 8$ such chunks. The executor makes recursive calls to apply \mathcal{F}_{512} to these chunks. The *i*-th recursive call, where $i = 2i_0 + i_1$, takes the block starting at *base* = $i_0 \times 1024 + i_1 \times 8$, $0 \le i_0 < 4$ and $0 \le i_1 < 2$.

For each chunk, the executor permutes the input vector with P_2 .

$$t_{2} = P_{2}x(base, 16, 8)$$

$$= (L_{16}^{64} \otimes I_{8}) \begin{pmatrix} x(base : 1 : base + 7) \\ x(base + 16 : 1 : base + 23) \\ \vdots \\ x(base + 1008 : 1 : base + 1015) \end{pmatrix}$$

$$= \begin{pmatrix} x(base : 1 : base + 7) \\ x(base + 256 : 1 : base + 7) \\ x(base + 512 : 1 : base + 519) \\ x(base + 768 : 1 : base + 519) \\ x(base + 768 : 1 : base + 775) \end{pmatrix}$$

$$= \begin{pmatrix} x(base + 16 : 1 : base + 775) \\ \vdots \\ x(base + 16 : 1 : base + 23) \\ \vdots \\ x(base + 496 : 1 : base + 247) \\ x(base + 496 : 1 : base + 503) \\ x(base + 752 : 1 : base + 759) \\ x(base + 1008 : 1 : base + 1015) \end{pmatrix}$$

$$(4.11)$$

The size of each chunk in Equation 4.11 is $S_2 = 32$. Within each chunk, $stride' = a_1a_2\overline{N_2} = 2 \times 16 \times 8 = 256$, block' = 8 and $base' = base + i \times 16, 0 \le i < 16$. Each chunk of size 32 can be represented as x(base', 256, 8), and the executor uses a no-twiddle codelet to apply \mathcal{F}_{32} to each of the chunks.

4.1.3 Dimensionless Executor

The program in Figure 4.1 implements the dimensionless FFT using a recursive executor with the addressing parameters from Theorem 3. It differs from the program in Figure 2.3 in that the permutations are implemented using addressing given by

Transform	$ y_1 - y_2 _{\infty}$
$F_2 \otimes F_{512}$	2.67×10^{-18}
$F_4 \otimes F_{256}$	1.47×10^{-18}
$F_8 \otimes F_{128}$	1.16×10^{-18}
$F_{16}\otimes F_{64}$	1.04×10^{-18}
$F_{32}\otimes F_{32}$	3.82×10^{-18}
$F_{64}\otimes F_{16}$	1.99×10^{-18}
$F_{128}\otimes F_8$	1.50×10^{-18}
$F_{256} \otimes F_4$	2.17×10^{-18}
$F_{512}\otimes F_2$	4.11×10^{-18}
$F_2 \otimes F_2 \otimes F_{256}$	1.47×10^{-18}
$F_2 \otimes F_8 \otimes F_{64}$	1.04×10^{-18}
$F_4 \otimes F_2 \otimes F_{128}$	1.16×10^{-18}
$F_{32} \otimes F_4 \otimes F_8$	3.82×10^{-18}
$F_2 \otimes F_{128} \otimes F_4$	3.44×10^{-19}
$F_2 \otimes \cdots \otimes F_2$	2.32×10^{-21}

Table 4.1: The precision between the simulator (Figure 4.1) and FFTW

stride and block parameters. The purpose of this program is to validate Theorem 3 and to prepare for the modification to FFTW needed to support the dimensionless FFT. Table 4.1 compares the results of several multi-dimensional DFTs computed using this implementation of the dimensionless FFT and the row-column algorithm implemented in FFTW. All transforms are of size 1024 and are applied to the vector $x = (1 : 1024)^T$. Let y_1 be the result computed by FFTW and y_2 be the result computed by this implementation, Table 4.1 reports $||y_1 - y_2||_{\infty}$, the norm of the difference of the two results. Table 4.1 shows that the results computed by the two program are nearly identical.

```
void my_executor(fftw_complex *in, fftw_complex *out, int N, int *n,
                 int rank, int stride, int howmany, int *table)
{ /* Input: rank: a positive integer
   * n: an integer array of size rank
   * N: an integer, it should be the product of n's
   * stride: the stride argument
   * howmany: the size within each block
   * table: the split table */
  fftw_complex *tw; /* the twiddle factor */
  int R, S, l, split, N1, N2, a, b, stride1;
  int i,j,k;
  if (N<64){/* no_twiddle codelet simulator */
    my_fftwnd_notw_codelet(in, out, stride, 1, N, howmany, n, rank);
  }else{
    /* find the factorization in the split table */
    R = (int) pow(2,table[log2(N)]);
    S = N/R;
    /* Set the corresponding variables
     * 1 is the index of n's that to be factorized, n[1] = ab
     * N1 is the product of n[0] to n[1-1]
     * N2 is the product of n[l+1] to n[rank] */
    set_variables(n, rank, R, S, &a, &b, &N1, &N2, &l);
    /* store n[1] temporarily */
    split = n[1];
    /* determine the stride permutation argument */
    if (N2 >= howmany)
      stride1 = a*N2;
    else
      stride1 = a*stride;
   n[1]=b;
    for (i=0; i<N1; ++i)</pre>
      for (j=0; j<a; ++j){</pre>
        // recursive call the executor
        my_executor(in+i*N/N1+j*N2, out+i*N/N1+j*b*N2, N/N1/a,
                    n+split, rank-split,stride1, b, table);}
    n[l] = split; /*restore the dimension information */
    /* create the twiddle factor */
    tw = twiddleMD(N1, split, b, N2);
    /* update the dimension information */
    n[1] = a;
    /* twiddle codelet simulator */
    my_fftwnd_twiddle_codelet(out, tw, R, S, n, l+1);
    /* restore the dimension information */
   n[1] = split;
   free(tw);
  }
```

Figure 4.1: The Executor Simulator

4.2 Modifications to FFTW to Support the Dimensionless FFT

This section describes the modifications to FFTW that are needed to support the dimensionless FFT. Three modifications are required: 1) the plan must be modified to include dimension information and the values of the twiddle factorization need to be changed according to the dimension. 2) The planner must be modified to generate the additional information to the plan. 3) The executor must be modified to incorporate to the addressing in Theorem 3.

4.2.1 Modified Plan

The following FFTW data structures are modified to include additional dimension information.

1. *fftw_plan* data structure

Figure 4.2 shows the modified $fftw_plan$ data structure in dimensionless FFT. Compared with Figure 3.3, the argument **n** is replaced with the following three arguments.

- rank is a nonnegative integer equal to the dimensions of the transform.
- **n** is a pointer to an integer array of size **rank**, the size of the transform in each dimension.
- **N** is the size of a given transform, and $N = \prod_{i=0}^{rank-1} n_i$.
- 2. *fftw_plan_node* data structure

The modified *fftw_plan_node* is shown in Figure 4.3. Changes similar to those made to *fftw_plan* are made to *fftw_plan_node*. (Compare to Figure 3.4)

3. *fftw_codelet_desc* data structure

Figure 4.4 shows the updated *fftw_codelet_desc*. The modifications are similar to those made to *fftw_plan*.

```
struct fftw_plan_struct {
    int N;
    int rank;
    int *n;
    int refcnt;
    fftw_direction dir;
    int flags;
    int wisdom_signature;
    enum fftw_node_type wisdom_type;
    struct fftw_plan_struct *next;
    fftw_plan_node *root;
    double cost;
    fftw_recurse_kind recurse_kind;
    int vector_size;
};
```

```
typedef struct fftw_plan_struct *fftw_plan;
```

Figure 4.2: Updated *fftw_plan* data structure

4. *fftw_twiddle* data structure

The updated $fftw_twiddle$ data structure is shown in Figure 4.5. The two additional arguments **a** and **b** allow $fftw_twiddle$ to store twiddle factors of the form $I_a \otimes T_s^n \otimes I_b$, rather than simply T_s^n as was the case originally.

4.2.2 Modified Recursive Planner

The FFTW planner uses dynamic programming to determine the best possible split to use for a given size transform. Each possible split is timed using the best plans found for the smaller sizes. The best plans for smaller sizes are stored in a linked list that is created during the search.

For the dimensionless FFT the same search is performed; however, the smaller transforms used for a given split depend on the dimension of the transform. Moreover, since all multi-dimensional transform are supported there are many possibilities for the smaller transforms. For example, there are $\theta(K^2)$ two-dimensional transforms

```
typedef struct fftw_plan_node_struct {
     enum fftw_node_type type;
     union {
          /* nodes of type FFTW_NOTW */
          struct {
            int size;
            int rank;
            int *n;
            fftw_notw_codelet *codelet;
            const fftw_codelet_desc *codelet_desc;
          } notw;
          /* nodes of type FFTW_TWIDDLE */
          struct {
            int size;
            int rank;
            int *n;
            int n1,n2,a,b;
            fftw_twiddle_codelet *codelet;
            fftw_twiddle *tw;
            struct fftw_plan_node_struct *recurse;
            const fftw_codelet_desc *codelet_desc;
          } twiddle;
     } nodeu;
     int refcnt;
```

} fftw_plan_node;

Figure 4.3: Updated *fftw_plan_node* data structure

```
typedef struct {
     const char *name;
                              /* name of the codelet */
                               /* pointer to the codelet itself */
    void (*codelet) ();
     int size;
                               /* size of the codelet */
    fftw_direction dir;
                               /* direction */
     enum fftw_node_type type; /* TWIDDLE or NO_TWIDDLE */
     int signature;
                               /* unique id */
     int ntwiddle;
                               /* number of twiddle factors */
     const int *twiddle_order; /*
                                 * array that determines the order
                                 * in which the codelet expects
                                 * the twiddle factors
                                 */
                                /* the dimensions */
     int *n;
                               /* rank, number of dimensions */
     int rank;
```

```
} fftw_codelet_desc;
```

Figure 4.4: Updated *fftw_codelet_desc* data structure

```
typedef struct fftw_twiddle_struct{
    int n;
    const fftw_codelet_desc *cdesc;
    fftw_complex *twarray;
    struct fftw_twiddle_struct *next;
    int refcnt;
    int a,b; /*eye(a) tensor twiddle(n,s) tensor eye(b) */
} fftw_twiddle;
```

and $\theta(K^3)$ three-dimensional transforms of size 2^K .

The planner was modified to include dimension information and the logic necessary to determine which smaller multi-dimensional transforms are required for each split. Furthermore a hash table was used to store the optimal plans formed for smaller sizes. The following functions were modified to support these changes.

- fftw_create_plan
- *fftw_create_plan_specific*
- planner
- $fftw_lookup$
- planner_wisdom
- planner_normal
- fftw_wisdom_lookup
- $\bullet ~\textit{fftw_wisdom_add}$
- $fftw_make_node_notw$
- fftw_make_node_twiddle
- $\bullet \ ftw_make_plan$
- *fftw_complete_twiddle*
- fftw_measure_runtime
- $\bullet \ \textit{fftw_compute_cost}$
- run_plan_hooks
- $\bullet ~ \textit{fftw_create_twiddle}$

 $\bullet \ \textit{fftw_compute_twiddle}$

4.2.3 Modified Recursive Executor

The executor was modified to incorporate the addressing required by Theorem 3. In particular an extra parameter is required to pass the block information for Theorem 3. The initial call used to compute $y = (F_{n_1} \otimes \cdots \otimes F_{n_t})x$ must set stride = block = $n_2 \cdots n_t$ to be satisfied. Furthermore, logic was added to determine the inputs for the recursive calls from the dimensions of the transform. The following functions were modified to support these changes.

- fftw
- fftw_executor_simple
- executor_many
Chapter 5: Codelets for the Dimensionless FFT

A collection of multi-dimensional DFT codelets are required for the base case of the recursive implementation of the dimensionless FFT. It is necessary to have codelets for all multi-dimensional DFTs of each base case size. Since there are 2^{n-1} multi-dimensional DFTs of size $N = 2^n$, the number of codelets needed for base cases up to size $B = 2^b = \sum_{i=1}^b 2^{i-1} = 2^b - 1$. For codelets up to size 64, 63 codelets are needed. Moreover, for each implementation platform, the codelets must be implemented and optimized. Thus, tools are needed for automatically implementing and optimizing the codelets.

FFTW provides a codelet generator for one-dimensional DFTs [9]. Their codelet generator takes an input integer N and produces optimized unrolled code to compute the FFT of size N. The codelet generator selects one of several possible FFT algorithms depending on the size and generates an expression DAG (directed acyclic graph) corresponding to the algorithm. The DAG is then passed to a simplifier that performs various transformations such as common sub-expression elimination. The simplified DAG is passed to an FFT specialized scheduler, which improves the locality, and the schedule is unparsed to produce C code.

The FFTW codelet generator is currently restricted to one-dimensional DFTs, and can not be used to build the multi-dimensional codelets library.

An alternative approach to generate optimized DFT code is provided by the SPI-RAL system [11]. Unlike the FFTW codelet generator, there is a language for describing algorithms and a compiler that translates the algorithms to code. This enables SPIRAL to generate code for a wide variety of transforms in addition to the DFT. In particular it can be used to generate all of the multi-dimensional DFTs that are required for the dimensionless FFT.

Because of this generality, SPIRAL was chosen to build the codelet library. This

chapter first gives an overview of the SPIRAL system [11], and then shows how to use SPIRAL to build a multi-dimensional codelet library.

5.1 Overview of the SPIRAL System

The following summary is take from [11], where additional details can be found.

SPIRAL is a system for automatically implementing and optimizing fast signal transforms. Given a transform of a specified size, SPIRAL produces an optimized implementation for a given computer platform. Algorithms are described by mathematical formulas, represented in the signal processing language (SPL), and SPIRAL uses a formula generator to generate alternative algorithm for the specified transform. Formulas are translated into programs, and a search engine is used to search for the fastest implementation as measured by runtime on a given machine. Using runtime as a performance measure allows SPIRAL to adapt the generated code to a specific machine. Given a signal transform, the formula generator produces an algorithm, encoded as an SPL formula, to compute the transform. The formula translator translates the SPL formula into a C or Fortran program to obtain a specific implementation. The runtime is fed back to the search engine, which controls the formula generator and translator to search for a fast formula and implementation. Figure 5.1 (taken from the SPIRAL website www.spiral.net) shows the architecture and flow of the SPIRAL system.

This section gives an overview of the formula generator, formula translator and search engine. It shows how SPIRAL implements and optimizes a given transform on a specific machine.

5.1.1 Formula Generator

A linear discrete signal transform can be represented by a transform matrix M. Computing the transform is equivalent to computing the matrix-vector product y = Mx,



Figure 5.1: The Process of SPIRAL

where x and y are vectors storing the input and output signals. Fast algorithms for computing a DSP transform represented by the transform matrix M can be viewed as a factorization of M into a product of highly structured sparse matrices.

In SPIRAL, algorithms are represented by **formulas** containing products of structured sparse matrices. Transforms are specified by parameterized symbols, called **non-terminals**, such as $\mathbf{DFT}_{\mathbf{N}}$, and algorithms are derived using **break-down rules**, that rewrite non-terminals in terms of other non-terminals and constructs in the SPL language. A sample rule corresponding to the Cooley-Tukey factorization is

$$\mathbf{DFT}_{\mathbf{N}} \to (\mathbf{DFT}_{\mathbf{R}} \otimes I_S) T_S^N (I_R \otimes \mathbf{DFT}_{\mathbf{S}}) L_R^N,$$
 (5.1)

where N = RS. The symbols on the right hand side can be other non-terminals (bold), or matrix operators and symbols such as L_R^N and T_S^N called **terminals** in the SPL language. Rewrite rules are applied recursively until there are no non-terminals. The resulting fully expanded formula represents an algorithm. In the base case, there are rules that replace non-terminals with symbols encoding the matrix corresponding to the specified transform. An example of such a rule is

$$\mathbf{DFT}_2 \to F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{5.2}$$

The derivation of an algorithm through a sequence of rules can be encoded in a tree called a **rule tree**. Since rules such as the rule in Equation 5.1 can be applied in many different ways (different factorizations of N), there are many different rule trees for a given non-terminal and hence it is possible to derive many different algorithms for a given transform. Figure 5.2 shows two possible rule trees for **DFT**₈. The



Figure 5.2: Ruletrees of formulas in Equation 5.3 and Equation 5.4

derivation for these trees are shown in Equation 5.3 and 5.4.

$$\mathbf{DFT}_{\mathbf{8}} \rightarrow (\mathbf{DFT}_{\mathbf{4}} \otimes I_{2})T_{2}^{8}(I_{4} \otimes \mathbf{DFT}_{2})L_{4}^{8}$$

$$\rightarrow (((\mathbf{DFT}_{\mathbf{2}} \otimes I_{2})T_{2}^{4}(I_{2} \otimes \mathbf{DFT}_{2})L_{2}^{4}) \otimes I_{2})T_{2}^{8}(I_{4} \otimes \mathbf{DFT}_{2})L_{4}^{8}$$

$$\rightarrow (((F_{2} \otimes I_{2})T_{2}^{4}(I_{2} \otimes F_{2})L_{2}^{4}) \otimes I_{2})T_{2}^{8}(I_{4} \otimes F_{2})L_{4}^{8}$$

$$\mathbf{DFT}_{\mathbf{8}} \rightarrow (\mathbf{DFT}_{\mathbf{2}} \otimes I_{4})T_{4}^{8}(I_{2} \otimes \mathbf{DFT}_{4})L_{2}^{8}$$

$$\rightarrow (\mathbf{DFT}_{\mathbf{2}} \otimes I_{4})T_{4}^{8}(I_{2} \otimes ((\mathbf{DFT}_{\mathbf{2}} \otimes I_{2})T_{2}^{4}(I_{2} \otimes \mathbf{DFT}_{2})L_{2}^{4}))L_{2}^{8}$$

$$\rightarrow (F_{2} \otimes I_{4})T_{4}^{8}(I_{2} \otimes (((F_{2}) \otimes I_{2})T_{2}^{4}(I_{2} \otimes F_{2})L_{2}^{4})L_{2}^{8}$$
(5.3)

After a rule tree is fully expanded and an algorithm is derived, the formula generator converts the rule tree to an SPL program, and passes it to the formula translator.

5.1.2 Formula Translator

SPL (Signal Processing Language) [10] is a language for representing structured matrix factorizations. The formulas in SPL are constructed from general matrices, parameterized matrices such as the identity matrix, and matrix operators such as matrix composition and the tensor product (Figure 5.3). An SPL program represents a formula corresponding to a factorization of a matrix M. For example, Figure 5.4 shows

```
(1) General matrices, such as
    (matrix ((a11 ... a1n) ... (am1 ... amn)))
    (diagonal (al1 ... ann))
(2) Parameterized matrices, such as
    (I N)
            ; identity matrix
    (F N)
            ; Fourier transform matrix
    (L N R)
            ; Stride permutation matrix
    (T N S)
            ; twiddle matrix
(3) Matrix operation, such as
    (compose A1 ... An)
                          ; matrix product
                         ; tensor product
    (tensor A1 ... An)
    (direct-sum A1 ... An) ; direct sum
```

Figure 5.3: Components of SPL

the SPL program corresponding to the algorithm shown in Equation 5.3, which is a factorization of F_8 . The formula can be interpreted as a program, which uses the structure specified by the factorization to compute the matrix-vector product y = Mx. The formula translator accepts an SPL program and produces a subroutine, in C or Fortran, for computing the corresponding matrix-vector product.

The formula translator first translates an SPL program into an abstract syntax tree (AST). Figure 5.5 shows the syntax tree for the SPL program in Figure 5.4. The AST is then translated to intermediate code, which is further optimized prior to generating the final code.

The translator inserts code for the parameterized matrices and combines the resulting code corresponding to the operators occurring in the formula:

1. (Compose A B)

In this case, the formula translator generates code to compute two matrix-vector products.

t = B x;

y = A t;

```
(compose
  (tensor
      (compose
      (tensor (F 2) (I 2))
      (T 4 2)
      (tensor (I 2) (F 2))
      (L 4 2)
    )
    (I 2))
  (T 8 2)
  (tensor (I 4)(F 2))
  (L 8 4)
)
```

Figure 5.4: SPL program for the algorithm in Equation 5.3



Figure 5.5: The AST corresponding for the SPL program in Figure 5.4

2. (tensor (I n) (A m))

The computation $I_n \otimes A_m$ performs the matrix-vector product for *n* consecutive blocks of size *m* of input vector *x*. Therefore, the formula translator generates the following code to compute it.

for i=0 to n-1
y(i*m:1:(i+1)*m-1) = A x(i*m:1:(i+1)*m-1);

3. (tensor (A m) (I n))

It is similar to the previous case with a stride permutation argument m. Therefore the corresponding code is:

```
for i=0 to n-1
    y(i:m:mn-1) = A x(i:m:mn-1);
```

4. (tensor A B)

Assume that A is an $n \times n$ matrix and B is an $m \times m$ matrix. Since $A \otimes B = (A \otimes I_m)(I_n \otimes B)$, the computation can be implemented using the previous two cases.

The resulting code is optimized through loop unrolling, intrinsic function evaluation, type transformation, and standard compiler optimization such as common subexpression elimination and construct folding. The details of the optimization are given in [10]. After the code is optimized, the formula translator outputs the subroutine in C or Fortran.

5.1.3 Search Engine

SPIRAL uses search to select the best algorithm for a given transform on a given computer architecture. Formulas are generated, translated to code, and the resulting code is executed and timed. The runtime is used to guide the generation of formulas in order to find a formula whose runtime is as fast as possible. Since there are too many algorithms (Table 3.1) to exhaustively generate all possibilities, various search strategies are used to find algorithms that are close to optimal. Search strategies supported by SPIRAL include exhaustive search, hill climbing, dynamic programming, and an evolutionary algorithm called STEER [11].

5.2 Building a Multi-dimensional Codelet Library

A library of multi-dimensional DFTs is built using the SPIRAL system. A program was written in SPIRAL to generate all of the necessary multi-dimensional DFTs, and the SPIRAL search engine was used to generate fast implementations. Note that in order to insert the generated code into the dimensionless FFT program, some post-processing was required.

5.2.1 Generating Multi-Dimensional DFTs

A multi-dimensional DFT $F_{N_1} \otimes \cdots \otimes F_{N_t}$ of size $N = N_1 \cdots N_t$, with $N_i = 2^{n_i}$ and $N = 2^n$, corresponds to a unique composition (ordered partition) $n_1 + \cdots + n_t$ of n. All possible multi-dimensional DFTs of size $N = 2^n$ can be obtained by generating all compositions of n. There is a one-to-one mapping from (n-1)-bit binary numbers onto the set of compositions of n.

The mapping can be obtained by listing n ones with a bit between each pair of consecutive ones. Consecutive ones separated by zero-bits are grouped, and bits set to one are used to separate the different groups. The composition is obtained by adding the ones in each group. All possible compositions of n are obtained by using the 2^{n-1} (n-1)-bit binary numbers. Figure 5.6 shows the 8 compositions of 4 and the mapping between them and the eight 3-bit binary numbers.

The function Int2Part(n,i) maps an integer $0 \le i < 2^{n-1}$ to the correspond-



Figure 5.6: Partitions of N=16

ing composition and the function *Part2MDFT* creates an SPL expression for the corresponding multi-dimensional DFT. For example, Int2Part(4,7) outputs the list [1, 1, 1, 1], and Part2MDFT([1,1,1,1]) outputs the SPL non-terminal *TensorSPL(Transform("DFT", 2), Transform("DFT", 2), Transform("DFT", 2), Transform("DFT", 2), Transform("DFT", 2), Transform("DFT", 2), Transform("DFT", 2), the multi-dimensional DFT F_2 \otimes F_2 \otimes F_2 \otimes F_2. SPL non-terminals for all multi-dimensional DFTs of size N = 2^n can be generated using these two functions.*

5.2.2 Finding Fast Implementations of Small Multi-dimensional DFTs

Fast implementations of all 63 multi-dimensional DFTs of size up to 64 were obtained using the functions from the previous section and SPIRAL. The resulting implementations were timed on the computer described in Section 6.1. Table 5.1 lists the runtime for all multi-dimensional DFTs of size 64.

5.2.3 Building the Codelet Library

The codelet library was built from the fastest implementations found by SPIRAL; however, the resulting code had to be modified to support the addressing required by the dimensionless FFT (see Section 4.1.1). Additional parameters also had to be inserted to support the required calling convention.

The following steps were used for each multi-dimensional DFT.

• Generate SPL for the algorithm found by SPIRAL's search engine.

For example, Figure 5.8 shows the SPL file (named as F2_2_2_2_spl) for $DFT_2 \otimes DFT_2 \otimes DFT_2 \otimes DFT_2$.

• Compile the SPL file using options to modify the addressing

Once the SPL files are output, they are compiled with the following command. With these compile flags, the SPL compiler adds two stride arguments, sx and

```
# Convert integer to ordered partition
# Input: n, i, positive integers
         0 <= i < 2^(n-1)
#
# Output: an ordered partition of n
Int2Part := function(n,i)
local N, count, iO, b, j;
  N := [];
  count := 1;
  i0 := i;
  for j in [1..n-1] do
    b := i0 mod 2;
    if b = 0 then
      count := count + 1;
    else
      Add(N, count);
      count := 1;
    fi;
    i0 := Int(i0/2);
  od;
  Add(N,count);
  return N;
end;
Part2MDFT := function(p)
local T;
  if Length(p) = 1 then
     return Transform("DFT", 2^p[1]);
  else
     T := List(p,n->Transform("DFT",2^n));
     return TensorSPL(T);
  fi;
end;
```

Figure 5.7: Function *Int2Part* and *Part2MDFT*

Dimension	64	2,32	4,16	2,2,16
Runtime	2384	2594	3147	3052
Dimension	8,8	2,4,8	4,2,8	2,2,2,8
Runtime	2384	2556	2289	2480
Dimension	16,4	2,8,4	4,4,4	2,2,4,4
Runtime	3033	2556	2632	2270
Dimension	8,2,4	$2,\!4,\!2,\!4$	4,2,2,4	2,2,2,2,4
Runtime	2480	2117	3242	2594
Dimension	32,2	2,16,2	4,8,2	2,2,8,2
Runtime	2480	2708	2937	2422
Dimension	8,4,2	$2,\!4,\!4,\!2$	4,2,4,2	2,2,2,4,2
Runtime	2422	2422	2918	2594
Dimension	16,2,2	$2,\!8,\!2,\!2$	4,4,2,2	$2,\!2,\!4,\!2,\!2$
Runtime	2632	2575	2041	2193
Dimension	8,2,2,2	2,4,2,2,2	4,2,2,2,2	2,2,2,2,2,2
Runtime	2613	2308	3185	2556

Table 5.1: Runtime for multi-dimensional DFTs of size 64, unit: nanosecond

- ; This file is generated from GAP by the
- ; function ExportSPL

(tensor

)

(F2) (F2) (F2) (F2) (F2)

```
/*
  -----+
! |
         Generated by SPL Compiler 3.29
                                           1 +-----
                                _____+
! Command-line options: -x language=c -o F2_2_2_2.c -x
 anystride -x codetype=real -x datatype=complex
!
*/
/* This file is generated from GAP by the*/
/* function ExportSPL*/
/*
! The SPL Program: (tensor (F 2)(tensor (F 2)(tensor (..)(..))))
! node size: 16 X 16
*/
#include <fftw.h>
void F2_2_2(x,y,sx,sy)
double *x, *y;
int sx;
int sy;
{
/* computation */
 f0 = x[0] - x[2*sx];
 .
 }
```

Figure 5.9: Computing $F_2 \otimes F_2 \otimes F_2 \otimes F_2$ with stride permutation

sy, to each implementation. The result is shown in Figure 5.9.

```
spl F2_2_2_2.spl -xlanguage=c -o F2_2_2_2.c -xanystride -xcodetype=real
-xdatatype=complex
```

After this step, two copies of the compiled C functions are used to generate the twiddle and no-twiddle codelets, respectively.

• Twiddle codelets

The following modifications were made to generate the twiddle codelets.

- Replace the argument $fftw_complex *y$ with $fftw_complex *w$ in the prototype. Although both arguments (y and w) are of the same type, y refers to the output while w refers to the twiddle factor.
- Remove the argument sy in the prototype, and replace all occurrences of sy with sx in the function.
- Add code to multiply the twiddle factor with the input vector before computing the DFT.
- Add the arguments *int m* and *int dist* in the function prototype. The argument *m* refers to the number of DFTs that are computed, and *dist* refers the initial address shift between the input/output arrays in each DFT. Then add a loop with *m* iterations to compute multiple DFTs.

A sample twiddle codelet is shown in Figure 5.10.

• no-twiddle codelets

The following modifications are made to generate the no-twiddle codelets.

- Add an argument *int block* to the function prototype. This argument refers the block size of permutation. (See Section 4.1.1)
- Replace each index k^*sx with $k/block^*sx+k\%block$.

A sample no-twiddle codelet is shown in Figure 5.11.

```
#include <fftw.h>
void ftw2_2_2_2(x,w,sx,m,dist)
double *w, *x;
 int sx;
 int m;
 int dist;
 {
 for (i=m; i>0; i=i-1, inout=inout+dist*2, w=w+dist*2){
     for (j=0; j<16; ++j){</pre>
       /* inout array multi-lies by the twiddle factor */
       tmp1 = inout[2*sx*j];
       tmp2 = inout[2*sx*j+1];
       tmp3 = w[2*j];
       tmp4 = w[2*j+1];
      inout[2*sx*j] = tmp1*tmp3 - tmp2*tmp4;
      inout[2*sx*j+1] = tmp2*tmp3 + tmp1*tmp4;
      }
  /* computation */
  f0 = inout[0] - inout[2*sx];
  •
  .
  .
 }
 }
```

Figure 5.10: Twiddle Codelets of $F_2 \otimes F_2 \otimes F_2 \otimes F_2$

```
#include <fftw.h>
void F2_2_2_2(x,y,sx,sy,b)
double *x, *y;
int sx;
int sy;
int b;
{
   /* computation */
   f0 = x[0] - x[2/block*sx+2%block];
   .
   .
   .
   .
   .
   .
}
```

Figure 5.11: No-twiddle Codelets of $F_2\otimes F_2\otimes F_2\otimes F_2$

Chapter 6: Performance

This chapter discusses an empirical study of the implementation of the dimensionless FFT presented in this thesis. A comparison is made with the row-column algorithm currently available in FFTW and it is shown that for some dimensions, the dimensionless FFT is faster and for others it is slower. All experiments were performed on a 550 MHz Pentium III processor with 16K L1 cache, 512K L2 cache and 512MB memory running Mandrake 9.0 Linux. Timings were performed with multi-dimensional DFTs of size 2^{20} . Each timing was performed five times and the average time was reported.

6.1 Performance of the Row-Column Algorithm

Recall that the motivation for using the dimensionless FFT was that for particular dimensions the row-column algorithm constrains the sizes of the recursive calls in the FFT and that these constraints may preclude the optimal breakdown strategy needed to improve locality and best utilize cache. The dimensionless FFT allows the same breakdown strategy to be used independent of dimension.

Figure 6.1 shows the performance of FFTW's row-column algorithm for different dimensions (all two-dimensional and selected three-dimensional DFTs were used - see Table 6.1). The figure shows that the runtime of the row-column algorithm varies dramatically as the dimension is changed. Furthermore, there are multidimensional DFTs where the row-column algorithm is substantially slower than the one-dimensional DFT of the same size.

6.2 Performance of the Dimensionless FFT

Figure 6.2 shows the ratio of the dimensionless FFT compared to the row-column algorithm in FFTW, using the same DFTs used in Figure 6.1. It shows that the



Figure 6.1: The runtime ratio of multi-dimensional DFT vs. one-dimensional DFT in FFTW

Transform	1	2	3	4	5	6
Dimension	2^{20}	$2, 2^{19}$	$2^2, 2^{18}$	$2^3, 2^{17}$	$2^4, 2^{16}$	$2^5, 2^{15}$
Transform	7	8	9	10	11	12
Dimension	$2^6, 2^{14}$	$2^7, 2^{13}$	$2^8, 2^{12}$	$2^9, 2^{11}$	$2^{10}, 2^{10}$	$2^{11}, 2^9$
Transform	13	14	15	16	17	18
Dimension	$2^{12}, 2^{8}$	$2^{13}, 2^7$	$2^{14}, 2^{6}$	$2^{15}, 2^5$	$2^{16}, 2^4$	$2^{17}, 2^3$
Transform	19	20	21	22	23	24
Dimension	$2^{18}, 2^2$	$2^{19}, 2$	$2^6, 2^6, 2^8$	$2^6, 2^7, 2^7$	$2^6, 2^8, 2^6$	$2^7, 2^6, 2^7$
Transform	25	26	27	28	29	
Dimension	$2^7, 2^7, 2^6$	$2^7, 2^8, 2^5$	$2^8, 2^6, 2^6$	$2^8, 2^7, 2^5$	$2^8, 2^8, 2^4$	

Table 6.1: Multi-dimensional DFTs that were tested in FFTW



Figure 6.2: The runtime ratio of dimensionless FFT to FFTW

dimensionless FFT is faster when computing the multi-dimensional DFTs whose runtime is substantially slower than the one-dimensional DFT using the row-column algorithm. However, the dimensionless FFT was not faster in all cases where the onedimensional FFT was faster than the row-column algorithm. A possible explanation for this is given by the fact that the one-dimensional FFT using the dimensionless infrastructure is about 1.5 times slower than FFTW's one-dimensional FFT. This suggests that additional overhead is introduced or the codelets generated by SPIRAL are slower. This is explained in the next section.

Unlike the row-column algorithm, the runtime for the dimensionless FFT should be independent of dimension (this is not quite true since there are fewer non-trivial multiplications in the twiddle codelets as the number of dimensions increases). Figure 6.3 shows that the runtimes for the dimensionless FFT do not vary as much as the row-column algorithm, and that the runtimes for multi-dimensional DFTs are less than the runtime for the one-dimensional DFT.

6.3 Performance Evaluation

In the previous section it was shown that the dimensionless FFT can outperform the row-column algorithm for some DFTs. However it did not provide the anticipated



Figure 6.3: The runtime ratio of multi-dimensional DFT to one-dimensional DFT in the dimensionless FFT

performance gain for some dimensions and a significant slowdown was seen for the one-dimensional DFT when computed using the dimensionless infrastructure. In this section a performance model is used to evaluate the cause for this discrepancy. It is shown that much of the degraded performance for the dimensionless FFT on the one-dimensional DFT, and other dimensions when expected gain was not obtained, is due to the performance of the codelets generated using SPIRAL. The SPIRAL codelets have similar performance [11] when unit stride is used; however, in larger FFTs when the twiddle codelets are called multiple times with non-unit stride, the performance is worse. This largely explains why the performance of the dimensionless FFT was not as good as expected, and shows that this is not an inherent limitation, but rather something that could be improved with better codelets.

6.3.1 Comparison of Codelets

In both FFTW and the dimensionless FFT, the executor calls a no-twiddle codelet multiple times to compute $y = (I_s \otimes F_n)L_s^{ns}x$ and $y = (I_s \otimes \mathcal{F}_n)Px$ respectively, where *n* is the size of the codelet, *s* is the number of calls, and *P* is a permutation. In both packages $s \leq 64$. In FFTW, the input vector is accessed at stride, while in the dimensionless FFT the input vector is accessed in a different way (See Section 4.1.1).

iterations	2	4	8	16	32	64
fn_2	6.6e-6	1.22e-5	2.22e-5	4.44e-5	8.70e-5	1.72e-4
fn_4	1.18e-5	2.26e-5	4.36e-5	8.60e-5	1.71e-4	3.55e-4
fn_8	2.26e-5	4.42e-5	8.66e-5	1.72e-4	3.76e-4	7.16e-4
fn_16	4.98e-5	9.18e-5	1.78e-4	3.63e-4	7.32e-4	1.42e-3
fn_32	9.92e-5	1.87e-4	3.71e-4	7.71e-4	1.53e-3	2.83e-3
fn_64	2.27e-4	4.20e-4	7.92e-4	1.49e-3	3.15e-3	5.80e-3

Table 6.2: The Runtime of FFTW's no-twiddle codelet in seconds

Table 6.3: The Runtime of dimensionless no-twiddle codelet in seconds

iterations	2 4		8 16		32	64
fn_2	6.8e-6	1.3e-5	2.44e-5	4.82e-5	9.70e-5	1.88e-4
fn_4	1.28e-5	2.4e-5	4.72e-5	9.26e-5	1.85e-4	3.95e-4
fn_8	2.6e-5	4.76e-5	9.34e-5	1.85e-4	4.24e-4	7.75e-4
fn_16	5.42e-5	1.09e-4	1.93e-4	3.96e-4	7.96e-4	1.63e-3
fn_32	1.17e-4	2.1e-4	4.13e-4	8.06e-4	1.55e-3	3.05e-3
fn_64	2.75e-4	5.13e-4	9.78e-4	1.89e-3	3.64e-3	7.41e-3

Therefore experiments were made to test the performance of FFTW's no-twiddle codelets and the no-twiddle codelets generated by SPIRAL. Table 6.2 shows the runtime of FFTW's no-twiddle codelets for different value of s. Table 6.3 shows the runtime of the one-dimensional no-twiddle codelets generated for the dimensionless FFT for same values of s as in Table 6.2. The comparison is summarized in Figure 6.4.

The same experiments were made on the twiddle codelets. In FFTW, the executor calls a twiddle codelet to compute $y = (F_n \otimes I_s)T_s^{ns}y$, where n is the codelet size. The twiddle codelet contains a loop to compute s iterations of F_n . For the dimensionless FFT, the executor uses twiddle codelets in the same way to compute $y = (\mathcal{F}_n \otimes I_s)Dy$,

iterations	2	4	8	16	32	64	128	256	512
ftw_2	2e-6	2e-6	2.6e-6	2e-6	3.6e-6	4.6e-6	2.22e-5	4.48e-5	1.08e-4
ftw_4	3e-6	2e-6	3e-6	4.2e-6	7.6e-6	2.5e-5	6.08e-5	1.31e-4	2.41e-4
ftw_8	3.6e-6	4.2e-6	5.2e-6	8e-6	2.84e-5	6.72e-5	1.39e-4	4.27e-4	8.51e-4
ftw_16	1.24e-5	1.58e-5	2.14e-5	4.14e-5	8.3e-5	1.69e-4	4.71e-4	9.65e-4	1.93e-3
ftw_32	3.4e-5	3.94e-5	6.32e-5	1.14e-4	2.11e-4	5.27e-4	1.14e-3	2.11e-3	4.38e-3
ftw_64	9.24e-5	1.26e-4	2.17e-4	3.71e-4	7.54e-4	1.46e-3	2.77e-3	7.07e-3	1.34e-2

Table 6.4: The Runtime of FFTW's twiddle codelet in seconds

Table 6.5: The Runtime of dimensionless twiddle codelet in seconds

iterations	2	4	8	16	32	64	128	256	512
ftw_2	2.2e-6	1.8e-6	2.4e-6	3e-6	3.6e-6	5.8e-6	2.42e-5	7.78e-5	1.23e-4
ftw_4	2.6e-6	3e-6	3.6e-6	5e-6	7.4e-6	2.52e-5	7.44e-5	1.55e-4	2.87e-4
ftw_8	3.6e-6	4.2e-6	6.2e-6	9.8e-6	2.98e-5	8.68e-5	1.67e-4	6e-4	1.3e-3
ftw_16	1.4e-5	1.5e-5	2e-5	4.54e-5	1.05e-4	2e-4	6.9e-4	1.47e-3	3e-3
ftw_32	3.38e-5	3.9e-5	7.08e-5	1.32e-4	2.49e-4	7.54e-4	1.57e-3	3.2e-3	6.6e-3
ftw_64	1.05e-4	1.54e-4	2.59e-4	4.93e-4	1.13e-3	2.18e-3	4.53e-3	1e-2	1.96e-2

where D is a generalized twiddle matrix. Table 6.4 and 6.5 show the performance of FFTW's twiddle codelets and the corresponding one-dimensional twiddle codelets used in the dimensionless FFT for different value of s. The twiddle codelets were timed under bigger iterations because there is no upper bound for s when calling the twiddle codelet. The comparison is given in Figure 6.5.

Both Figures 6.4 and 6.5 show that the one-dimensional codelets generated by SPIRAL are slower than FFTW's codelets in most cases. This explains the slowdown for the one-dimensional DFT when computed using the dimensionless FFT. Although SPIRAL generates code competitive to FFTW, the post-processing mentioned in Section 5.2.3 slows down the performance.



Figure 6.4: The ratio of the runtime of the dimensionless no-twiddle codelets to FFTW's no-twiddle codelets



Figure 6.5: The ratio of the runtime of the dimensionless twiddle codelets to FFTW's twiddle codelets

6.3.2 A Performance Model

This section gives a simple model to estimate the overall performance from a given plan and the performance of the codelets.

In FFTW, when a given plan factorizes N = RS, the executor will first make Rrecursive calls or call a no-twiddle codelet R times to compute the DFT of size S. It then calls S times a twiddle codelet of size R to compute $F_R s$ times. Let T(N)denote the time to compute a DFT of size N, $fn_{S,R}$ denote the runtime of a notwiddle codelet to compute $(I_R \otimes F_S)L_R^N$, and $ftw_{R,S}$ denote the time for a twiddle codelet to compute $(F_R \otimes I_S)T_S^N$. The runtime can be estimated by the following recurrence when N = RS

$$T(N) = \begin{cases} RT(S) + ftw_{R,S}, \\ fn_{S,R} + ftw_{R,S}, \text{ in the base case} \end{cases}$$
(6.1)

Figure 6.6 shows a plan returned by FFTW to compute a one-dimensional DFT of size 2^{20} . According to Equation 6.1, the runtime is estimated as

$$T(2^{20}) = 2^{2}T(2^{18}) + ftw_{2^{2},2^{18}}$$

$$= 2^{2}(2^{5}T(2^{13}) + ftw_{2^{5},2^{13}}) + ftw_{2^{2},2^{18}}$$

$$= 2^{7}T(2^{13}) + 2^{2}ftw_{2^{5},2^{13}} + ftw_{2^{2},2^{18}}$$

$$\vdots$$

$$= ftw_{4,2^{18}} + 4 \times ftw_{32,2^{13}} + 2^{7} \times ftw_{4,2^{11}}$$

$$2^{9} \times ftw_{4,2^{9}} + 2^{11} \times ftw_{16,32} + 2^{11} \times fn_{32,16}.$$
(6.2)

Equation 6.2 also can be used to estimate the runtime for the dimensionless FFT by substituting the runtime for the dimensionless codelets. Most of the runtimes that are needed to estimate the performance are given in Tables 6.2, 6.4, 6.3 and 6.5. Additional experiments were made to determine the other runtimes that are needed.



Figure 6.6: A FFTW plan of input size 2^{20}

Using Equation 6.2,

$$\begin{split} T_{FFTW} &= 0.382976 + 0.931264 = 1.31424 \text{ seconds}, \\ T_{dimless} &= 0.43008 + 1.198184 = 1.628264 \text{ seconds}, \end{split}$$

and

$$R_{ets} = \frac{T_{dimless}}{T_{FFTW}} = 1.2389.$$

Although the estimated runtimes differ from the experimental results, the runtime ratio is relatively close to the experimental result. This shows how the performance of codelets affects the overall performance.

Chapter 7: Conclusion

This thesis presented a divide and conquer algorithm for computing multi-dimensional DFTs that works independent of the dimensions. The benefit is that the decomposition is not constrained by the number of points in each dimension as is the case for the standard row-column algorithm. The dimensionless FFT was implemented by modifying the FFTW package for computing one-dimensional FFTs (only minor modifications were required). The resulting program improved the performance of some multi-dimensional DFTs as compared to FFTW's row-column algorithm. While the performance gain was not as great as expected, it was shown that additional performance gain is possible through the use of better multi-dimensional codelets.

Bibliography

- James W. Cooley and J.W.Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series", *Math. Comput.*, vol.19, pp.297-301, 1965
- [2] L. Auslander, J.R. Johnson, and R.W. Johnson, "Dimensionless Fast Fourier Transforms", Tech.Rep. DU-MCS-97-01, Drexel University, 1997, http://www.cs.drexel.edu
- [3] J.Johnson, R.W.Johnson, D.Rodriguez and R.Tolimieri, "A Methodology for Designing, Modifying and Implementing Fourier Transform Algorithms on Various Architectures", *Circuit, Systems and Signal Processing*, vol.9, no.4 pp.249-500, 1990
- [4] Charles Van Loan, Computational Frameworks for the Fast Fourier Transform, SIAM, Philadelphia, PA, 1992
- [5] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, Introduction to Algorithms, second edition, 2001
- [6] Matteo Frigo and Steven G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT", ICASSP Conference Proceedings, 1998, vol.3, pp.1381-1384
- [7] P. Kumhom, J. R. Johnson and P. Nagvajara, "Design, Optimization, and Implementation of a Universal FFT Processor", in *Proc. 13th IEEE International* ASIC/SOC Conference, Washington DC, September 2000
- [8] D. Mirković and S. L. Johnson, "Automatic Performance Tuning in the UHFFT Library", Proc. ICCS. 2001, vol. 2073, pp.71-80, Springer
- [9] Matteo Frigo, "A Fast Fourier Transform Compiler", Proc. 1999 ACM SIG-PLAN Conference on Programming Language Design and Implementation, Atlanta, May 1999

- [10] J. Xiong, J. Johnson, R. Johnson and D. Padua, "SPL: A Language and Compiler for DSP Algorithms", Proc. PLDI 2001, pp. 298-308
- [11] Markus Puschel, Bryan Singer, Jianxin Xiong, Jose M.F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Robert W. Johnson, "SPIRAL: A Generator for Platform-Adapted Libraries of Sigal Processing Algorithms" Journal of High Performance Computing and Applications, special issue on Automatic Performance Tuning, 18(1), 2004, pp. 21-45
- [12] William T. Vetterling, Saul A. Teukolsky, Brian P. Flannery, William H. Press, Numerical Recipes in C++: The Art of Scientific Computing, Cambridge University, February 2002, http://www.nr.com