Construction of a High-Performance FFT

Mathematics, Design, and Implementation Guide

Author: **Eric Postpischil (http://edp.org)**
Version: **2.1**
Date: **August 8, 2004**

The FFT algorithm for computing the DFT is well known and provides an O(n log n)-time implementation of the DFT. However, constructing a high-performance FFT implementation that executes at the best possible speed requires careful and efficient organization.

This paper describes the mathematical composition of an FFT, some overall design considerations for implementing high-performance FFTs, and specific considerations for implementing a high-performance FFT on an AltiVec processor.

Version	Date	Changes
2.0	August 25, 2003	First public release.
2.1	August 8, 2004	More rigorous definition of r. Minor edits.

Contents

Source Code Displays

1 Introduction

1.1 What Is an FFT?

This paper is intended for the engineer who wants to design and implement an FFT or to understand an existing implementation. It helps if you already know what an FFT is. However, it is not essential. This paper is about how to compute an FFT, not how to use it, and the computations are laid out in detail.

FFT stands for Fast Fourier Transform. It is an algorithm for performing a DFT. DFT stands for Discrete Fourier Transform. The DFT is a mathematical operation. You will find a definition for it in section 2.1.4. If you are unfamiliar with the DFT, you are probably a software engineer who has been asked to implement or maintain some FFT code for some signal processing applications. In that case, you can study the mathematics in this paper to understand the FFT structure or read any of the numerous books and web pages about the FFT, what it does, and how it is used.

Strictly, the FFT is a specific algorithm for performing fast DFTs on vectors whose lengths are powers of two. “FFT” is also used to describe other algorithms for performing DFTs on vectors of other lengths. This paper addresses only vector lengths that are powers of two. The basic algorithm used in this paper is described in section 2.3.

1.2 What Are We Going To Do?

This paper shows you how to design and implement a high-performance FFT, particularly on a computer processor with AltiVec technology. High-performance means executing an FFT not just in O(n log n) time but organizing the work for efficient execution so that an FFT can be performed in world-class time. The design illustrated in this document, if implemented well, can perform an FFT on a 1024-element vector in less than 9,400 CPU cycles on a Motorola PowerPC CPU 7400.

Section 2 analyzes the mathematical structure of the DFT, shows an FFT procedure, and proves the FFT procedure computes the DFT. The mathematics is developed explicitly. The advantage of this, aside from knowing our algorithm is correct, is that it makes it easier for us to reason about the algorithm. The effect is that other decisions later on—How do we generate the weights?—are easier because we can write a simple formula that shows what must be calculated. Also, this assists in demonstrating that the FFT algorithm is largely composed of simple parts, albeit connected in some complicated ways.

Section 3 converts the algorithm into simple C code and then shows how to reorganize the code for efficient execution. This method of showing how the code is developed is repeated in this paper for two reasons:

- Showing the development provides a better understanding of the design than showing a completed work, particularly since some parts of the complete design are intricate.

- Laying out the decisions separately makes them easier to change for other circumstances, such as a different target architecture.

Section 4 shows incremental design improvements and methods to implement them.

Section 5 discusses generating constants needed by the FFT routines.

Section 6 reorganizes the basic FFT loops for more efficient execution.

Section 7 shows how to design an FFT for efficient performance on long vectors that do not fit completely in cache memory at one time.

Section 8 adds support for the reverse DFT.

Section 9 completes the FFT, showing code to call the subroutines of earlier sections.

1.3 Target Architecture

The overall design described in this paper is suitable for implementation on a variety of computer architectures, because features like simplifying code structure, reducing memory use, and eliminating unneeded calculations are generally beneficial regardless of computer architecture. At certain points, choices will be made specifically for the family of PowerPC processors (from IBM and Motorola) using AltiVec technology (from Motorola).

AltiVec technology has several features of interest in implementing a high-performance FFT.

The floating-point instructions include a fused multiply-add operation that executes in the same time as a multiply or add operation. It is therefore advantageous to structure calculations to minimize multiply-add operations rather than merely minimizing multiply operations.

The architecture provides single-instruction multiple-data (SIMD) instructions that perform the same calculation on four sets of floating-point numbers at the same time. E.g., the calculation expressed by this C code:

```
for (i = 0; i < 4; ++i)
   d[i] = a[i] * b[i] + c[i];
```

can be computed by the single instruction “`vmaddfp d, a, b, c`”, provided that the array contents are in processor registers named `a`, `b`, `c`, and `d`.

Along with this multiple-data capability come:

- the ability to load and store data to and from memory in blocks and

- the restriction that memory access should be done on addresses with 16-byte alignment for best performance.

These features affect our design decisions by giving us incentive to group data in blocks of four floating-point numbers.

1.4 Target Processor

In section 7, particular characteristics of the Motorola PowerPC CPU 7400 will be used to illustrate design choices and construct the FFT. Relevant information about this CPU is in section 7.1.1.

The reader is expected to be familiar with cache operations, such as touches, streams, invalidates, and flushes.

1.5 Introductory Notes

1.5.1 Source Code Notation and Mathematical Notation

References to C and assembly language source code are marked with a fixed-width font, as in the assembly-language instruction `vmaddfp` or the C expression `1<<N-n[p]`.

The usual mathematical notation is used extensively in section 2 and sporadically throughout this paper. At times it is necessary to mix these two notations, to refer to the mathematical value that a certain software entity has. Italics denote mathematical variables, such as n, and distinguish them from software entities, such as `n`.

1.5.2 Complex Number Representation

The representations of complex numbers and arrays of complex numbers are not made explicit in much of this paper. Many of design features and criteria discussed are not sensitive to the choice of representations.

Common arrangements for arrays of complex numbers are:

- Have two arrays. One holds the real components of the complex numbers, and the other holds the imaginary components. This is called split or separated data.
- Have one array in which each element is a structure containing two floating-point numbers, one the real component and the other the imaginary component. This is called interleaved data.

In demonstration source code, the real or imaginary components of complex numbers are sometimes referred to in ways that, due to C semantics, suggest a certain representation. E.g., a reference to "`v.re[k]`" implies `v` is a structure containing a member `re` (and likely another member `im`) that is an array of floating-point numbers, thus suggesting separated data. Conversely, "`v[k].re`" implies an array of structures, thus suggesting interleaved data. The reader should understand that usually either arrangement is acceptable, with suitable changes in the source code, and I may switch back and forth between them to use the representation that is simpler in whatever feature is being discussed.

In discussing the data being transformed, the term “element” refers to an entire complex element, either as a whole or as essentially parallel operations on its real and imaginary components. When the individual components are relevant, the real and imaginary components are referred to explicitly.

1.5.3 Miscellaneous

“Low bit” and “low bits” refer to the least significant bits of a value. “High bit” and “high bits” refer to the most significant bits of a value. The values involved are typically bit fields smaller than whole processor registers or architectural words, so the most or least significant bits involved are those of the value and not necessarily of the whole word.

1.5.4 Bit-Reversal Permutation

It is well known that the FFT produces results in a permuted order, an order called a bit-reversal permutation. This is defined formally in sections 2.1.5 and 2.3. An introduction here may also be useful. An array **a'** containing 2^N elements is said to be the bit-reversal permutation of an array **a** also containing 2^N elements if:

> For each k and k' such that the N-bit binary notation for k (including leading zeroes) is the bit-by-bit reversal of the N-bit binary notation for k', $a_k = a'_{k'}$.

That is, each element a_k is found in **a'** by reversing the bits in the index k.

Note two properties of the bit-reversal:

- The bit-reversal of a number is symmetric; the bit-reversal of the bit-reversal is the original number. The same is true of the entire permutation; the bit-reversal permutation of a bit-reversal permutation is the identity permutation.
- The bit-reversal depends on N. For example, the bit-reversal of 11 considered as a 4-bit number (1011_2) is 13 (1101_2). The bit-reversal of 11 considered as a 6-bit number (001011_2) is 52 (110100_2).

2 Mathematical Composition of an FFT

2.1 Definitions

2.1.1 Domains

The domain for variables used as indices is the set of nonnegative integers. This includes the variables j, j_0, j_1, k, k_0, k_1, k_2, m, n, N, p, and q. That domain should be understood in the theorems below. Other variables are drawn from the set of complex numbers or are explicitly described.

The square root of -1 is denoted with i and not with j.

Indices begin at zero. A vector with 2^N elements has indices k satisfying $0 \le k < 2^N$.

2.1.2 Notation

Bold font indicates a vector: **a**.

A subscript indicates an element of a vector: a_j.

Brackets indicate construction of a vector: $[j^2]_{0 \le j < 4}$ is a vector containing the elements 0, 1, 4, and 9.

If **a** has, for example, 2^m elements, then **a** is identical to $[a_j]_{0 \le j < 2^m}$.

2.1.3 Roots of 1

For convenience, we define $\mathbf{1}^x$ to be $e^{2\pi i x}$. Note that $\mathbf{1}^{p/q}$ is one of the qth roots of 1. Specifically, it is the pth such root in the counterclockwise (+i) direction from the real axis. Thus, $\mathbf{1}^{0/4} = 1$, $\mathbf{1}^{1/4} = i$, $\mathbf{1}^{2/4} = -1$, and $\mathbf{1}^{3/4} = -i$.

$\mathbf{1}^x$ is cyclic with period 1, since, if k is an integer, $\mathbf{1}^{k+x} = e^{2\pi i (k+x)} = e^{2\pi i k} e^{2\pi i x} = 1 e^{2\pi i x} = \mathbf{1}^x$.

$\mathbf{1}^{-x}$ is the complex conjugate of $\mathbf{1}^x$, written $\overline{\mathbf{1}^x}$.

2.1.4 Discrete Fourier Transform (DFT)

The DFT of a 2^N-element vector **h** is the vector **H**:

$$H_k = \sum_{0 \le j < 2^N} \mathbf{1}^{\frac{jk}{2^N}} h_j \text{ for } 0 \le k < 2^N.$$

This is identical to the conventional definition that uses $e^{\frac{jk}{2^N} 2\pi i}$ for the coefficient rather than $\mathbf{1}^{\frac{jk}{2^N}}$.

2.1.5 Bit-Reversal Function, *r*(*k*)

Given an integer k, let $[b_i]$ be the string of bits (0 or 1) such that $k = \sum_i b_i 2^i$. Thus $[b_i]$ is the binary numeral for k. The sum may be taken over all integers i. Only finitely many bits will be non-zero, so the sum effectively has a finite number of terms although limits on i are not explicitly written.

Define $r(k) = \sum_i b_i 2^{-i-1}$.

A description of r is:

r(*k*) is the number obtained by writing the binary digits of *k* in reverse order after a ".". E.g., $r(12) = r(1100_2) = .0011_2 = 3/16$.

The way *r* maps integers to fractions is convenient in the FFT, particularly since *r* is independent of the length of the vector being transformed. We will also multiply *r*(*k*) by 2^m to produce an integer result:

Lemma (1): If $k<2^m$, $2^m r(k)$ is the number obtained by writing *k* as an *m*-bit number in binary, including leading 0s, and reversing the digits (that is, exchanging the *i*th digit with the *m*-*i*-1th digit).

Proof: $2^m r(k)$ is $2^m \sum_i b_i 2^{-i-1} = \sum_i b_i 2^{m-i-1}$. By substituting *m*-*i*-1 for *i*, we obtain $\sum_i b_{m-i-1} 2^i$.

Corollary (2): If $k < 2^m$, then $2^m r(k)$ is an integer.

Lemma (3): If $k_1 < 2^m$, then $r(2^m k_0 + k_1) = r(k_1)+r(2^m k_0)$.

Proof: Let $[b_{0,i}]$ and $[b_{1,i}]$ be binary numerals for k_0 and k_1, respectively, and note that $[b_{0,i-m}]$ is the binary numeral for $2^m k_0$. The binary numeral for $2^m k_0+k_1$ is $[b_{0,i-m}+b_{1,i}]$ because $b_{0,i-m}+b_{1,i}$ is always a binary digit; $b_{0,i-m}$ and $b_{1,i}$ are never both 1. ($b_{1,i}$ is 1 only for some *i* less than *m*, and $b_{0,i-m}$ is 1 only for some *i* not less than *m*.) Then:

$$\begin{aligned} r\left(2^m k_0 + k_1\right) &= \sum_i \left(b_{0,i-m} + b_{1,i}\right) 2^i \\ &= \sum_i b_{0,i-m} 2^i + \sum_i b_{1,i} 2^i \\ &= r\left(2^m k_0\right) + r\left(k_1\right). \end{aligned}$$

Lemma (4): $2^m r(2^m k) = r(k)$.

Proof: If $[b_i]$ is the binary numeral for *k*, $[b_{i-m}]$ is the binary numeral for $2^m k$. Then $2^m r(2^m k)$ is $2^m \sum_i b_{i-m} 2^{-i-1} = \sum_i b_{i-m} 2^{m-i-1}$. Substituting *m*+*i* for *i* gives $\sum_i b_i 2^{-i-1}$, which is *r*(*k*).

2.2 1^x with *r*(*k*)

Lemma (5): If $k_1 < 2^m$, then $\mathbf{1}^{2^m \cdot r\left(2^m k_0 + k_1\right)} = \mathbf{1}^{r(k_0)}$.

Proof:

$\mathbf{1}^{2^m \cdot r\left(2^m k_0 + k_1\right)} = \mathbf{1}^{2^m \cdot r(k_1) + 2^m \cdot r\left(2^m k_0\right)}$, by Lemma (3).

$= \mathbf{1}^{2^m \cdot r\left(2^m k_0\right)}$, since $\mathbf{1}^x$ is cyclic and $2^m r(k_1)$ is an integer by Corollary (2).

$= \mathbf{1}^{r(k_0)}$, by Lemma (4).

2.3 Introduction to the FFT Procedure

Let **v** be a sequence of vectors, so that $\mathbf{v}_i$ is the ith vector in **v** and $v_{i,k}$ is the kth element in the ith vector in **v**. Each vector will be of length 2^N, so the element index k satisfies $0 \le k < 2^N$.

Let $\mathbf{v_0} = \mathbf{h}$, where **h** is a vector we are interested in computing the DFT of.

We define $v_{n,k}$ for $0 < n \le N$ by dividing k by 2^{N-n} and using the quotient k_0 and the remainder k_1:

Definition: $$v_{n,k} = v_{n,2^{N-n}k_0+k_1} = \sum_{0 \le j < 2^n} \mathbf{1}^{j \cdot r(k_0)} v_{0,2^{N-n}j+k_1}.$$

The last vector of this sequence, $\mathbf{v}_N$, is the bit-reversal permutation of **H**, the DFT of **h**, as **H** is defined in section 2.1.4. To see this, consider an element $v_{N,k}$. Following the definition of $v_{n,k}$, we divide the element index k by 2^{N-N} to get the quotient k and the remainder 0, which gives:

$$v_{N,k} = \sum_{0 \le j < 2^N} \mathbf{1}^{j \cdot r(k)} v_{0,j+0} = \sum_{0 \le j < 2^N} \mathbf{1}^{\frac{j \cdot 2^N r(k)}{2^N}} v_{0,j} = \sum_{0 \le j < 2^N} \mathbf{1}^{\frac{j \cdot 2^N r(k)}{2^N}} h_j = H_{2^N r(k)}.$$

Thus, $\mathbf{v}_N$ is **H** indexed with a bit-reversal function (refer to Lemma (1) about $2^N r(k)$).

The last vector, $\mathbf{v}_N$, is the result we want, and the intermediate vectors form a route for getting there. Any vector in the sequence can be computed from any previous vector in the sequence. To see this, we will show how elements of $\mathbf{v}_{n+m}$ can be computed from 2^m elements of $\mathbf{v}_n$, for $0 \le n \le n+m \le N$. To do this, we need to take an index k into the vector and decompose it into three parts, k_0, k_1, and k_2, such that $k = 2^{N-n}k_0 + 2^{N-n-m}k_1 + k_2$ and $0 \le k_0 < 2^n$, $0 \le k_1 < 2^m$, and $0 \le k_2 < 2^{N-n-m}$. Given that, we will show below that:

Equation (6): $$d_{k_1} = \sum_{0 \le j_1 < 2^m} \mathbf{1}^{j_1 \cdot r(k_1)} \omega^{j_1} a_{j_1},$$

where $\omega = \mathbf{1}^{r(2^m k_0)}$, $d_{k_1} = v_{n+m,2^{N-n}k_0+2^{N-n-m}k_1+k_2}$, and $a_{j_1} = v_{n,2^{N-n}k_0+2^{N-n-m}j_1+k_2}$.

Equation (6) is the classic butterfly operation of the FFT:

- A few elements of an already-computed vector $\mathbf{v}_n$ are extracted to form a vector **a**, which has 2^m elements.

- The elements of **a** are multiplied by certain coefficients[1] to form a new vector $[\omega^{j_1} a_{j_1}]_{0 \le j_1 < 2^m}$.
- The DFT of $[\omega^{j_1} a_{j_1}]_{0 \le j_1 < 2^m}$ is computed to give a new vector **d**.
- The elements of **d** become elements in a new vector $\mathbf{v}_{n+m}$.

2^m is called the radix of the butterfly.

An advantage of this formulation is that the coefficients and the elements of $\mathbf{v}_{n+m}$ and $\mathbf{v}_n$ are explicitly identified. Some formulations of the FFT show that the FFT can be performed using butterfly operations in this form but leave out details or include them only as part of a complete algorithm from which it is difficult to identify individual butterfly operations.

2.4 Proof of the FFT Procedure

Now we prove the claim. First, write k as $2^{N-n}k_0 + 2^{N-n-m}k_1 + k_2$ using the k_0, k_1, and k_2 described above. Observe that dividing k by $2^{N-(n+m)}$ gives a quotient $2^m k_0+k_1$ and a remainder k_2. So by definition, $v_{n+m,k}$ is:

$$v_{n+m,2^{N-n-m}\left(2^m k_0+k_1\right)+k_2} = \sum_{0 \le j < 2^{n+m}} \mathbf{1}^{j \cdot r\left(2^m k_0+k_1\right)} v_{0,2^{N-n-m} j+k_2}.$$

Divide j by 2^m to get a quotient j_0 and a remainder j_1. Then:

$$\begin{aligned}
v_{n+m,2^{N-n-m}\left(2^m k_0+k_1\right)+k_2} &= \sum_{0 \le 2^m j_0+j_1 < 2^n} \mathbf{1}^{\left(2^m j_0+j_1\right) \cdot r\left(2^m k_0+k_1\right)} v_{0,2^{N-n-m}\left(2^m j_0+j_1\right)+k_2} \\
&= \sum_{0 \le j_1 < 2^m} \sum_{0 \le j_0 < 2^n} \mathbf{1}^{\left(2^m j_0+j_1\right) \cdot r\left(2^m k_0+k_1\right)} v_{0,2^{N-n-m}\left(2^m j_0+j_1\right)+k_2} \\
&= \sum_{0 \le j_1 < 2^m} \sum_{0 \le j_0 < 2^n} \mathbf{1}^{\left(2^m j_0+j_1\right) \cdot r\left(2^m k_0+k_1\right)} v_{0,2^{N-n} j_0+\left(2^{N-n-m} j_1+k_2\right)} \\
&= \sum_{0 \le j_1 < 2^m} \sum_{0 \le j_0 < 2^n} \mathbf{1}^{j_1 \cdot r\left(2^m k_0+k_1\right)} \mathbf{1}^{2^m j_0 \cdot r\left(2^m k_0+k_1\right)} v_{0,2^{N-n} j_0+\left(2^{N-n-m} j_1+k_2\right)} \\
&= \sum_{0 \le j_1 < 2^m} \mathbf{1}^{j_1 \cdot r\left(2^m k_0+k_1\right)} \sum_{0 \le j_0 < 2^n} \mathbf{1}^{2^m j_0 \cdot r\left(2^m k_0+k_1\right)} v_{0,2^{N-n} j_0+\left(2^{N-n-m} j_1+k_2\right)}.
\end{aligned}$$

So far, we have used standard algebraic derivations. The next step uses properties of r and $\mathbf{1}^x$. By Lemma (5), $\mathbf{1}^{2^m \cdot r\left(2^m k_0+k_1\right)} = \mathbf{1}^{r(k_0)}$. That gives us:

Equation (7): $$v_{n+m,2^{N-n-m}\left(2^m k_0+k_1\right)+k_2} = \sum_{0 \le j_1 < 2^m} \mathbf{1}^{j_1 \cdot r\left(2^m k_0+k_1\right)} \sum_{0 \le j_0 < 2^n} \mathbf{1}^{j_0 \cdot r(k_0)} v_{0,2^{N-n} j_0+\left(2^{N-n-m} j_1+k_2\right)}.$$

[1] These coefficients are commonly called "twiddles." I will call them weights.

Consider $v_{n,2^{N-n}k_0+\left(2^{N-n-m}j_1+k_2\right)}$. To use the definition of v_n, divide $2^{N-n}k_0 + (2^{N-n-m}j_1 + k_2)$ by 2^{N-n} to get quotient k_0 and remainder $(2^{N-n-m}j_1 + k_2)$, and then the definition gives:

$$v_{n,2^{N-n}k_0+\left(2^{N-n-m}j_1+k_2\right)} = \sum_{0\le j<2^n} \mathbf{1}^{j\cdot r(k_0)} v_{0,2^{N-n}j+\left(2^{N-n-m}j_1+k_2\right)}.$$

Change j to j_0 in that equation and substitute it into Equation (7) to get:

$$v_{n+m,2^{N-n-m}\left(2^m k_0+k_1\right)+k_2} = \sum_{0\le j_1<2^m} \mathbf{1}^{j_1\cdot r\left(2^m k_0+k_1\right)} v_{n,2^{N-n}k_0+\left(2^{N-n-m}j_1+k_2\right)}.$$

By Lemma (3), $r(2^m k_0 + k_1) = r(k_1)+r(2^m k_0)$, so:

$$\begin{aligned} v_{n+m,2^{N-n-m}\left(2^m k_0+k_1\right)+k_2} &= \sum_{0\le j_1<2^m} \mathbf{1}^{j_1\cdot r(k_1)}\mathbf{1}^{j_1\cdot r\left(2^m k_0\right)} v_{n,2^{N-n}k_0+2^{N-n-m}j_1+k_2} \\ &= \sum_{0\le j_1<2^m} \mathbf{1}^{j_1\cdot r(k_1)}\left(\mathbf{1}^{r\left(2^m k_0\right)}\right)^{j_1} v_{n,2^{N-n}k_0+2^{N-n-m}j_1+k_2}. \end{aligned}$$

This is readily seen to be equivalent to Equation (6). We have proven that Equation (6) can be used to compute each vector $\mathbf{v}_n$ from previous vectors, so a sequence of such computations will compute the DFT of **h**.

2.5 Conclusions

The key statements from the above sections are:

$$v_{0,k} = h_k,$$

$$v_{n+m,2^{N-n}k_0+2^{N-n-m}k_1+k_2} = \sum_{0\le j_1<2^m} \mathbf{1}^{j_1\cdot r(k_1)}\left(\mathbf{1}^{r\left(2^m k_0\right)}\right)^{j_1} v_{n,2^{N-n}k_0+2^{N-n-m}j_1+k_2}, \text{ and}$$

$$v_{N,k} = H_{2^N r(k)}.$$

This suffices to show the FFT takes O(n log n) time (here n is the number of elements) and how to implement it simply (by choosing an m, constructing a general butterfly implementation, and iterating through the values of n, k_0, and k_2). For a high-performance FFT, it is just our starting point.

3 Initial Design

3.1 Starting the FFT Kernel

3.1.1 Implement C Code From the Mathematics

The conclusions above show how to implement an FFT that executes in O(n log n) time. To perform an FFT on a vector of 2^N elements, first decide what value of m to use in each step from some $\mathbf{v}_n$ to some $\mathbf{v}_{n+m}$. E.g., for a vector of 2^9 elements, we might use m’s of 3, 2, 2, and 2 to go from $\mathbf{v}_0$ to $\mathbf{v}_3$ to $\mathbf{v}_5$ to $\mathbf{v}_7$ to $\mathbf{v}_9$. For each of these $\mathbf{v}$’s after $\mathbf{v}_0$, calculate:

$$v_{n+m,2^{N-n}k_0+2^{N-n-m}k_1+k_2} = \sum_{0\le j_1<2^m} \mathbf{1}^{j_1\cdot r(k_1)}\left(\mathbf{1}^{r\left(2^m k_0\right)}\right)^{j_1} v_{n,2^{N-n}k_0+2^{N-n-m}j_1+k_2},$$

using the corresponding values of n and m.

To be formal, let m_0, m_1, m_2,…, m_{P-1} be a sequence of positive integers that sum to N. Let where $n_0 = 0$ and $n_{p+1}=n_p + m_p$. Then $n_P=N$, and the following set of calculations is sufficient to perform an FFT on a vector of length 2^N.

$$\left\{ v_{n_{p+1},2^{N-n_p}k_0+2^{N-n_p-m_p}k_1+k_2} = \sum_{0\le j_1<2^{m_p}} \mathbf{1}^{j_1\cdot r(k_1)}\left(\mathbf{1}^{r\left(2^{m_p} k_0\right)}\right)^{j_1} v_{n_p,2^{N-n_p}k_0+2^{N-n_p-m_p}j_1+k_2} \right\}_B,$$

where B represents the variables bounds and is: $0 \le p < P$, $0 \le k_0 < 2^{n_p}$, $0 \le k_1 < 2^{m_p}$, and $0 \le k_2 < 2^{N-n_p-m_p}$. Although this expression is tedious, we can translate it directly into C code:

FFT Directly From Mathematics

```
for (p  = 0; p  < P            ; ++p )
for (k0 = 0; k0 < 1<<n[p]      ; ++k0)
for (k1 = 0; k1 < 1<<m[p]      ; ++k1)
for (k2 = 0; k2 < 1<<N-n[p]-m[p]; ++k2)
{
   complex sum = 0.;
   for (j1 = 0; j1 < 1<<m[p]; ++j1)
      sum += one(j1*r(k1)) * one(j1*r((1<<m[p])*k0)) *
         v [n[p]] [(1<<N-n[p])*k0 + (1<<N-n[p]-m[p])*j1 + k2];
   v [n[p+1]] [(1<<N-n[p])*k0 + (1<<N-n[p]-m[p])*k1 + k2] = sum;
}
```

where `one(x)` and `r(k)` are functions to compute $\mathbf{1}^x$ and $r(k)$. (The exponentiation by j_1 has been written as a multiplication in the exponent of $\mathbf{1}$.)

3.1.2 Group Butterfly Calculations Together

The C code specifies an execution order, but the mathematical expression is a set of operations. They may be performed in any order, subject to the natural constraint that each element $\mathbf{v}_{n,k}$ must be calculated before it is used. We will rearrange the calculations to benefit computing speed. (In fact, I chose P to stand for "pass," reflecting that the FFT can be performed in P separate passes over the data, exactly as in the loops above. We will not stay with this, although I will sometimes refer to calculations in a certain pass, such as the first pass, the last pass, or some pass p. These refer to logical positions in the calculation and not necessarily chronological positions in the execution sequence.)

First, note that for given values of `p`, `k0`, and `k2`, the calculations for different values of `k1` use the same elements of `v[n[p]]`. (`k1` does not appear in the subscript to `v[n[p]]`.) Because of this, it is efficient to group these calculations together, since that allows all the inputs to be read once and used repeatedly. Since `k1` and `k2` are independent of each other, we may freely swap the order of their loops:

```
for (p  = 0; p  < P             ; ++p )
for (k0 = 0; k0 < 1<<n[p]       ; ++k0)
for (k2 = 0; k2 < 1<<N-n[p]-m[p]; ++k2)
for (k1 = 0; k1 < 1<<m[p]       ; ++k1)
...
```

3.1.3 Create a Butterfly Subroutine

Define a subroutine:

FFT_Butterflies

```
static void FFT_Butterflies(
   int m,               // Butterfly radix.
   ComplexArray vOut,   // Address of output vector.
   ComplexArray vIn,    // Address of input vector.
   int k0,              // k0 from equation.
   int c0               // Coefficient for k0.
)
{
   // Coefficient for k1 is coefficient for k0 divided by 1<<m.
   const int c1 = c0 >> m;
   int j1, k1, k2;

   for (k2 = 0; k2 < c1  ; ++k2)
   for (k1 = 0; k1 < 1<<m; ++k1)
   {
      complex sum = 0.;
      for (j1 = 0; j1 < 1<<m; ++j1)
         sum += one(j1*r(k1)) * one(j1*r((1<<m)*k0)) *
            vIn[c0*k0 + c1*j1 + k2];
      vOut[c0*k0 + c1*k1 + k2] = sum;
   }
}
```

By using this subroutine, our FFT code becomes:

First FFT Kernel

```
for (p  = 0; p  < P      ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(m[p], v[n[p+1]], v[n[p]], k0, 1<<N-n[p]);
```

We will call this the FFT kernel. It will change and grow as we improve the implementation.

The code above for `FFT_Butterflies` does not show all the inputs `vIn[c0*k0 + c1*j1 + k2]` being read prior to the loop on `k1`, but that will be a feature of the butterfly routines we construct later. For now, we will state that feature is added to the `FFT_Butterflies` routine without showing it. A second benefit of the feature is that by reading all input elements before writing any output element, the routine may be used "in-place," that is, with the same memory used for `vIn` and `vOut`.

3.2 Structuring the FFT

3.2.1 General

We must choose the values of m_p. These are largely influenced by our target processor architecture. To begin, I require that N be at least 4, so that some reductions can be made later, in section 3.3.5. FFTs for fewer than 16 ($N < 4$) elements can be implemented separately.

In most passes, a radix-4 butterfly (m is 2) is attractive, as it is efficient and a high-performance implementation is feasible. A high-performance general radix-8 butterfly is difficult or impossible to implement (see below). Conversely, a radix-2 butterfly is easy but inefficient. So for most passes, we will use radix-4 butterflies.

What would be required to implement a general radix-8 butterfly? A general radix-8 butterfly has 16 input numbers (real and imaginary components of eight complex numbers), 14 numbers for weights, and one additional constant ($\sqrt{2}/2$). Those numbers occupy 31 processor registers. (The AltiVec registers hold four floating-point numbers each, but we wish to use the parallelism of the processor and perform four butterflies at once. Each number needed by one butterfly will occupy one of the four spaces in a register, and the parallel numbers of other butterflies will occupy others.) The PowerPC CPU 7400, like all existing AltiVec processors, executes instructions in a pipeline. To obtain high performance, multiple instructions must be executed simultaneously, and so multiple calculations must be in progress at one time. With 31 registers occupied and 32 total, only one register is left to work with.

It is possible to start some calculations of the radix-8 butterfly without having all the input data, and it is possible to perform calculations at less than the best possible speed. A radix-8 butterfly is more efficient than a radix-4 butterfly in that two passes of a radix-8 butterfly yield the same mathematical results as three passes of a radix-4 butterfly but require the data to be read and written only two times instead of three. It is conceivable that an FFT structured with radix-8 passes could compete for performance with an FFT structured with radix-4 passes. I have not fully explored this possibility and do not consider it further in this paper.

3.2.2 First Pass

Although we will use m=2 for most passes, we will consider the first m, m_0, separately because of a significant difference in the butterfly calculations in the initial pass (p is zero). When p is zero, n_p is zero. Recall the bounds on k_0 are $0 \le k_0 < 2^{n_p}$, so, when p is zero, we have $0 \le k_0 < 1$, so k_0 is zero and only zero. Then the weight used, $\omega = \mathbf{1}^{r\left(2^m k_0\right)}$, is 1. Since multiplying by 1 is a waste of time, a special butterfly implementation that omits the multiplications by the weight will be faster than a general implementation that multiplies by the weight, while still getting correct results in this case.

When the multiplications by the weight are not needed, the number of processor registers required by an implementation of the butterfly calculations is reduced, since registers are not needed to hold the values associated with the weight. In this case, high-performance radix-4, radix-8, and radix-16 butterfly implementations are all feasible. Generally, a higher-radix butterfly is preferred, for two reasons. One, a good FFT composed of higher-radix butterflies uses no more, and perhaps fewer, calculations than an FFT composed of lower-radix butterflies. Two, with higher-radix butterflies, fewer passes are needed, and so the number of times data must be read from and written to memory (or cache) is lower.

Because we are using butterflies with m=2 (radix-4) for all passes after the first, we need at least two butterfly implementations for the first pass, one with an even m and one with an odd m. The m’s must sum to N, which can be even or odd. Thus, we must use the radix-8 butterfly for odd N, and we may use either the radix-4 or radix-16 butterfly for even N. The radix-16 butterfly provides a slight performance advantage over the radix-4 butterfly, but the cost of implementing it might not be worth the slight gain. I will use the radix-4 butterfly in this paper. The changes required to support a radix-16 butterfly in the initial pass are small. (Among other changes, the minimum value of N will need to be increased from 4 to 5.)

3.2.3 Summary

This then gives us an FFT structure. For even N, use a radix-4 butterfly on the first pass and all remaining passes. For odd N, use a radix-8 butterfly on the first pass and a radix-4 butterfly on all remaining passes.

3.3 Preparing the Kernel

3.3.1 Separate the First Pass

The FFT kernel is:

```
for (p  = 0; p  < P      ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(m[p], v[n[p+1]], v[n[p]], k0, 1<<N-n[p]);
```

To use a special butterfly routine for the first pass, we should separate that iteration from the rest of the loop. That gives:

```
for (p  = 0; p  < 1      ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(m[p], v[n[p+1]], v[n[p]], k0, 1<<N-n[p]);

for (p  = 1; p  < P      ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(m[p], v[n[p+1]], v[n[p]], k0, 1<<N-n[p]);
```

Some simplifications are now possible. The first loop (on `p`) is a single iteration, and so is the second (on `k0`) since `n[p]` is 0. Instances of `k0` and `n[p]` in these loops may be replaced with 0. In the second set of loops, `m[p]` is always 2, so it will be replaced. Then we have:

```
p = 0;
FFT_Butterflies(m[p], v[n[p+1]], v[n[p]], 0, 1<<N);

for (p  = 1; p  < P      ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(2, v[n[p+1]], v[n[p]], k0, 1<<N-n[p]);
```

3.3.2 Eliminate Fictitious Mathematical Vectors

As stated earlier, the butterfly routines can be written to work in place. So we are not required to have separate memory for `v[n[p]]` and `v[n[p+1]]`. Instead, we can pass the same memory location for the butterfly input and output vectors. Before the butterfly, the memory will contain elements of $\mathbf{v}_{n_p}$. After the butterfly, the memory will contain elements of $\mathbf{v}_{n_{p+1}}$. We will use two arrays, named `vIn` for the original input array and `vOut` for the final output array. On the first pass, data is read from `vIn` and written to `vOut`. On subsequent passes, data is both read from and written to `vOut`, so all subsequent calculations are performed in place. Note that `vIn` may be the same array as `vOut` or may be different. The new code is:

```
p = 0;
FFT_Butterflies(m[p], vOut, vIn, 0, 1<<N);

for (p  = 1; p  < P      ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n[p]);
```

3.3.3 Specialize Values for the First Pass

We decided to use an initial radix-8 butterfly if N is odd and an initial radix-4 butterfly if N is even, so the first call to `FFT_Butterflies` with can be expanded with 3 or 2 substituted for `m[p]`:

```
if (N & 1)
   FFT_Butterflies(3, vOut, vIn, 0, 1<<N);
```

```
else
   FFT_Butterflies(2, vOut, vIn, 0, 1<<N);

for (p  = 1; p  < P      ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n[p]);
```

3.3.4 Discuss the Last Two Passes

The last two passes (when *p* is *P*-2 and *P*-1) are also special, partly for reasons to do with the target computer architecture. We classify passes after the first:

- general passes (*p* < *P*-2), in which there are many (more than four) iterations on `k2` for each iteration on `k0`,
- the penultimate pass (*p* = *P*-2), in which there are four iterations on `k2` for each iteration on `k0`, and
- the final pass (*p* = *P*-1), in which there is one iteration on `k2` for each iteration on `k0`.

The general passes have the feature that one weight is used for many iterations on `k2`, because the weight depends only on `k0` and not on `k2`. Thus, we will be able to load the values associated with a weight once each time `k0` changes and use them for many values of `k2`.

In the penultimate pass, there are four iterations on `k2` per iteration on `k0`. With AltiVec instructions, one iteration of the butterfly instruction sequence will calculate butterflies for four values of `k2`. Thus, the weight used will change in each iteration of the instruction sequence. In this case, it is better to use code designed to reload the weight values frequently.

In the final pass, there is one iteration on `k2` per iteration on `k0`. `k2` is always zero, and the coefficient `c1` in the `FFT_Butterflies` routine is 1. This means the input elements for one butterfly are adjacent to each other in the array, as can be seen by examining the subscripts in the butterfly code. AltiVec instructions are not well suited to data packed so closely together, so a special routine is necessary. To complicate matters further, we will want to do additional processing in the final pass.

3.3.5 Separate the Last Two Passes

The details of designing specialized routines to calculate butterflies in the last two passes will be examined in sections 4.3.4 and 4.3.5. For now, we want to prepare the kernel by separating those passes:

```
if (N & 1)
   FFT_Butterflies(3, vOut, vIn, 0, 1<<N);
else
   FFT_Butterflies(2, vOut, vIn, 0, 1<<N);

for (p  = 1; p  < P-2    ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
```

```
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n[p]);

for (      ; p  < P-1    ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n[p]);

for (      ; p  < P      ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n[p]);
```

As before, some simplifications become available. The values of `1<<N-n[p]` are known constants in the final two sets of loops, where p is P-2 and P-1. n_p=N, n_{p-1}+m_{p-1}=n_p, and m_{p-1}=2, so n_{p-1}=N-2. Similarly, n_{p-2}=N-4. Thus `1<<N-n[p]` is 16 and 4 in the final two sets of loops. The values of `n[p]` are not constants but are known to be `N-4` and `N-2`. Making these substitutions gives:

```
for (      ; p  < P-1   ; ++p )
for (k0 = 0; k0 < 1<<N-4; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 16);

for (      ; p  < P     ; ++p )
for (k0 = 0; k0 < 1<<N-2; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 4);
```

The loop “`for (; p < P; ++p)`” can become “`if (p < P)`” because there is at most one iteration (since `p` is at least P-1 after the previous set of loops). This discards one execution of “`++p`”, but there is no subsequent code that uses `p`, so the increment is superfluous. Further, if we require that P be at least 2, the condition is necessarily true, so the test may be omitted. Requiring P be 2 implies we have at least m_0 and m_1, each of which will be at least 2, so N is at least 4. Our kernel now works only for vectors of at least 16 elements. (If a radix-16 butterfly is used in the first pass in lieu of a radix-4 butterfly, the radix-8 butterfly becomes the smaller possibility in the first pass. Then m_0 is at least 3, so N is at least 5, and there must be at least 32 elements.)

The kernel is now:

```
if (N & 1)
   FFT_Butterflies(3, vOut, vIn, 0, 1<<N);
else
   FFT_Butterflies(2, vOut, vIn, 0, 1<<N);

for (p  = 1; p  < P-2    ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n[p]);

for (      ; p  < P-1   ; ++p )
for (k0 = 0; k0 < 1<<N-4; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 16);

for (k0 = 0; k0 < 1<<N-2; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 4);
```

As we saw in the last set of loops, the "`for (; p < P-1; ++p)`" in the penultimate set can be changed to "`if (p < P-1)`", and the `++p` is again superfluous and may be discarded:

Expanded FFT Kernel

```
if (N & 1)
   FFT_Butterflies(3, vOut, vIn, 0, 1<<N);
else
   FFT_Butterflies(2, vOut, vIn, 0, 1<<N);

for (p  = 1; p  < P-2    ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n[p]);

if (p < P-1)
for (k0 = 0; k0 < 1<<N-4; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 16);

for (k0 = 0; k0 < 1<<N-2; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 4);
```

3.3.6 Use Butterfly Specializations

Now that the special cases have been separated in the kernel, we take advantage of them by using specializations of the butterfly routine customized for high performance in each case.

The initial radix-8 and radix-4 butterflies will be performed by routines that are essentially `FFT_Butterflies` specialized to `m`=2 or `m`=3 and `k0`=0. These are named `FFT8_0Weights` (described in section 4.3.3) and `FFT4_0Weights` (described in section 4.3.2).

The general radix-4 butterflies will be performed by a routine specialized to `m`=2 and `vIn`=`vOut`. Rather than require this routine to calculate `one(j1*r((1<<m)*k0))`, we will pass it precalculated values to use. As discussed in section 3.3.4, only one weight is used per value of `k0`, so only one weight is passed. This routine is named `FFT4_1WeightPerCall` (described in section 4.3.1). In section 4.2, I discuss what the contents of the weights array should be.

In the penultimate pass, the loop on `k2` in `FFT_Butterflies` is executed only four times in C code, only once when implemented as AltiVec instructions. At the same time, `FFT_Butterflies` is called many times, since the upper bound on `k0` is larger than in previous passes. To reduce the overhead of routine calls, we use a routine that incorporates the loop on `k0`. In addition, the routine will be designed to efficiently load the weights, one per instruction sequence iteration. This routine is named `FFT4_1WeightPerIteration` (described in section 4.3.4). Instead of being passed a single value of `k0`, it is passed the upper bound on `k0`, and, instead of being passed a single weight, it is passed the array of weights.

The final pass similarly incorporates the loop on `k0` in its butterfly routine. This routine is named `FFT4_Final` (described in section 4.3.5). Using these new routines, our kernel becomes:

FFT Kernel Using Specialized Butterfly Routines

```
if (N & 1)
   FFT8_0Weights(vOut, vIn, 1<<N);
else
   FFT4_0Weights(vOut, vIn, 1<<N);

for (p  = 1; p  < P-2    ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT4_1WeightPerCall(vOut, k0, 1<<N-n[p], weights[k0]);

if (p < P-1)
   FFT4_1WeightPerIteration(vOut, 1<<N-4, weights);

FFT4_Final(vOut, 1<<N-2, weights);
```

4 Designing Butterfly Routines

The bulk of a high-performance FFT implementation is the butterfly routines. The `FFT_Butterflies` subroutine given in section 3.1.3 is quite general and does not provide high performance. We must implement the routines described in section 3.3.6.

These routines incorporate improvements including:

- Read each input element before writing any output to the same memory location, so the routine can be used "in-place."
- Load weights from a table instead of calculating them.
- Compute butterflies of specific radices with specialized code.
- Incorporate a loop iterating on `k0`.
- Omit multiplications by a weight when the weight is 1.
- Incorporate other desired processing, including rearranging data in memory.

As mentioned, a butterfly routine can read of its input elements before writing any output element. This is a straightforward modification and will not be demonstrated for the general butterfly routine. It will be a feature of all of the specific high-performance variations we write.

4.1 Prepared Constants

4.1.1 Internal Weights Are Built into Routine

The `FFT_Butterflies` routine contains two expressions that refer to `one` and `r`. The first of these is `one(j1*r(k1))`. This depends solely on `j1` and `k1`, each of which is between 0 and *m*. Thus, the expression `one(j1*r(k1))` takes on a fixed set of values that is determined by *m*. When writing a butterfly variation for a specific value of *m*, those values can be incorporated into the routine.

For example, in a radix-4 butterfly (*m* is 2), `one(j1*r(k1))` takes on the values 1, -1, *i*, and -*i* at various times. Rather than compute `one(j1*r(k1))`, the routine simply multiplies by 1, -1, *i*, or -*i* at the appropriate points.

In a radix-8 butterfly, we also see the values $\pm\sqrt{2}/2 \pm i\sqrt{2}/2$. With the constant $\sqrt{2}/2$ prepared at the time the source code is compiled or assembled, no calculation is needed at run-time for the values of `one(j1*r(k1))`.

4.1.2 External Weights Are Stored in An Array

The second expression is `one(j1*r((1<<m)*k0))`. The values of this expression depend on `k0`, so they differ from iteration to iteration in a loop on `k0`. Still, we wish to avoid computing them when the FFT is performed. A simple arrangement is to calculate all the values $\left(\mathbf{1}^{r\left(2^m k_0\right)}\right)^{j_1}$ takes on and store them in an array, say an array named `weights`. The value of $\left(\mathbf{1}^{r\left(2^m k_0\right)}\right)^{j_1}$ could be in `weights[k0][j1]`.

However, we will see in section 4.2 that these values are not all used directly in a high-performance implementation of a radix-4 butterfly. Instead, for each value of k_0, we use six floating-point numbers derived from the values $\left(\mathbf{1}^{r\left(2^m k_0\right)}\right)^{j_1}$ for $0<j_1<4$. These six values will be stored in some structure in the array element `weights[k0]`.

Calculating the values and storing them in an array saves no computation time when performing a single FFT. There is a savings when numerous FFTs on vectors of the same length are performed, as these weight calculations need be performed only once, prior to performing the first FFT. Reading the values from memory will usually be much faster than calculating them. Hence there is a great advantage to storing the values.

Note that the value of *m* is assumed in `weights`. A specific preparation of the array `weights` provides values only for butterflies of a specific radix. To provide values for multiple radices, multiple arrays or more-complicated arrangements would be needed.

4.1.3 Common Weights

The elements of `weights` are independent of the length of the vector being transformed. The number of elements we need from the array depends on the length of the vector (`k0` reaches higher values for longer vectors), but the contents of each element are the same. For example, every FFT for which `k0` reaches the value 7 uses the same value in `weights[7][1]`. Thus, one array of weights (for a specific radix) arranged in this way may be easily shared by FFTs of every length.

4.2 General Radix-4 Butterfly Algorithm

4.2.1 Goedecker's Algorithm

The general radix-4 butterfly, with an input vector **a**, an output vector **d**, and some weight ω, is Equation (6) with m=2:

$$d_{k_1} = \sum_{0 \le j_1 < 4} \mathbf{1}^{j_1 \cdot r(k_1)} \omega^{j_1} a_{j_1} .$$

Implementations of this calculation that minimize the number of multiplications have been known for some time, but the AltiVec architecture, like many others, features a fused multiply-add operation. Using this operation, S. Goedecker gives us a 22-instruction sequence for calculating the radix-4 butterfly.[2] Goedecker's algorithm requires that we prepare the weights in a different form. In place of the six real and imaginary components of ω^1, ω^2, and ω^3, we use six values calculated from them:

`w1r` = Re(ω).
`w1i` = Im(ω)/Re(ω).
`w2r` = Re(ω^2).
`w2i` = Im(ω^2)/Re(ω^2).
`w3r` = Re(ω^3)/Re(ω).
`w3i` = Im(ω^3)/Re(ω^3).

When we wish to perform a radix-4 butterfly, we retrieve those six prepared values and read the four complex numbers of **a** into processor registers named `a0r`, `a0i`, `a1r`, `a1i`, `a2r`, `a2i`, `a3r`, and `a3i`, whose names indicate the real and imaginary components of the elements of **a** in the natural way. Then Goedecker's algorithm is:

Goedecker's Algorithm

```
b1r = - a1i * w1i + a1r.
b1i = + a1r * w1i + a1i.
b2r = - a2i * w2i + a2r.
b2i = + a2r * w2i + a2i.
b3r = - a3i * w3i + a3r.
b3i = + a3r * w3i + a3i.
c0r = + b2r * w2r + a0r.
c0i = + b2i * w2r + a0i.
c2r = - b2r * w2r + a0r.
c2i = - b2i * w2r + a0i.
c1r = + b3r * w3r + b1r.
c1i = + b3i * w3r + b1i.
c3r = - b3r * w3r + b1r.
c3i = - b3i * w3r + b1i.
d0r = + c1r * w1r + c0r.
d0i = + c1i * w1r + c0i.
d1r = - c1r * w1r + c0r.
d1i = - c1i * w1r + c0i.
d2r = - c3i * w1r + c2r.
d2i = + c3r * w1r + c2i.
d3r = + c3i * w1r + c2r.
d3i = - c3r * w1r + c2i.
```

[2] S. Goedecker, "Fast Radix 2, 3, 4, and 5 Kernels for Fast Fourier Transformations on Computers with Overlapping Multiply-Add Instructions," *SIAM Journal of Scientific Computing* 18, no. 6 (November 1997): 1605-1611, `http://epubs.siam.org/sam-bin/dbq/article/28194`.

Upon completion of this sequence, `d0r`, `d0i`, `d1r`, `d1i`, `d2r`, `d2i`, `d3r`, and `d3i` contain the real and imaginary components of **d**, as may be verified by working through the algebra. Note that each of the 22 lines corresponds to one AltiVec `vmaddfp` or `vnmsubfp` instruction. (Except those instructions operate on four sets of data, where I have shown only one.)

4.2.2 Division by Zero

The astute reader will have wondered what happens when Re(ω), Re(ω^2), or Re(ω^3) is zero. The short answer is to never let them be zero. In practice, this turns out to be simple, an unintended side effect of finite floating-point precision. Re(ω) is zero when ω is i (or -i). ω is $\mathbf{1}^{r\left(2^m k_0\right)}$, so it is i when $r(2^m k_0)$ is ¼. Then $\mathbf{1}^{1/4}$ is, by definition, $e^{2\pi i \, 1/4}$, which is $e^{\pi/2 \; i}$. When preparing the weights, the real part of this is calculated by evaluating cos(π/2), which is ideally zero. However, a computer's floating-point representation of π/2 is imperfect, and a small-nonzero value results. This avoids division by zero, but it introduces a small error into the FFT calculation. However, this error is no different from the many errors caused by rounding errors in all the other weights, which are also calculated imprecisely. The FFT calculation is necessarily slightly imprecise.

4.3 Butterfly Routines

4.3.1 FFT4_1WeightPerCall

`FFT4_1WeightPerCall` implements `FFT_Butterflies` with `m`=2 and `vIn`=`vOut` and with weight values provided so that it need not calculate them.

If we make the first two modifications (replacing `m` with 2 and `vIn` with `vOut`) directly to `FFT_Butterflies` and change the arguments, we get:

```
static void FFT4_1WeightPerCall(
   ComplexArray vOut,   // Address of output vector.
   int k0,              // k0 from equation.
   int c0,              // Coefficient for k0.
   CommonWeight weight  // Values for weight calculations.
)
{
   // Coefficient for k1 is coefficient for k0 divided by 1<<m.
   const int c1 = c0 >> 2;
   int j1, k1, k2;

   for (k2 = 0; k2 < c1; ++k2)
   for (k1 = 0; k1 < 4 ; ++k1)
   {
      complex sum = 0.;
      for (j1 = 0; j1 < 4; ++j1)
         sum += one(j1*r(k1)) * one(j1*r(4*k0)) *
            vOut[c0*k0 + c1*j1 + k2];
      vOut[c0*k0 + c1*k1 + k2] = sum;
   }
}
```

Now we will implement Goedecker's algorithm. Essentially, the loops on `k1` and `j1` are replaced by Goedecker's algorithm from section 4.2.1, including the necessary reads of input elements into symbols `a0r`, `a0i`, `a1r`, `a1i`, `a2r`, `a2i`, `a3r`, and `a3i` and writes of output from symbols `d0r`, `d0i`, `d1r`, `d1i`, `d2r`, `d2i`, `d3r`, and `d3i`. If `weight` is set correctly, the code above and the code below calculate the same results (aside from differences in floating-point rounding).

FFT4_1WeightPerCall

```
static void FFT4_1WeightPerCall(
   ComplexArray vOut,   // Address of output vector.
   int k0,              // k0 from equation.
   int c0,              // Coefficient for k0.
   CommonWeight weight  // Values for weight calculations.
)
{
   // Coefficient for k1 is coefficient for k0 divided by 1<<m.
   const int c1 = c0 >> 2;
   int k2;
   float a0r, a0i, a1r, a1i, a2r, a2i, a3r, a3i,
                   b1r, b1i, b2r, b2i, b3r, b3i,
         c0r, c0i, c1r, c1i, c2r, c2i, c3r, c3i,
         d0r, d0i, d1r, d1i, d2r, d2i, d3r, d3i;

   for (k2 = 0; k2 < c1; ++k2)
   {
      a0r = vOut.re[c0*k0 + c1*0 + k2];
      a0i = vOut.im[c0*k0 + c1*0 + k2];
      a1r = vOut.re[c0*k0 + c1*1 + k2];
      a1i = vOut.im[c0*k0 + c1*1 + k2];
      a2r = vOut.re[c0*k0 + c1*2 + k2];
      a2i = vOut.im[c0*k0 + c1*2 + k2];
      a3r = vOut.re[c0*k0 + c1*3 + k2];
      a3i = vOut.im[c0*k0 + c1*3 + k2];
      b1r = - a1i * weight.w1i + a1r;
      b1i = + a1r * weight.w1i + a1i;
      b2r = - a2i * weight.w2i + a2r;
      b2i = + a2r * weight.w2i + a2i;
      b3r = - a3i * weight.w3i + a3r;
      b3i = + a3r * weight.w3i + a3i;
      c0r = + b2r * weight.w2r + a0r;
      c0i = + b2i * weight.w2r + a0i;
      c2r = - b2r * weight.w2r + a0r;
      c2i = - b2i * weight.w2r + a0i;
      c1r = + b3r * weight.w3r + b1r;
      c1i = + b3i * weight.w3r + b1i;
      c3r = - b3r * weight.w3r + b1r;
      c3i = - b3i * weight.w3r + b1i;
      d0r = + c1r * weight.w1r + c0r;
      d0i = + c1i * weight.w1r + c0i;
      d1r = - c1r * weight.w1r + c0r;
      d1i = - c1i * weight.w1r + c0i;
      d2r = - c3i * weight.w1r + c2r;
      d2i = + c3r * weight.w1r + c2i;
      d3r = + c3i * weight.w1r + c2r;
      d3i = - c3r * weight.w1r + c2i;
      vOut.re[c0*k0 + c1*0 + k2] = d0r;
      vOut.im[c0*k0 + c1*0 + k2] = d0i;
      vOut.re[c0*k0 + c1*1 + k2] = d1r;
      vOut.im[c0*k0 + c1*1 + k2] = d1i;
      vOut.re[c0*k0 + c1*2 + k2] = d2r;
      vOut.im[c0*k0 + c1*2 + k2] = d2i;
      vOut.re[c0*k0 + c1*3 + k2] = d3r;
      vOut.im[c0*k0 + c1*3 + k2] = d3i;
   }
}
```

4.3.2 FFT4_0Weights

`FFT4_0Weights` implements `FFT_Butterflies` with `m`=2 and `k0`=0. The calculations for a weightless radix-4 butterfly are straightforward and can be derived from `FFT4_1WeightPerCall` by replacing `w1r`, `w2r`, and `w3r` with 1 and `w1i`, `w2i`, and `w3i` with 0 and simplifying the resulting code:

FFT4_0Weights

```
static void FFT4_0Weights(
   ComplexArray vOut,   // Address of output vector.
   ComplexArray vIn,    // Address of input vector.
   int c0               // Coefficient for k0.
)
{
   // Coefficient for k1 is coefficient for k0 divided by 1<<m.
   const int c1 = c0 >> 2;
   int k2;
   float a0r, a0i, a1r, a1i, a2r, a2i, a3r, a3i,
         c0r, c0i, c1r, c1i, c2r, c2i, c3r, c3i,
         d0r, d0i, d1r, d1i, d2r, d2i, d3r, d3i;

   for (k2 = 0; k2 < c1; ++k2)
   {
      a0r = vIn.re[c1*0 + k2];
      a0i = vIn.im[c1*0 + k2];
      a1r = vIn.re[c1*1 + k2];
      a1i = vIn.im[c1*1 + k2];
      a2r = vIn.re[c1*2 + k2];
      a2i = vIn.im[c1*2 + k2];
      a3r = vIn.re[c1*3 + k2];
      a3i = vIn.im[c1*3 + k2];
      c0r = + a2r + a0r;
      c0i = + a2i + a0i;
      c2r = - a2r + a0r;
      c2i = - a2i + a0i;
      c1r = + a3r + a1r;
      c1i = + a3i + a1i;
      c3r = - a3r + a1r;
      c3i = - a3i + a1i;
      d0r = + c1r + c0r;
      d0i = + c1i + c0i;
      d1r = - c1r + c0r;
      d1i = - c1i + c0i;
      d2r = - c3i + c2r;
      d2i = + c3r + c2i;
      d3r = + c3i + c2r;
      d3i = - c3r + c2i;
      vOut.re[c1*0 + k2] = d0r;
      vOut.im[c1*0 + k2] = d0i;
      vOut.re[c1*1 + k2] = d1r;
      vOut.im[c1*1 + k2] = d1i;
      vOut.re[c1*2 + k2] = d2r;
      vOut.im[c1*2 + k2] = d2i;
      vOut.re[c1*3 + k2] = d3r;
      vOut.im[c1*3 + k2] = d3i;
   }
}
```

4.3.3 FFT8_0Weights

`FFT8_0Weights` implements `FFT_Butterflies` with `m`=3 and `k0`=0. It may be said that the calculations for a weightless radix-8 butterfly are both complicated and straightforward, as they are very symmetric yet intricate:

FFT8_0Weights

```
static void FFT8_0Weights(
   ComplexArray vOut,   // Address of output vector.
   ComplexArray vIn,    // Address of input vector.
   int c0               // Coefficient for k0.
)
{
   // Prepare a constant, sqrt(2)/2.
   const float sqrt2d2 = .7071067811865475244;
   // Coefficient for k1 is coefficient for k0 divided by 1<<m.
   const int c1 = c0 >> 3;
   int k2;
   float a0r, a0i, a1r, a1i, a2r, a2i, a3r, a3i,
         a4r, a4i, a5r, a5i, a6r, a6i, a7r, a7i,
         b0r, b0i, b1r, b1i, b2r, b2i, b3r, b3i,
         b4r, b4i, b5r, b5i, b6r, b6i, b7r, b7i,
         c0r, c0i, c1r, c1i, c2r, c2i, c3r, c3i,
         c4r, c4i, c5r, c5i, c6r, c6i, c7r, c7i,
         d0r, d0i, d1r, d1i, d2r, d2i, d3r, d3i,
         d4r, d4i, d5r, d5i, d6r, d6i, d7r, d7i,
         t5r, t5i, t7r, t7i;

   for (k2 = 0; k2 < c1; ++k2)
   {
      a0r = vIn.re[c1*0 + k2];
      a0i = vIn.im[c1*0 + k2];
      a1r = vIn.re[c1*1 + k2];
      a1i = vIn.im[c1*1 + k2];
      a2r = vIn.re[c1*2 + k2];
      a2i = vIn.im[c1*2 + k2];
      a3r = vIn.re[c1*3 + k2];
      a3i = vIn.im[c1*3 + k2];
      a4r = vIn.re[c1*4 + k2];
      a4i = vIn.im[c1*4 + k2];
      a5r = vIn.re[c1*5 + k2];
      a5i = vIn.im[c1*5 + k2];
```

```
        a6r = vIn.re[c1*6 + k2];
        a6i = vIn.im[c1*6 + k2];
        a7r = vIn.re[c1*7 + k2];
        a7i = vIn.im[c1*7 + k2];
        b0r = a0r + a4r;             // w = 1.
        b0i = a0i + a4i;
        b1r = a1r + a5r;
        b1i = a1i + a5i;
        b2r = a2r + a6r;
        b2i = a2i + a6i;
        b3r = a3r + a7r;
        b3i = a3i + a7i;
        b4r = a0r - a4r;
        b4i = a0i - a4i;
        b5r = a1r - a5r;
        b5i = a1i - a5i;
        b6r = a2r - a6r;
        b6i = a2i - a6i;
        b7r = a3r - a7r;
        b7i = a3i - a7i;
        c0r = b0r + b2r;             // w = 1.
        c0i = b0i + b2i;
        c1r = b1r + b3r;
        c1i = b1i + b3i;
        c2r = b0r - b2r;
        c2i = b0i - b2i;
        c3r = b1r - b3r;
        c3i = b1i - b3i;
        c4r = b4r - b6i;             // w = i.
        c4i = b4i + b6r;
        c5r = b5r - b7i;
        c5i = b5i + b7r;
        c6r = b4r + b6i;
        c6i = b4i - b6r;
        c7r = b5r + b7i;
        c7i = b5i - b7r;
        t5r = c5r - c5i;
        t5i = c5r + c5i;
        t7r = c7r + c7i;
        t7i = c7r - c7i;
        d0r = c0r + c1r;             // w = 1.
        d0i = c0i + c1i;
        d1r = c0r - c1r;
        d1i = c0i - c1i;
        d2r = c2r - c3i;             // w = i.
        d2i = c2i + c3r;
        d3r = c2r + c3i;
        d3i = c2i - c3r;
        d4r = + t5r * sqrt2d2 + c4r;  // w = sqrt(2)/2 * (+1+i).
        d4i = + t5i * sqrt2d2 + c4i;
        d5r = - t5r * sqrt2d2 + c4r;
        d5i = - t5i * sqrt2d2 + c4i;
        d6r = - t7r * sqrt2d2 + c6r;  // w = sqrt(2)/2 * (-1+i).
        d6i = + t7i * sqrt2d2 + c6i;
        d7r = + t7r * sqrt2d2 + c6r;
        d7i = - t7i * sqrt2d2 + c6i;
        vOut.re[c1*0 + k2] = d0r;
```

```
      vOut.im[c1*0 + k2] = d0i;
      vOut.re[c1*1 + k2] = d1r;
      vOut.im[c1*1 + k2] = d1i;
      vOut.re[c1*2 + k2] = d2r;
      vOut.im[c1*2 + k2] = d2i;
      vOut.re[c1*3 + k2] = d3r;
      vOut.im[c1*3 + k2] = d3i;
      vOut.re[c1*4 + k2] = d4r;
      vOut.im[c1*4 + k2] = d4i;
      vOut.re[c1*5 + k2] = d5r;
      vOut.im[c1*5 + k2] = d5i;
      vOut.re[c1*6 + k2] = d6r;
      vOut.im[c1*6 + k2] = d6i;
      vOut.re[c1*7 + k2] = d7r;
      vOut.im[c1*7 + k2] = d7i;
   }
}
```

For readers who wish to analyze the radix-8 butterfly code, it is structured as a sequence of three radix-2 passes. The comments show the value of ω where each iteration on k_0 begins.

Discussion of the derivation of the above code is beyond the scope of this paper. *Maple* code that generates the assignment statements is given in appendix A.

4.3.4 FFT4_1WeightPerIteration

`FFT_1WeightPerIteration` implements a loop on `k0` calling `FFT_Butterflies` with `m`=2, `vIn`=`Vout`, and `c1`=4 and with an array of weight values provided so that it need not calculate them. `FFT_1WeightPerIteration` compute the same results as:

```
for (k0 = 0; k0 < 1<<N-4; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 16);
```

Here is a simple implementation:

```
static void FFT4_1WeightPerIteration(
   ComplexArray vOut,              // Address of output vector.
   int u0,                         // Upper bound on k0.
   const CommonWeight weights[]    // Array of weight values.
)
{
   int j1, k0, k1, k2;

   for (k0 = 0; k0 < u0; ++k0)
   for (k2 = 0; k2 < 4 ; ++k2)
   for (k1 = 0; k1 < 4 ; ++k1)
   {
      complex sum = 0.;
      for (j1 = 0; j1 < 4; ++j1)
         sum += one(j1*r(k1)) * one(j1*r(4*k0)) *
            vOut[16*k0 + 4*j1 + k2];
      vOut[16*k0 + 4*k1 + k2] = sum;
   }
}
```

Here is an implementation using Goedecker’s algorithm:

FFT4_1WeightPerIteration

```
static void FFT4_1WeightPerIteration(
   ComplexArray vOut,              // Address of output vector.
   int u0,                         // Upper bound on k0.
   const CommonWeight weights[]    // Array of weight values.
)
{
   int k0, k2;
   float a0r, a0i, a1r, a1i, a2r, a2i, a3r, a3i,
                   b1r, b1i, b2r, b2i, b3r, b3i,
         c0r, c0i, c1r, c1i, c2r, c2i, c3r, c3i,
         d0r, d0i, d1r, d1i, d2r, d2i, d3r, d3i;

   for (k0 = 0; k0 < u0; ++k0)
   {
      // Load values for current weight.
      CommonWeight weight = weights[k0];

      for (k2 = 0; k2 < 4 ; ++k2)
      {
         a0r = vOut.re[16*k0 + 4*0 + k2];
         a0i = vOut.im[16*k0 + 4*0 + k2];
         a1r = vOut.re[16*k0 + 4*1 + k2];
         a1i = vOut.im[16*k0 + 4*1 + k2];
         a2r = vOut.re[16*k0 + 4*2 + k2];
         a2i = vOut.im[16*k0 + 4*2 + k2];
         a3r = vOut.re[16*k0 + 4*3 + k2];
         a3i = vOut.im[16*k0 + 4*3 + k2];
         b1r = - a1i * weight.w1i + a1r;
         b1i = + a1r * weight.w1i + a1i;
         b2r = - a2i * weight.w2i + a2r;
         b2i = + a2r * weight.w2i + a2i;
         b3r = - a3i * weight.w3i + a3r;
         b3i = + a3r * weight.w3i + a3i;
         c0r = + b2r * weight.w2r + a0r;
         c0i = + b2i * weight.w2r + a0i;
         c2r = - b2r * weight.w2r + a0r;
         c2i = - b2i * weight.w2r + a0i;
         c1r = + b3r * weight.w3r + b1r;
         c1i = + b3i * weight.w3r + b1i;
         c3r = - b3r * weight.w3r + b1r;
         c3i = - b3i * weight.w3r + b1i;
         d0r = + c1r * weight.w1r + c0r;
         d0i = + c1i * weight.w1r + c0i;
         d1r = - c1r * weight.w1r + c0r;
         d1i = - c1i * weight.w1r + c0i;
         d2r = - c3i * weight.w1r + c2r;
         d2i = + c3r * weight.w1r + c2i;
         d3r = + c3i * weight.w1r + c2r;
         d3i = - c3r * weight.w1r + c2i;
         vOut.re[16*k0 + 4*0 + k2] = d0r;
```

```
            vOut.im[16*k0 + 4*0 + k2] = d0i;
            vOut.re[16*k0 + 4*1 + k2] = d1r;
            vOut.im[16*k0 + 4*1 + k2] = d1i;
            vOut.re[16*k0 + 4*2 + k2] = d2r;
            vOut.im[16*k0 + 4*2 + k2] = d2i;
            vOut.re[16*k0 + 4*3 + k2] = d3r;
            vOut.im[16*k0 + 4*3 + k2] = d3i;
         }
      }
   }
```

Note that when this is implemented with AltiVec instructions, the loop on `k2` will vanish, as all four iterations of the loop are performed by a single iteration of AltiVec instructions, as indicated by the name `FFT4_1WeightPerIteration`.

4.3.5 FFT4_Final

`FFT_1Final` implements a loop on `k0` calling `FFT_Butterflies` with `m`=2, `vIn=Vout`, and `c1`=1 and with an array of weight values provided so that it need not calculate them. `FFT_Final` should compute the same results as:

```
for (k0 = 0; k0 < 1<<N-2; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 4);
```

Here is a simple implementation:

```
static void FFT4_Final(
   ComplexArray vOut,               // Address of output vector.
   int u0,                          // Upper bound on k0.
   const CommonWeight weights[]     // Array of weight values.
)
{
   int j1, k0, k1, k2;

   for (k0 = 0; k0 < u0; ++k0)
   for (k2 = 0; k2 < 1 ; ++k2)
   for (k1 = 0; k1 < 4 ; ++k1)
   {
      complex sum = 0.;
      for (j1 = 0; j1 < 4; ++j1)
         sum += one(j1*r(k1)) * one(j1*r(4*k0)) *
            vOut[4*k0 + j1 + k2];
      vOut[4*k0 + k1 + k2] = sum;
   }
}
```

We can reduce this further since `k2` is always zero:

```
static void FFT4_Final(
   ComplexArray vOut,               // Address of output vector.
   int u0,                          // Upper bound on k0.
   const CommonWeight weights[]     // Array of weight values.
)
```

```
{
   int j1, k0, k1;

   for (k0 = 0; k0 < u0; ++k0)
   for (k1 = 0; k1 < 4 ; ++k1)
   {
      complex sum = 0.;
      for (j1 = 0; j1 < 4; ++j1)
         sum += one(j1*r(k1)) * one(j1*r(4*k0)) *
            vOut[4*k0 + j1];
      vOut[4*k0 + k1] = sum;
   }
}
```

Here is an implementation using Goedecker's algorithm:

FFT4_Final

```
static void FFT4_Final(
   ComplexArray vOut,               // Address of output vector.
   int u0,                          // Upper bound on k0.
   const CommonWeight weights[]     // Array of weight values.
)
{
   int k0;
   float a0r, a0i, a1r, a1i, a2r, a2i, a3r, a3i,
                   b1r, b1i, b2r, b2i, b3r, b3i,
         c0r, c0i, c1r, c1i, c2r, c2i, c3r, c3i,
         d0r, d0i, d1r, d1i, d2r, d2i, d3r, d3i;

   for (k0 = 0; k0 < u0; ++k0)
   {
      // Load values for current weight.
      CommonWeight weight = weights[k0];

      a0r = vOut.re[4*k0 + 0];
      a0i = vOut.im[4*k0 + 0];
      a1r = vOut.re[4*k0 + 1];
      a1i = vOut.im[4*k0 + 1];
      a2r = vOut.re[4*k0 + 2];
      a2i = vOut.im[4*k0 + 2];
      a3r = vOut.re[4*k0 + 3];
      a3i = vOut.im[4*k0 + 3];
      b1r = - a1i * weight.w1i + a1r;
      b1i = + a1r * weight.w1i + a1i;
      b2r = - a2i * weight.w2i + a2r;
      b2i = + a2r * weight.w2i + a2i;
      b3r = - a3i * weight.w3i + a3r;
      b3i = + a3r * weight.w3i + a3i;
      c0r = + b2r * weight.w2r + a0r;
      c0i = + b2i * weight.w2r + a0i;
      c2r = - b2r * weight.w2r + a0r;
      c2i = - b2i * weight.w2r + a0i;
      c1r = + b3r * weight.w3r + b1r;
      c1i = + b3i * weight.w3r + b1i;
      c3r = - b3r * weight.w3r + b1r;
      c3i = - b3i * weight.w3r + b1i;
```

```
        d0r = + c1r * weight.w1r + c0r;
        d0i = + c1i * weight.w1r + c0i;
        d1r = - c1r * weight.w1r + c0r;
        d1i = - c1i * weight.w1r + c0i;
        d2r = - c3i * weight.w1r + c2r;
        d2i = + c3r * weight.w1r + c2i;
        d3r = + c3i * weight.w1r + c2r;
        d3i = - c3r * weight.w1r + c2i;
        vOut.re[4*k0 + 0] = d0r;
        vOut.im[4*k0 + 0] = d0i;
        vOut.re[4*k0 + 1] = d1r;
        vOut.im[4*k0 + 1] = d1i;
        vOut.re[4*k0 + 2] = d2r;
        vOut.im[4*k0 + 2] = d2i;
        vOut.re[4*k0 + 3] = d3r;
        vOut.im[4*k0 + 3] = d3i;
    }
}
```

4.3.5.1 AltiVec Implementation

Readers familiar with the AltiVec architecture will appreciate that previous butterfly routines are nearly ideal for AltiVec implementation. `FFT4_Final` presents some interesting problems, though. Consider the symbols `a0r`, `a1r`, `a2i`, and `a3i`. The values for these symbols are read from array elements with indices `4*k0+0`, `4*k0+1`, `4*k0+2`, and `4*k0+3`. These elements are adjacent to each other, and AltiVec instructions provide no good way to perform arithmetic on adjacent elements. Additional instructions must be used to move the elements around within the processor registers.

This problem interacts fortuitously with another problem. The FFT finishes with its elements permuted from the desired order. That is, the FFT procedure returns $\mathbf{v}_N$, which is the bit-reversal permutation of $\mathbf{H}$. After $\mathbf{v}_N$ is computed, we would like to rearrange the array elements into the desired order. This rearrangement also requires moving elements within processor registers. As it happens, quite to our benefit, the same rearrangements serve both to provide the desired order and to arrange the elements conveniently for high-performance calculation.

However, this rearrangement changes the order in which we process elements, with consequences to the weight array. We could use the same weight array as is used in other routines, but the calculations of the memory addresses of the weights will be more complicated, and we will need to rearrange the weights within the processor registers to match the data. The FFT can be performed faster if the weights are prearranged as needed.

This is discussed further in section 6.4.

5 Generating Weights

The butterfly routines need prepared weights. Here is code to generate them.

5.1 Prerequisites

A simple constant is used:

TwoPi

```
static const double TwoPi = 2 * 3.1415926535897932384626433;
```

The values needed to perform a butterfly with one weight can be stored in this structure:

CommonWeight

```
typedef struct {
   float w1r, w1i, w2r, w2i, w3r, w3i;
} CommonWeight;
```

5.2 Subroutines

The weight-generation routines need some subroutines. Here is a subroutine to calculate the integer base-two logarithm of *n*, that is $\lfloor \log_2 n \rfloor$:

ilog2

```
static inline int ilog2(unsigned int n)
{
   int c;
   for (c = 0; n >>= 1; ++c)
      ;
   return c;
}
```

With GCC and a PowerPC execution target, the same function may be implemented more efficiently with the routine below. Many processors have an instruction similar to `cntlzw`, which counts the number of leading zero bits in a word (of 32 bits).

```
static inline int ilog2(unsigned int n)
{
   int c;
   asm("cntlzw %0, %1; subfic %0, %0, 31" : "=r"(c) : "r"(n) );
   return c;
}
```

A method is needed to calculate bit-reversals. The following routine calculates the bit-reversal of a 32-bit number by reversing the bit-reversals of its four eight-bit bytes, which are looked up in a table. The table is generated with the code in section B.4.

rw, Reverse Word

```
static unsigned int rw(unsigned int k)
{
   static const unsigned char b[256] = {
  0, 128,  64, 192,  32, 160,  96, 224,  16, 144,  80, 208,  48, 176, 112, 240,
  8, 136,  72, 200,  40, 168, 104, 232,  24, 152,  88, 216,  56, 184, 120, 248,
  4, 132,  68, 196,  36, 164, 100, 228,  20, 148,  84, 212,  52, 180, 116, 244,
 12, 140,  76, 204,  44, 172, 108, 236,  28, 156,  92, 220,  60, 188, 124, 252,
  2, 130,  66, 194,  34, 162,  98, 226,  18, 146,  82, 210,  50, 178, 114, 242,
 10, 138,  74, 202,  42, 170, 106, 234,  26, 154,  90, 218,  58, 186, 122, 250,
  6, 134,  70, 198,  38, 166, 102, 230,  22, 150,  86, 214,  54, 182, 118, 246,
 14, 142,  78, 206,  46, 174, 110, 238,  30, 158,  94, 222,  62, 190, 126, 254,
  1, 129,  65, 193,  33, 161,  97, 225,  17, 145,  81, 209,  49, 177, 113, 241,
  9, 137,  73, 201,  41, 169, 105, 233,  25, 153,  89, 217,  57, 185, 121, 249,
  5, 133,  69, 197,  37, 165, 101, 229,  21, 149,  85, 213,  53, 181, 117, 245,
 13, 141,  77, 205,  45, 173, 109, 237,  29, 157,  93, 221,  61, 189, 125, 253,
  3, 131,  67, 195,  35, 163,  99, 227,  19, 147,  83, 211,  51, 179, 115, 243,
 11, 139,  75, 203,  43, 171, 107, 235,  27, 155,  91, 219,  59, 187, 123, 251,
  7, 135,  71, 199,  39, 167, 103, 231,  23, 151,  87, 215,  55, 183, 119, 247,
 15, 143,  79, 207,  47, 175, 111, 239,  31, 159,  95, 223,  63, 191, 127, 255
   };
   unsigned char
      b0 = b[k >> 0*8 & 0xff],
      b1 = b[k >> 1*8 & 0xff],
      b2 = b[k >> 2*8 & 0xff],
      b3 = b[k >> 3*8 & 0xff];
   return b0 << 3*8 | b1 << 2*8 | b2 << 1*8 | b3 << 0*8;
}
```

The function $r(k)$, which rotates bits around a ".", can be computed from `rw(k)`, which reverses bits in a 32-bit field, by shifting the bits right 32 bits in floating-point:

r, Calculate *r*(*k*)

```
static float r(unsigned int k)
{
   return 1./4294967296. * rw(k);
}
```

5.3 Generate Common Weights

The routine below generates the array of common weights.

The numbers stored in each array element are the numbers needed for Goedecker's algorithm, described in section 4.2.1. In each iteration the weight is $\omega = \mathbf{1}^{r\left(2^m k_0\right)}$, from Equation (6) in section 2.3. Since we use m=2, we have $\omega = \mathbf{1}^{r(4k_0)}$. Thus `r(4*k0)` is used in the code below to generate the numbers.

The caller of this routine passes the length of the vector to be transformed. This is the number of elements in the vector to be transformed, not the number of elements in the weight array.

This routine generates only $2^N/16$ weights. In full, $2^N/4$ weights are needed, as the upper limit on k_0 in the final pass is $2^N/4$, as readily seen in the call to `FFT4_Final` in section 3.3.6. Indeed, the loop condition below should be "`k0 < n/4`" to support the kernel as written so far. However, in section 6.4, we will modify `FFT4_Final` in ways that preclude it from using the common weights. It will get its own weights array, and the common array generated here will be used only by the other butterfly routines. In this case, the greatest upper limit on k_0 is $2^N/16$, as seen in the call to `FFT4_1WeightPerIteration` in section 3.3.6.

Here is the routine. See comments below about the arguments.

GenerateCommonWeights

```
static int GenerateCommonWeights(
   CommonWeight **weights,    // Pointer to array address.
   int *length,               // Pointer to supported length.
   int NewLength              // New length to support (1<<N).
)
{
   int k0;

   // Try to allocate space and check result.
   CommonWeight *p = (CommonWeight *)
      realloc(*weights, NewLength/16 * sizeof **weights);
   if (p == NULL)
      return 1;

   for (k0 = *length/16; k0 < NewLength/16; ++k0)
   {
      const double x = TwoPi * r(4*k0);
      p[k0].w1r = cos(x);
      p[k0].w1i = tan(x);
      p[k0].w2r = cos(x+x);
      p[k0].w2i = tan(x+x);
      p[k0].w3r = 2. * p[k0].w2r – 1.;
      p[k0].w3i = tan(3.*x);
   }

   // Pass address and supported length back to caller.
   *weights = p;
   *length = NewLength;

   return 0;
}
```

This routine could simply take a vector length as input and return an array of weight values. However, to facilitate operations by the caller, it provides services to alter an existing array and to record the supported length. In addition to the vector length to be supported, the routine is passed two pointers. The first gives the location where an existing weight array is stored, which may be `NULL`. The second gives the location where the length associated with the existing array is stored.

This routine then uses `realloc` to get the space it needs. This will newly allocate (if the pointer is `NULL`) or reallocate memory. The routine then fills in elements that were not in the previous array (if any). It may be used to shorten an array but is more commonly used to create an array or lengthen an existing array.

6 More Kernel Changes

The code described in previous sections will provide a high-performance FFT, but we can still do better.

6.1 Group Butterflies by Weight

In our latest FFT kernel (section 3.3.6), the second set of loops performs general butterflies with one weight per call:

```
for (p  = 1; p  < P-2    ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT4_1WeightPerCall(vOut, k0, 1<<N-n[p], weights[k0]);
```

Implicit in this call is that the contents of `weights[k0]` are read from memory into registers. There is one such read for every iteration on `k0`, and iterations on `k0` are repeated in subsequent iterations on `p`. We can eliminate some of the reads by iterating on `k0` first and then on `p`, that is, by swapping the order of the loops.

6.1.1 Calculate New Loop Bounds

The inner body of the two loops is executed a number of times, each time with a pair of values for `p` and `k0`. Consider the set of all such pairs. The current code executes the body with each of those pairs, in a certain order.

Our goal is to execute the body with the same set of pairs, but in a different order. To do rearrange the loops and get the same pairs, we must calculate new bounds on the variable used in each loop. Since the upper bound on `k0` depends on `p`, this requires some mathematics.

The existing code shows us trivially that the set of pairs (p, k_0) for which the body is executed contains those pairs satisfying $1 \le p < P - 2$ and $0 \le k_0 < 2^{n_p}$.

n_p increases strictly as p increases, so p<P-2 and $k_0 < 2^{n_p}$ imply p is at most P-3 and therefore $k_0 < 2^{n_{P-3}}$. So we can say $0 \le k_0 < 2^{n_{P-3}}$, and that gives us bounds for an outer loop on `k0`. Next we consider the bounds for an inner loop `p`. Those bounds must depend on the value of `k0`.

It can be shown that the loop bounds for p are $\max\left(1, \left\lfloor \left(\log_2 k_0 + 4 - m_0\right)/2 \right\rfloor\right) \le p < P - 2$. However, it is simpler to keep the lower bound for p in an auxiliary variable `pLower` and increase `pLower` whenever the constraint $k_0 < 2^{n_p}$ is violated:

```
pLower = 1;
for (k0 = 0; k0 < 1<<n[P-3]; ++k0)
{
   if (! (k0 < 1<<n[pLower]) )
      ++pLower;
   for (p = pLower; p < P-2; ++p)
      FFT4_1WeightPerCall(vOut, k0, 1<<N-n[p], weights[k0]);
}
```

Observe that the initial values for `pLower` and `k0` satisfy $k_0 < 2^{n_p}$, because $0 < 2^{n_1}$, and a single increment to `pLower` when the constraint is violated suffices to restore it because `k0` never increases by more than one per iteration.

6.1.2 Check the New Calculation Order

We have reordered the calculations and should ensure that we have not violated the necessary order. The problem may be phrased in the following way. Consider an element in `vOut` with index *k*. When `p` is 0, this element will be read once, used in calculations, and then written once. This will occur again when `p` is 1, 2, 3, and so on. These uses of the element must occur in that order, so that when it is read for `p`=*p*, it contains the result calculated when `p`=*p*-1. How do we know the new loop order satisfies this?

The element with index *k* is read once and written once per value of `p`, specifically when `k0` has the value $\lfloor k/2^{N-n_p} \rfloor$. As *p* increases, the values of `k0` form a non-decreasing sequence. Then we can easily see that sorting the pairs of values (`p`, `k0`) lexicographically first by `p` and then by `k0` (the original loop order, iterating on `p` and then `k0`) yields the same order as sorting the pairs first by `k0` and then by `p` (the new loop order). Thus although the references to different array elements have been reordered, the references to any one array element *k* are in the same correct order as they were originally.

6.1.3 Optimize the Code

A few calculations can be saved by creating an auxiliary loop to handle the increments to `pLower`. The code above evaluates "`k0 < 1<<n[pLower]`" and "`k0 < 1<<n[P-3]`" in each iteration on `k0`. As long as `pLower < P-2`, the former implies the latter, so, each time `k0` changes, we need test only the former. When it fails, then we will increment `pLower`, and we must test the latter. Observe that "`k0 < 1<<n[pLower]`" fails just as we have incremented `k0` to the value `1<<n[pLower]`. If "`pLower < P-2`", then "`k0 < 1<<n[P-3]`", and vice-versa, so we can use "`pLower < P-2`" as our test:

```
for (pLower = 1, k0 = 0; pLower < P-2         ; ++pLower)
for (                  ; k0     < 1<<n[pLower]; ++k0    )
for (p = pLower        ; p      < P-2         ; ++p     )
   FFT4_1WeightPerCall(vOut, k0, 1<<N-n[p], weights[k0]);
```

To summarize, the new loops above execute all the same calls to `FFT4_1WeightPerCall` as the original code (repeated below for reference) but in a different and more efficient order.

```
for (p  = 1; p  < P-2    ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT4_1WeightPerCall(vOut, k0, 1<<N-n[p], weights[k0]);
```

6.2 Separate the Weightless Butterflies

We used special butterfly routines for the first pass because there is a significant gain from eliminating multiplications when weights are not needed. Now that we have rearranged the second set of loops in the kernel, we have again grouped together a set of butterflies in which k_0 is zero. We again separate these from the rest:

```
pLower = 1;
for (p = pLower; p      < P-2         ; ++p     )
   FFT4_0Weights(vOut, vOut, 1<<N-n[p]);

for (k0 = 1    ; pLower < P-2         ; ++pLower)
for (          ; k0     < 1<<n[pLower]; ++k0    )
for (p = pLower; p      < P-2         ; ++p     )
   FFT4_1WeightPerCall(vOut, k0, 1<<N-n[p], weights[k0]);
```

The first loop is always executed and is no longer guarded by the loop tests "`pLower < P-2`" or "`k0 < 1<<n[pLower]`". However, the former is implied by "`p < P-2`", which is evaluated, and the latter is true because `k0` is implicitly 0, so this separation of the first loop is safe.

6.2.1 Create A Variant of FFT4_0Weights

`FFT4_0Weights` has both an input array and an output array as arguments. We only need one array in this instance and could use another specialization of the routine. The performance gain is likely to be slight or zero, as the address calculations for the second array might be computed entirely in parallel with the floating-point data calculations.

However, there may be a more important reason for using a separate variant of this routine. The initial pass is an opportune place to perform additional processing, such as rearranging the data in memory so that it is arranged in a way that is efficient for the remaining routines. In such a case, you will need a variant of `FFT4_0Weights` that does the additional processing and another variant that does not do the additional processing.

6.3 Update the Kernel

Our FFT kernel now is:

```
if (N & 1)
   FFT8_0Weights(vOut, vIn, 1<<N);
else
   FFT4_0Weights(vOut, vIn, 1<<N);

pLower = 1;
for (p = pLower; p      < P-2         ; ++p     )
   FFT4_0Weights(vOut, vOut, 1<<N-n[p]);

for (k0 = 1    ; pLower < P-2         ; ++pLower)
for (          ; k0     < 1<<n[pLower]; ++k0    )
for (p = pLower; p      < P-2         ; ++p     )
   FFT4_1WeightPerCall(vOut, k0, 1<<N-n[p], weights[k0]);

if (p < P-1)
   FFT4_1WeightPerIteration(vOut, 1<<N-4, weights);

FFT4_Final(vOut, 1<<N-2, weights);
```

This code refers to `n[p]`, representing n_p. We do not actually need an array to hold values of n_p; we can calculate them. There is a 1-1 map between p and n_p, and the operations we use on them are isomorphic under the map. (Notably, inequalities involving p are isomorphic because n_p is a strictly increasing function of p.) So every reference to p may be replaced by an equivalent reference to n_p.

We will replace all references to `p`, `pLower`, and `n[p]` by equivalent expressions of new variables `n` and `nLower`. `n` will contain the value previously expressed by `n[p]`, and `nLower` will contain the value previously expressed by `n[pLower]`. The substitutions to make are:

- `pLower = 1` becomes `nLower = N&1 ? 3 : 2`.
- `p = pLower` becomes `n = nLower`.
- `p < P-2` becomes `n < N-4`.
- `++p` becomes `n += 2`.
- `++pLower` becomes `nLower += 2`.
- `n[p]` becomes `n`.
- `n[pLower]` becomes `nLower`.

The new code is:

FFT Kernel with Reordered Loops and Separated Loop for k_0=0

```
if (N & 1)
   FFT8_0Weights(vOut, vIn, 1<<N);
else
   FFT4_0Weights(vOut, vIn, 1<<N);

nLower = N&1 ? 3 : 2;
for (n = nLower; n      < N-4      ; n += 2     )
   FFT4_0Weights(vOut, vOut, 1<<N-n);

for (k0 = 1    ; nLower < N-4      ; nLower += 2)
for (          ; k0     < 1<<nLower; ++k0       )
for (n = nLower; n      < N-4      ; n += 2     )
   FFT4_1WeightPerCall(vOut, k0, 1<<N-n, weights[k0]);

if (n < N-2)
   FFT4_1WeightPerIteration(vOut, 1<<N-4, weights);

FFT4_Final(vOut, 1<<N-2, weights);
```

With all references to p gone, the entire FFT structure is now built into the kernel. We could have made these substitutions earlier, but the reasoning in section 6.1.1 depends on p being an integer and would be harder to express in terms of n. Also, these substitutions specialize the kernel for a particular scheme of n’s. By developing the kernel to this point before making the substitutions, it could instead be specialized to other schemes.

6.4 Incorporate Bit-Reversal Permutation

The result of the FFT, $\mathbf{v}_N$, is the bit-reversal permutation of the desired result, **H**. (This is demonstrated in section 2.3). As mentioned in section 4.3.5.1, the final pass is an opportune place to rearrange the results in memory to produce **H** instead of $\mathbf{v}_N$. Here is one scheme for doing so.

6.4.1 Read Groups of Elements and Write in Bit-Reversed Locations

To do one butterfly, `FFT4_Final` reads and writes four elements with indices $4k_0+k_1$ for the four values of k_1, 0, 1, 2, and 3. In the final pass, $0 \le k_0 < N/4$ (as seen by the fact that `FFT4_Final` is called with `1<<N-2` as the upper bound on `k0`). Essentially, k_0 has N-2 bits in the final pass. Separate k_0 into its highest two bits, k_H, and its remaining N-4 low bits, k_L, so $k_0=2^{N-4}k_\mathrm{H}+k_\mathrm{L}$.

Let k'_H be the bit-reversal of the two bits of k_H. Let k'_L be the bit-reversal of the N-4 bits of k_L. Let k'_1 be the bit-reversal of the two bits of k_1. Observe that the bit-reversal of the N-bit number $4(2^{N-4}k_\mathrm{H}+k_\mathrm{L})+k_1$ is $4\left(2^{N-4}k'_1+k'_\mathrm{L}\right)+k'_\mathrm{H}$.

`FFT4_Final` iterates through all values of k_0, performing one butterfly on four elements in each iteration. Instead, iterate through values of k_L, performing four butterflies on 16 elements in each iteration.

Specifically, in each iteration, read the 16 elements with indices $4(2^{N-4}k_\mathrm{H}+k_\mathrm{L})+k_1$. Perform four butterflies on these elements, with the appropriate four weights. Write the results to the 16 array elements with indices $4\left(2^{N-4}k'_1+k'_\mathrm{L}\right)+k'_\mathrm{H}$. That is, write the result with index $4(2^{N-4}k_\mathrm{H}+k_\mathrm{L})+k_1$ in $\mathbf{v}_N$ to the bit-reversed index $4\left(2^{N-4}k'_1+k'_\mathrm{L}\right)+k'_\mathrm{H}$, which is its desired location in **H**.

When the iterations are completed, the output array will contain results in the order desired, matching **H** rather than $\mathbf{v}_N$.

6.4.2 Problems

Attempting to do this in-place will destroy the array, because k'_L will in many iterations be a value that k_L has not yet reached. Then data needed in the future is overwritten. An easy solution is to use a separate array for output in the final pass, if memory is available. Another solution is to read the 16 old elements just before we overwrite them with new results. Doing that presents another problem: What do we do with the 16 elements just read? It also presents an opportunity: Make use of them. First, it will help to define some terminology.

6.4.3 Terminology

Let the term "k_L-elements" refer to the 16 elements that are indexed by using k_L and the 16 combinations of values of k_H and k_1. That is, the k_L-elements are those whose indices in the array are:

$$\{4(2^{N-4} k_H + k_L) + k_1 \mid 0 \le k_1 < 4 \wedge 0 \le k_H < 4\}.$$

Such an index is essentially the bit-wise concatenation of k_H, k_L, and k_1.

Let the term "k_L-reversed-elements" refer to the 16 elements that are indexed by using the bit-reversal of k_L and the 16 combinations of the bit-reversals of the values of k_H and k_1. That is, the k_L-reversed-elements are those whose indices in the array are:

$$\{4(2^{N-4} k_1' + k_L') + k_H' \mid 0 \le k_1 < 4 \wedge 0 \le k_H < 4\}.$$

Similarly, one of these indices is the bit-wise concatenation of k_1', k_L', and k_H'.

Observe that the k_L-reversed-elements are also the k_L'-elements. That is:

$$\{4(2^{N-4} k_1' + k_L') + k_H' \mid 0 \le k_1 < 4 \wedge 0 \le k_H < 4\} =$$
$$\{4(2^{N-4} k_1 + k_L') + k_H \mid 0 \le k_1 < 4 \wedge 0 \le k_H < 4\}.$$

This is a subtle statement, for the sets look very similar, so it is unsurprising that they are equal. It embodies the fact that the set of values {0, 1, 2, 3} for k_1 equals the set of bit-reversed values {0, 2, 1, 3} for k_H' and vice-versa. It is important because it means that the 16 old k_L-reversed-elements we read just before overwriting them are precisely the k_L'-elements we can use for new butterfly operations.

6.4.4 Solution

We are ready to redesign `FFT4_Final` to perform butterflies and permute the results efficiently. After processing some k_L-elements, we will read k_L'-elements and process those. When those are done, they are stored in the k_L-elements. At that point, the k_L-elements were already done, so we are free to go on to a new value of k_L.

If we write `FFT4_Final` with a loop whose body performs four butterflies on 16 elements, there are three cases to distinguish in each iteration:

- $k_L = k_L'$. The k_L-elements are the k_L'-elements, so there is no need to read the k_L'-elements and perform more butterflies. We just go on to a new value of k_L.
- $k_L \ne k_L'$ and we have just done the k_L-elements. We must read the k_L'-elements for the next iteration.
- $k_L \ne k_L'$ and we have just done the k_L'-elements. We must go on to a new value of k_L.

Going on to a new value of k_L is a problem, as we must skip elements that were already processed when k_L' indexed those elements in prior iterations.

Fortunately, the three cases can all be implemented in simple code that uses a table to determine which location to read next and which location to write next. Using a table both eliminates the computation of bit-reversals during the FFT execution and eliminates testing and branching to handle separate cases.

We will prepare a table that contains values of k_L in the order we would like to process them and the corresponding values of k_L'. Like the weights, this table can be prepared before the first FFT is executed.

Consider this pseudo-code:

```
q = 0;
Read $k_L$-elements using $k_L$ = IndexTable[q].read;
Perform butterflies on input to get output.
for (q = 1; q < 1<<N-4; ++q)
{
   Read $k_L$-elements using $k_L$ = IndexTable[q].read;
   Write $k_L$-reversed-elements using $k_L'$ = IndexTable[q-1].write;
   Perform butterflies on input to get output.
}
Write $k_L$-reversed-elements using $k_L'$ = IndexTable[q-1].write;
```

As discussed, this code reads data, performs butterflies, and then reads the next set of input before writing output. It reads the next set of input in each iteration. This is necessary in the second of the three cases above. It is unnecessary in the other cases but causes no harm.

In section 6.4.6, I demonstrate C code that is nearly identical to the pseudo-code:

```
q = 0;
ReadElements(IndexTable[q].read);
PerformButterflies(weights[q]);
for (q = 1; q < cH; ++q)
{
   ReadElements(IndexTable[q].read);
   WriteReversedElements(IndexTable[q-1].write);
   PerformButterflies(weights[q]);
}
WriteReversedElements(IndexTable[q-1].write);
```

What should be stored in the index table? We have two requirements:

- Each value of k_L such that $k_L = k_L'$ is stored as a single table entry, with the same value in the `read` and `write` members.
- Each value of k_L such that $k_L \neq k_L'$ must be stored as a pair of entries. In one entry, `read` contains k_L and `write` contains k_L'. In the other, `read` contains k_L' and `write` contains k_L. The order of these two entries does not matter.

Other than this, the table entries may be ordered as desired. Changing the order within these constraints will not alter the results that are computed, but it might change performance, as we will see in section 7.4.

6.4.5 Index Table Implementation

This routine generates a table of indices for the final pass. For definitions of the routines `rw` and `ilog2`, see section 5.2. See comments in section 5.3 about the arguments.

FinalIndices

```
typedef struct {
   unsigned short int read, write;
} FinalIndices;
```

GenerateFinalIndices

```
static int GenerateFinalIndices(
   FinalIndices **indices,    // Pointer to index array address.
   int NewLength              // New length to support (1<<N).)
{
   // Prepare to bit-reverse a number of N-4 bits (see below).
   const int shift = 32 – (ilog2(NewLength) – 4);
   int kL;

   // Try to allocate space and check result.
   FinalIndices *p = (FinalIndices *)
      realloc(*indices, NewLength/16 * sizeof **indices);
   if (p == NULL)
      return 1;

   // Pass address back to caller.
   *indices = p;

   // Iterate through all values of kL.
   for (kL = 0; kL < NewLength/16; ++kL)
   {
      // rw(kL) reverses kL as a 32-bit number.  To get it as
      // the reversal of an N-4 bit number, shift right to
      // remove 32-(N-4) bits.
      const int kLprime = rw(kL) >> shift;

      // If kLprime < kL, then kL in a previous iteration had the
      // value kLprime has now, and we do not want to repeat it.
      if (kL <= kLprime)
      {
         // If kL == kLprime, add one table entry.
         // If kL != kLprime, add table entries in both orders.
         *(p++)     = Construct( kL, kLprime );
         if (kL < kLprime)
             *(p++) = Construct( kLprime, kL );
      }
   }
   return 0;
}
```

The routine `Construct` used in the above code is used to construct a `FinalIndices` object. It is unneeded in C 1999 (ISO/IEC 9899-1999) but is needed by older compilers:

Construct

```
static FinalIndices Construct(unsigned int read, unsigned int write)
{
   FinalIndices result = { read, write };
   return result;
}
```

Using `short int` for the indices has some advantage in an AltiVec implementation, but it limits the vector length that the FFT can operate on. A `short int` is commonly 16 bits. Limiting k_L to 16 bits limits the entire index to 20 bits, so only vectors of up to 2^{20}=1,048,576 elements can be supported.

6.4.6 C Implementation

The C code fragment in section 6.4.4 will become the body of our new `FFT4_Final` routine.

6.4.6.1 FFT4_Final

Here is the new `FFT4_Final` routine. The weights required by this routine are described in section 6.4.8, and I add some code that will be explained below:

FFT4_Final With Bit-Reversal Permutation

```
static void FFT4_Final(
   ComplexArray vOut,               // Address of output vector.
   int u0,                          // Upper bound on k0.
   const FinalIndices IndexTable[], // Array of index pairs.
   const FinalWeights weights[]     // Array of weight values.
)
{
   typedef float FloatBlock[4];
   FloatBlock     a0r, a0i, a1r, a1i, a2r, a2i, a3r, a3i,
                            b1r, b1i, b2r, b2i, b3r, b3i,
                  c0r, c0i, c1r, c1i, c2r, c2i, c3r, c3i,
                  d0r, d0i, d1r, d1i, d2r, d2i, d3r, d3i;
   int q = 0;

   ReadElements(IndexTable[q].read);
   PerformButterflies(weights[q]);
   for (q = 1; q < u0 >> 2; ++q)
   {
      ReadElements(IndexTable[q].read);
      WriteReversedElements(IndexTable[q-1].write);
      PerformButterflies(weights[q]);
   }
   WriteReversedElements(IndexTable[q-1].write);
}
```

The various declarations above (such as `a0r`) are present even though they appear to be unused because I will use macros to show the operations in the routine, and the macros will use the declared identifiers. The macros expand to code in the context of the routine and have access to all of its identifiers. This is usually poor style for code to be used in actual programs, but it serves well here to illustrate the algorithm.

Note that `q` is iterated from zero to `u0>>2`. In the original version of `FFT4_Final`, in section 4.3.5, `u0` iterations were performed. In this new version, four butterflies are performed in each version, and so only `u0>>2` iterations are needed.

6.4.6.2 ReadElements

`ReadElements`, below, reads the k_L-elements. Previous radix-4 butterfly routines operated on four elements at a time, kept in objects of type `float`. We now do four butterflies on 16 elements (four sets of four) so we will keep them in objects of type "`float [4]`" and use array indices to access the elements within those objects.

We should study the array indices carefully. In the original version of `FFT4_Final`, in section 4.3.5, the index had the form "`4*k0 + k1`". In this version, we have separated `k0` into `kH` and `kL`. We defined k_H and k_L so that $k_0=2^{N-4}k_H+k_L$, so $4k_0+k_1$ becomes $2^{N-2}k_H+4k_L+k_1$. 2^{N-2} is passed to `FFT4_Final` in the parameter `u0` (see section 6.3). Thus, we may use the form "`u0*kH + 4*kL + k1`". For example, when k_H is 2 and k_1 is 1, the array index is "`u0*2 + 4*kL + 1`".

Observe that the real (or imaginary) components of the four elements associated with one value of k_H and four values of k_1 are placed by `ReadElements` one apiece into `a0r`, `a1r`, `a2r`, and `a3r`, in order and ready for butterfly calculations. However, the components associated with four values of k_H (0, 1, 2, and 3) and one value of k_1 are placed four apiece into one of the objects (`a0r`, `a1r`, `a2r`, or `a3r`) in bit-reversed order (0, 2, 1, and 3).

This does not affect the butterfly calculations (as long as the correct weight is used in each position). Each of the four butterflies operates on one element from `a0r`, one from `a1r`, one from `a2r`, and one from `a3r`, and the contents of other elements do not affect the butterfly. The advantage of putting the elements in this order is that they are then in the order in which they must be written to memory. That makes the write operations simpler.

ReadElements

```
#define ReadElements(kL)                                    \
{                                                           \
    a0r[0] = vOut.re[u0*0 + 4*kL + 0];                      \
    a1r[0] = vOut.re[u0*0 + 4*kL + 1];                      \
    a2r[0] = vOut.re[u0*0 + 4*kL + 2];                      \
    a3r[0] = vOut.re[u0*0 + 4*kL + 3];                      \
    a0r[1] = vOut.re[u0*2 + 4*kL + 0];                      \
    a1r[1] = vOut.re[u0*2 + 4*kL + 1];                      \
    a2r[1] = vOut.re[u0*2 + 4*kL + 2];                      \
    a3r[1] = vOut.re[u0*2 + 4*kL + 3];                      \
    a0r[2] = vOut.re[u0*1 + 4*kL + 0];                      \
    a1r[2] = vOut.re[u0*1 + 4*kL + 1];                      \
    a2r[2] = vOut.re[u0*1 + 4*kL + 2];                      \
    a3r[2] = vOut.re[u0*1 + 4*kL + 3];                      \
   a0r[3] = vOut.re[u0*3 + 4*kL + 0];                 \
   a1r[3] = vOut.re[u0*3 + 4*kL + 1];                 \
   a2r[3] = vOut.re[u0*3 + 4*kL + 2];                 \
   a3r[3] = vOut.re[u0*3 + 4*kL + 3];                 \
   a0i[0] = vOut.im[u0*0 + 4*kL + 0];                 \
   a1i[0] = vOut.im[u0*0 + 4*kL + 1];                 \
   a2i[0] = vOut.im[u0*0 + 4*kL + 2];                 \
   a3i[0] = vOut.im[u0*0 + 4*kL + 3];                 \
   a0i[1] = vOut.im[u0*2 + 4*kL + 0];                 \
   a1i[1] = vOut.im[u0*2 + 4*kL + 1];                 \
   a2i[1] = vOut.im[u0*2 + 4*kL + 2];                 \
   a3i[1] = vOut.im[u0*2 + 4*kL + 3];                 \
   a0i[2] = vOut.im[u0*1 + 4*kL + 0];                 \
   a1i[2] = vOut.im[u0*1 + 4*kL + 1];                 \
   a2i[2] = vOut.im[u0*1 + 4*kL + 2];                 \
   a3i[2] = vOut.im[u0*1 + 4*kL + 3];                 \
   a0i[3] = vOut.im[u0*3 + 4*kL + 0];                 \
   a1i[3] = vOut.im[u0*3 + 4*kL + 1];                 \
   a2i[3] = vOut.im[u0*3 + 4*kL + 2];                 \
   a3i[3] = vOut.im[u0*3 + 4*kL + 3];                 \
}
```

6.4.6.3 WriteReversedElements

`WriteReversedElements` writes the k'_{L}-reversed-elements. Because the elements in each array are in the desired order, each array can be written to memory with a simple loop. Note that k_{H} was bit-reversed to k'_{H} by rearranging the elements in `ReadElements`, and k_{L} was bit-reversed to k'_{L} by reading it from a table, but k_1 has not been bit-reversed yet. That is done here, by using 0, 2, 1, and 3 in the highest bits of the element indices:

WriteReversedElements

```
#define WriteReversedElements(kLprime)                        \
{                                                             \
   int kHprime;                                               \
   for (kHprime = 0; kHprime < 4; ++kHprime)                  \
   {                                                          \
      vOut.re[u0*0 + 4*kLprime + kHprime] = d0r[kHprime];   \
      vOut.re[u0*2 + 4*kLprime + kHprime] = d1r[kHprime];   \
      vOut.re[u0*1 + 4*kLprime + kHprime] = d2r[kHprime];   \
      vOut.re[u0*3 + 4*kLprime + kHprime] = d3r[kHprime];   \
      vOut.im[u0*0 + 4*kLprime + kHprime] = d0i[kHprime];   \
      vOut.im[u0*2 + 4*kLprime + kHprime] = d1i[kHprime];   \
      vOut.im[u0*1 + 4*kLprime + kHprime] = d2i[kHprime];   \
      vOut.im[u0*3 + 4*kLprime + kHprime] = d3i[kHprime];   \
   }                                                          \
}
```

6.4.6.4 PerformButterflies

Finally, `PerformButterflies` does the calculations:

PerformButterflies

```
#define PerformButterflies(weight)                    \
{                                                     \
   int i;                                            \
   for (i = 0; i < 4; ++i)                           \
   {                                                 \
      b1r[i] = - a1i[i] * weight.w1i[i] + a1r[i];    \
      b1i[i] = + a1r[i] * weight.w1i[i] + a1i[i];    \
      b2r[i] = - a2i[i] * weight.w2i[i] + a2r[i];    \
      b2i[i] = + a2r[i] * weight.w2i[i] + a2i[i];    \
      b3r[i] = - a3i[i] * weight.w3i[i] + a3r[i];    \
      b3i[i] = + a3r[i] * weight.w3i[i] + a3i[i];    \
      c0r[i] = + b2r[i] * weight.w2r[i] + a0r[i];    \
      c0i[i] = + b2i[i] * weight.w2r[i] + a0i[i];    \
      c2r[i] = - b2r[i] * weight.w2r[i] + a0r[i];    \
      c2i[i] = - b2i[i] * weight.w2r[i] + a0i[i];    \
      c1r[i] = + b3r[i] * weight.w3r[i] + b1r[i];    \
      c1i[i] = + b3i[i] * weight.w3r[i] + b1i[i];    \
      c3r[i] = - b3r[i] * weight.w3r[i] + b1r[i];    \
      c3i[i] = - b3i[i] * weight.w3r[i] + b1i[i];    \
      d0r[i] = + c1r[i] * weight.w1r[i] + c0r[i];    \
      d0i[i] = + c1i[i] * weight.w1r[i] + c0i[i];    \
      d1r[i] = - c1r[i] * weight.w1r[i] + c0r[i];    \
      d1i[i] = - c1i[i] * weight.w1r[i] + c0i[i];    \
      d2r[i] = - c3i[i] * weight.w1r[i] + c2r[i];    \
      d2i[i] = + c3r[i] * weight.w1r[i] + c2i[i];    \
      d3r[i] = + c3i[i] * weight.w1r[i] + c2r[i];    \
      d3i[i] = - c3r[i] * weight.w1r[i] + c2i[i];    \
   }                                                 \
}
```

6.4.7 AltiVec Implementation

The C code in section 6.4.6 converts very nicely to AltiVec instructions. The `WriteReversedElements` and `PerformButterflies` macros are straightforward, but `ReadElements` requires some work. `ReadElements` permutes the elements as it reads them, which is often a miserable task in AltiVec work. Fortunately, the permutations we need work well.

6.4.7.1 ReadElements, Part I

Before permuting the elements, they must be read from memory. `ReadElements` is shown with array index expressions that imply a good deal of address arithmetic. These calculations can be simplified:

- Values of 0, 1, 2, and 3 for k_H correspond to certain addresses in the data array, four addresses for the real components and four for the imaginary components. Calculate these addresses once per FFT and store them in registers named `highbits00r`, `highbits01r`, `highbits10r`, `highbits11r`, `highbits00i`, `highbits01i`, `highbits10i`, and `highbits11i`.
- When a value for `kL` is assigned, calculate the byte offset of `4*kL` elements (that is, the number of bytes from an element with some index `k` to the element with index `k + 4*kL`). Store this offset in a register named `index`.

With these preparations, the AltiVec instruction "`lvx y1r, highbits01r, index`" loads four real components into the register `y1r`. Those four components are `vOut.re[u0*1+4*kL+0]`, `vOut.re[u0*1+4*kL+1]`, `vOut.re[u0*1+4*kL+2]`, and `vOut.re[u0*1+4*kL+3]`. (See section 6.4.6.2 regarding the use of `u0` as the coefficient for `kH`, which has the value 1 in this example.) Address arithmetic is thus simple, and the instructions needed to load all 16 complex elements are:

AltiVec ReadElements, Part I

```
lvx   y0r, highbits00r, index
lvx   y1r, highbits01r, index
lvx   y2r, highbits10r, index
lvx   y3r, highbits11r, index
lvx   y0i, highbits00i, index
lvx   y1i, highbits01i, index
lvx   y2i, highbits10i, index
lvx   y3i, highbits11i, index
```

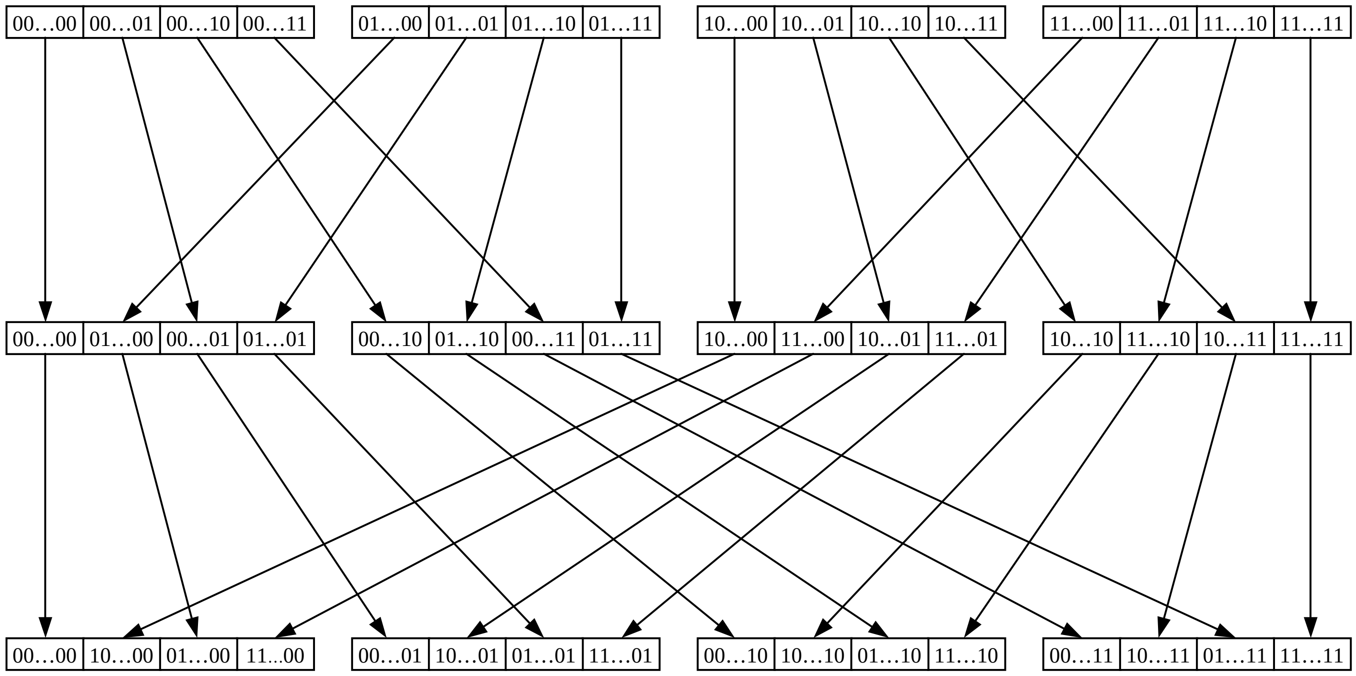
6.4.7.2 ReadElements, Part II

Next we need to rearrange the elements within the registers, into the bit-reversed order. Here are instructions for the real components:

AltiVec ReadElements, Part II

```
vmrghw   z0r, y0r, y1r  # Merge lesser of two highest bits.
vmrglw   z1r, y0r, y1r
vmrghw   z2r, y2r, y3r
vmrglw   z3r, y2r, y3r
vmrghw   a0r, z0r, z2r  # Merge higher of two highest bits.
vmrglw   a1r, z0r, z2r
vmrghw   a2r, z1r, z3r
vmrglw   a3r, z1r, z3r
```

The `r` suffix designates real components. Similar code will load, rearrange, and store the imaginary components. This diagram illustrates the effects of the merge instructions:



Observe that the elements within each block are now in order by the bit-reversals of the highest two bits (00…, 10…, 01…, 11…). These elements are ready for writing to memory in the order they are in **H**. However, the four blocks are in order by the lowest two bits (…00, …01, …10, …11), not the bit-reversals of those bits. This makes sense because we still want to perform a butterfly operation on this data, and it will be the same butterfly we have used so far, taking as input elements indexed 0, 1, 2, and 3, which we have placed in registers named `a0r`, `a0i`, `a1r`, `a1i`, `a2r`, `a2i`, `a3r`, and `a3i`.

6.4.7.3 WriteReversedElements

When we have the results in registers named `d0r`, `d0i`, `d1r`, `d1i`, `d2r`, `d2i`, `d3r`, and `d3i`, we will write those in bit-reversed order:

AltiVec WriteReversedElements

```
stvx  d0r, highbits00r, index
stvx  d1r, highbits10r, index
stvx  d2r, highbits01r, index
stvx  d3r, highbits11r, index
stvx  d0i, highbits00i, index
stvx  d1i, highbits10i, index
stvx  d2i, highbits01i, index
stvx  d3i, highbits11i, index
```

6.4.7.4 PerformButterflies

An AltiVec implementation of `PerformButterflies` is:

AltiVec PerformButterflies

```
vnmsubfp b1r, a1i, w1i, a1r
vmaddfp  b1i, a1r, w1i, a1i
```

```
vnmsubfp b2r, a2i, w2i, a2r
vmaddfp  b2i, a2r, w2i, a2i
vnmsubfp b3r, a3i, w3i, a3r
vmaddfp  b3i, a3r, w3i, a3i

vmaddfp  c0r, b2r, w2r, a0r
vmaddfp  c0i, b2i, w2r, a0i
vnmsubfp c2r, b2r, w2r, a0r
vnmsubfp c2i, b2i, w2r, a0i
vmaddfp  c1r, b3r, w3r, b1r
vmaddfp  c1i, b3i, w3r, b1i
vnmsubfp c3r, b3r, w3r, b1r
vnmsubfp c3i, b3i, w3r, b1i

vmaddfp  d0r, c1r, w1r, c0r
vmaddfp  d0i, c1i, w1r, c0i
vnmsubfp d1r, c1r, w1r, c0i
vnmsubfp d1i, c1i, w1r, c0i
vnmsubfp d2r, c3i, w1r, c2r
vmaddfp  d2i, c3r, w1r, c2i
vmaddfp  d3r, c3i, w1r, c2r
vnmsubfp d3i, c3r, w1r, c2i
```

The above code presumes that weight values have been loaded into registers named `w1r`, `w1i`, `w2r`, `w2i`, `w3r`, and `w3i`.

6.4.8 Generate Final Weights

Earlier butterfly routines required six values for one weight for one value of k_0 at a time. `FFT4_Final` now performs butterflies for four values of k_0 at a time, so it needs values for four weights. Further, the values of k_0 are not consecutive, and their order varies depending on N, so FFTs of different lengths need different groups of weights.

In one iteration on `q`, `FFT4_Final` performs butterflies using values for k_0 of $2^{N\text{-}4}0+k_\text{L}$, $2^{N\text{-}4}2+k_\text{L}$, $2^{N\text{-}4}1+k_\text{L}$, and $2^{N\text{-}4}0+k_\text{L}$, where k_L has the value loaded from `IndexTable`. The butterfly data is in the processor registers in that order (0, 2, 1, and 3), so the weight values should be available in that order.

This is all the information we need to generate weights for `FFT4_Final`. First, the six weight values for four butterflies are packaged in groups of four, like this:

FinalWeights

```
typedef struct {
   float w1r[4], w1i[4], w2r[4], w2i[4], w3r[4], w3i[4];
} FinalWeights;
```

Next, the values are calculated and stored by `GenerateFinalWeights`, below.

The expression “`r4kL + kHprime*rn`” used in the code below equals $r(4(2^{N\text{-}4}k_\text{H}+k_\text{L}))$, which is r($4k_0$), as required. (See section 5.3.) To see this, note that `r4kL` is assigned the value $r(4k_\mathrm{L})$, `rn` is assigned the value `1/n`, which is $1/2^N$, and `kHprime` represents k'_H. Then:

`r4kL + kHprime*rn` $= r(4k_\mathrm{L}) + k'_\mathrm{H} / 2^N$.

$= r(4k_L) + 2^2 r(k_H)/2^N$, by definition of k'_H and Lemma (1).

$= r(4k_\mathrm{L}) + r(2^{N-2} k_H)$, by Lemma (4).

$= r(4k_L + 2^{N-2} k_\mathrm{H})$, by Lemma (3).

$= r(4k_0)$, by definition of k_L and k_H.

GenerateFinalWeights

```
static int GenerateFinalWeights(
   FinalWeights **weights,    // Pointer to weight array address.
   int NewLength,             // New length to support (1<<N).
   FinalIndices *indices      // Index array address.)
{
   const double rn = 1./NewLength;
   int kHprime, q;

   // Try to allocate space and check result.
   FinalWeights *p = (FinalWeights *)
      realloc(*weights, NewLength/16 * sizeof **weights);
   if (p == NULL)
      return 1;

   for (q = 0; q < NewLength/16; ++q)
   {
      const int kL = indices[q].read;
      const double r4kL = r(4*kL);
      for (kHprime = 0; kHprime < 4; ++kHprime)
      {
         const double x = TwoPi * (r4kL + kHprime*rn);
         p[q].w1r[kHprime] = cos(x);
         p[q].w1i[kHprime] = tan(x);
         p[q].w2r[kHprime] = cos(x+x);
         p[q].w2i[kHprime] = tan(x+x);
         p[q].w3r[kHprime] = 2. * p[q].w2r[kHprime] – 1.;
         p[q].w3i[kHprime] = tan(3.*x);
      }
   }

   // Pass address back to caller.
   *weights = p;

   return 0;
}
```

6.4.9 Update Kernel

The new `FFT4_Final` routine must be passed a table of indices and an array of weights different from the previous weights, so the kernel has to pass the new arguments:

FFT Kernel with Final Indices and Weights

```
if (N & 1)
   FFT8_0Weights(vOut, vIn, 1<<N);
else
   FFT4_0Weights(vOut, vIn, 1<<N);

nLower = N&1 ? 3 : 2;
for (n = nLower; n      < N-4      ; n += 2     )
   FFT4_0Weights(vOut, vOut, 1<<N-n);

for (k0 = 1    ; nLower < N-4      ; nLower += 2)
for (          ; k0     < 1<<nLower; ++k0       )
for (n = nLower; n      < N-4      ; n += 2     )
   FFT4_1WeightPerCall(vOut, k0, 1<<N-n, weights[k0]);

if (n < N-2)
   FFT4_1WeightPerIteration(vOut, 1<<N-4, weights);

FFT4_Final(vOut, 1<<N-2, finalIndices, finalWeights);
```

6.5 FFT Kernel Routine

The inputs to the FFT kernel are `vIn`, `vOut`, `N`, `weights`, `finalIndices`, and `finalWeights`. We can take the code fragment we have developed and make it into a complete routine:

FFT Kernel Routine

```
static void FFT_Kernel(
   ComplexArray vOut,              // Address of output vector.
   ComplexArray vIn,               // Address of input vector.
   int N,                          // N.
   const CommonWeight *weights,    // Common weight values.
   const FinalIndices *finalIndices,// Index pairs.
   const FinalWeights *finalWeights // Final weight values.
)
{
   int n, nLower, k0;

   if (N & 1)
      FFT8_0Weights(vOut, vIn, 1<<N);
   else
      FFT4_0Weights(vOut, vIn, 1<<N);

   nLower = N&1 ? 3 : 2;
   for (n = nLower; n      < N-4      ; n += 2     )
      FFT4_0Weights(vOut, vOut, 1<<N-n);

   for (k0 = 1    ; nLower < N-4      ; nLower += 2)
   for (          ; k0     < 1<<nLower; ++k0       )
   for (n = nLower; n      < N-4      ; n += 2     )
      FFT4_1WeightPerCall(vOut, k0, 1<<N-n, weights[k0]);

   if (n < N-2)
      FFT4_1WeightPerIteration(vOut, 1<<N-4, weights);
```

```
    FFT4_Final(vOut, 1<<N-2, finalIndices, finalWeights);
}
```

7 Out-of-Cache Performance

7.1 Introduction

The FFT kernel developed so far is excellent when all the memory needed fits within processor cache, including the transform data, the weights and table of indices, and any miscellaneous data. When the memory does not fit within processor cache, problems occur.

7.1.1 Motorola PowerPC CPU 7400 Cache Architecture

Much discussion in previous sections is generally applicable to a variety of computer architectures. To discuss designing for high-performance in the presence of cache architecture issues, it is necessary to be more specific. This paper addresses designing for the Motorola PowerPC CPU 7400 or similar CPUs. This specific CPU will be assumed throughout the rest of section 7.

The level-1 (L1) data cache in the Motorola PowerPC CPU 7400 is 32,768 bytes. The cache is partitioned into 128 sets. Each set contains eight blocks, and each block is 32 bytes. This cache architecture is not uncommon, and other processors may have a similar architecture with different dimensions.

Each memory address maps to one set. That is, when the contents of a memory address are brought into cache, they must go into the set assigned to the address. Any of the eight blocks within the set may be used. If all blocks are in use, the CPU makes a block available by selecting a block and discarding the data in it or writing it to memory, as appropriate. (The CPU approximates selecting the least-recently-used block to reuse.)

Using C notation, the byte with address `a` is:

- the byte numbered `a%32` in a block and
- mapped to the cache set numbered `a/32 % 128`.

Concerning cache set mapping, observe that data separated by multiples of 4096 bytes (32·128) map to the same cache set (if `a` and `b` differ by a multiple of 4096, then `a/32 % 128` equals `b/32 % 128`).

Cache blocks are also called cache lines. 32-byte cache blocks are different from the 16-byte blocks loaded by `lvx` instructions and from the programmer-specified blocks in `dst` instructions.

I will refer to the group of 32 bytes in memory that would be loaded into a cache block together as a cache block even while it is only in memory and not in cache.

7.1.2 Cache Problems

The cache architecture imposes three important constraints:

- The number of elements that can fit in cache is limited. For complex numbers of eight bytes (two floating-point numbers of four bytes each), 4096 elements can fit in cache.
- The number of elements that can fit in a cache set is limited. Elements from eight different addresses that map to the same cache set can fit in the set.
- It is not possible to read less than one cache block from memory (in the absence of special control operations). Reading any byte from memory causes all bytes in the same block to be read and stored in cache.

We will find that although the entire FFT cannot be perform entirely in cache, the FFT can be partitioned into sets of butterfly operations such that each set can be performed entirely in cache.

7.2 The Cache Size Problem

Our goal is to partition the FFT into sets of butterfly operations such that each set can be performed entirely in cache.

A radix-2^m butterfly requires 2^m input elements, along with some amount of constant data. Choosing a sufficiently small m yields a butterfly for which all the data fits in cache. Then each iteration of the loop on `k0` can be performed in cache.

If m is even smaller, it may be that the data for several radix-2^m butterflies fit in cache. Then several iterations of the loop on `k0` can be performed in cache. In that case, we have a choice:

- We could read the data for one butterfly from memory into cache, perform the butterfly, and write the results to memory.
- We could read the data for several butterflies, perform those butterflies, and write the results to memory.

Depending on the bus characteristics, there may be advantages to reading more data sequentially at one time. If so, we prefer the latter choice, and we will cluster iterations of the loop on `k0`.

7.3 The Cache Set Size Problem

In the early passes of an FFT, n is small, so 2^{N-n-m} is large. Consider array element indices of the form $2^{N-n}k_0 + 2^{N-n-m}k_1 + k_2$, which are used by butterfly operations. A single radix-4 butterfly uses four values of k_1. These values are stored in elements that are separated by large multiples of a power of two (2^{N-n-m} elements), so they are assigned to the same cache set. One cache set can hold values from eight locations. So all data for a single radix-4 butterfly can fit in a single cache set. After performing such a butterfly, we could increment k_2 and repeat the process until the entire pass were completed. Thus it is possible to perform an entire butterfly pass while reading and writing each element only once, not having to reload any element.

However, we wish to perform more than one pass of radix-4 butterflies.

Consider a butterfly in the third radix-4 pass of an FFT. It needs four elements for input. Each of those elements is calculated in the second radix-4 pass from four different input elements. So if we wish to calculate two radix-4 passes without reloading data from memory, there must be 16 elements in cache at the same time, and those 16 elements are widely separated in the array, by multiples of a large power of two. Therefore, they cannot fit in cache simultaneously unless the cache associativity is at least 16.

If you were so fortunate as to have a cache with an associativity of 16, attempting three radix-4 passes would require an associativity of 64.

However, doing one radix-4 pass on one set of data that has been read into cache is unacceptable. While data is in cache, we want to take the FFT operation from $\mathbf{v}_{\mathrm{n}}$ to $\mathbf{v}_{n+m}$ for m of a fairly large size. To perform this radix-2^m butterfly, we need 2^m elements, and we need them to fit in a cache set. Since they do not fit in a cache set in their original memory locations, we must move them.

Suppose we have a buffer of length b elements where we can store data temporarily. We can copy the elements we need for one radix-2^m butterfly into the buffer, perform the butterfly, and copy the elements back to their original locations (or to new locations if we like). If the buffer will hold more elements than we need for one butterfly, we can do several butterflies at one time. The plan is:

(1) Gather elements together: Copy the data for the butterflies from spread-out addresses in the data array to sequential addresses in the buffer.
(2) Do calculations: For each set of 2^m elements in the buffer, perform a radix-2^m butterfly[3].
(3) Scatter elements back: Copy the data from the buffer to spread-out addresses in the data array.

The primary effect of this copying is to move the data from addresses where they map to the same cache set to addresses where they map to different cache sets. Once the data is in the buffer, it may be accessed freely in any order without casting other buffered data out of cache. So we may perform radix-2^m butterflies efficiently, and the data needs to be read from the data array once and written back to it once. The buffer may reside entirely cache, so it never needs to be written to or read from memory, although (hopefully small) portions of it may be cast out and reread as unintended byproducts of other memory operations.

[3] This does not mean to perform a radix-2^m butterfly as one operation as shown in `FFT_Butterflies`, but to perform it by any efficient means, such as a composition of radix-8 and radix-4 butterflies.

How big should the buffer be, what value should *b* have? There are two advantages to increasing *b*:

- The larger *b* is, the larger *m* may be, and the more calculations may be done per element per buffer turnover.
- Using a larger *b* without a larger *m* may gain some advantage in data transfer on the bus between memory and cache, if the bus has characteristics such as transferring sequential addresses more quickly than disordered addresses.

In the latter case, consider that using a larger *b* without a larger *m* means the data for more butterflies can be held in the buffer. If *m* is increased by one, the number of different (and nonsequential) locations that must be read is doubled. However, if *b* is increased while *m* remains the same, more data can be read (sequentially) from each of the 2^m locations. Thus, increasing *b* may increase data transfer rates, while increasing *m* increases the computations per buffer turnover.

Half of cache is a good choice. If the buffer filled all of cache, other necessary data, such as weights, could not be kept in cache. If we make the buffer smaller, we lose the above performance advantages.

The truly ambitious implementer could use a buffer between half and all of cache.

A design to use such a buffer for the first stage of an FFT is in section 7.6.1.

7.4 The Cache Block Size Problem

Reading any byte from memory causes all bytes in the same cache block to be read and stored in cache. For the most part, this is not a problem for the FFT. Data is operated on in AltiVec blocks of four floating-point numbers at a time. Four single-precision floating-point numbers take 16 bytes, half of the 32-byte cache block. If real and imaginary components of the complex data are stored in the same cache block, then four complex elements occupy exactly one cache block, and the cache block size coincides well with the FFT operations.

If the real and imaginary components are stored separately, some attention must be paid to cache block use.

Most FFT operations iterate sequentially through values of `k0` and `k2`. As the operations iterate through the data, they will use first one half and then the other half of each cache block, thus completing the use of all the data in the block while it is in cache.

An exception is `FFT4_Final`, which processes four-element blocks in an order partially dictated by the bit-reversal permutation. This order does not necessarily use both halves of a cache block in successive iterations.

However, it can be made to do so. As stated in section 6.4.4, there are two constraints about storing index pairs, but we otherwise have a good deal of freedom in arranging the index table. We can cluster indices that reside in the same cache blocks. Recall that each index in the table (k_L) addresses a group of four elements ($4k_L$+0, $4k_L$+1, $4k_L$+2, and $4k_L$+3). With separated real and imaginary components, we need eight array elements to fill a cache block, so we need two indices (k_L and k_L+1) to be clustered in the index table.

For example, suppose that k_L is even and the arrays of real and imaginary components each begin on cache block boundaries. After we use the k_L-elements (see terminology in section 6.4.3), we will want to use the (k_L+1)-elements whose components are in the same blocks. (Note that we have now added the constraint that the arrays should be aligned to cache-block boundaries for best performance. When k_L is even, we are depending on the k_L-elements to be in the same blocks as the (kL+1)-elements and not the (kL-1)-elements.)

However, organizing the table is not as simple as pairing each even k_L with k_L+1. When the k_L-elements are used, the k'_L-reversed-elements are used as well. If $k_L \neq k'_L$, then the k'_L-elements and k_L-reversed-elements are also used with the preceding or following index table entry. We must cluster all of these elements with their cache-block partners.

Fortunately, this is accomplished with changes to `GenerateFinalIndices`:

Cache-Block Clustering GenerateFinalIndices

```
static int GenerateFinalIndices(
   FinalIndices **indices,    // Pointer to index array address.
   int NewLength              // New length to support (1<<N).)
{
   // Prepare to bit-reverse a number of N-4 bits (see below).
   const int shift = 32 - (ilog2(NewLength) - 4);
   int kL;

   // Try to allocate space and check result.
   FinalIndices *p = (FinalIndices *)
      realloc(*indices, NewLength/16 * sizeof **indices);
   if (p == NULL)
      return 1;

   // Pass address back to caller.
   *indices = p;

/* This routine generates indices for the kL part of the element
   index, which is the index minus the two high bits and the two
   log bits.  This routine is never called with length less than
   16, so those four bits are always there to remove.

   In addition, we want to cluster the indices by cache blocks,
   so we need to remove another low bit, and therefore another
   high bit.  This requires that the length be at least 64.

   For smaller lengths, all the elements do not form a whole
   cluster, so we will generate those indices with separate code.
*/

   // Handle small sizes.
   if (NewLength < 64)
   {
      *(p++)     = Construct(0, 0); // (0, 0) for lengths 16 and 32.
      if (16 < NewLength)
          *(p++) = Construct(1, 1); // (1, 1) for length 32.
   }

   // Do other sizes, with whole clusters of cache blocks.
   else
   {
      // Provide names for high bit of zero (h0) and one (h1).
      const unsigned int h0 = 0, h1 = rw(1) >> shift;

      // Iterate through all values of kL excluding high bit and low bit.
      for (kL = 0; kL < NewLength/16/2; kL += 2)
      {
         // rw(kL) reverses kL as a 32-bit number.  To get it as
         // the reversal of an N-4 bit number, shift right to
         // remove 32-(N-4) bits.
         const int kLprime = rw(kL) >> shift;

         // If kLprime < kL, then kL in a previous iteration had the
         // value kLprime has now, and we do not want to repeat it.
         if (kL <= kLprime)
         {
            // Use shorter names for forward kL (F) and reverse kL (R).
            const unsigned int F = kL, R = kLprime;

/* To convince yourself the following code is correct,
   first check that each pair of addresses are bit-reversals
   of each other (h0|R|1 is paired with h1|F|0, and so on).

   Next, in the kL != kLprime case, check that each entry
   is preceded or followed by its reversal (a pair with
   a write to h1|R|0 is adjacent to a pair with a read
   from h1|R|0, and so on).

   Finally, in the kL == kLprime case, check that each of
   the four executed entries either is its own reversal
   (h0|F|0 equals h0|R|0 when F == R) or its preceded or
   followed by its reversal (same as kL != kLprime case).
*/
            *(p++)     = Construct( h0|F|0, h0|R|0 );
            if (kL != kLprime)
            {
                *(p++) = Construct( h0|R|0, h0|F|0 );
                *(p++) = Construct( h1|F|0, h0|R|1 );
                *(p++) = Construct( h0|R|1, h1|F|0 );
                *(p++) = Construct( h1|F|1, h1|R|1 );
            }
            *(p++)     = Construct( h1|R|1, h1|F|1 );
            *(p++)     = Construct( h0|F|1, h1|R|0 );
            *(p++)     = Construct( h1|R|0, h0|F|1 );
         }
      }
   }
```

```
    return 0;
}
```

In addition to cluster final-pass processing by cache blocks, cache control operations can be inserted into a variant of `FFT4_Final`. This may speed up performance by flushing data soon after we are done writing it. This releases the unneeded cache blocks, ensuring those blocks will be reused first and avoiding the possibility the processor will select for replacement blocks containing data that is still needed.

7.5 Structuring the Multiple-Stage FFT

In section 3.2, we chose values of m_p based on our target processor architecture. To solve cache problems, we consider values of m_p based on our cache architecture. Instead of the term “passes” used in the initial kernel, I use the term “stages” in describing the out-of-cache FFT. Mathematically, stages and passes are identical except that we generally use larger values of m for stages.

In each stage, we will perform radix-2^{m_p} butterflies. Those butterflies will in turn be composed of butterflies using the existing butterfly routines. The first stage will be performed using a first pass of `FFT4_0Weights` or `FFT8_0Weights` followed by passes of radix-4 butterflies. In every other stage, only radix-4 butterflies will be used. Thus m_p must be even for every stage after the first, and m_0 must be even or odd according to whether N is even or odd. Also, m_0 must be at least 2 (the smallest m available in an implemented butterfly routine), although we will never want to use a value this low.

In the first stage, m must be not greater than $\log_2(b)$, so that the data for one butterfly fits in the buffer described in section 7.3. m might be smaller because completely packing the buffer with the data for one butterfly means that input elements to the butterfly will be adjacent in memory, and then it is difficult to access them with AltiVec instructions. There is even some question whether $\log_2(b)$-2 is too high for m, as then we must use `FFT4_1WeightPerIteration` for some of the computation with the buffer rather than `FFT4_1WeightPerCall`. For this design, I choose to limit m to $\log_2(b)$-4. (See also sections 7.5.2 and 7.6.1.3.)

After the first stage, it is possible to perform additional stages using the gather-scatter technique. Such stages would also have their values of m constrained by $\log_2(b)$. Such stages are not needed except for FFTs on extraordinarily long vectors and are not examined in this paper.

The penultimate stage is flexible, but the final stage has severe constraints, so I will discuss the last stage and then return to the penultimate stage.

In the last stage, we wish to do the bit-reversal permutation. The bit-reversal wreaks havoc with cache. Cache and bus performance are generally enhanced by sequential access. Bus performance may be hindered by non-sequential access, and cache performance is hindered by repeated access to more than eight addresses differing only in their high bits. However, while doing a bit-reversal permutation, accessing eight consecutive elements in one place causes necessitates accesses to eight different places that map to the same cache set. Any more overflow a cache set and cause thrashing.

In section 7.4, I discussed clustering elements in cache blocks. If real and imaginary components are stored separately and are four-byte floating-point numbers, there are eight components in one cache block. The cluster of butterflies needs to read eight such cache blocks and write results to eight other cache blocks (at the reversed addresses), intermingled with also reading the latter blocks and writing the former blocks. Fortunately, the cache associativity is eight and the former and latter blocks are usually (not always!) mapped to different cache sets.

This means eight elements strain the cache associativity as far as it will go. To do two radix-4 passes, we would need 16 elements. Therefore, `FFT4_Final` is the only butterfly operation we can put in the final stage without breaking it. The final pass is the final stage, so m_{P-1}=2. With three stages, P is 3, so m_{P-1}=m_2=2.

Given a first stage with some m_0 and a last stage with m_2=2, the penultimate stage is determined: m_1=N-2-m_0.

If m_1 is small enough that all the data for at least one radix-2^{m_1} butterfly in the penultimate stage fits in cache, then three stages suffice to perform the FFT with good cache behavior in each stage. If not, then more stages are required.

7.5.1 Summary

Summarizing our multiple-stage FFT design:

- m_0 is odd or even according to whether N is odd or even.
- m_0 is no larger than the cached buffer of b elements will permit.
- m_0 may be slightly smaller due to AltiVec inefficiencies with elements located too closely together.
- m_2 is 2.
- m_1 is whatever is left over.
- m_1 is no larger than the cache will permit.
- m_1 may be zero, a degenerate case indicating the stage is not used.

7.5.2 PowerPC CPU 7400 Design

Level-1 cache on the PowerPC CPU 7400 is 32,768 bytes. If our buffer for gathering data is half of cache, then b is 2048. (2048 complex elements of eight bytes each occupy 16,384 bytes.) Because support for butterfly data spaced more closely together than AltiVec blocks (16 bytes, four elements) will not be included in our implementation of the first-stage of the multiple-stage FFT, m_0 must not be greater than $\log_2(b/4)$, which is 9. So m_0 could be 9 or 8, according to whether N is odd or even.

However, these values require the use of `FFT4_1WeightPerIteration` in the first stage, as discussed in section 7.6.1.3. We may find values of 7 or 6 preferable. The value of m_0

is a flexible part of the FFT design, easily changed by adjusting a compile- or assembly-time value, so it can be left for tuning after measurements are made on a target system.

The elements needed for a radix-2^{m_1} butterfly have indices $2^{N-n_1-m_1}k_1+k_2$ for $0 \le k_1 < 2^{m_1}$. In the penultimate stage, N-n_1-m_1 is 2, so the indices are $4k_1+k_2$ and range from k_2 to $4\left(2^{m_1}-1\right)+k_2$, spanning $4\left(2^{m_1}-1\right)$ elements. So $4\left(2^{m_1}-1\right)$ must be not more than the number of elements that we can have in cache. 4096 elements would fit but would not leave room for weights, so $4\left(2^{m_1}-1\right)$ must be less than 4096. Then m_1 must be less than 10 and even, so it is at most 8.

At the limits, m_0 is 9, m_1 is 8, and m_2 is 2, so N is 19, and the longest vector for which we can efficiently compute the DFT on a PowerPC CPU 7400 without a fourth stage has 2^{19}=524,288 elements. If m_0 is limited to 7, then the longest vector for which we can efficient compute the DFT has 2^{17}=131,072 elements.

7.6 Stage Designs

Now that we have a design for the multiple-stage FFT, we can design the stages themselves. We return to our first FFT kernel, from section 3.1.2:

```
for (p  = 0; p  < P      ; ++p )
for (k0 = 0; k0 < 1<<n[p]; ++k0)
   FFT_Butterflies(m[p], v[n[p+1]], v[n[p]], k0, 1<<N-n[p]);
```

We have two or three stages, so we can unroll the loop on `p`:

```
for (k0 = 0; k0 < 1<<n[0]; ++k0)
   FFT_Butterflies(m[0], v[n[1]], v[n[0]], k0, 1<<N-n[0]);

if (0 < m[1])
for (k0 = 0; k0 < 1<<n[1]; ++k0)
   FFT_Butterflies(m[1], v[n[2]], v[n[1]], k0, 1<<N-n[1]);

for (k0 = 0; k0 < 1<<n[2]; ++k0)
   FFT_Butterflies(m[2], v[n[3]], v[n[2]], k0, 1<<N-n[2]);
```

Let `m0` have the value of m_0. Since m_1 is N-2-m_0, m_2 is 2, n_0 is 0, n_1 is m_0, and n_2 is N-2, and the code becomes:

```
for (k0 = 0; k0 < 1     ; ++k0)
   FFT_Butterflies(m0, vOut, vIn, k0, 1<<N);

if (0 < N-2-m0)
for (k0 = 0; k0 < 1<<m0 ; ++k0)
   FFT_Butterflies(N-2-m0, vOut, vOut, k0, 1<<N-m0);

for (k0 = 0; k0 < 1<<N-2; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 4);
```

The first loop has only one iteration:

```
FFT_Butterflies(m0, vOut, vIn, 0, 1<<N);

if (0 < N-2-m0)
for (k0 = 0; k0 < 1<<m0  ; ++k0)
   FFT_Butterflies(N-2-m0, vOut, vOut, k0, 1<<N-m0);

for (k0 = 0; k0 < 1<<N-2; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 4);
```

We will write new routines to perform each section in the above code:

```
FFT_FirstStage(m0, vOut, vIn, 1<<N);

if (0 < N-2-m0)
   FFT_PenultimateStage(vOut, m0, N);

FFT_FinalStage(vOut, 1<<N-2);
```

Actually, the new routines will need prepared constants to compute efficiently:

Multiple-Stage Kernel

```
FFT_FirstStage(m0, vOut, vIn, 1<<N, weights);

if (0 < N-2-m0)
   FFT_PenultimateStage(vOut, m0, N, weights);

FFT_FinalStage(vOut, 1<<N-2, finalIndices, finalWeights);
```

7.6.1 First Stage

When called via:

```
FFT_FirstStage(m0, vOut, vIn, 1<<N, weights);
```

`FFT_FirstStage` must perform the calculations defined by:

FFT_FirstStage Prototype

```
FFT_Butterflies(m0, vOut, vIn, 0, 1<<N);
```

where `m0` and `N` are large. To do this efficiently, we will create a new specialization of `FFT_Butterflies` for this situation. We will gather data into a buffer, calculate some butterflies, and scatter the data back to a data array.

7.6.1.1 Gather and Scatter

Here are subroutines to gather the data into a buffer and scatter it back to an array. To get the 2^m elements needed for all the values of k_1 in a butterfly, we iterate `k1` through each value. To get all the data for a cluster, we iterate on `c`. Data is gathered from spread-apart locations in the data array (using `c1*k1`) and collected in close-together locations in the buffer (using `cluster*k1`). At each location, sequential data is copied by iterating `c`. Each time these routines are called, the caller passes a different value of `k2`, using it to step through the data array.

Gather

```
static void Gather(
   ComplexArray destination,  // Destination of copying.
   ComplexArray source,       // Source of copying.
   int u1,                    // Upper limit on k1, equals 1<<m.
   int c1,                    // Coefficient for k1.
   int k2,                    // Current value of k2.
   int cluster                // Butterfly sets per cluster.
)
{
   int k1, c;

   for (k1 = 0; k1 < u1     ; ++k1)
   for (c  = 0; c  < cluster; ++c )
      destination[cluster*k1 + c] = source[c1*k1 + k2+c];
}
```

Scatter

```
static void Scatter(
   ComplexArray destination,  // Destination of copying.
   ComplexArray source,       // Source of copying.
   int u1,                    // Upper limit on k1, equals 1<<m.
   int c1,                    // Coefficient for k1.
   int k2,                    // Current value of k2.
   int cluster                // Butterfly sets per cluster.
)
{
   int k1, c;

   for (k1 = 0; k1 < u1     ; ++k1)
   for (c  = 0; c  < cluster; ++c )
      destination[c1*k1 + k2+c] = source[cluster*k1 + c];
}
```

7.6.1.2 Calculating Butterflies

Here is a first version of `FFT_FirstStage`:

First FFT_FirstStage

```
static void FFT_FirstStage(
   int m,                           // log2 of butterfly radix.
   ComplexArray vOut,               // Address of output vector.
   ComplexArray vIn,                // Address of input vector.
   int c0,                          // Coefficient for c0.
   const CommonWeight weights[]     // Array of weight values.
)
{
   // Coefficient for k1 is coefficient for c0 divided by 1<<m.
   const int c1 = c0 >> m;
   const int u1 = 1<<m;
   // Cluster size is how many sets fit in buffer at one time.
   const int cluster = b >> m;

   int k2;

   // Process values of k2 in clusters.
   for (k2 = 0; k2 < c1; k2 += cluster)
   {
      Gather(buffer, vIn, u1, c1, k2, cluster);
      FFT_Butterflies(m, buffer, buffer, 0, b);
      Scatter(vOut, buffer, u1, c1, k2, cluster);
   }
}
```

The code is simple enough, but why is `b` passed to `FFT_Butterflies`? That formal argument is `c0`, the coefficient for `k0`, which is 2^{N-n} in the mathematics. In this initial pass, n is 0, so we would normally pass 2^N, or `1<<N`.

In `FFT_Butterflies`, `c1` is derived from the formal argument `c0` (actual argument `b`), and `c1` is used in two ways. It is the coefficient for `k1`, used to locate elements in the array, and it is the upper bound of the loop on `k2`, so it specifies the number of iterations for `k2`.

In both cases, the normal value of `c1` would not work. First, we have moved elements from their original locations; they are at different indices in `buffer`. Second, we have gathered only `cluster` sets of data, not all of them.

We can see that passing `b` as the actual argument satisfies both purposes. `FFT_Butterflies` calculates "`c1 = c0 >> m`". Having been passed `b` for `c0`, this gives `b>>m`, which equals `cluster`. As we can see from the `Gather` code, `cluster` is both the coefficient for `k1` used to place elements in the buffer and the number of sets of data.

As written, this code requires that `cluster` divide `c1`, so that `k2` ends at exactly `c1` after a whole number of clusters. `cluster` is `b>>m`, so `b>>m` must divide `c1`, which means `b` must divide `c1<<m`. `c1` is `c0>>m`, and the formal argument `c0` is passed `1<<N` as the actual argument (in section 7.6), so `b` must divide `1<<N`. Since `1<<N` is a power of two, this amounts to saying `b` must be a power of two. This restriction may be lifted by separating a final iteration from the loop to handle a partial cluster. Such a modification would have to be propagated to the more efficient code below. That is not shown in this paper.

Another constraint on `b` is that it must be a multiple of `1<<m`. `b` is divided by `1<<m` to set `cluster`, and then each call to `Gather` gathers data for `cluster` butterflies, so it gathers `cluster<<m` elements. If `b` is a multiple of `1<<m`, then `cluster<<m` is `b`. If not, only `cluster<<m` elements are gathered, and `cluster<<m` should be passed to `FFT_Butterflies` in lieu of `b`. This is the same as reducing `b` to the nearest multiple of `1<<m`.

7.6.1.3 Specializing the Butterflies

`FFT_FirstStage` contains a call to `FFT_Butterflies`:

```
FFT_Butterflies(m, buffer, buffer, 0, b);
```

This should be replaced with a specialization optimized for this situation. Observe that this call transforms the contents of `buffer` from $\mathbf{v}_0$ to $\mathbf{v}_m$. We already have optimized code that does this. The first sets of loops in the FFT kernel transform $\mathbf{v}_0$ (in `vIn`) to $\mathbf{v}_{n_p}$ (in `vOut`). We can take this code from the kernel:

```
if (N & 1)
   FFT8_0Weights(vOut, vIn, 1<<N);
else
   FFT4_0Weights(vOut, vIn, 1<<N);

nLower = N&1 ? 3 : 2;
for (n = nLower; n      < N-4      ; n += 2     )
   FFT4_0Weights(vOut, vOut, 1<<N-n);

for (k0 = 1    ; nLower < N-4      ; nLower += 2)
for (          ; k0     < 1<<nLower; ++k0       )
for (n = nLower; n      < N-4      ; n += 2     )
   FFT4_1WeightPerCall(vOut, k0, 1<<N-n, weights[k0]);
```

and make appropriate substitutions. To know what substitutions to make, let us review the code. This code:

- reads from `vIn` and writes to `vOut`,
- evaluates `N & 1` to decide whether radix-8 or radix-4 is used first,
- uses `N-4` in various loop tests to limit `n` to `N-4` (thus yielding $\mathbf{v}_{N\text{-}4}$), and
- passes `1<<N` and `1<<N-n` for the `c0` argument, which is used for element spacing and loop counting.

To use this code for our new purpose, we will make the following substitutions.

- We want to operate on the data in buffer, so `vIn` and `vOut` become `buffer`.
- We will start with radix-8 or radix-4 according to whether the formal argument `m` is odd or even, so `N & 1` becomes `m & 1`.
- We want to calculate $\mathbf{v}_m$ rather than $\mathbf{v}_{N\text{-}4}$, so the loop limits change from `N-4` to `m`.
- The vector has length `b` instead of `1<<N`, so `1<<N` becomes `b`, and `1<<N-n` becomes `b>>n`.

This yields:

```
if (m & 1)
   FFT8_0Weights(buffer, buffer, b);
else
```

```
   FFT4_0Weights(buffer, buffer, b);

nLower = m&1 ? 3 : 2;
for (n = nLower; n      < m        ; n += 2     )
   FFT4_0Weights(buffer, buffer, b>>n);

for (k0 = 1    ; nLower < m        ; nLower += 2)
for (          ; k0     < 1<<nLower; ++k0       )
for (n = nLower; n      < m        ; n += 2     )
   FFT4_1WeightPerCall(buffer, k0, b>>n, weights[k0]);
```

Note that if `b>>n` reaches 16, the final loop on `n` is better done with a call to `FFT4_1WeightPerIteration` (which is specialized for this case) than with a loop calling `FFT4_1WeightPerCall`. This design, while it will calculate correct results if `b>>n` is 16, is not the most efficient in that case. In the current design, this routine will not be called on to do this. `n` reaches the value `m-2`, and `m` is passed the value `m0`, so `b>>n` reaches `b>>m0-2`. We will select *b* and m_0 to keep $b/2^{m_0}$ above 16.

7.6.1.4 Finished Routine

Putting the new code into the routine gives:

FFT_FirstStage

```
static void FFT_FirstStage(
   int m,                            // log2 of butterfly radix.
   ComplexArray vOut,                // Address of output vector.
   ComplexArray vIn,                 // Address of input vector.
   int c0,                           // Coefficient for c0.
   const CommonWeight weights[]      // Array of weight values.
)
{
   // Coefficient for k1 is coefficient for c0 divided by 1<<m.
   const int c1 = c0 >> m;
   const int u1 = 1<<m;

   // Cluster size is how many sets fit in buffer at one time.
   const int cluster = b >> m;

   int n, nLower, k, k0, k2;

   // Process values of k2 in clusters.
   for (k2 = 0; k2 < c1; k2 += cluster)
   {
      Gather(buffer, vIn, u1, c1, k2, cluster);

      if (m & 1)
         FFT8_0Weights(buffer, buffer, b);
      else
         FFT4_0Weights(buffer, buffer, b);

      nLower = m&1 ? 3 : 2;
      for (n = nLower; n      < m        ; n += 2     )
         FFT4_0Weights(buffer, buffer, b>>n);
      for (k0 = 1    ; nLower < m        ; nLower += 2)
      for (          ; k0     < 1<<nLower; ++k0       )
      for (n = nLower; n      < m        ; n += 2     )
         FFT4_1WeightPerCall(buffer, k0, b>>n, weights[k0]);

      Scatter(vOut, buffer, u1, c1, k2, cluster);
   }
}
```

7.6.2 General Stages

If an FFT is being performed on a very long vector, more than three stages are required. (See the end of section 7.5.) After the first stage, the input elements to butterflies are still far apart and must still be gathered together. However, the code in `FFT_FirstStage` cannot be used as is, as it is structured for n=0. The code in `FFT_PenultimateStage`, below, is structured for general n, but it does not gather and scatter data. To support high-performance with very long vectors, another stage would have to be designed.

A routine implementing such a stage would suffice to handle vectors of any length, as it could be used as many times as necessary to process any number of intermediate stages. However, such a routine is not discussed in this paper.

For the reader who would design such a routine, note that an argument `k0` must be added to the `Gather` and `Scatter` routines. The routines in section 7.6.1.1 implicitly have `k0`=0, since they are used only in the first stage.

7.6.3 Penultimate Stage

In the penultimate stage, n is large, so 2^{N-n-m} is small, and the cache set size is not a problem. This means we do not need the gather-scatter technique used in the first stage. However, cache size is still a problem, so the penultimate stage must be done in sets of butterfly operations that can each be performed in cache.

When called via:

```
FFT_PenultimateStage(vOut, m0, N, weights);
```

`FFT_PenultimateStage` must perform the calculations defined by:

FFT_PenultimateStagePrototype

```
for (k0 = 0; k0 < 1<<m0  ; ++k0)
   FFT_Butterflies(N-m0-2, vOut, vOut, k0, 1<<N-m0);
```

This code computes $\mathbf{v}_{N-2}$ from $\mathbf{v}_{m_0}$. Another way to compute $\mathbf{v}_{N-2}$ from $\mathbf{v}_{m_0}$, given that $N-2-m_0$ is even, is:

```
for (n = m0; n < N-2; n += 2)
for (k0 = 0; k0 < 1<<n; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n);
```

That is, compute $\mathbf{v}_n$ for $n = m_0+2$, m_0+4, $m_0+6, \ldots, N-2$, where each $\mathbf{v}_n$ being computed form the previous one in this sequence by radix-4 butterflies. (For a reminder about this loop structure, compare this code to the first FFT kernel in section 3.1.3.) To be general, I will replace the actual argument `m0` with a formal argument `nStage`, representing the value of `n` for which $\mathbf{v}_n$ is input to the stage.

The inner loop can be partitioned into groups of iterations such that all the input data for each group fits in cache. Let `g` be the number of elements used in a group of iterations. Generally, `g` should be as large as it can be without excluding other data, such as weights, from cache. `g` might or might not be the same as `b`, the number of elements in the buffer used in the first stage.

If `g` divides `1<<N`, then the code to process the butterflies in groups is:

```
for (n  = nStage; n  < N-2       ; n += 2)
for (k  = 0     ; k  < 1<<N      ; k += g)
for (k0 = k>>N-n; k0 < (k+g)>>N-n; ++k0  )
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n);
```

We could also write code that works for any value of `g`:

```
for (n  = nStage; n < N-2; n += 2)
{
   // Do whole groups up to last.
   for (k  = 0     ; k  < (1<<N)-g  ; k += g)
   for (k0 = k>>N-n; k0 < (k+g)>>N-n; ++k0  )
      FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n);

   // Do last group, whether partial or whole.
   for (k0 = k>>N-n; k0 < 1<<n      ; ++k0  )
      FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n);
}
```

By removing the constraint, the latter code may allow use of a larger `g`, and that may improve performance by grouping bus transactions into longer sequential accesses. For simplicity, I will use the former code.

We have now grouped the butterflies within each iteration of `n` so that the data of the group fits in cache, but, in one iteration on `n`, many such groups are processed. As multiple iterations on `n` are executed, the data must be reloaded into cache as each group is begun. Fortunately, `k` and `n` are independent, so we can easily swap the order of their loops:

```
for (k  = 0     ; k  < 1<<N      ; k += g)
for (n  = nStage; n  < N-2       ; n += 2)
for (k0 = k>>N-n; k0 < (k+g)>>N-n; ++k0  )
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n);
```

Now we have partitioned all the butterflies in the penultimate stage into groups whose data fits in cache, and we iterate through the groups only once. Therefore, this design for the penultimate stage will read each element from memory into cache exactly once and write an element from cache to memory exactly once (assuming no external factors interfere with cache operations and that the stage starts and finishes with no elements in cache).

Cache control operations could be inserted before each iteration on `k` to read the data that will be needed for the iteration and after each iteration to write data from cache to memory and make room in cache for new data.

Within this design, we can still reorganize the calculations for computational efficiency, without affecting the cache grouping. As with the FFT kernel, the penultimate pass is best performed by specialized code. The penultimate pass of the FFT is the final pass of this stage. So, within the loop in `k`, we separate the last iteration on `n`:

```
for (k  = 0; k < 1<<N; k += g)
{
   for (n  = nStage; n  < N-4       ; n += 2)
   for (k0 = k>>N-n; k0 < (k+g)>>N-n; ++k0  )
      FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n);

   for (k0 = k>>4  ; k0 < (k+g)>>4  ; ++k0  )
      FFT_Butterflies(2, vOut, vOut, k0, 16);
}
```

Note that the original for-loop on n performs this last iteration only if "`n < N-2`" evaluates to true. This occurs if `nStage < N-2`, which is true if there is any work for the routine to do at all. Our design calls `FFT_PenultimateStage` only if there is work for it, that is, if `0 < N-2-m0`. So we can omit the test "`n < N-2`" in this code.

In the FFT kernel, we found it useful to reorder the loops to group butterflies by weight. That is possible here in the first iteration on `k`, when `k` is 0, so we will separate that iteration. In the other iterations on `k`, weights are not used repeatedly in different passes. (Compare the upper bound on `k0` when `n` is *n* to the lower bound on `k0` in the next pass, when `n` is *n*+2: `(k+g)>>N-`*n* versus `k>>N-(`*n*`+2)`. If `k` is at least `g`, as it is after the first iteration, the latter is at least twice the former. So `k0` always begins a new loop at a higher value than it ended the previous loop.) The new code with the first iteration on `k` separated is:

Early FFT_PenultimateStage

```
for (n  = nStage; n  < N-4   ; n += 2)
for (k0 = 0     ; k0 < g>>N-n; ++k0  )
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n);

for (k0 = 0     ; k0 < g>>4  ; ++k0  )
   FFT_Butterflies(2, vOut, vOut, k0, 16);

for (k = g; k < 1<<N; k += g)
{
   for (n  = nStage; n  < N-4       ; n += 2)
   for (k0 = k>>N-n; k0 < (k+g)>>N-n; ++k0  )
```

```
      FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n);

   for (k0 = k>>4  ; k0 < (k+g)>>4  ; ++k0  )
      FFT_Butterflies(2, vOut, vOut, k0, 16);
}
```

The loop:

```
for (k0 = 0; k0 < g>>4; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 16);
```

is equivalent to:

```
FFT4_1WeightPerIteration(vOut, g>>4, weights);
```

However, the loop:

```
for (k0 = k>>4; k0 < (k+g)>>4 ; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 16);
```

cannot be directly computed with `FFT4_1WeightPerIteration`, because it does not start `k0` at 0. We need a variation of `FFT4_1WeightPerIteration` that takes both lower and upper bounds:

```
FFT4_1WeightPerIterationB(vOut, k>>4, (k+g)>>4, weights);
```

The reader can see by inspecting `FFT4_1WeightPerIteration` in section 4.3.4 that the following is an implementation of `FFT4_1WeightPerIterationB`:

FFT4_1WeightPerIterationB

```
static void FFT4_1WeightPerIterationB(
   ComplexArray vOut,               // Address of output vector.
   int l0,                          // Lower bound on k0.
   int u0,                          // Upper bound on k0.
   const CommonWeight weights[]     // Array of weight values.
)
{
   FFT4_1WeightPerIteration(vOut + (l0<<4), u0-l0, weights + l0);
}
```

Now our second-stage code becomes:

```
for (n  = nStage; n  < N-4   ; n += 2)
for (k0 = 0 ; k0 < g>>N-n; ++k0  )
   FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n);

FFT4_1WeightPerIteration(vOut, g>>4, weights);

for (k = g; k < 1<<N; k += g)
{
   for (n  = nStage; n  < N-4       ; n += 2)
   for (k0 = k>>N-n; k0 < (k+g)>>N-n; ++k0  )
      FFT_Butterflies(2, vOut, vOut, k0, 1<<N-n);

   FFT4_1WeightPerIterationB(vOut, k>>4, (k+g)>>4, weights);
}
```

The two calls to `FFT_Butterflies` are efficiently computed by `FFT4_1WeightPerCall`, so we will replace them:

```
for (n  = nStage; n  < N-4   ; n += 2)
for (k0 = 0 ; k0 < g>>N-n; ++k0  )
   FFT4_1WeightPerCall(vOut, k0, 1<<N-n, weights[k0]);

FFT4_1WeightPerIteration(vOut, g>>4, weights);

for (k = g; k < 1<<N; k += g)
{
   for (n  = nStage; n  < N-4       ; n += 2)
   for (k0 = k>>N-n; k0 < (k+g)>>N-n; ++k0  )
      FFT4_1WeightPerCall(vOut, k0, 1<<N-n, weights[k0]);

   FFT4_1WeightPerIterationB(vOut, k>>4, (k+g)>>4, weights);
}
```

Finally, we wish to change the orders of the first two loops and separate the k_0=0 iteration, as we did in the FFT kernel. The derivations are the same as for the kernel, so they are left as an exercise for the reader. The only change in the resulting code is that `nLower` is initialized to `nStage` instead of “`N&1 : 3 : 2`” (each value is the starting *n* in the respective FFT structure) and the loop bound on `k0` is changed from `1<<nLower` to `g>>N-nLower`. The new code is:

FFT_PenultimateStage

```
static void FFT_PenultimateStage(
   ComplexArray vOut,              // Address of output vector.
   int nStage,                     // n at start of stage.
   int N,                          // N.
   const CommonWeight weights[]    // Array of weight values.
)
{
   int n, nLower, k, k0;

   nLower = nStage;
   for (n = nLower; n      < N-4        ; n += 2     )
      FFT4_0Weights(vOut, vOut, 1<<N-n);

   for (k0 = 1    ; nLower < N-4        ; nLower += 2)
   for (          ; k0     < g>>N-nLower; ++k0       )
   for (n = nLower; n      < N-4        ; n += 2     )
      FFT4_1WeightPerCall(vOut, k0, 1<<N-n, weights[k0]);

   FFT4_1WeightPerIteration(vOut, g>>4, weights);
   for (k = g; k < 1<<N; k += g)
   {
      for (n  = nStage; n  < N-4       ; n += 2)
      for (k0 = k>>N-n; k0 < (k+g)>>N-n; ++k0  )
         FFT4_1WeightPerCall(vOut, k0, 1<<N-n, weights[k0]);

      FFT4_1WeightPerIterationB(vOut, k>>4, (k+g)>>4, weights);
   }
}
```

7.6.4 Final Stage

When called via:

```
FFT_FinalStage(vOut, 1<<N-2, finalIndices, finalWeights);
```

`FFT_FinalStage` must perform the calculations defined by:

FFT_FinalStage Prototype

```
for (k0 = 0; k0 < 1<<N-2; ++k0)
   FFT_Butterflies(2, vOut, vOut, k0, 4);
```

and it must perform the bit-reversal permutation. This is identical to the function that `FFT4_Final` was created to implement efficiently, so `FFT_FinalStage` could be:

FFT_FinalStage

```
static void FFT_FinalStage(
   ComplexArray vOut,               // Address of output vector.
   int u0,                          // Upper bound on k0.
   const FinalIndices IndexTable[], // Array of index pairs.
   const FinalWeights weights[]     // Array of weight values.
)
{
   FFT4_Final(vOut, u0, IndexTable, weights);
}
```

However, `FFT_FinalStage` has to operate on data that does not all fit in cache simultaneously, and thus we may want to implement it as a new variant of `FFT4_Final` that includes cache-control operations.

It was mentioned in section 6.4 that the data rearrangement needed for the final butterflies meshes nicely with the data rearrangement of the bit-reversal permutation. There is an additional advantage in the out-of-cache FFT because the butterfly operations are largely calculations and intraregister data movement, while the bit-reversal permutation is largely memory reading and writing, so they can execute simultaneously.

7.7 Cache Operations

Previous sections show an FFT algorithm design that organizes the work in a way suitable for the cache architecture, primarily by arranging for data to be operated on in groups small enough to fit in cache. The fact that the data can be read into cache, kept there, and written to memory in an efficient manner does not mean that the processor will do so. We may need to direct the processor in these activities by using explicit cache operations.

Because the usefulness of cache operations is partially dependent on L2 and memory bus speeds and characteristics, this paper only lists potential performance enhancements from cache operations that might exist and does not give a definitive design.

Also, L2 cache control on the Motorola PowerPC CPU7400 and some other AltiVec processors is imperfect and does not provide all the operations we would desire.

In addition, the benefits of various operations will vary at different vector lengths. A complete design for best performance at every length therefore requires tailoring cache operations to each length.

7.7.1 Cache Operations

AltiVec processors offer a variety of cache operations. The details are beyond the scope of this paper. The operations may be categorized:

- **Load.** Data is loaded into cache in advance of its use in a computation.
- **Allocate.** Blocks are created in cache with zero or undefined data without reading from memory.
- **Mark.** Data is marked most- or least-recently-used to influence the processor's choice of blocks to remove from cache when bringing in new blocks.
- **Store.** Data is written from cache to memory (if it has been modified in cache).
- **Remove.** Data is removed from cache.

Cache operations will be discussed in these terms without addressing variants. For example, the 7400 has separate methods to load data intended only for reading and to load data intended for reading and writing. Another example is that there are instructions to store data without removing it (`dcbst`), to store data and remove it (`dcbf`), and to remove data without storing it (`dcbi`). In the former example, the choice is determined by the situation and is obvious. In the latter case, the specific instruction used is an implementation detail—the categories given above describe the operation sufficiently.

7.7.2 Allocate Buffer in Cache

The gathering step will copy data to a buffer in cache. Because the existing contents of the buffer will be completely overwritten, there is no need to read them from memory.

On the Motorola PowerPC CPU 7400, quickly issuing store instructions that fill a cache block results in the processor gathering all the stores together and writing the resulting block to cache. (This store-gathering is unrelated to the data gathering of our FFT algorithm.) If this does not occur, only part of a cache block is written. The remainder of the block must come from memory, and so reads from memory are performed. For high performance, these reads should be avoided.

On processors like the 7400, store-gathering usually provides the desired behavior. If it does not (the program does not issue store instructions sufficiently quickly) or on other processors, the buffer may be allocated in cache, to avoid reading it from memory. This step can be done once before the first gather operation and need not be repeated if the buffer is kept in cache.

7.7.3 Load Data Being Gathered

In step (1), when data is being gathered, it may be useful to load into cache parts of the data array shortly before they are read. This may also be unnecessary as the copying is limited by the rate at which data can be read from memory, and there is little other work to do while waiting for data to arrive.

7.7.4 Remove Data After Gathering

After data has been read from the array and written to a buffer, the cache blocks with images of the array are not needed for calculations and can be discarded. One might think these blocks will be needed again soon, when the scattering is done to copy the data from the buffer back to the array. However:

- The blocks generally do not remain in cache, due to the cache set size problem, which is the reason we are gathering data.
- There is no advantage in having the blocks in cache because when we write the results, they can be written to memory without reading the existing contents (using either store-gathering or another means, depending on the processor).
- Keeping the blocks in cache may result in other data being cast out of cache, such as parts of the buffer used early in the gathering or parts of the table of weights.

Thus, there may be an advantage to discarded the data after it is read, either by explicitly removing it or by marking it least-recently-used.

7.7.5 Write Results Without Reading

When the first-stage results are copied from the buffer back to the array, the issue about writing entire cache blocks exists. Again, store-gathering or another method should prevent unnecessary reads.

7.7.6 Remove Data After Scattering

After results have been copied from the buffer back to the array, they will not be used again in the first stage, so they could be removed from cache. However, they will be used again in subsequent stages.

7.7.7 Remove Buffer

When the first stage is done, the buffer contains results from the last set of butterflies. If we go on and do other work, the processor will eventually recognize that data in cache has not been used and will select it for removal. To remove the data from cache, the processor will write it to memory. We do not want that to occur. To avoid it, we could remove the buffer.

7.7.8 Penultimate Stage

The penultimate stage might benefit from loading the data of each group before it is used, loading the weights used by each group of data before they are used, and either storing and removing the data after each group is done or marking the data least-recently-used.

The weights could also be removed. However, some weights are used by more than one group in the penultimate stage, so we would wish to remove only weights that will not be used again, or perhaps to refrain from removing weights until the end of the stage.

7.7.9 Final Stage

The final stage is driven by a table of indices as is not readily amenable to cache operations. Because the table entries are non-sequential, it is not possible to usefully issue sequential load operations for the data in final stage. Some attempt could be made to read the table ahead of loading the data and issue individual loads.

The weights are read sequentially, since the weight table is prepared to match the index table, so weights could be loaded in advance.

After being used, results and weights could be removed from cache or marked least-recently-used.

7.7.10 After the FFT

When the FFT is complete, the application using it will go on to other things, and the FFT may be able to enhance performance by leaving the cache in a state useful to the application. Because the last operation in the FFT is the non-sequential final pass with bit-reversal, there is not a good description of what remains in cache.

If the memory bus in use performs sequential accesses more efficiently than non-sequential accesses, it may be useful to ensure that cache is stored after completion of an FFT on a long vector. Then, when the application goes on to other work, it may load data sequentially and benefit from the faster execution of sequential accesses. If modified data were left in cache by the FFT, the reading of new data would have to be interleaved with the writing of FFT results, resulting in non-sequential accesses on the memory bus.

8 Reverse DFT

The reverse-DFT of a 2^N-element vector **H** is the vector **h**:

$$h_k = \frac{1}{2^N} \sum_{0 \le j < 2^N} \mathbf{1}^{-\frac{jk}{2^N}} H_j \text{ for } 0 \le k < 2^N.$$

Some algebra will show that the reverse-DFT is the inverse of the DFT, that the reverse-DFT of the DFT of **h** is **h**.

The original DFT may be called the forward-DFT.

Aside from swapping **H** and **h**, the definition of the reverse-DFT differs from the definition of the DFT in two ways: The exponent is negated and all results are multiplied by $1/2^N$.

8.1 Conjugating Elements

Let $\mathbf{V}^*$ denote the vector formed by exchanging the real and imaginary components of each element of a vector **V**. That is, if $V_k=a+bi$, then $V^*_k=b+ai$. Let DFT(**V**) denote the DFT of a vector **V**. Then $\mathbf{h}=(\mathrm{DFT}(1/2^N \cdot \mathbf{H}^*)^*$. In other words, we can perform a reverse DFT by exchanging the real and imaginary components of a vector **H**, multiplying the result by $1/2^N$, taking the DFT, and exchanging the real and imaginary components again.

To see that this is so, observe that $b+ai=i(a-bi)=i\overline{(a+bi)}$, where $\overline{a+bi}$ denotes the complex conjugate of $a+bi$. That is, swapping the real and imaginary components is equivalent to conjugating and multiplying by i. Then $(\mathrm{DFT}(1/2^N \cdot \mathbf{H}^*)^*$ is:

$$\overline{i\mathrm{DFT}\left(\frac{1}{2^N} i\overline{\mathbf{H}}\right)}.$$

Element k of this vector is:

$$\overline{i \sum_{0\le j<2^N} \mathbf{1}^{\frac{jk}{2^N}} \frac{1}{2^N} i\overline{H_j}} = i\, \overline{\sum_{0\le j<2^N} \mathbf{1}^{\frac{jk}{2^N}} \frac{1}{2^N} i\overline{H_j}} = i \sum_{0\le j<2^N} \overline{\mathbf{1}^{\frac{jk}{2^N}}}\, \overline{\frac{1}{2^N}}\, \overline{i}\, \overline{\overline{H_j}}$$

$$= i \sum_{0\le j<2^N} \mathbf{1}^{-\frac{jk}{2^N}} \frac{1}{2^N}(-i)H_j = i\frac{1}{2^N}(-i) \sum_{0\le j<2^N} \mathbf{1}^{-\frac{jk}{2^N}} H_j = \frac{1}{2^N} \sum_{0\le j<2^N} \mathbf{1}^{-\frac{jk}{2^N}} H_j\ .$$

Thus element k of $(\mathrm{DFT}(1/2^N \cdot \mathbf{H}^*)^*$ is indeed element k of the reverse-DFT of **H**. Thus, we can compute a reverse-DFT using a DFT if we implement two additional things: exchanging the real and imaginary components before and after the DFT and multiplying by $1/2^N$.

If arrays of complex numbers are implemented with two pointers to arrays, one for the real components and one for the imaginary components, then exchanging the components in each element of **H** is implemented simply by exchanging the two pointers.

If arrays of complex numbers are implemented as arrays of pairs of real and imaginary components, then the data must actually be exchanged before and after the DFT. This need not involve any additional work. An alternate version of `FFT4_0Weights` or `FFT8_0_Weights` can exchange the components as it loads them, and an alternate version of `FFT4_Final` can exchange the components as it stores them.

The former is used in the demonstration code. The latter is easily implemented, although it requires duplicate some amount of code.

8.2 Scaling in the Butterfly Routines

This leaves the matter of multiplying the data by $1/2^N$. Our choices for this are quite flexible. The DFT is linear, so the multiplications can be inserted before or after the DFT with the same results. The multiplications can even be inserted inside the FFT computation if done in a consistent manner.

Consider this variant of `FFT4_0Weights`:

FFT4_0WeightsScale

```
static void FFT4_0WeightsScale(
   ComplexArray vOut,   // Address of output vector.
   ComplexArray vIn,    // Address of input vector.
   int c0,              // Coefficient for k0.
   float scale          // Scale for reverse transform.
)
{
   // Coefficient for k1 is coefficient for k0 divided by 1<<m.
   const int c1 = c0 >> 2;
   int k2;
   float a0r, a0i, a1r, a1i, a2r, a2i, a3r, a3i,
         c0r, c0i, c1r, c1i, c2r, c2i, c3r, c3i,
         d0r, d0i, d1r, d1i, d2r, d2i, d3r, d3i;

   for (k2 = 0; k2 < c1; ++k2)
   {
      a0r = vIn.re[c1*0 + k2] * scale;
      a0i = vIn.im[c1*0 + k2] * scale;
      a1r = vIn.re[c1*1 + k2];
      a1i = vIn.im[c1*1 + k2];
      a2r = vIn.re[c1*2 + k2];
      a2i = vIn.im[c1*2 + k2];
      a3r = vIn.re[c1*3 + k2];
      a3i = vIn.im[c1*3 + k2];
      c0r = + a2r * scale + a0r;
      c0i = + a2i * scale + a0i;
      c2r = - a2r * scale + a0r;
      c2i = - a2i * scale + a0i;
      c1r = + a3r + a1r;
      c1i = + a3i + a1i;
      c3r = - a3r + a1r;
      c3i = - a3i + a1i;
      d0r = + c1r * scale + c0r;
      d0i = + c1i * scale + c0i;
      d1r = - c1r * scale + c0r;
      d1i = - c1i * scale + c0i;
      d2r = - c3i * scale + c2r;
      d2i = + c3r * scale + c2i;
      d3r = + c3i * scale + c2r;
      d3i = - c3r * scale + c2i;
      vOut.re[c1*0 + k2] = d0r;
      vOut.im[c1*0 + k2] = d0i;
      vOut.re[c1*1 + k2] = d1r;
      vOut.im[c1*1 + k2] = d1i;
      vOut.re[c1*2 + k2] = d2r;
      vOut.im[c1*2 + k2] = d2i;
      vOut.re[c1*3 + k2] = d3r;
      vOut.im[c1*3 + k2] = d3i;
   }
}
```

The intent in this routine is that the caller will pass $1/2^N$ in `scale`, and the routine will produce results as if all the input data were multiplied by `scale`. This could be accomplished simply by multiplying each input number by `scale`. However, the above code takes advantage of the availability of a fused multiply-add instruction that will perform a multiplication and an addition in the same time as an add. All but two of the multiplications have been incorporated into existing additions.

Notice that `a0r` and `a0i` are multiplied by `scale`, yielding scaled results. Then, wherever `FFT4_0Weights` originally added an unscaled number to a scaled number, `FFT4_0WeightsScale` multiplies the unscaled number by scale as it adds it to the scaled number, yielding a consistent scaled result. By the end of the routine, all results are properly scaled.

Similar changes can be made to `FFT8_0Weights` to produce `FFT8_0WeightsScale`. `FFT8_0Weights` already includes multiplications, by a symbol named `sqrt2d2`, but the contents of that symbol can be multiplied by `scale` to get the desired result, as shown here:

FFT8_0WeightsScale

```
static void FFT8_0WeightsScale(
   ComplexArray vOut,   // Address of output vector.
   ComplexArray vIn,    // Address of input vector.
   int c0,              // Coefficient for k0.
   float scale          // Scale for reverse transform.
)
{
   // Prepare a constant, sqrt(2)/2, with the scaling incorporated.
   const float sqrt2d2 = .7071067811865475244 * scale;
   // Coefficient for k1 is coefficient for k0 divided by 1<<m.
   const int c1 = c0 >> 3;
   int k2;
   float a0r, a0i, a1r, a1i, a2r, a2i, a3r, a3i,
         a4r, a4i, a5r, a5i, a6r, a6i, a7r, a7i,
         b0r, b0i, b1r, b1i, b2r, b2i, b3r, b3i,
         b4r, b4i, b5r, b5i, b6r, b6i, b7r, b7i,
         c0r, c0i, c1r, c1i, c2r, c2i, c3r, c3i,
         c4r, c4i, c5r, c5i, c6r, c6i, c7r, c7i,
         d0r, d0i, d1r, d1i, d2r, d2i, d3r, d3i,
         d4r, d4i, d5r, d5i, d6r, d6i, d7r, d7i,
         t5r, t5i, t7r, t7i;

   for (k2 = 0; k2 < c1; ++k2)
   {
      a0r = vIn.re[c1*0 + k2] * scale;
      a0i = vIn.im[c1*0 + k2] * scale;
      a1r = vIn.re[c1*1 + k2];
```

```
        a1i = vIn.im[c1*1 + k2];
        a2r = vIn.re[c1*2 + k2];
        a2i = vIn.im[c1*2 + k2];
        a3r = vIn.re[c1*3 + k2];
        a3i = vIn.im[c1*3 + k2];
        a4r = vIn.re[c1*4 + k2];
        a4i = vIn.im[c1*4 + k2];
        a5r = vIn.re[c1*5 + k2];
        a5i = vIn.im[c1*5 + k2];
        a6r = vIn.re[c1*6 + k2];
        a6i = vIn.im[c1*6 + k2];
        a7r = vIn.re[c1*7 + k2];
        a7i = vIn.im[c1*7 + k2];
        b0r = a0r + a4r * scale;      // w = 1.
        b0i = a0i + a4i * scale;
        b1r = a1r + a5r;
        b1i = a1i + a5i;
        b2r = a2r + a6r;
        b2i = a2i + a6i;
        b3r = a3r + a7r;
        b3i = a3i + a7i;
        b4r = a0r - a4r * scale;
        b4i = a0i - a4i * scale;
        b5r = a1r - a5r;
        b5i = a1i - a5i;
        b6r = a2r - a6r;
        b6i = a2i - a6i;
        b7r = a3r - a7r;
        b7i = a3i - a7i;
        c0r = b0r + b2r * scale;      // w = 1.
        c0i = b0i + b2i * scale;
        c1r = b1r + b3r;
        c1i = b1i + b3i;
        c2r = b0r - b2r * scale;
        c2i = b0i - b2i * scale;
        c3r = b1r - b3r;
        c3i = b1i - b3i;
        c4r = b4r - b6i * scale;      // w = i.
        c4i = b4i + b6r * scale;
        c5r = b5r - b7i;
        c5i = b5i + b7r;
        c6r = b4r + b6i * scale;
        c6i = b4i - b6r * scale;
        c7r = b5r + b7i;
        c7i = b5i - b7r;
        t5r = c5r - c5i;
        t5i = c5r + c5i;
        t7r = c7r + c7i;
        t7i = c7r - c7i;
        d0r = c0r + c1r * scale;      // w = 1.
        d0i = c0i + c1i * scale;
        d1r = c0r - c1r * scale;
        d1i = c0i - c1i * scale;
        d2r = c2r - c3i * scale;      // w = i.
        d2i = c2i + c3r * scale;
        d3r = c2r + c3i * scale;
        d3i = c2i - c3r * scale;
```

```
      d4r = + t5r * sqrt2d2 + c4r;  // w = sqrt(2)/2 * (+1+i).
      d4i = + t5i * sqrt2d2 + c4i;
      d5r = - t5r * sqrt2d2 + c4r;
      d5i = - t5i * sqrt2d2 + c4i;
      d6r = - t7r * sqrt2d2 + c6r;  // w = sqrt(2)/2 * (-1+i).
      d6i = + t7i * sqrt2d2 + c6i;
      d7r = + t7r * sqrt2d2 + c6r;
      d7i = - t7i * sqrt2d2 + c6i;
      vOut.re[c1*0 + k2] = d0r;
      vOut.im[c1*0 + k2] = d0i;
      vOut.re[c1*1 + k2] = d1r;
      vOut.im[c1*1 + k2] = d1i;
      vOut.re[c1*2 + k2] = d2r;
      vOut.im[c1*2 + k2] = d2i;
      vOut.re[c1*3 + k2] = d3r;
      vOut.im[c1*3 + k2] = d3i;
      vOut.re[c1*4 + k2] = d4r;
      vOut.im[c1*4 + k2] = d4i;
      vOut.re[c1*5 + k2] = d5r;
      vOut.im[c1*5 + k2] = d5i;
      vOut.re[c1*6 + k2] = d6r;
      vOut.im[c1*6 + k2] = d6i;
      vOut.re[c1*7 + k2] = d7r;
      vOut.im[c1*7 + k2] = d7i;
   }
}
```

FFT4_0WeightsScale executes two more multiplications than FFT4_0Weights. These are necessary with this implementation of the reverse-DFT. For highest performance with the DFT, an implementation might either use FFT4_0Weights or implement FFT4_0WeightsScale in assembly language in a way that allows it to avoid the additional time for the unnecessary multiplications when doing a DFT. The same is true of FFT8_0WeightsScale.

8.3 Changing the Kernels

Having created these variants, it is necessary to use them. The kernel changes one last time:

FFT Kernel with Scaling for Reverse Transform

```
static void FFT_Kernel(
   ComplexArray vOut,               // Address of output vector.
   ComplexArray vIn,                // Address of input vector.
   int N,                           // N.
   int direction,                   // Transform direction.
   const CommonWeight *weights,     // Common weight values.
   const FinalIndices *finalIndices,// Index pairs.
   const FinalWeights *finalWeights // Final weight values.
)
{
   const float scale = direction == -1 ? 1./(1<<N) : 1.;

   int n, nLower, k0;
   if (N & 1)
      FFT8_0WeightsScale(vOut, vIn, 1<<N, scale);
   else
      FFT4_0WeightsScale(vOut, vIn, 1<<N, scale);

   nLower = N&1 ? 3 : 2;
   for (n = nLower; n      < N-4      ; n +=2      )
      FFT4_0Weights(vOut, vOut, 1<<N-n);

   for (k0 = 1    ; nLower < N-4      ; nLower += 2)
   for (          ; k0     < 1<<nLower; ++k0       )
   for (n = nLower; n      < N-4      ; n += 2     )
      FFT4_1WeightPerCall(vOut, k0, 1<<N-n, weights[k0]);

   if (n < N-2)
      FFT4_1WeightPerIteration(vOut, 1<<N-4, weights);

   FFT4_Final(vOut, 1<<N-2, finalIndices, finalWeights);
}
```

`FFT_MultipleStages` also changes:

Multiple-Stage Kernel with Scaling for Reverse Transform

```
static void FFT_MultipleStages(
   ComplexArray vOut,                // Address of output vector.
   ComplexArray vIn,                 // Address of input vector.
   int N,                            // N.
   int direction,                    // Transform direction.
   const CommonWeight *weights,      // Common weight values.
   const FinalIndices *finalIndices,// Index pairs.
   const FinalWeights *finalWeights // Final weight values.
)
{
   const float scale = direction == -1 ? 1./(1<<N) : 1.;

   int m0 = N&1 ? 9 : 8;

   FFT_FirstStage(m0, vOut, vIn, 1<<N, scale, weights);

   if (0 < N-2-m0)
      FFT_PenultimateStage(vOut, m0, N, weights);

   FFT_FinalStage(vOut, 1<<N-2, finalIndices, finalWeights);
}
```

The auxiliary routine `FFT_FirstStage` must pass `scale` along:

FFT_FirstStage with Scaling for Reverse Transform

```
static void FFT_FirstStage(
   int m,                            // log2 of butterfly radix.
   ComplexArray vOut,                // Address of output vector.
   ComplexArray vIn,                 // Address of input vector.
   int c0,                           // Coefficient for c0.
   float scale,                      // Scale for reverse transform.
   const CommonWeight weights[]      // Array of weight values.
)
{
   // Coefficient for k1 is coefficient for c0 divided by 1<<m.
   const int c1 = c0 >> m;
   const int u1 = 1<<m;

   // Cluster size is how many sets fit in buffer at one time.
   const int cluster = b >> m;

   int n, nLower, k0, k2;

   // Process values of k2 in clusters.
   for (k2 = 0; k2 < c1; k2 += cluster)
   {
      Gather(buffer, vIn, u1, c1, k2, cluster);

      if (m & 1)
         FFT8_0WeightsScale(buffer, buffer, b, scale);
      else
         FFT4_0WeightsScale(buffer, buffer, b, scale);

      nLower = m&1 ? 3 : 2;
      for (n = nLower; n      < m        ; n += 2     )
         FFT4_0Weights(buffer, buffer, b>>n);

      for (k0 = 1    ; nLower < m        ; nLower += 2)
      for (          ; k0     < 1<<nLower; ++k0       )
      for (n = nLower; n      < m        ; n += 2     )
         FFT4_1WeightPerCall(buffer, k0, b>>n, weights[k0]);

      Scatter(vOut, buffer, u1, c1, k2, cluster);
   }
}
```

8.4 Alternatives

Butterfly routines other than `FFT4_0Weights` or `FFT8_0Weights` could be chosen for performing the scaling multiplications. A significant disadvantage of using any other routine is that all other routines include multiplications by weights and cannot incorporate the scaling multiplications without extra computations unless the scale is incorporated into the weights. This would require separate tables of weights for the forward-DFT and the reverse-DFT. If that is acceptable, then `FFT4_Final` may be another good candidate for the scaling multiplications because:

- It is used in only one pass (and we do not want to scale the data more than once).
- It uses separate weights (so only the final weights have to be doubled for the forward- and reverse-DFT, not the common weights).
- Its implementation might have some compute time available for additional multiplications, since the routine is burdened with loads, stores, and permutations of elements.

When performing the DFT with the in-cache kernel, the data of the vector being transformed is loaded only in the butterfly routines. Adding more loads of the data would hurt performance, so the scaling for the reverse-DFT must be incorporated into one of the butterfly routines. In the multiple-stage kernel, the data is also loaded in the `Gather` and `Scatter` routines. Since these routines are memory copy operations and are free of computations, they may be able to do the scaling multiplications with little or no extra time consumed. This possibility is not examined further in this paper.

9 Executing the FFT

All the parts that will execute the FFT have been designed. Now we need to call those parts, to design one central routine that will execute the entire FFT. This requires obtaining the constants that the FFT will use, choosing the single-stage in-cache `FFT_Kernel` or the out-of-cache `FFT_MultipleStages`, and executing the chosen routine.

9.1 Constants

To manage the constants, we use a structure that holds pointers to each of the types we need:

ConstantsSet

```
typedef struct {
   const CommonWeight   *commonWeights;
   const FinalWeights   *finalWeights;
   const FinalIndices   *finalIndices;
} ConstantsSet;
```

Before calling `FFT_Kernel` or `FFT_MultipleStages`, the FFT needs to get the constants. We will use a routine named `GetConstants` to manage the tables. This routine will:

- Allocate space for and generate any tables of constants needed.
- Keep tables for future use.
- Return existing tables when available rather than generating them again.
- Keep one table of common weights for all lengths up to the longest requested length.
- Keep one table of indices and one table of final weights for each requested length.

For the common weights, a pointer to the existing table is kept in `CommonWeights`, and the longest vector length supported by that table is kept in `CommonLength`. Given a requested to provide a table for a vector of length `length`, we compare it to `CommonLength` to see if the existing table is long enough. If it is not, we generate a new table. Then the table is returned in the structure of table pointers (or 1 is returned to indicate an error):

```
if (CommonLength < length)
   if (GenerateCommonWeights(&CommonWeights, &CommonLength,
         length) != 0)
      return 1;
set->commonWeights = CommonWeights;
```

For final pass indices, a separate table is needed for each supported vector length. So an array of pointers is kept in `FinalIndices`, and a method is needed to select an element in the table based on the vector length. The precise method is unimportant, and this statement suffices to provide an element index given a vector length:

```
const int hash = ilog2(length);
```

Having selected an element in the array, we check it to see if there is already a table of indices for this vector length. If there is not, we generate one. Then the pointer to the table is returned in the structure pointers:

```
if (FinalIndices[hash] == NULL)
   if (GenerateFinalIndices(&FinalIndices[hash], length) != 0)
      return 1;
set->finalIndices = FinalIndices[hash];
```

The preparation of the table of final-pass weights is similar. The complete routine is shown below. `d` is passed as a parameter but not used. This allows for the possibility that scaling for the reverse-DFT could be incorporated into the tables of weights in a modified design, and `GetConstants` would need to return different tables for different values of `d`.

GetConstants

```
static int GetConstants(
   ConstantsSet *set,   // Structure in which to return pointers.
   int length,          // Length of vector to be transformed.
   int d                // Direction of transform.
)
{
   static CommonWeight  *CommonWeights = NULL;
   static int           CommonLength = 0;
   static FinalWeights  *FinalWeights[32] = { NULL };
   static FinalIndices  *FinalIndices[32] = { NULL };

   const int hash = ilog2(length);

   if (CommonLength < length)
      if (GenerateCommonWeights(&CommonWeights, &CommonLength,
            length) != 0)
         return 1;
   set->commonWeights = CommonWeights;

   if (FinalIndices[hash] == NULL)
      if (GenerateFinalIndices(&FinalIndices[hash], length) != 0)
         return 1;
   set->finalIndices = FinalIndices[hash];

   if (FinalWeights[hash] == NULL)
      if (GenerateFinalWeights(&FinalWeights[hash], length,
            FinalIndices[hash]) != 0)
         return 1;
   set->finalWeights = FinalWeights[hash];

   return 0;
}
```

9.2 FFT Routine

Finally we can write our main FFT routine:

FFT

```
int FFT(
   float *re,     // Address of real components.
   float *im,     // Address of imaginary components.
   int N,         // Base-two logarithm of length of vector.
   int d          // Direction of transform.
)
{
   ConstantsSet constants;
   ComplexArray v(re, im);

   /* To perform a transform in the reverse direction, first
      swap the real and imaginary components.  Scaling will be
      done later.
   */
   if (d < 0)
      v = ComplexArray(im, re);

   // This FFT does not support N < 4.
   if (N < 4)
      return 1;

   /* This FFT does not support long vectors that overflow the
      field size in the indices.
   */
   if (CHAR_BIT * sizeof constants.finalIndices->read + 4 < N)
      return 1;

   // Get the constants.
   if (0 != GetConstants(&constants, 1<<N, d))
      return 1;

   // If n is small, do the single-stage FFT.
   if (1<<N < 32768 / (sizeof *re + sizeof *im))
      FFT_Kernel(v, v, N, d, constants.commonWeights,
         constants.finalIndices, constants.finalWeights);

   // If n is large, do the multiple-stage FFT.
   else
      FFT_MultipleStages(v, v, N, d, constants.commonWeights,
         constants.finalIndices, constants.finalWeights);

   return 0;
}
```

A Generating Radix-8 Butterfly with *Maple*

The following *Maple* (version 7.00) code generates the assignment statements used in the weightless radix-8 butterfly (section 4.3.3), except that the use of `t5r`, `t5i`, `t7r`, and `t7i` was added manually to eliminate common subexpressions.

`one(x)` is a convenient notation for $e^{2\pi i x}$.

```
> one := x -> exp(2*Pi*I*x):
```

`r(n)` gives the number obtained by writing `n` in binary and rotating its bits around the binary point.

```
> r := proc(n) option remember;
    if n=0 then 0 else (irem(n,2)+r(iquo(n,2)))/2 fi end:
```

`vp(N, n, m, k)` gives the k^{th} element of $\mathbf{v}_{n+m}$ as an expression of elements in $\mathbf{v}_n$. E.g., for a 1024-element FFT, `v(10, 3, 0, 4)` gives element four of the third pass in terms of original (pass zero) input elements, and `v(10, 2, 1, 4)` gives element four of the third pass (2+1) in terms of elements in pass 2.

```
> vp := proc(N, n, m, k)
    local k0, k1, k2, j1;
```

Separate *k* into bit fields of length *n*, *m*, and *N*-*n*-*m*.

```
    k0 := iquo(k, 2^(N-n));
    k1 := iquo(irem(k, 2^(N-n)), 2^(N-n-m));
    k2 := irem(k, 2^(N-n-m));
```

Write $v_{n+m,k}$ as a sum of elements in $\mathbf{v}_n$.

```
    v[n+m, k] = sum(
          one(j1*r(k1)) * one(r(2^m*k0)) ^ j1
          * v[n, 2^(N-n)*k0 + 2^(N-n-m)*j1 + k2],
       j1=0 .. 2^m-1);
end:
```

`ExpandParts` converts each reference to an element $v_{n,k}$ into real and imaginary parts with new names. The name takes the form `<letter><number><part>`, where:
`<letter>` is derived from *n*: 0 becomes `a`, 1 becomes `b`, etc.
`<number>` is the value of *k*.
`<part>` is “`r`” or “`i`” for real or imaginary.
For example, `v[2, 4]` is converted to `b4r + I*b4i`.

```
> ExpandParts := proc(e)
```

Apply the procedure `x->`… to each occurrence of `v[integer, anything]` in the expression `e`.

```
    subsindets(e, v[integer, anything],
```

`x->`… applies `y->`… to the `` `r` `` and the `` `i` `` in `` `r`+I*`i` ``.

```
       x -> subsindets(`r`+I*`i`, symbol,
```

`y->`… concatenates a null string (to ensure string type), a letter derived from the first subscript of `x`, the value of the second subscript of `x`, and the name in `y` (which is `` `r` `` or `` `i` ``).

```
          y -> cat(``, StringTools[Char](97+op(1, x)),
             op(2, x), y)
       )
    );
end:
```

`ExpandPartsCompound` separates, expands, and simplifies the real and imaginary parts of a list of equations.

`ExpandParts` is applied to the list, and then each object in the list is replaced with two objects, one for its real components and one for its imaginary components.

Since each object in the list is expected to be an equation, and `Re` and `Im` cannot be applied to equations, `map` is used to apply `Re` and `Im` to the parts of the object.

```
> ExpandPartsCompound := proc(l)
```
Third, simplify the expression to get the separate components.
```
    evalc(map(
```
Second, request the real and imaginary components.
```
        t -> (map(Re, t), map(Im, t)),
```
First, expand and rename the real and imaginary parts.
```
        ExpandParts(l)
    )):
end:
```

Use `vp` to express each element *k* of each radix-2 pass *n* of a 2^3-element FFT in terms of elements of the previous pass.

```
> t0 := subs(N=3, '[seq(seq(vp(N, n-1, 1, k), k=0..2^N-1),
    n=1..N)]'):
```

Print each equation "linearly," suitable for cutting and pasting.

```
> map(lprint, ExpandPartsCompound(t0)):
b0r = a0r+a4r
b0i = a0i+a4i
b1r = a1r+a5r
b1i = a1i+a5i
b2r = a2r+a6r
b2i = a2i+a6i
b3r = a3r+a7r
b3i = a3i+a7i
b4r = a0r-a4r
b4i = a0i-a4i
b5r = a1r-a5r
b5i = a1i-a5i
b6r = a2r-a6r
b6i = a2i-a6i
b7r = a3r-a7r
b7i = a3i-a7i
c0r = b0r+b2r
c0i = b0i+b2i
c1r = b1r+b3r
c1i = b1i+b3i
c2r = b0r-b2r
c2i = b0i-b2i
c3r = b1r-b3r
c3i = b1i-b3i
c4r = b4r-b6i
c4i = b4i+b6r
c5r = b5r-b7i
```

```
c5i = b5i+b7r
c6r = b4r+b6i
c6i = b4i-b6r
c7r = b5r+b7i
c7i = b5i-b7r
d0r = c0r+c1r
d0i = c0i+c1i
d1r = c0r-c1r
d1i = c0i-c1i
d2r = c2r-c3i
d2i = c2i+c3r
d3r = c2r+c3i
d3i = c2i-c3r
d4r = c4r+1/2*2^(1/2)*c5r-1/2*2^(1/2)*c5i
d4i = c4i+1/2*2^(1/2)*c5i+1/2*2^(1/2)*c5r
d5r = c4r-1/2*2^(1/2)*c5r+1/2*2^(1/2)*c5i
d5i = c4i-1/2*2^(1/2)*c5i-1/2*2^(1/2)*c5r
d6r = c6r-1/2*2^(1/2)*c7r-1/2*2^(1/2)*c7i
d6i = c6i-1/2*2^(1/2)*c7i+1/2*2^(1/2)*c7r
d7r = c6r+1/2*2^(1/2)*c7r+1/2*2^(1/2)*c7i
d7i = c6i+1/2*2^(1/2)*c7i-1/2*2^(1/2)*c7r
```

B Notes About C Source Code

B.1 Indentation

Because there is a limited width available to display code in this paper, I contract some of normal indenting when showing loops or conditional statements. For example, code that is more usually written:

```
for (k0 = l0; k0 < u0; ++k0)
   for (k1 = l1; k1 < u1; ++k1)
      for (k2 = l2; k2 < u2; ++k2)
         function(k0, k1, k2);
```

may instead be written:

```
for (k0 = l0; k0 < u0; ++k0)
for (k1 = l1; k1 < u1; ++k1)
for (k2 = l2; k2 < u2; ++k2)
   function(k0, k1, k2);
```

I hope the reader will not find this confusing. The latter set of loops might be thought of as one three-dimensional loop instead of three one-dimensional loops.

B.2 Complex Number Representation

The code displays use a type `complex` that is not defined in this paper but implements normal complex arithmetic.

C++ implementations of `complex` and `ComplexArray` are given in the demonstration code that supplements this paper. These implementations provide convenience features that make the demonstration code appear simple but would be atrocious to implement in a real application. Those features solely illustrate the design, particularly the intermediate stages of development. None of them are needed in a final implementation.

B.3 Memory Allocation and Alignment

The routines for generating weights are shown using `realloc` to allocate memory. In an AltiVec implementation, these arrays should be aligned to multiples of 16 bytes, and so an allocation routine that guarantees this should be used, such as the `vec_realloc` described in Motorola's *AltiVec: The Programming Interface Manual*.

B.4 Bit-Reversed Bytes

This code generates the table of bit-reversed bytes in routine `rw` in section 5.2:

Generate Bit-Reversed Bytes for rw

```
#include <stdio.h>

static int rw(int i) {
   int r, t;
   for (r = t = 0; t < 8; ++t, i>>=1)
      r = r << 1 | i & 1;
   return r;
}

int main(void) {
   int i;
   for (i = 0; i < 256; ++i)
      printf("%3d,%c", rw(i), i % 16 == 15 ? '\n' : ' ');
   return 0;
}
```