Freescale Semiconductor
Application Note

Document Number: AN2114
Rev. 4, 04/2013

Complex Fixed-Point Fast Fourier Transform Optimization for AltiVec™

This document compares the performance of fast Fourier transform (FFT) with and without AltiVec™ technology to demonstrate how mathematically-intensive code can be adapted for use with AltiVec and how AltiVec increases code performance. To locate published updates for this document, see the website listed on the last page of this document.

Contents

1 Overview

Fourier transforms convert a signal to and from the frequency domain. Just as a glass prism may display the spectrum of an incoming light wave, Fourier transforms break a signal down into its frequency components. This process involves writing a signal as a summation of sines and cosines. Equation 1 shows a typical algorithm used for this purpose, the discrete Fourier transform (DFT), and Equation 2 calculates the DFT of a discrete signal using a set of symmetric points around a unit circle, W_N.

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{kn}$$

Equation 1. DFT Equation

where x[n] = discrete-time signal
X[k] = frequency domain components
N = Number of Points
k = 0,1,2,..., N-1
W_N is a multiplicand factor and is shown in Equation 2

$$W_N[n] = e^{-j\left(\frac{2\Pi n}{N}\right)}$$

Equation 2. W_N Value

The DFT is of the order $O(N^2)$ for a signal of length N (an algorithm is said to have an order of N, O(N), if it computes in a scalar multiple of N iterations). The fast Fourier transform (FFT) reduces the number of calculations of the DFT by dividing the initial function into repeated subfunctions and continues this process until the subfunction is no longer divisible. For a detailed description of the derivation of the FFT formula and the various types of FFTs available, see *Inside The FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*, by Eleanor Chu and Alan George.

This equation shows the decimation of the DFT algorithm from Equation 1.

$$X[k] = \left(\sum_{n=0}^{\frac{N}{2}-1} \left(x[n] + x\left[n + \frac{N}{2}\right] \right) \cdot W_{\frac{N}{2}}^{kn} \right) + \left(\sum_{n=0}^{\frac{N}{2}-1} \left(\left(x[n] - x\left[n + \frac{N}{2}\right] \right) W_N^n \right) \cdot W_{\frac{N}{2}}^{kn} \right)$$

Equation 3. DFT subdivided into the Radix-2 Decimation in Frequency (DIF)

In Equation 3, the Radix-2 Decimation-in-Frequency (DIF), FFT divides the DFT problem into two subproblems, each of which equals half the original sum. Note that, in this example, the FFT is a DIF because it decimates the frequency components (X[k]) of the DFT problem. In comparison, if the FFT is a DIT, it decimates the time components (x[n]).

The Radix-2 DIF FFT is not the fastest algorithm. If faster algorithms are desired, check on a higher radix algorithm, such as Radix-4, which divides the equation into four subproblems.

This implementation of the FFT uses integer data types, specifically signed short integers, which have a range of –32768 to 32767. For the purposes of calculation, sinusoidal values that natively range from –1

to 1 are scaled between these values. Fixed-point FFT implementations such as these have the following advantages over floating-point FFT implementations:

- Integer math tends to execute faster than floating-point calculations, and this is also true for AltiVec. With AltiVec, not only do the integer instructions complete in less time, but eight signed shorts fit in a vector versus only four for floating-point values.
- Fixed-point FFTs are used because input data frequently comes directly from an analog-to-digital converter (A/D). A/Ds typically provide a stream of fixed-point data, so using a fixed-point FFT eliminates the need to convert the data into another format.

However, some problems are intrinsic to using fixed-point mathematics. Signed short integers cover a much smaller range of values than floating-point values. Therefore, overflow and underflow during mathematical operations are a concern. Precautions to prevent overflow and underflow can add extra instructions, decreasing performance, and/or may scale the input data limiting the accuracy and precision of the algorithm. By providing saturation logic with many of its instructions, Altivec assists in decreasing the performance hit. The methods used to keep the data within bounds are described in the sections below.

2 Signal Flow Graph for the Scalar and Vector FFTs

This section details two functions that perform the FFT. The first performs a fixed-point, signed short, complex Radix-2 DIF FFT without using AltiVec instructions. The second function performs the same transform using AltiVec instructions. These two functions are written as similarly as possible to highlight the strengths of the AltiVec technology.

When coding FFTs, the signal flow graph (SFG) of the equations resemble a butterfly. The butterfly equivalent to Equation 3 is shown in Figure 1. Note how the original x[n] values can be replaced to constitute the new sets of coefficients.

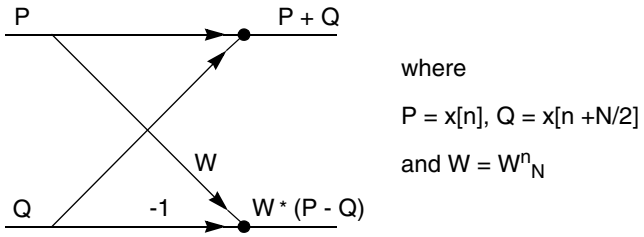

Figure 1. Single Butterfly Representation of a Signal Flow Graph

Figure 2 shows an SFG for a data set of size N = 8. Inputs are on the left and calculations flow from left to right. This DIF FFT takes the input signal in order and produces the output in bit-reversed order. In bit reversed order, each output index is represented as a binary number and the indices' bits are reversed. For example, in an eight-point DIF FFT, sequential order of the indices' bits is 000, 001, 010, 011, and so on. Reversing these bits yields 000, 100, 010, 110 and so on. This sequence corresponds to 0, 4, 2, 6, and so on in decimal notation, which is the output order shown in Figure 2.

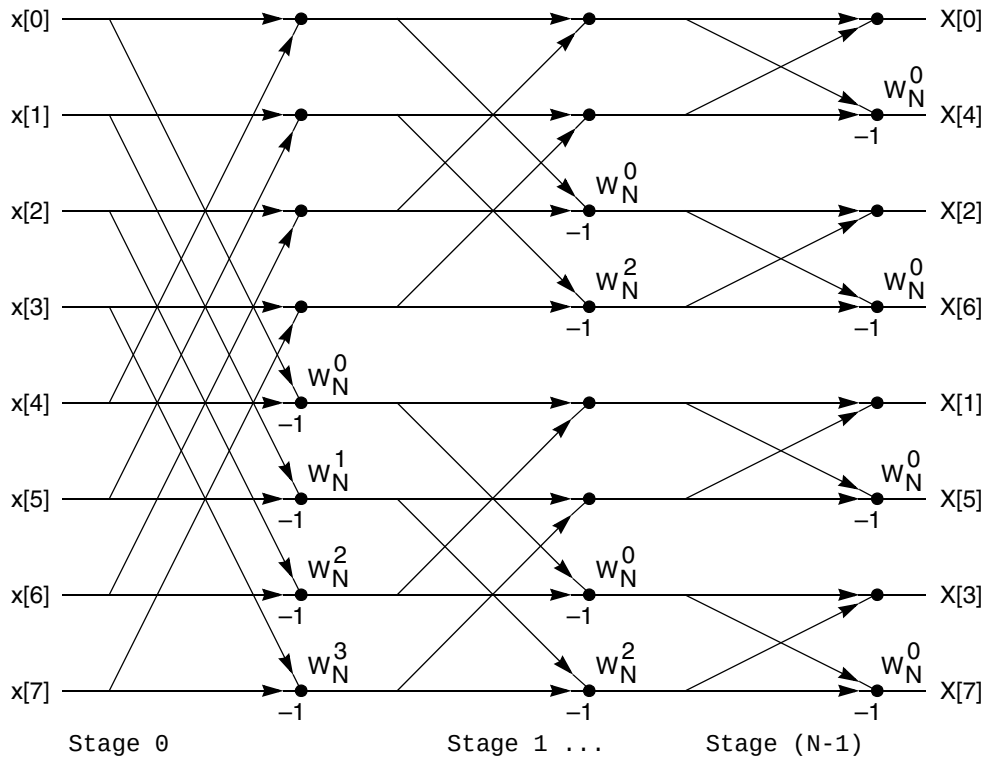

Figure 2. Radix-2, DIF, FFT Computational Lattice Structure for N = 8

Each butterfly involves one complex addition and one complex subtraction followed by a complex multiply (the value W involved in the complex multiply is commonly called a twiddle value). One advantage of the butterfly structure is that result values can safely overwrite the input values in memory. This allows the Radix-2 FFT to be complete as an in-place computation. A single iteration of in-place computation forms a stage. As Figure 2 shows, within a stage, there are N/2 butterflies. There are $\log_2(N)$ stages; therefore, the Radix-2 FFT is an $O(N*\log_2(N))$ operation.

The illustrated implementation uses three nested “for” loops. The outer loop iterates over the stages. The middle loop iterates over blocks, which is where butterflies intersect. As Figure 2 shows, stage 0 has one block, Stage 1 has two blocks, Stage 2 has four blocks, and so on. Finally, the inner most loop iterates over individual butterflies in a block.

Since the element size is a power of two, there are $\log_2 N$ stages. The size information is described as P where:

$N = 2^P$ or more easily calculated as N = (1 << P).

2.1 Outer Loop

2.1.1 Scalar

The outer loop iterates P times, once for each stage. For AltiVec code, however, the last two stages are done separately because, unlike prior stages, the two last stages have smaller blocks and both butterfly points end up on the same vector. To be consistent with the vector code, the scalar code’s last two stages

are done separately. This ensures an accurate comparison. The first stage, however, is still processed in the three nested loops.

The code below illustrates the scalar outer loop.

Outer Scalar Loop

```
for ( stage=0; stage < (P-2); stage++ )
{
/* Calculations for each block go here */
}
```

2.1.2 Vector

Below is the code for the vector version of the outer loop.

Outer Vector Loop

```
for ( stage=0; stage < (P-2); stage++ )
{
/* Calculations for each block go here */
}
```

2.2 Inner Loop

2.2.1 Scalar

Within a stage, two values must be calculated: the block offset and the stride. The block offset in the stage block always points at the top of the current butterfly. The stride is the distance between the top of a butterfly and its bottom. Incrementing to the next block within a stage requires knowing the size of a block. This is a simple calculation: block size is always twice the value of stride. This relationship is used in the “for” loop to increment through the blocks within a stage. Because the block pointer should never point outside the dataset, block<N is the stopping condition for the loop. Stride is initialized to N/2. The code below includes the second inner loop.

Inner Scalar Loop

```
for( stage=0; stage < (P-2); stage++ )
{
        for ( block=0; block<N; block+= stride*2 )
        {
                /* work on the butterflies here */
        }
}
```

2.2.2 Vector

The vector code looks different than but works similarly to the scalar code. The variable “block” is used as a counter, not as an address offset. The pointer to the current blocks of butterflies is called nBlk (the incrementing of nBlk is discussed in Section 2.3.2, “Vector”). The terminating case for this loop is when block equals numBlks. Because the number of blocks doubles for each subsequent stage, numBlks doubles after completing a stage.

Note that the “stride” in the vector version is also different. In the scalar case, the stride is twice the block size; but in the vector case, because four data points can be operated on concurrently, stride is one quarter of the value in the scalar case (each data point consists of real and imaginary portions, yielding the eight signed shorts which fit into a vector). The code below shows the loop.

Inner Vector Loop

```
for ( stage=0; stage < (P-2); stage++ )
{
        nBlk = 0;
        for (block=0; block<numBlks; block++)
        {
                /* work on the butterflies here */
                /* nBlk points to the current block */
        }
        numBlks = numBlks * 2;
}
```

2.3 Innermost Loop

2.3.1 Scalar

The innermost loop calculates the butterflies. Two butterflies (vertically) are processed because symmetry in the unit circle allows twiddle values loaded for one butterfly to be used again, with some manipulation, for calculation in another butterfly. The number of butterflies in a stage is always equal to the stride. Because two butterflies are processed in a loop, the stopping condition is stride/2. Two pairs of variables are needed to point at the two butterflies and these pairs are a stride amount apart. One pair of variables, pa and pb**,** point at the top and bottom of one butterfly, while variables qa and qb point to the top and bottom of the second butterfly. Below is the scalar code.

Complete FFT Scalar Loops

```
for( stage=0; stage < (P-2); stage++ )
{
for( block=0; block<N; block += stride*2 )
{
        pa = block;
        pb = block + stride/2;
        qa = block + stride;
        qb = blockblock + stride + stride/2;
        for( j = 0; j < stride/2; j++)
        {
         /* work on two butterflies here */
        }
}
}
```

2.3.2 Vector

Though the vector code uses a different indexing scheme than does the scalar code, it’s overall behavior is the same. The location of the next block within a given stage is the address of the current block, nBlk, plus

twice the stride. Again, because the stride differs from the scalar version, the terminating condition for the innermost loop is different. Two vectors (or four butterflies) are processed within the inner loop. The vector code is illustrated below.

Complete FFT Vector Loops

```
for ( stage=0; stage < (P-2); stage++ )
{
  nBlk = 0;
  for( block=0; block<N/2; block += stride )
  {
      for( j = 0; j < stride; j++)
      {
       /* work on four butterflies here */
      }
      nBlk = nBlk + 2 * stride;
  //twice as many blocks in next stage
  numBlks = numBlks *2
  //stride is half in next stage
  stride = stride / 2
}
```

2.3.3 Scalar (detail)

The butterfly calculation is processed in the innermost loop. As mentioned above, pa, pb, qa, and qb are used to point at the tops and bottoms of the two butterflies. pfs is the complex element array of data. Initially, the values are inputs; but, because operated on in-place, they store the outputs as well. The twiddle values are pre-calculated and stored in pfw. As Figure 2 shows, the twiddle values for the first stage are accessed from consecutive places. For the second stage, the twiddle values are accessed as every other value from a table and so forth.

Data scaling instructions both start the algorithm and are scattered through the inner loop. The initial scaling is necessary because data may range from -32768 to 32767. Without scaling there exists the possibility of overflow or underflow when performing the first round of additions and subtractions. To guarantee that subsequent additions and subtractions remain within range, the input data is shifted right by 1 bit, equivalent to dividing the dataset by two. This is a necessary step but results in a loss of precision.

The scalar code shown below provides a more complicated case for scaling. Multiplying two 16-bit numbers can possibly yield a 32-bit number. Here, a temporary 32-bit variable, tmpMult, holds the result of the 16-bit multiplication. tmpMult is then shifted right 15 bits to get the upper portion of the result. Note that tmpMult is shifted by only 15 bits and not 16 because the upper two bits are sign bits. Shifting by 16 would yield a result half of the correct size. This is a straightforward scaling algorithm, always scaling even if the numbers do not exceed bounds.

Also note that re and im are parts of the structure representing the real and imaginary values.

Core Scalar Calculations

```
//Scale inputs
pfs[pa+j].re = pfs[pa+j].re >> 1;
pfs[pa+j].im = pfs[pa+j].im >> 1;
pfs[qa+j].re = pfs[qa+j].re >> 1;
pfs[qa+j].im = pfs[qa+j].im >> 1;
pfs[pb+j].re = pfs[pb+j].re >> 1;
pfs[pb+j].im = pfs[pb+j].im >> 1;
pfs[qb+j].re = pfs[qb+j].re >> 1;
pfs[qb+j].im = pfs[qb+j].im >> 1;

//add
ft1a.re = pfs[pa+j].re + pfs[qa+j].re;
ft1a.im = pfs[pa+j].im + pfs[qa+j].im;
ft1b.re = pfs[pb+j].re + pfs[qb+j].re;
ft1b.im = pfs[pb+j].im + pfs[qb+j].im;

//sub
ft2a.re = pfs[pa+j].re - pfs[qa+j].re;
ft2a.im = pfs[pa+j].im - pfs[qa+j].im;
ft2b.re = pfs[pb+j].re - pfs[qb+j].re;
ft2b.im = pfs[pb+j].im - pfs[qb+j].im;
pfs[pa+j] = ft1a;       //store adds
pfs[pb+j] = ft1b;

//cmul
tmpMult = ((int) ft2a.re * (int) pfw[iw].re);
tmpMult = tmpMult - ((int)ft2a.im * (int)pfw[iw].im);
tmpMult = tmpMult >> 15;
pfs[qa+j].re = (signed short)  tmpMult;

tmpMult = ((int) ft2a.re * (int) pfw[iw].im);
tmpMult = tmpMult + ((int)ft2a.im * (int)pfw[iw].re);
tmpMult = tmpMult >> 15;
pfs[qa+j].im = (signed short)  tmpMult;

//twiddled cmul
tmpMult = ((int) ft2b.re * (int) pfw[iw].im);
tmpMult = tmpMult + ((int)ft2b.im * (int)pfw[iw].re);
tmpMult = tmpMult >> 15;
pfs[qb+j].re = (signed short)  tmpMult;

tmpMult = ((int) -ft2b.re * (int) pfw[iw].re);
tmpMult = tmpMult + ((int)ft2b.im * (int)pfw[iw].im);
tmpMult = tmpMult >> 15;
pfs[qb+j].im = (signed short)  tmpMult;

iw += edirts;
```

2.3.4 Vector (detail)

As mentioned earlier, the innermost vector loop processes four butterflies at a time. By doing more calculations within the inner loop, expensive memory transactions are minimized. Because complex numbers are represented as interleavings of real and imaginary values, the vec_perm instruction is used to extract the twiddle values as well as the data. Note that the input values are scaled in much the same way, using vec_sra to shift the input values right by one bit. However, no scaling is done for the multiplication because it is handled by the instruction vec_madds. Vec_madds multiplies two signed short vectors, divides by 2^15, and then adds another vector to the result. We are interested only in the multiplication portion of the vec_madds instruction, so a vector constant of zeroes is chosen as the addend.

There are other important portions of the code specific to fixed-point mathematics. To reverse the signs of some values, they must be multiplied not by a real -1, but by the scaled fixed-point equivalent of-32767. Even those values whose sign remains the same in vec_madds instructions must be multiplied by 32767 (the fixed-point equivalent of 1). This can be observed in the values of the various constants, among them vFixKn1.

Below is the vector code for the innermost loop.

Core Vector Calculations

```
//load the vectors
ve = data[ nBlk + n ];                  //[5]
vo = data[ nBlk + stride + n ];         //[6]

vw = vec_ld( (nw*4)+ (n*16), w); //nw = num of twiddles values to skip.
//4 bytes/(twiddle value)

//scaling input
ve = vec_sra(ve, vScaleInput);
vo = vec_sra(vo, vScaleInput);

//do the regular math
vep = vec_adds( ve, vo );               //[7]
vop = vec_subs( ve, vo );               //[8]

vn = vec_madds(vw, vFixKn1, vk0s);
vMultiply = vec_madds(vop, vn, vk0s);
vSwap = (VSH) vec_perm(vMultiply, vMultiply, vkm1);
vReal = vec_adds(vMultiply, vSwap);

vs = (VSH) vec_perm(vw, vw, vkm1);
vMultiply = vec_madds(vop, vs, vk0s);
vSwap = (VSH) vec_perm(vMultiply, vMultiply, vkm1);
vImag = vec_adds(vMultiply, vSwap);

vopt= (VSH)vec_perm(  vReal, vImag, vkm2 );   //[17]

//store the vectors
data[ nBlk + n ] = vep;                 //[19]
data[ nBlk + stride + n ] = vopt;       //[20]
```

As mentioned earlier, two stages are removed from the iterations over the Signal Flow Graph. The last two stages are removed because the final twiddle values (1,0) and (0,-1) do not require a

complex multiply. Instead, a vector-permute and vector-negate instructions are used, though multiply instructions do show up in the vector version to negate values.

After all iterations are done, the FFT of the original signal is found in the same memory location as the original signal. The element at index zero of the result is the DC component of the signal. Lower index values correspond to lower frequency bins of the FFT after bit reversal.

3 Fast Fourier Transform: Example 1

The concepts discussed above are illustrated in the following example.

This table shows eight data points, with real and imaginary parts, that represent input for the FFT. However, for purposes of a fixed-point FFT, these values are scaled between the signed short range of 32767 to –32767. The scaled values that serve as input to the FFT are labeled as Fix Real and Fix Imag. In an application, whether the input data must be converted or is natively integer depends on the input source.

Table 1. Values for x[n]

n	Real	Imag	Fix Real	Fix Imag
0	0.00	.019	0	630
1	.707	–.092	23169	–3005
2	–1.00	.653	–32767	21401
3	.377	0.00	12364	0
4	0.00	0.00	0	0
5	–.032	–.882	–1057	–28904
6	.119	.207	3890	6789
7	.890	0.00	29169	0

where N = 8

First the twiddle values are calculated using Equation 1. For a DFT calculation eight twiddle values across the unit circle are needed. For the FFT discussed in this paper, symmetry is used to reduce these numbers to two values (see Figure 3).

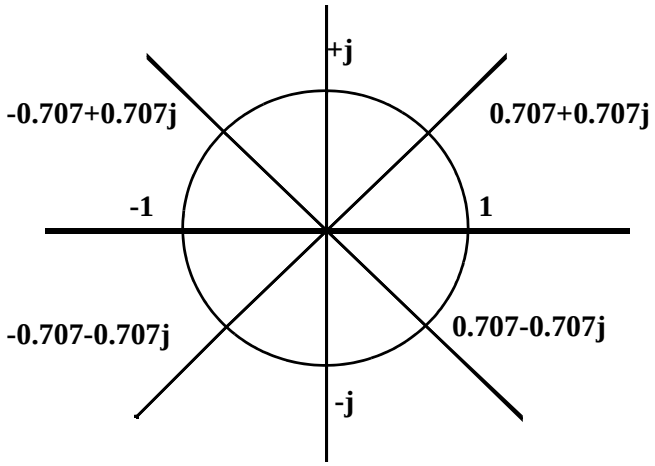

Figure 3. DFT Twiddle Values for N = 8

The two selected values {(1,0), (0.707, -0.707)} are W_0 and W_1in Equation 2. Each W_n starts from an angle position of 0 and increases in the negative direction. Therefore, W_2 and W_3 (the only ones needed for N = 8 FFT—see Figure 4) can be determined from the following:

- Real(W_0) = -Imaginary(W_2)
- Real(W_1) = Imaginary(W_3)
- Imaginary(W_1) = - Real(W_3)

The vector version assumes that initial twiddles have been used to generate a complete twiddle table, containing all of twiddle values for each stage. Also, like the input data, twiddle values for both scalar and vector versions must be scaled to fixed-point values. Figure 4 shows the new table of twiddle values. Using these values, the FFT is calculated and the result (after bit reversing the output addresses) is shown in Table 2.

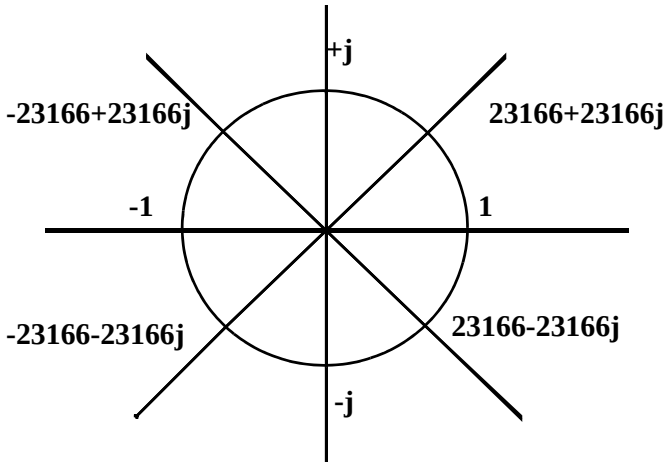

Figure 4. Scaled DFT Twiddle Values for N = 8

Table 2. Values for x[n] and X[k]

	INPUT		OUTPUT	
n, K	Real	Imag	Real	Imag
0	0	630	8688	–777
1	23169	–3005	15477	12581
2	–32767	21401	–761	–2035
3	12364	0	–6327	–14896
4	0	0	–23128	15179
5	–1057	–28904	–8176	6056
6	3890	6789	15195	–11745
7	29169	0	–978	–3119

4 Fast Fourier Transform: Example 2

The purpose of this example is to provide conceptual and visual insight into the FFT algorithm. Because input and output data for larger data sets are better described in plots, the actual numerical results are of lesser importance here.

The input data in this example consists of floating-point values that are more familiar for the sinusoidal input signal. The fixed-point version behaves similarly, though the amplitudes of the spikes in its output may vary. Note, however, the position (the sample number) of the spikes in the FFT output are unchanged. For this example, the signal considered is:

$$x(t) = 5\sin(2\Pi x 2t) + \sin(2\Pi x 20t)$$

Equation 4. Signal with Two Frequency Components

For the above to be changed into a discrete signal, it needs to be sampled at a certain frequency. If the signal is sampled at 256 samples per second, the discrete signal obtained is as shown in Equation 5. Note that the discrete time variable, n, is equivalent to the time domain variable, t, multiplied by the sampling frequency. Figure 5 shows the plot of the signal.

$$x[n] = 5\sin\left(\frac{2\Pi x 2n}{256}\right) + \sin\left(\frac{2\Pi x 20n}{256}\right)$$

Equation 5. Discrete Signal with Two Frequency Components

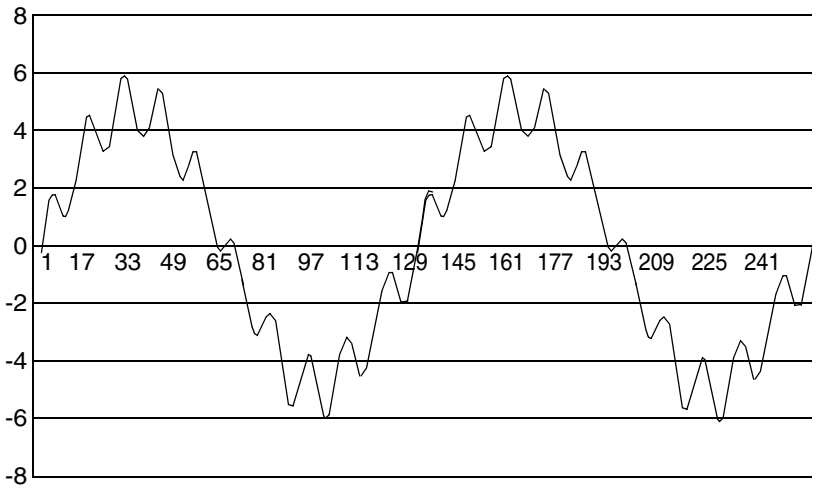

Figure 5. Plot of Equation 5

The sinusoid of greater magnitude in the figure corresponds to the first sine term in Equation 5. The nested sinusoids correspond to the second sine term in the Equation 5. When this signal is transformed through the FFT function, the two frequency components are isolated. The output plot is shown in Figure 6.

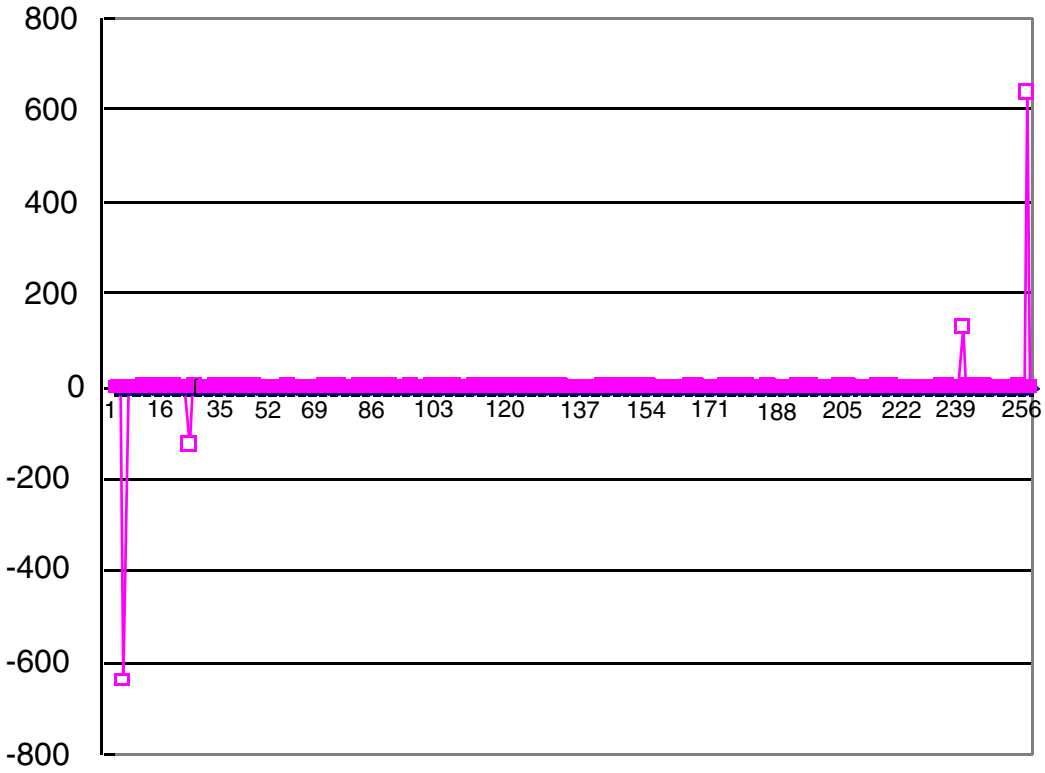

Figure 6. FFT Output for the Signal in Figure 5

The even part of the input signal corresponds to the imaginary part of the output. The odd part of the input signal correspond to real output. Because the input was entirely composed of sine waves (which are pure odd signals), the output is entirely composed of imaginary components. According to the Fourier transform properties (Oppenheim and Schafer p52), the output of a real signal has an odd-symmetric imaginary component. Figure 6 confirms this property. The spikes at positions 2 and 254 correspond to the first term of the input signal. The spikes at positions 20 and 246 correspond to the second term in the input signal.

Equation 6 shows the mathematically derived solution. This equation represents the first two spikes in Figure 6 and their reflection on the y-axis. In the equation, ∂ is a signal input to extract the system's impulse response. To read more about FFT functions and other signal processing techniques and terms, see Section 7, "References."

$$X_K = -\frac{5}{2}j[256\delta[K-2] - 256\delta[K+2]] - \frac{1}{2}j[256\delta[K-20] - 256\delta[K+20]]$$

Equation 6. Mathematical Representation of the FFT output

To relate the discrete-time frequency and the continuous-time frequency, the following relation is used.

$$W(\text{discrete}) = W(\text{continuous}) \times \text{SamplingPeriod}$$

Equation 7. Discrete and Continuous Frequency

For the discrete time-frequency components, (k) is used, as in 2πk/N, where N is the number of points displayed (see *DSP First* p340 - 343). Therefore, the first two spikes in the discrete frequency domain correspond to 2π(2)/256 and 2π(20)/256. Because the sampling frequency is 256 samples per second, the continuous-time frequency is obtained by multiplying 2π(2)/256 and 2π(20)/256 by 256. This gives continuous-time radial frequencies of 2π(2) and 2π(20), or 2Hz and 20Hz, which Figure 5 confirms.

5 Performance

Performance results are given in clock cycles for a Freescale T4240 microprocessor. To obtain the execution time, divide clock cycles by Megahertz. Performance for other AltiVec implementations may vary. Also, performance can vary depending on the C compiler used and can improve as the quality of a compiler improves from release to release. It is assumed that all required instructions and data are present in the L1 cache.

Both versions of the Radix-2 DIF FFT function are executed to find the number of clock cycles necessary for completion. These values are recorded in Table 3. From the clock cycles, the performance gain is calculated as the scalar clock cycles divided by vector clock cycles.

Table 3. Radix-2 DIF FFT Performance (Vector versus Scalar)

N	Scalar Clock Cycles	Vector Clock Cycles	Scalar/Vector
64	5,204	1,700	3.1
128	11,708	3,182	3.7
256	27,178	6,930	3.9
512	62,117	15,307	4.1
1024	139,946	33,336	4.2
2048	323,747	71,949	4.5
4096	740,735	157,357	4.7
8192	1,684,617	356,696	4.7
16384	3,866,877	880,851	4.4
32768	8,929,538	1,884,394	4.7
65536	20,343,323	4,026,735	5.1
131072	44,285,306	10,292,499	4.3
262144	96,563,909	22,167,880	4.4

The gain values in Table 3 show that the AltiVec version of the Radix-2 DIF FFT consistently outperforms the scalar version. A performance gain of over 4.0 is seen for larger FFTs where

inner-loop computation is a larger part of the execution time than overhead. Performance degrades at N = 8192 because 8192 complex shorts and associated twiddles consume the 32 KB L1 data cache of the T4240. Larger dataset sizes are influenced by hierarchical memory performance, which reduces incremental performance for both scalar and vector routines - though vector remains faster.

To read more about FFT functions and other signal processing techniques and terms, see Section 7, “References.”

6 Appendix

6.1 Scalar FFT Function Source Code

```
int sc_fft_dif( cplx *pfs, cplx *pfw, unsigned int n, unsigned int log_n )
{
       unsigned int stage,blk,j,iw=0;
       unsigned int pa,pb,qa,qb;
       unsigned int stride,edirts;
       cplx ft1a,ft1b,ft2a,ft2b,ft3a,ft3b;
       int tmpMult;

       //INIT
       stride = n/2;
       edirts = 1;

       //DIF FFT
       for( stage=0; stage<log_n-2; stage++ ) {
         for( blk=0; blk<n; blk+=stride*2 )  {
              pa = blk;
              pb = blk + stride/2;
              qa = blk + stride;
              qb = blk + stride/2 + stride;
              iw = 0;

              for( j=0; j<stride/2; j++ ) {//2bufflies/loop

              //Scale inputs
              pfs[pa+j].re = pfs[pa+j].re >> 1;
              pfs[pa+j].im = pfs[pa+j].im >> 1;
              pfs[qa+j].re = pfs[qa+j].re >> 1;
              pfs[qa+j].im = pfs[qa+j].im >> 1;
              pfs[pb+j].re = pfs[pb+j].re >> 1;
              pfs[pb+j].im = pfs[pb+j].im >> 1;
              pfs[qb+j].re = pfs[qb+j].re >> 1;
              pfs[qb+j].im = pfs[qb+j].im >> 1;

              //add
              ft1a.re = pfs[pa+j].re + pfs[qa+j].re;
              ft1a.im = pfs[pa+j].im + pfs[qa+j].im;
              ft1b.re = pfs[pb+j].re + pfs[qb+j].re;
              ft1b.im = pfs[pb+j].im + pfs[qb+j].im;

              //sub
              ft2a.re = pfs[pa+j].re - pfs[qa+j].re;
              ft2a.im = pfs[pa+j].im - pfs[qa+j].im;
              ft2b.re = pfs[pb+j].re - pfs[qb+j].re;
              ft2b.im = pfs[pb+j].im - pfs[qb+j].im;
```

```
                pfs[pa+j] = ft1a;//store adds
                pfs[pb+j] = ft1b;

                //cmul
                tmpMult = ((int) ft2a.re * (int) pfw[iw].re);
                tmpMult = tmpMult - ((int)ft2a.im * (int)pfw[iw].im);
                tmpMult = tmpMult >> 15;
                pfs[qa+j].re = (signed short)  tmpMult;

                tmpMult = ((int) ft2a.re * (int) pfw[iw].im);
                tmpMult = tmpMult + ((int)ft2a.im * (int)pfw[iw].re);
                tmpMult = tmpMult >> 15;
                pfs[qa+j].im = (signed short)  tmpMult;

                //twiddled cmul
                tmpMult = ((int) ft2b.re * (int) pfw[iw].im);
                tmpMult = tmpMult + ((int)ft2b.im * (int)pfw[iw].re);
                tmpMult = tmpMult >> 15;
                pfs[qb+j].re = (signed short)  tmpMult

                tmpMult = ((int) -ft2b.re * (int) pfw[iw].re);
                tmpMult = tmpMult + ((int)ft2b.im * (int)pfw[iw].im);
                tmpMult = tmpMult >> 15;
                pfs[qb+j].im = (signed short)  tmpMult;

                iw += edirts;
                }
          }
        }
        stride = stride>>1;
        edirts = edirts<<1;
}

//last two stages
for( j=0; j<n; j+=4 ) {
        //Scaling
        pfs[j].re = pfs[j].re >> 1;
        pfs[j].im = pfs[j].im >> 1;
        pfs[j+1].re = pfs[j+1].re >> 1;
        pfs[j+1].im = pfs[j+1].im >> 1;
        pfs[j+2].re = pfs[j+2].re >> 1;
        pfs[j+2].im = pfs[j+2].im >> 1;
        pfs[j+3].re = pfs[j+3].re >> 1;
        pfs[j+3].im = pfs[j+3].im >> 1;

        //upper two

        ft1a.re = pfs[j  ].re + pfs[j+2].re;
        ft1a.im = pfs[j  ].im + pfs[j+2].im;
        ft1b.re = pfs[j+1].re + pfs[j+3].re;
        ft1b.im = pfs[j+1].im + pfs[j+3].im;
        ft2a.re = ft1a.re + ft1b.re;
        ft2a.im = ft1a.im + ft1b.im;
        ft2b.re = ft1a.re - ft1b.re;
        ft2b.im = ft1a.im - ft1b.im;

        //lower two
        //notwiddle
        ft3a.re = pfs[j].re - pfs[j+2].re;
        ft3a.im = pfs[j].im - pfs[j+2].im;
        //twiddle
```

```
        ft3b.re =  pfs[j+1].im - pfs[j+3].im;
        ft3b.im = -pfs[j+1].re + pfs[j+3].re;

        //store
        pfs[j  ]    = ft2a;
        pfs[j+1]    = ft2b;
        pfs[j+2].re = ft3a.re + ft3b.re;
        pfs[j+2].im = ft3a.im + ft3b.im;
        pfs[j+3].re = ft3a.re - ft3b.re;
        pfs[j+3].im = ft3a.im - ft3b.im;

        }
        return 0;
}
```

6.2 Vector FFT Function Source Code

```
int av_dif(int numStages, VSH *data, VSH *w)
{

  //VARIABLES
  int     numBlks;        //num blocks in current stage
  int     stride;         //dist between even and odd pts in butterfly
  int     stage;          //the current stage
  int     block;          //the current block
  int     nBlk;           //offset of current block
  int     n;              //offset of current butterfly from nBlk
  int     nw;             //twiddle block index
  int     lastStages;   //Number of times to iterate through last 2 stages code
  VSH     ve;             //vector of even (upper) butterfly data
  VSH     vo;             //vector of  odd (lower) butterfly data
  VSH     vw;             //vector of twiddles
  VSH     vep;            //result of even add
  VSH     vop;            //result of odd subtract
  VSH     vopt;           //vop with twiddles applied
  VSH     vn, vs;         //intermediate results
  VSH     vm1,vm2,vpt;    //intermediate results
  VSH     v,vp,vpp;       //src vec, intermediate and final results
  VSH     vMultiply;      //Intermediate for vec_madds results
  VSH     vSwap;          //Swapped values for adding
  VSH     vReal, vImag;   //Real and imag. int. results

  //CONSTANTS
  const VSH vk0s  = (VSH)( 0, 0, 0, 0, 0, 0, 0, 0 );
  const VUC vkm1  = (VUC)( 2, 3, 0, 1, 6, 7, 4, 5,        //[15]
                           10, 11, 8, 9, 14, 15, 12, 13 );
  const VUC vkm2  = (VUC)( 0, 1, 16, 17, 4, 5, 20,        //[18]
                           21, 8, 9, 24, 25, 12, 13, 28, 29 );
  const VUC vkm3  = (VUC)( 8, 9, 10, 11, 12, 13, 14, 15,  //[24]
                           0, 1, 2, 3, 4, 5, 6, 7 );
  const VUC vkm4  = (VUC)( 0, 1, 2, 3, 4, 5, 6, 7,        //[30]
                           8, 9, 10, 11, 14, 15, 12, 13 );
  const VUC vkm5  = (VUC)( 4, 5, 6, 7, 0, 1, 2, 3,        //[34]
                           12, 13, 14, 15, 8, 9, 10, 11 );
  const VSH vFixKn2 = (VSH)(32767, 32767, 32767, 32767, -32767, -32767, -32767, -32767);
  const VSH vFixNegate7 = (VSH) (32767, 32767, 32767, 32767, 32767, 32767, 32767, -32767);
  const VSH vFixKn3 = (VSH) (32767, 32767, -32767, -32767, 32767, 32767, -32767, -32767);
```

```
const VSH vFixKn1 = (VSH) (32767, -32767, 32767, -32767, 32767, -32767, 32767, -32767);
const VUH  vScaleInput = (VUH) (1, 1, 1, 1, 1, 1, 1, 1);

//INIT
numBlks = 1;                            //stage 0 has 1 block
nw      = 0;                            //twiddle block index
stride  = (int)(1 << numStages) / SPV;  //dist b/n even & odd vecs

//FFT : iterate over all (but last 2) stages
for( stage=0; stage<(numStages-2); stage++ ) {
  //start at the top of the lattice
  nBlk = 0;

  //iterate over all blocks in this stage
  for( block=0; block<numBlks; block++ ) {

    //iterate over all butterflies in this block
    for( n=0; n<stride; n++) {

      //load the vectors
      ve = data[ nBlk + n ];                //[5]
      vo = data[ nBlk + stride + n ];       //[6]

      vw = vec_ld( (nw*4)+ (n*16), w); //nw = num of twiddles values to skip.
      //4 bytes/(twiddle value)

      //scaling input
      ve = vec_sra(ve, vScaleInput);
      vo = vec_sra(vo, vScaleInput);

      //do the regular math
      vep = vec_adds( ve, vo );             //[7]
      vop = vec_subs( ve, vo );             //[8]

      vn = vec_madds(vw, vFixKn1, vk0s);
      vMultiply = vec_madds(vop, vn, vk0s);
      vSwap = (VSH) vec_perm(vMultiply, vMultiply, vkm1);
      vReal = vec_adds(vMultiply, vSwap);

      vs = (VSH) vec_perm(vw, vw, vkm1);
      vMultiply = vec_madds(vop, vs, vk0s);
      vSwap = (VSH) vec_perm(vMultiply, vMultiply, vkm1);
      vImag = vec_adds(vMultiply, vSwap);

      vopt= (VSH)vec_perm(  vReal, vImag, vkm2 );   //[17]


      //store the vectors
      data[ nBlk + n ] = vep;               //[19]
      data[ nBlk + stride + n ] = vopt;     //[20]

    }
    //offset to next block is 2 strides away
    nBlk = nBlk + 2 * stride;
  }
  //twice as many blocks in next stage
  numBlks = numBlks * 2;
  //next twiddle block is 4 strides away
  nw   = nw + 4 * stride;
  //stride is half in next stage
  stride  = stride  / 2;
}
```

```
  // SPECIAL CASE: last 2 stages combined into one. [38]
  //iterate over all vectors in dataset
  lastStages = ((1<<(numStages+1))/SPV);
  for( n=0; n<lastStages; n++) {
    v = data[n];
    v = vec_sra(v, vScaleInput);
    vm1 = (VSH)vec_perm( v, v, vkm3 );      //[25]
    vp = vec_madds(v, vFixKn2, vm1);           //[28]
    vpt = (VSH)vec_perm( vp, vp, vkm4 );    //[31]
    vpt = vec_madds (vpt, vFixNegate7, vk0s);
    vm2 = (VSH)vec_perm( vpt, vpt, vkm5 );  //[35]
    vpp = vec_madds( vpt, vFixKn3, vm2 );      //[37]
    data[n] = vpp;
  }

  return 0;
}
```

7 References

1. Eleanor Chu and Alan George. *Inside The FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. ISBN 0-8493-0270-b, CRC Press, 2000.
2. *The FFT Demystified*. http://www.eptools.com/tn/T0001/INDEX.HTM, Engineering Productivity Tools Ltd., 1999.
3. J. H. McClellan, R. W. Schafer and M. A. Yoder. *DSP First*. ISBN 0-13-243171-8, Prentice-Hall, 1998.
4. A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. p599, ISBN 0-130216292-X, Prentice-Hall, 1989.

8 Revision History

This table summarizes the revision history of this document.

Table 4. Revision History

Rev. Number	Date	Substantive Change(s)
4	04/2013	• Modified Figure 1, “Single Butterfly Representation of a Signal Flow Graph,” to show W multiplies (P-Q). • In Table 2, “Values for x[n] and X[k],” updated sign for n = 2. • Updated performance results in Table 3, “Radix-2 DIF FFT Performance (Vector versus Scalar),” from MPC7410 to T4240 • Minor typographical corrections and editorial changes
3	10/2006	Rebranding for Freescale; non-technical formatting.
2	2005	Modified table 3. Updated the source code for scalar FFT function and vector FFT function. Added revision history.
0–1	2004	Initial release

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN2114
Rev. 4
04/2013