Application Report
SPRACN4–August 2019

Using DSPLIB FFT Implementation for Real Input and Without Data Scaling

DSP Processors

ABSTRACT

DSPLIB for TI C64x+ and C66x DSP architecture provides optimized implementation for FFT with assumptions for format and scaling of the input data. The application report provides guidance for users of DSPLIB for the following use cases:

- Efficient compute of single precision FFT using real input without reformatting data to complex format
- Fixed point FFT compute with no data scaling to avoid overflow

The content discussed benefits all DSPLIB users on C64x+, C674x and C66x DSP devices.

Contents

List of Figures

List of Tables

Trademarks

All trademarks are the property of their respective owners.

1 Real Input Introduction

Algorithms to perform forward and Inverse Fast Fourier Transforms (FFT and IFFT) typically assume complex input and output data. However, many applications use only real-valued data in the time domain. A simple but inefficient solution to this problem is to pad N-length real input signals to N-length complex input signals with zero-valued imaginary components.

x_{real} = { 1, 2, 3, ... }
x_{cplx} = { 1, 0, 2, 0, 3, 0, ...}

The complex FFT can then be applied to the double-length sequence. However, this method is obviously inefficient. This topic explains how to use the complex-valued FFT and IFFT algorithms to efficiently process real-valued sequences without padding those sequences. There are two key advantages to this approach:

- Lower memory footprint - M bytes per sequence instead of 2M bytes
- Fewer cycles - length N/2 FFT and IFFT computation instead of length N

For more information on the derivation of this method, see *Implementing Fast Fourier Transform Algorithms of Real-Valued Sequences With the TMS320 DSP Platform*.

Example C code for this method is also available for download.

1.1 Prerequisites

- Install the C674x DSPLIB or C66x DSPLIB based on the DSP being used.
  - Familiarize yourself with the FFT and IFFT kernels
  - Look at demo code (*_d.c) for usage examples

1.2 Computing a Length N/2 Complex FFT From a Length N Real Input Sequence

Let g(n) be an N-point real sequence (N must be even). Compute the N-point complex FFT, but only use an N/2-point FFT computation. This can be accomplished by using the following steps:

1. Form the the N/2-point complex valued sequence, x(n) = x_1(n) + jx_2(n), where x_1(n) = g(2n) and x_2(n) = g(2n + 1).
2. Compute the N/2-point complex FFT on the complex valued sequence x(n) to obtain X(k) = FFT{x(n)}. Note that the FFT should be performed with bit reversal.
3. Perform an additional computation to get G(k) from X(k).

$G_r(k) = X_r(k)A_r(k) - X_i(k)A_i(k) + X_r(N/2-k)B_r(k) + X_i(N/2-k)B_i(k)$, for k = 0, 1, ..., N/2–1 and X(N/2) = X(0)

$G_i(k) = X_i(k)A_r(k) + X_r(k)A_i(k) + X_r r(N/2-k)B_i(k) - X_i(N/2-k)B_r(k)$

Note that only N/2 points of the N-point sequence of G(k) are computed in the above equations. Because the FFT of a real-sequence has symmetric properties, you can easily compute the remaining N/2 points of G(k) with the following equations:

$G_r(N/2) = X_r(0) - X_i(0)$
$G_i(N/2) = 0$
$G_r(N-k) = G_r(k)$, for k = 1, 2, ..., N/2–1
$G_i(N-k) = -G_i(k)$

As you can see, the above equations require A(k) and B(k), which are sine and cosine coefficients. These values can be precomputed with the following C code, where even indices contains the real part and odd indices contain the imaginary part.

```
for (i = 0; i < N/2; i++)
{
  A[2 * i]     = 0.5 * (1.0 - sin (2 * PI / (double) (2 * N) * (double) i));
  A[2 * i + 1] = 0.5 * (-1.0 * cos (2 * PI / (double) (2 * N) * (double) i));
  B[2 * i]     = 0.5 * (1.0 + sin (2 * PI / (double) (2 * N) * (double) i));
  B[2 * i + 1] = 0.5 * (1.0 * cos (2 * PI / (double) (2 * N) * (double) i));
}
```

1.3 Returning to a Length N Real Sequence Using a Length N/2 Complex IFFT

Let G(k) be a N-point complex valued sequence derived from a real-valued sequence g(n). You want to get back g(n) = IFFT{G(k)}. However, you want to do this with an N/2-point IFFT. This can be accomplished using the following procedure:

$X_r(k) = G_r(k)IA_r(k) - G_i(k)IA_i(k) + G_r(N/2-k)IB_r(k) + G_i(N/2-k)IB_i(k)$, for k = 0, 1, ..., N/2–1 and G(N/2) = G(0)

$X_i(k) = G_i(k)IA_r(k) + G_r(k)IA_i(k) + G_r(N/2-k)IB_i(k) - G_i(N/2-k)IB_r(k)$

1. Find X(k) using the above equations.
2. Compute the N/2-point inverse FFT of X(k) to get x(n) = x1(n) + jx2(n). Note that the IFFT should be performed with bit reversal.
3. Get g(n) from x(n) using the following equations:

$g(2n) = x_1(n)$, for n = 0, 1, ..., N/2–1

$g(2n+1) = x_2(n)$

The above equations look similar to those used in our forward FFT computation. However, the pre-computed coefficients are slightly different. The following C code can be used to initialize IA(k) and IB(k), again, even indices contains the real part and odd indices contain the imaginary part.

```
for (i = 0; i < N/2; i++)
{
  IA[2 * i]     = 0.5 * (1.0 - sin (2 * PI / (double) (2 * N) * (double) i));
  IA[2 * i + 1] = 0.5 * (1.0 * cos (2 * PI / (double) (2 * N) * (double) i));
  IB[2 * i]     = 0.5 * (1.0 + sin (2 * PI / (double) (2 * N) * (double) i));
  IB[2 * i + 1] = 0.5 * (-1.0 * cos (2 * PI / (double) (2 * N) * (double) i));
}
```

Note that IA(k) is the complex conjugate of A(k) and IB(k) is the complex conjugate of B(k).

1.4 Benchmark of the Efficient Compute of FFT

Table 1 compares the performance of two methods of calculating the FFT of an N-point real sequence. Complex FFT refers to using an N-point complex FFT in the standard way, while Real FFT refers to using an N/2-point complex FFT as described in this topic. As expected, the Real FFT method yields superior performance.

Table 1. Performance Comparison

N	Cycle Count		
	Complex FFT	Real FFT	Improvement
128	1134	827	27.07%
256	2094	1709	18.38%
512	4944	3181	35.65%
1024	9680	7055	27.11%
2048	22770	13839	39.22%
4096	45298	31025	31.50%
8192	104724	61745	41.04%

NOTE:

- The data was measured on C6748 DSP with the data in L2 SRAM.
- L1D and L1P cache was enabled.
- Compiler CGT 6.x and CGT 7.4.x was used due to need to support legacy COFF binary format.

2 Fixed Point FFT With No Data Scaling

The FFT (DSP_fft16x16) and iFFT (DSP_ifft16x16) implementation provided with the C64x+ DSPLIB [1] apply scaling of data to avoid overflow. For more information, see the *TMS320C64x+ DSP Little-Endian Library Programmer's Reference*.

All stages are radix-4 except the last one, which can be radix-2 or radix-4, depending on the size of the FFT. All stages except the last one scale by two the stage output data.

It is desirable in certain use cases that the data scaling is not applied in the FFT routines. This article suggests modifications to the provided FFT source in the DSPLIB such that the data is not scaled. Also, an example is provided to demonstrate the affect of suggested change.

2.1 Suggested Change

The change to both the routines (DSP_fft16x16 and DSP_ifft16x16) is similar. The below description suggests modifications to the serial assembly (SA) implementation of the kernels. The kernels are located at:

- [DSPLIB_INSTALLATION_DIR]\packages\ti\dsplib\src\DSP_fft16x16\c64P\DSP_fft16x16_sa.sa
- [DSPLIB_INSTALLATION_DIR]\packages\ti\dsplib\src\DSP_fft16x16\c64P\DSP_fft16x16_sa.sa

Change 1: Identify the below code in the SA files:

```
;-------------------------------------------------------;
; Compute first set of outputs: ;
; ;
; x0[0]= xh0_0 + xh20_0 + 1 >> 1 ;
; x0[1]= xh1_0 + xh21_0 + 1 >> 1 ;
; x0[2]= xh0_1 + xh20_1 +1 >> 1 ;
; x0[3]= xh1_1 + xh21_1 +1 >> 1 ;
;-------------------------------------------------------;
AVG2.2 B_xh1_0_xh0_0, A_xh21_0_xh20_0, B_x_1o_x_0o
AVG2.2 B_xh1_1_xh0_1, A_xh21_1_xh20_1, B_x_3o_x_2°
```

Update the code to:

```
ADD2.2 B_xh1_0_xh0_0, A_xh21_0_xh20_0, B_x_1o_x_0o
ADD2.2 B_xh1_1_xh0_1, A_xh21_1_xh20_1, B_x_3o_x_2°
```

Note replacement of AVG2 instruction with ADD2.

Change 2: Identify the below code in the SA files:

```
;-------------------------------------------------------;
; The following code computes intermediate results for: ;
; ;
; si10' = -si10 twiddle table has -sin factors ;
;
; x2[h2 ] = (co10 * xt0_0 + si10'* yt0_0 + 0x8000) >> 16 ;
; x2[h2+1] = (co10 * yt0_0 - si10'* xt0_0 + 0x8000) >> 16 ;
; x2[h2+2] = (co11 * xt0_1 + si11'* yt0_1 + 0x8000) >> 16 ;
; x2[h2+3] = (co11 * yt0_1 - si11'* xt0_1 + 0x8000) >> 16 ;
;-------------------------------------------------------;
FFT Implementation With No Data Scaling 2
CMPYR .M1 A_co10_si10, B_yt1_0_xt1_0, A_xh2_1_0;
CMPYR .M1 A_co11_si11, B_yt1_1_xt1_1, A_xh2_3_2;
;-------------------------------------------------------;
;
; x2[l1 ] = (co20 * xt1_0 + si20'* yt1_0 + 0x8000) >> 16 ;
; x2[l1+1] = (co20 * yt1_0 - si20'* xt1_0 + 0x8000) >> 16 ;
; x2[l1+2] = (co21 * xt1_1 + si21'* yt1_1 + 0x8000) >> 16 ;
; x2[l1+3] = (co21 * yt1_1 - si21'* xt1_1 + 0x8000) >> 16 ;
;
; These four results are retained in registers and a ;
; double word is formed so that it can be stored with ;
; one STDW. ;
;-------------------------------------------------------;
; This equation ONLY has minus sign for x, y components
```

```
CMPYR .M1 A_co20_si20, A_myt0_0_mxt0_0, A_xl1_1_0;
CMPYR .M1 A_co21_si21, A_myt0_1_mxt0_1, A_xl1_3_2;
;-------------------------------------------------------;
; The following code computes intermediate results for: ;
;
; x2[l2 ] = (co30 * xt2_0 + si30'* yt2_0 + 0x8000) >> 16 ;
; x2[l2+1] = (co30 * yt2_0 - si30'* xt2_0 + 0x8000) >> 16 ;
; x2[l2+2] = (co31 * xt2_1 + si31'* yt2_1 + 0x8000) >> 16 ;
; x2[l2+3] = (co31 * yt2_1 - si31'* xt2_1 + 0x8000) >> 16 ;
;-------------------------------------------------------;
CMPYR .M2 B_co30_si30, B_yt2_0_xt2_0, B_xl2_1_0
CMPYR .M2 B_co31_si31, B_yt2_1_xt2_1, B_xl2_3_2
```

Update the code to:

```
;-------------------------------------------------------;
; The following code computes intermediate results for: ;
;
; si10' = -si10 twiddle table has -sin factors ;
; ;
; x2[h2 ] = (co10 * xt0_0 + si10'* yt0_0 + 0x8000) >> 16 ;
; x2[h2+1] = (co10 * yt0_0 - si10'* xt0_0 + 0x8000) >> 16 ;
; x2[h2+2] = (co11 * xt0_1 + si11'* yt0_1 + 0x8000) >> 16 ;
; x2[h2+3] = (co11 * yt0_1 - si11'* xt0_1 + 0x8000) >> 16 ;
;-------------------------------------------------------;
CMPYR1 .M1 A_co10_si10, B_yt1_0_xt1_0, A_xh2_1_0;
CMPYR1 .M1 A_co11_si11, B_yt1_1_xt1_1, A_xh2_3_2;
;-------------------------------------------------------;
;
; x2[l1 ] = (co20 * xt1_0 + si20'* yt1_0 + 0x8000) >> 16 ;
; x2[l1+1] = (co20 * yt1_0 - si20'* xt1_0 + 0x8000) >> 16 ;
; x2[l1+2] = (co21 * xt1_1 + si21'* yt1_1 + 0x8000) >> 16 ;
; x2[l1+3] = (co21 * yt1_1 - si21'* xt1_1 + 0x8000) >> 16 ;
; ;
FFT Implementation With No Data Scaling 3
; These four results are retained in registers and a ;
; double word is formed so that it can be stored with ;
; one STDW. ;
;-------------------------------------------------------;
; This equation ONLY has minus sign for x, y components
CMPYR1 .M1 A_co20_si20, A_myt0_0_mxt0_0, A_xl1_1_0;
CMPYR1 .M1 A_co21_si21, A_myt0_1_mxt0_1, A_xl1_3_2;
;-------------------------------------------------------;
; The following code computes intermediate results for: ;
;
; x2[l2 ] = (co30 * xt2_0 + si30'* yt2_0 + 0x8000) >> 16 ;
; x2[l2+1] = (co30 * yt2_0 - si30'* xt2_0 + 0x8000) >> 16 ;
; x2[l2+2] = (co31 * xt2_1 + si31'* yt2_1 + 0x8000) >> 16 ;
; x2[l2+3] = (co31 * yt2_1 - si31'* xt2_1 + 0x8000) >> 16 ;
;-------------------------------------------------------;
CMPYR1 .M2 B_co30_si30, B_yt2_0_xt2_0, B_xl2_1_0
CMPYR1 .M2 B_co31_si31, B_yt2_1_xt2_1, B_xl2_3_2
```

Note the CMPYR instruction has been replaced with CMPYR1 intrinsic The updates to the FFT routines can be incorporated in the application in two ways:

- The DSPLIB SW can be recompiled so that the generated library includes the updated kernels. To do this, recompile the library project [DSPLIB_INSTALLATION_DIR] \dsplib_v210\dsplib64plus.pjt. The updated library will be generated at [DSPLIB_INSTALLATION_DIR]\Release\dsplib64plus_rebuild.lib.
- The updated kernels can be directly included in the application project. Including the updated kernels will override the kernels that are included in the dsplib library.

2.2 Example Application

Find attached with this article an example that demonstrates the impact of the above suggested changes (fft_scaling_example.zip). The example should be unarchived at [DSPLIB_INSTALLATION_DIR]\dsplib_v210\example. Note the example assumes that the updated FFT and iFFT files are located at [DSPLIB_INSTALLATION_DIR]\dsplib_v210\src\DSP_fft16x16 and [DSPLIB_INSTALLATION_DIR] \dsplib_v210\src\DSP_ifft16x16. Following files are included:

- fft_example.pjt: Test project that demonstrates the impact of above suggested changes
- lnk.cmd: Linker command file
  - Generates input data that is summation of Sine wave data of various freqs. The complex input data is separated into real and imag components for easier visualization (xin_real_16x16 and xin_imag_16x16).
  - Generates twiddle factors for the FFT and iFFT kernels. Note that the two kernels use different twiddle factors. Two separate twiddle factor kernels are provided with the DSPLIB release. For the FFT kernel, use the function: [DSPLIB_INSTALLATION_FOLDER]\examples\fft_ex\DSP_fft16x16\gen_twiddle_fft16x16.c and for the iFFT kernel, use the functio: [DSPLIB_INSTALLATION_FOLDER]\examples\fft_ex\DSP_ifft16x16\gen_twiddle_ifft16x16.c.
  - Calls the FFT routine. The generated complex fft data is separated into real and imag components for easier visualization (y_real_16x16 and y_imag_16x16).
  - Calls the iFFT routine. The generated complex output is separated into real and imag components for easier visualization (xout_real_16x16 and xout_imag_16x16)
- Images: The package includes images that help visualize the impact of the changes. Each image displays side by side three data buffers; xin_real_16x16, y_real_16x16 and xout_real_16x16. Three images are provided:
  - FFTOutput_bothKernelsApplyScaling.JPG: Helps visualize the case where both FFT and iFFT routines apply scaling. This is the default operation.
  - FFTOutput_onlyIFFTApplyScaling.JPG: Helps visualize the case where only iFFT routines apply scaling. This is the case where the FFT routine has been updated to not apply any scaling.
  - FFTOutput_noScaling.JPG: Helps visualize the case where no scaling is applied. This is the case where both the FFT routine and the iFFT routine have been updated to not apply any scaling.

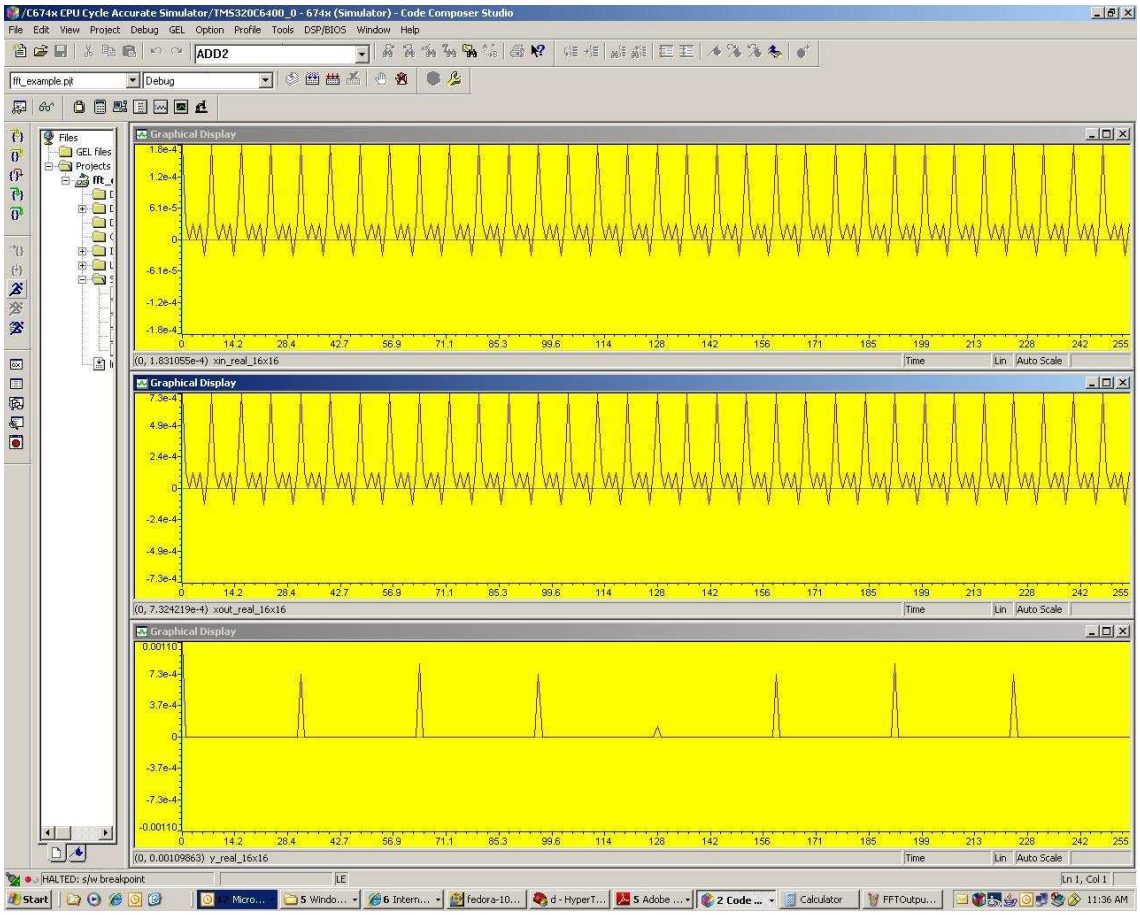

Figure 1. Both Kernels Apply Scaling

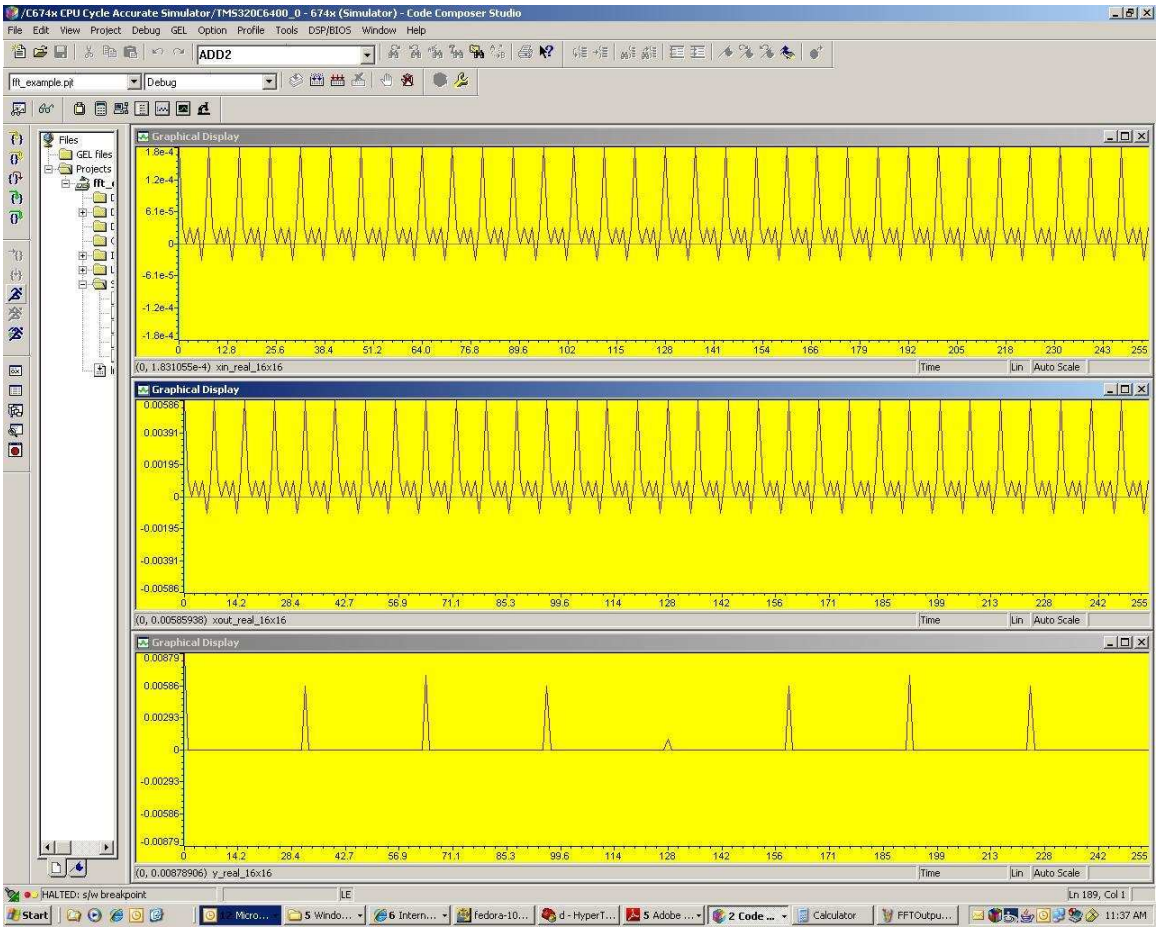

Figure 2. Only IFFT Apply Scaling

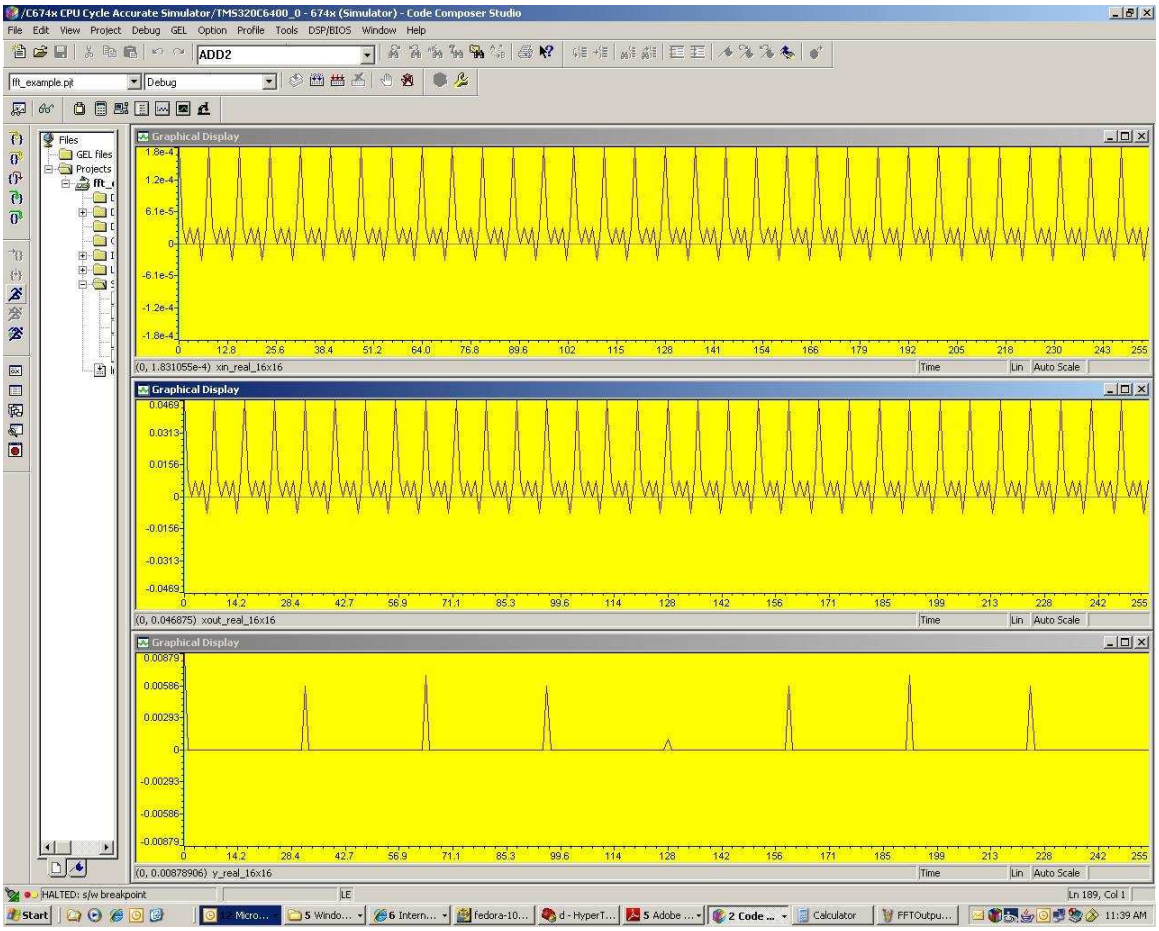

Figure 3. No Scaling

3 Summary

- N = 256. Since 256 = 4^4, the processing will happen as follows: 3x radix 4 iterations in the main loop (scaling by 2) and 1x radix 4 iteration at the end. So the total scaling will be 2^3. This is because every stage applies scaling of 2 but the last stage doesn't apply any scaling. Thus, scaling by 2*2*2 = 8 is expected. The images confirm the understanding. Note the X Output magnitude increased by 8 times when scaling is removed from FFT and then from iFFT.
- When both the kernels are updated to not apply scaling, the X output magnitude is 256 times the input. This is correct as 1/N scaling is not applied overall. Thus, X Output to be 256 times X Input is expected.

Thus, the suggested changes correctly remove the scaling from the two kernels.

4 References

- Texas Instruments: *Implementing Fast Fourier Transform Algorithms of Real-Valued Sequences With the TMS320 DSP Platform*
- Texas Instruments: C64x+ DSP Library
- Texas Instruments: *TMS320C64x+ DSP Little-Endian Library Programmer's Reference*

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265