Application Report

SPRABB7–June 2010

Algorithms for Optical Coherence Tomography on TMS320C64x+

Murtaza Ali and Renuka Parlapalli

ABSTRACT

Optical coherence tomography (OCT) imaging is a high-resolution, sub-surface non-invasive imaging technique that has become increasingly popular in hospitals and clinics for various applications for structural and quantitative imaging.

As an increasing number of OCT equipment manufacturers choose to use digital signal processors (DSP) to design their OCT systems, it becomes extremely important to implement highly efficient algorithms on the DSP. This application report discusses major requirements, API's and key challenges in implementing certain identified algorithms for OCT. It introduces TI's efficient software implementation of these algorithms on C64x+™ based DSP devices. Performance is given in cycles per scanline and total number of scanlines that can be processed per second.

Contents

List of Figures

List of Tables

1 Introduction

Digital signal processors (DSPs) and system-on-chips (SOC's) are specially designed single-chip digital microcomputers that process digitized electrical signals generated by electronic sensors. Current OCT systems are very large with most of the signal processing carried out on power hungry computers. To avoid re-tooling and to provide flexibility of the implementation of the signal processing on a system, it becomes important to use architectures that are easily programmable at low-power consumption. Programmable multi-core DSP's can serve as the main processing engine in such systems with architectures that provide scalability, power efficiency, real-time capability and high performance. Also, analogous to the recent trend of portable systems in ultrasound imaging, systems with smaller form factor have gained increased momentum in OCT. This allows for systems to be used directly at the point of care.

OCT primarily uses the principle of low coherence interferometry to obtain depth information from light that is backscattered from a sample. By stacking scans in X and/or Y directions, two or three dimensional imaging is feasible. Some major applications of OCT include:

- Ophthalmology – For very fine imaging of the retina to diagnose several eye diseases.
- Dentistry – For high resolution structural imaging of the teeth.
- Cardiology/Intravascular Imaging – For structural images using catheters and coronary intervention procedures.
- Endoscopy – For in-vivo tissue morphology of the gastrointestinal tract.
- Intra-Operative Surgery – For tumor margining that helps discriminating between malignant and normal tissues allowing cancer diagnosis through either non-invasive or minimally invasive procedures during surgery.

OCT technology uses a standard Michelson interferometer (shown in Figure 1) with a low coherent light source [1]. Using a beam splitter, it splits the light source into the reference path and the sample path, which are recombined after back-reflection from the reference mirror and the multiple layers of the sample to form an interference signal. The broadband nature of light causes interference of the optical fields only when the path lengths of the reference and the sample arm are matched to within the coherence length of the light. This interference signal carries information about the sample at a depth determined by the reference path length. A visual image is constructed based on the signal processing of the interference signal implemented on the DSP.

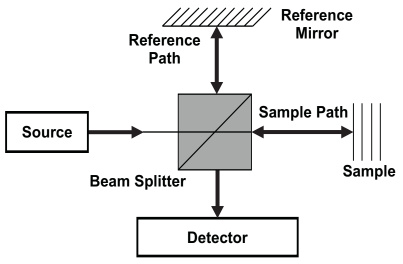

Figure 1. Schematic of Basic OCT System

Current optical coherence tomography systems can be broadly categorized as spectral domain and swept source systems. These systems are briefly described below.

- Spectral Domain OCT Systems (SD-OCT Systems)

  A broadband source of light is used as an input to the interferometer. The depth information is obtained by measuring the spectral density in the detection arm of the interferometer using a spectrometer, where the interference beam is dispersed by a diffraction grating and the individual wavelength components are detected by an array detector or a line scan camera.
- Swept Source OCT Systems (SS-OCT Systems)

  Swept source systems are an alternate way to obtain the spectrogram by using a frequency-swept laser or a tunable laser with just a single detector and without dispersion components, which is referred to as SS-OCT. Instead of sampling the received spectrum over a finite wavelength, the sample is probed with a narrow band but frequency varying source.

Once the interference signal is obtained, signal processing remains the same for both of the systems indicated above. Signal processing is critical as it allows you to make meaningful diagnosis of the data. To achieve this, a set of generic algorithms have been identified that comprise the signal processing chain of a basic OCT system including resampling, Fast Fourier Transform (FFT), magnitude computation and full range log compression. Programmable multi-core DSPs and SOCs, with C64x+ architectures are well-suited for implementing complex mathematical algorithms and can efficiently address all the processing needs of such a system.

2 OCT Algorithm Implementation on DSP

2.1 Basic Introduction on OCT Algorithms

Figure 2 shows the basic signal processing chain needed to create a structural image from the recorded interference signal.

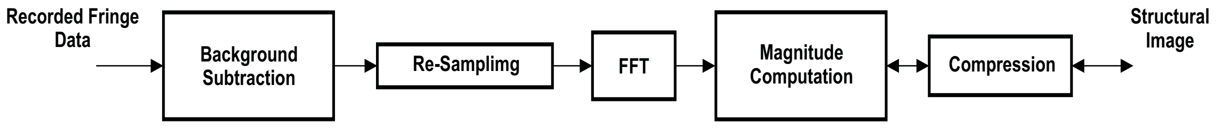

Figure 2. Baseline Signal Processing Chain in OCT Systems

Using TI's Embedded Processor Software Toolkit for Medical Imaging (MED-STK) (SPRABB8) (referred to as MED-STK throughout the remainder of this document) provides optimized functions to implement such a signal processing chain on DSPs based on the C64x+ architecture.

Different algorithms optimized for the C64x+ architecture and benchmarked for OCT systems are outlined in the following sections.

2.1.1 Background Subtraction

This is a simple operation where the background is subtracted from the acquired data. This subtraction takes care of the DC component in the signal that is due to the reflectance from the reference arm. Additional variations due to fixed pattern noise in the line scan camera (for spectral domain OCT) and variations in power spectral densities of the source can also be suppressed by this method.

2.1.2 Resampling

In spectral domain OCT, spectrometers are used that measure optical intensity as a function of wavelength. The signal obtained at the interferometer exit, though equidistant in λ space, is non-equidistant in frequency (or k) space. The spectrum should be evenly sampled in the frequency domain (or k-space) to implement the FFT algorithm. In SS -OCT, the frequency sweeping is usually non-linear in frequency (or k-space). Therefore, in either of the systems, the captured information is not linearly spaced in frequency and a re-sampling technique is usually employed to resample the recorded discrete intensities from the acquired domain to linear frequency domain. Note that some advanced techniques have been proposed to eliminate re-sampling, e.g., use of secondary interferometer in swept source systems to derive the non-linear clock. However, re-sampling continues to be a common way to provide the desired data at the input of the FFT. The cubic spline interpolation algorithm is used as defined in [2] to perform the re-sampling function. Figure 3 represents the details of the internal structure of the cubic spline algorithm.

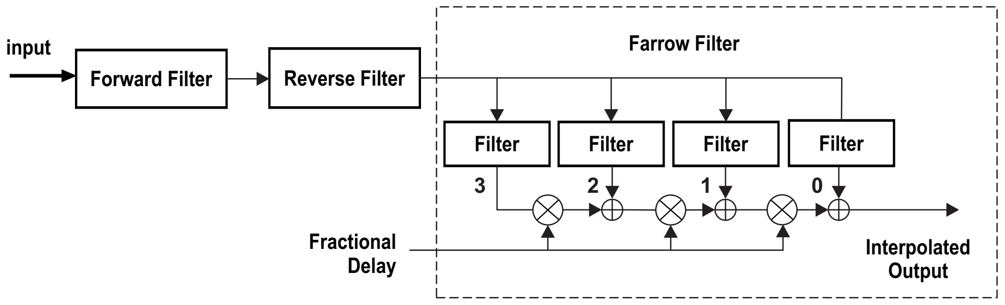

Figure 3. Structure of Cubic Spline Algorithm

A causal IIR filter followed by a non-causal IIR filter is used in the pre-processing step, as defined in [2]. Therefore, the whole A-scan needs to be available before this cubic spline algorithm can be used.

2.1.3 Fast Fourier Transform (FFT)

After resampling, the data is linear in the k-space and an FFT is performed to reconstruct the axial scan as a function of depth. The MED-STK has several variations of the FFT available from the TI DSPLib [3]. The routines use 32 bit internal precision math operations with 16-bit twiddle factors and only allows power of 2 FFT sizes. Therefore, for the sizes of FFT used in OCT with 16-bit input, there is no possibility of internal saturation. The 16-bit output is derived using your programmable right shift value.

2.1.4 Magnitude Computation

The FFT output is a complex number. The structural information is contained in the magnitude of the FFT output. The corresponding optimized function approximates a square root using a four term Taylor series approximation. The function provides 15.5 bits of accuracy.

2.1.5 Log Compression

A 16 bit value provides 96 dB dynamic range. However, human visualization range is about 40-60 dB. The 16-bit data is compressed using a non-linear function to reduce the dynamic range for visualization. The log function is a common non-linearity used in OCT. There are two approximations of log compression to map the 16-bit input to 8-bit data for display: one using linear approximation and the other using quadratic approximation. They have different cycle counts and different accuracies. The linear approximation has lesser cycle count but it is less accurate compared to the quadratic approximation. Function interfaces for both full range compression and thresholded compression are also provided. The latter can be used to display only a portion of the whole input dynamic range. In addition, a table-based dynamic range compression scheme is also available in the MED-STK where any non-linear function can be used by defining the appropriate table.

2.2 Key Requirements for DSP Based OCT Algorithms

2.2.1 API

A DSP-based software implementation of these OCT algorithms consists of well defined APIs. In general, for all the OCT algorithms through the API, you should be able to accurately specify all the physical parameters related to the image of interest such as number of scanlines, number of samplers per scanline, number of frames, etc.

Specific algorithms may have additional parameters that you are required to specify. For such a case, through the API, you should be able to accurately specify these parameters.

2.2.2 Flexibility

The DSP-based OCT algorithms should be flexible in terms of having the ability to operate in different modes. For example, one should be able to switch between different cases/modes of the OCT signal processing chain like B-mode imaging, Doppler, polarization sensitive, etc.

The same DSP used for the main signal chain can be utilized for calibration and different estimation algorithms needed to identify system parameters like background signal, the re-sampling points, the phase corrections for dispersion compensation, etc. These parameters are either pre-computed during calibration or computed automatically before the image acquisition process.

2.2.3 Efficiency

The implementation should be highly efficient so that minimum DSP CPU bandwidth is consumed for these algorithms, allowing more space for future OCT algorithms to be implemented.

2.3 Key Challenges of implementing OCT Algorithms on DSP

There are several challenges in implementing highly efficient algorithms for OCT as listed in the following sections.

2.3.1 I/O Bandwidth Requirement

The I/O bandwidth requirements for CPUs to access all the necessary data is algorithm dependent to a certain extent. For instance, in the resampling algorithms, eight lines of data have to be accessed simultaneously for the efficient implementation of the algorithm.

The highly efficient DMA in C64x+ architecture-based devices can be used to mitigate the impact of data movement between the external memory and the internal memory of the device. The DMA also allows multi-dimensional data movements and data transposition (as need by cubic spline API) can be done effectively without involving the CPU core.

The effect of cache on benchmarking depends on memory organization and system function partitioning. All of the benchmarking provided here is done on a TMS320C6455 device with the data located in its L2 memory. Therefore, the first order cache effects have already been taken into consideration in this document.

2.3.2 Precision Requirement

It is important to have the correct precision requirement to implement algorithms on a DSP. The system will suffer from poor images if the precisions used throughout the system are not sufficient. On the other hand, if more than necessary precision is used, that will unnecessarily increase the cycle count and reduce the number of scanlines that can be processed through the system. The APIs for 16-bit input and output have been optimized except for the compression module, which outputs 8-bit data for visualization. Since the analog to digital converter samples the interference data with 10-14 bit precision, 16 bit is sufficient for high-quality image production for this application.

3 TI DSP-Based OCT Algorithms

Efficient algorithms for OCT have been implemented on TI devices that are based on the C64x+ DSP at Texas Instruments and are offered as part of the MED-STK. The algorithms have been implemented to achieve high efficiency, high accuracy and high flexibility. The key features for the set of the APIs available in the MED-STK and applicable to OCT are provided. For more details on the API, see the toolkit documentation.

3.1 API Definition

3.1.1 Background Subtraction

This is the vectorized subtraction API that can be used.

```
/**
 *  @brief      Performs real-real element wise vector subtraction
 *
 *  @param[in]  pDataIn1    pointer to the first 16 bit input vector (real)
 *  @param[in]  pDataIn2    pointer to the second 16 bit input vector (real)
 *  @param[in]  len         length of input vectors
 *  @param[out] pDataOut    pointer to the 16 bit output vector
 *
*/
void util_vectorSubRxR_16b_16b_16b(const int_least16_t *pDataIn1,
                                   const int_least16_t *pDataIn2,
                                   const int_least16_t len,
                                   int_least16_t *pDataOut);
```

3.1.2 Resampling

This API is designed to take in the interpolating points (both the integer part and the fractional part) as parameters and then produce the interpolated output for a given input vector. Specifically, the API is as follows:

```
/**
 *  @brief      Performs cubic spline interpolation with 16 bit
 *              input/output for eight lines in parallel
 *
 *  @param[in,out]  pDataIn          Pointer to the input Data
 *  @param[in]  pCubicSplineParams   Pointer to parameter structure
 *  @param[out] pDataOut             Pointer to the output Data
 *
*/
void interp_cubicSpline_16b_16b_8p(int_least16_t *pDataIn,
                            const CubicSplineParams_t *pCubicSplineParams,
                            int_least16_t *pDataOut);
```

The parameter structure is defined below:

```
/**
 *  @brief Specifies the cubic spline interpolation parameter structure
 */
typedef struct _CubicSplineParams_ {
  /// number of samples in input vector;
  int_least16_t     numInputSamples;
  //// number of samples in output vector
  int_least16_t     numOutputSamples;
  /// pointer to the integer part of interpolating locations;
  int_least16_t     *pIntegralPos;
  /// pointer to the fractional part of interpolating locations
  int_least16_t     *pFractionalPos;
} CubicSplineParams_t;
```

To improve the efficiency in terms of MHz consumed, this API is designed to operate on a set of eight lines in parallel. The set of eight lines is chosen based on C64x+ core architecture (available instructions, parallelism and registers). This requires that the input and output data are specially arranged. Let the data sequence be represented by $x_{k,l}$, where $0 \leq l \leq 7$ is the line number and k is the index of the data sequence and the data is assumed to be placed in memory in line order as shown in the following sequencing.

$$x_{0,0}, x_{0,1}, x_{0,2}, x_{0,3}, x_{0,4}, x_{0,5}, x_{0,6}, x_{0,7}, x_{1,0}, x_{1,1}, x_{1,2}, x_{1,3}, x_{1,4}, x_{1,5}, x_{1,6}, x_{1,7}, x_{2,0}, x_{2,1}, \ldots$$

The output data is also placed in memory in the same sequence.

The input and output is 16 bit and all accumulation is done internally in 16 bit as well. Care should be taken on input scaling so that overflow or saturation does not occur internally. The overflow or saturation will not only depend on the input scaling but also on the smoothness of the input vector. Note that interpolation assumes inherent smoothness of the input vector.

For the cubic spline interpolation API, the first set of IIR stages provides gain of 1/6, with the final Farrow stage providing gain of 6 back which then provides an overall gain of unity across the signal chain.

3.1.3 FFT

The following set of FFT APIs are useful for implementation in OCT systems.

```
/**
 *  @brief      Complex to complex N-point FFT of N-point sequence
 *              with 16 bit input/output and 32 bit internal precision;
 *
 *  @param[in]  pDataIn                  Pointer to 16-bit complex input data
 *  @param[in]  pFft_16b_16b_32i_Params  Pointer to FFT parameters
 *  @param[out] pDataOut                 Pointer to 16-bit complex output data
 *
 */
void fft_forwardC2C1_16b_16b_32i(const cplx_least16_t * pDataIn,
    const Fft_16b_16b_32i_Params_t     *pFft_16b_16b_32i_Params,
    cplx_least16_t *pDataOut);

/**
 *  @brief      Complex to complex 2N-point FFT of N-point sequence
 *              with 16 bit input/output and 32 bit internal precision;
 *
 *  @param[in]  pDataIn                  Pointer to 16-bit complex input data
 *  @param[in]  pFft_16b_16b_32i_Params  Pointer to FFT parameters
 *  @param[out] pDataOut                 Pointer to 16-bit complex output data
 *
 */
void fft_forwardC2C2_16b_16b_32i(const cplx_least16_t * pDataIn,
    const Fft_16b_16b_32i_Params_t *pFft_16b_16b_32i_Params,
    cplx_least16_t *pDataOut);

/**
 *  @brief      Real to complex N-point FFT of N-point sequence
 *              with 16 bit input/output and 32 bit internal precision;
 *
 *  @param[in]  pDataIn                  Pointer to 16-bit real input data
 *  @param[in]  pFft_16b_16b_32i_Params  Pointer to FFT parameters
 *  @param[out] pDataOut                 Pointer to 16-bit complex output data
 *
*/
void fft_forwardR2C1_16b_16b_32i(const int_least16_t * pDataIn,
    const Fft_16b_16b_32i_Params_t *pFft_16b_16b_32i_Params,
    cplx_least16_t *pDataOut);

/**
 *  @brief      Real to complex 2N-point FFT of N-point sequence
 *              with 16 bit input/output and 32 bit internal precision;
 *
 *  @param[in]  pDataIn                  Pointer to 16-bit real input data
 *  @param[in]  pFft_16b_16b_32i_Params  Pointer to FFT parameters
 *  @param[out] pDataOut                 Pointer to 16-bit complex output data
 *
 */
void fft_forwardR2C2_16b_16b_32i(const int_least16_t * pDataIn,
    const Fft_16b_16b_32i_Params_t *pFft_16b_16b_32i_Params,
    cplx_least16_t *pDataOut);




/**
 *  @brief      Complex to complex N-point IFFT of N-point sequence
 *              with 16 bit input/output and 32 bit internal precision;
 *
 *  @param[in]  pDataIn                  Pointer to 16-bit complex input data
 *  @param[in]  pFft_16b_16b_32i_Params  Pointer to FFT parameters
 *  @param[out] pDataOut                 Pointer to 16-bit complex output data
 *
 */
void fft_inverseC2C1_16b_16b_32i(const cplx_least16_t * pDataIn,
    const Fft_16b_16b_32i_Params_t *pFft_16b_16b_32i_Params,
    cplx_least16_t *pDataOut);

/**
 *  @brief      Complex to complex 2N-point IFFT of N-point sequence
 *              with 16 bit input/output and 32 bit internal precision;
 *
 *  @param[in]  pDataIn                  Pointer to 16-bit complex input data
 *  @param[in]  pFft_16b_16b_32i_Params  Pointer to FFT parameters
 *  @param[out] pDataOut                 Pointer to 16-bit complex output data
 *
 */
void fft_inverseC2C2_16b_16b_32i(const cplx_least16_t * pDataIn,
    const Fft_16b_16b_32i_Params_t *pFft_16b_16b_32i_Params,
    cplx_least16_t *pDataOut);

/**
 *  @brief      Complex to real N-point IFFT of N-point sequence
 *              with 16 bit input/output and 32 bit internal precision;
 *
 *  @param[in]  pDataIn                  Pointer to 16-bit complex input data
 *  @param[in]  pFft_16b_16b_32i_Params  Pointer to FFT parameters
 *  @param[out] pDataOut                 Pointer to 16-bit real output data
  *
*/
void fft_inverseC2R1_16b_16b_32i(const cplx_least16_t * pDataIn,
    const Fft_16b_16b_32i_Params_t *pFft_16b_16b_32i_Params,
    int_least16_t *pDataOut);

/**
 *  @brief      Complex to real 2N-point IFFT of N-point sequence
 *              with 16 bit input/output and 32 bit internal precision;
 *
 *  @param[in]  pDataIn                  Pointer to 16-bit complex input data
 *  @param[in]  pFft_16b_16b_32i_Params  Pointer to FFT parameters
 *  @param[out] pDataOut                 Pointer to 16-bit real output data
 *
 */
void fft_inverseC2R2_16b_16b_32i(const cplx_least16_t * pDataIn,
    const Fft_16b_16b_32i_Params_t *pFft_16b_16b_32i_Params,
    int_least16_t *pDataOut);
```

All the above APIs use the following parameter structure:

```
/**
 *  @brief Specifies FFT/IFFT parameters (16 bit input/output and 32 bit
 *         internal precision)
 *
 */
typedef struct _Fft_16b_16b_32i_Params_{
  /// twiddle factors following DSPLIB FFT;
  cplx_least16_t    *pDspLibTwiddle;
/// additional twiddle factor for pre or post processing
/// required for various types;
  cplx_least16_t    *pTwiddle;
  /// 32 bit input buffer to DSPLIB FFT mapped to complex data structure;
  cplx_least32_t    *pBufferIn;
  /// 32 bit output buffer to DSPLIB FFT  mapped to complex data structure;
  cplx_least32_t    *pBufferOut;
  /// desired FFT length; must be multiple of 2;
  int_least16_t     fftLen;
  /// left shift used to move 16 bit input to 32 bit internal buffer;
  uint_least8_t     lShift;
  /// right shift used to move 32 bit internal buffer to 16 bit output;
  uint_least8_t     rShift;
} Fft_16b_16b_32i_Params_t;
```

3.1.4 Magnitude Computation

The complex magnitude API that produces the structural information is as follows:

```
/**
 *  \brief      Computes the fixed point magnitude of 16 bit complex data.
 *              It performs the  following mathematical operation
 *              mag(input)*2^mag_qFmt for input values.
 *  \param[in]  pInputValues  The pointer to the input values
 *  \param[in]  numValues     The number of input values
 *  \param[in]  mag_qFmt      The power of 2 used to scale the magnitude values
 *  \param[out] pMagValues    The pointer to magnitude values
 */
void util_cplxMag_16b_16b(const cplx_least16_t *pInputValues,
                          const uint_least32_t numValues,
                          const uint_least8_t mag_qFmt,
                          uint_least16_t *pMagValues);
```

3.1.5 Log Compression

The API for full range log compression using quadratic approximation is as follows:

```
/**
 *  @brief      Performs full range log compression with 16 bit input
 *              and 8 bit output (all unsigned) using quadratic approximation
 *              for log
 *
 *  @param[in]  pDataIn       pointer to the 16 bit real and positive
 *                            input data vector
 *  @param[in]  len           length of input vector
 *  @param[out] pDataOut      pointer to the 8 bit log compressed output
 *                            data vector
 *
 */
void util_logCompFullQuadratic_16b_8b(const uint_least16_t * pDataIn,
                                      const int_least32_t len,
                                      uint_least8_t *pDataOut);
```

3.1.6 Dispersion Compensation

The vector complex-complex multiplication API given below can be used to perform dispersion compensation, which requires correction in the phase by multiplying the data vector by a phase compensation vector.

```
/**
 *  @brief      Performs complex-complex element wise vector multiplication
 *
 *  @param[in]  pDataIn1    pointer to the first 16 bit input vector (complex)
 *  @param[in]  pDataIn2    pointer to the second 16 bit input vector (complex)
 *  @param[in]  len         length of input vectors
 *  @param[in]  rShift      right shift used to move 32 bit value to 16 bit output;
 *                          valid values are between 0 and 31
 *  @param[out] pDataOut    pointer to the 16 bit output vector
 *
*/
void util_vectorMultCxC_16b_16b_16b(const cplx_least16_t *pDataIn1,
                                    const cplx_least16_t *pDataIn2,
                                    const int_least16_t len,
                                    const uint_least8_t rShift,
                                    cplx_least16_t *pDataOut);
```

4 Performance of TI DSP Based OCT Algorithms

4.1 Benchmark of Individual Algorithms

Table 1 lists the cycle count needed to run each API mentioned in earlier sections for various parameters of interest. The cycle count was determined by running the optimized APIs on a TMS320C6455 DSK.

Table 1. Benchmark on Individual Algorithms

	Parameters		
Item	**Input Sample Size**	**Output Sample Size**	**Cycle Count per Scanline on TMS320C6455 EVM**
Background Subtraction	2048	2048	1557
Complex/Complex Vector Multiplication	2048	2048	4130
Cubic Spline	4096	2048	13456
	2048	2048	8373
Complex to complex FFT	2048	2048	30029
Real to Complex FFT	2048	1024	15241
Real to Complex FFT with 2x Zero Padding	2048	2048	32804
Complex to Complex IFFT with 2x Zero Padding	1024	2048	27873
Complex to Real IFFT with 2x Zero Padding	1024	4096	31916
Magnitude	2048	2048	9064
	1024	1024	4611
Full Range Log Compression (quadratic approximation)	2048	2048	6715
	1024	1024	3387

4.2 Performance With B-Mode Imaging

The individual APIs provided can be used to build the signal processing chains necessary to produce a B-mode image from the raw collected data. The actual signal processing chain is application dependent. The next sections discuss several cases that identify the performances achievable in TIs DSPs.

4.2.1 Case 1

Figure 4 shows the simplest form of signal chain to create the B-mode imaging. Key features of this chain are as follows:

- Input data: 2048 points per scanline
- 2048 point real to complex FFT used
- Output data: 1024 points per scanline

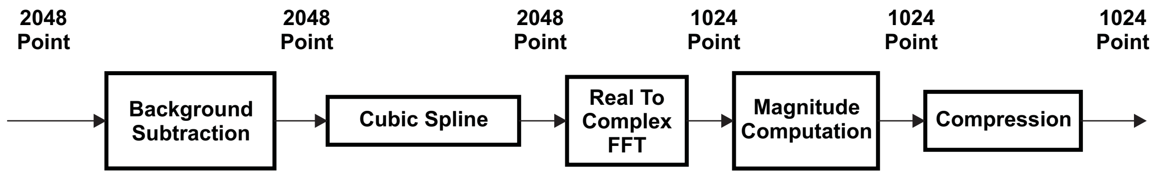

Figure 4. OCT Signal Processing Chain for Case 1

The total cycle count per scanline for this chain is tabulated in Table 2.

Table 2. Cycle Count per Scanline for Signal Chain in Case 1

Algorithm	Cycle Count/Scanline
Background subtraction	1557
Re-sampling	8373
FFT	15241
Magnitude	4611
Compression	3387
Total	**33169**

4.2.2 Case 2

In this case (shown in Figure 5), the interference data is zero padded prior to taking the FFT. This is usually done to have a smooth point spread function (PSF) with more points available on the PSF. Key features of this chain are as follows:

- Input data: 2048 points per scanline
- A 4096-pont real to complex FFT with 2x upsampling (input is 2048 point vector) is used here
- Output data: 2048 points per scanline

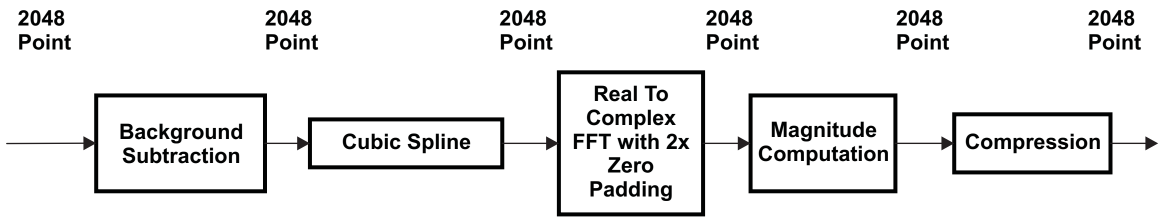

Figure 5. OCT Signal Processing Chain for Case 2

The total cycle count per scanline for this chain is tabulated in Table 3.

Table 3. Cycle Count per Scanline for Signal Chain in Case 2

Algorithm	Cycle Count/Scanline
Background subtraction	1557
Re-sampling	8373
FFT	30029
Magnitude	9064
Compression	6715
Total	**55738**

4.2.3 Case 3

In this case (shown in Figure 6), a 2x interpolation on the raw interference data is performed by first taking an FFT of the input, zero padding the output and then taking an IFFT before performing the re-sampling via cubic interpolation. Sometimes this is done to get better interpolation. Key features of this chain are as follows:

- Input data: 2048 points per scanline
- A total of three FFTs are needed
  - Two 2048 real to complex FFT
  - One 4096 point complex to real IFFT with 2x upsampling (input is 1024 point vector)
- Output data: 1024 points per scanline

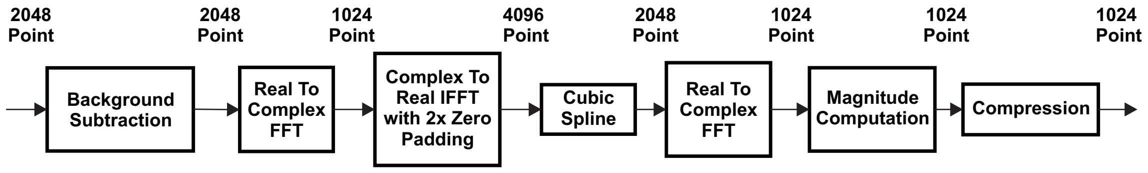

Figure 6. OCT Signal Processing Chain for Case 3

The total cycle count per scanline for this chain is tabulated in Table 4.

Table 4. Cycle Count per Scanline for Signal Chain in Case 3

Algorithm	Sub-Algorithm	Cycle Count/Scanline
Background subtraction		1557
Re-sampling	FFT	15241
	IFFT with 2x zero padding	31916
	Cubic spline	13456
FFT		15241
Magnitude		4611
Compression		3387
Total		**85409**

4.2.4 Case 4

In this case, dispersion compensation is added to the signal chain (see Figure 7) [4], [5]. To perform the dispersion compensation, the complex analytic continuation of the real interference data is formed through the Hilbert transform. The complex output is then multiplied by a phase rotation vector to correct for non-linear phase effects. You need to perform a complex to complex FFT for image generation.

The Hilbert transform itself can be performed by:

1. Real to complex FFT
2. Zeroing out the negative frequencies
3. Taking an IFFT

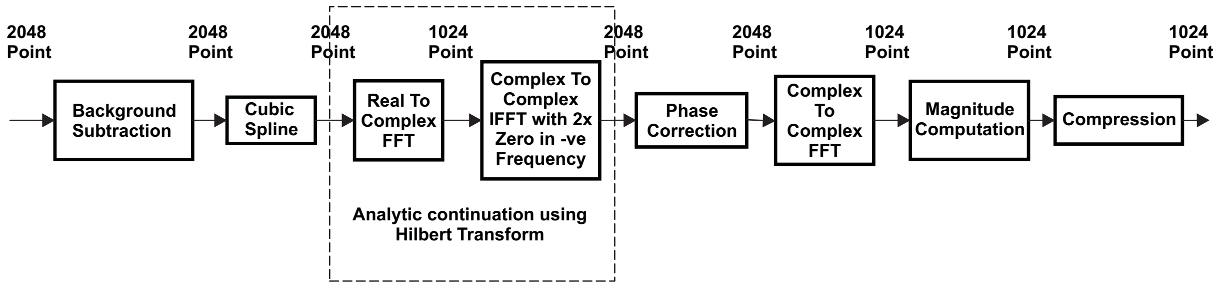

Figure 7. OCT Signal Processing Chain for Case 4

Note that the optimized API needed for the second IFFT in the Hilbert transform had not been supplied in the MED-STK. Complex to complex IFFT with 2x oversampling available in the toolkit is designed to maintain the positive and negative frequencies. However, for benchmarking purpose, you can use the same cycle count for this API since the processing steps on both are essentially the same. Also note that only half of the samples at the output of the image formation FFT carries sample structure information; therefore, the magnitude computation and dynamic range compression only need to be carried out for half the samples of FFT output. The Hilbert transform can possibly be done using FIR filter approximation and this has the potential to improve the MHz performance of the signal chain. Further study is needed for understanding the trade-off between such approximations and image quality.

The total cycle count per scanline for this chain is tabulated in Table 5.

Table 5. Cycle Count per Scanline for Signal Chain in Case 4

Algorithm	Sub-Algorithm	Cycle Count/Scanline
Background subtraction		1557
Re-sampling		8373
Analytic continuation using Hilbert transform	FFT	15241
	IFFT with 2x zero padding	27873
Phase correction		4130
FFT		30029
Magnitude		4611
Compression		3387
Total		**95201**

4.2.5 Overall Benchmark

Based on the cycle counts above, the performance summary of OCT systems on several DSP devices from TI in terms of number of scanlines that can be processed per second is shown in Table 6. 80% loading is used on each of the devices to account for possible additional cycles needed in memory I/O and control overhead.

Table 6. OCT System Performance on Several Devices Based on C64x+ Architecture

Device	No. of Core	Core Clock (MHz)	No. of Scanlines per Second (80% loading)			
			Case 1	Case 2	Case 3	Case 4
TMS320C6455	1	1200	28 K	17 K	11 K	10 K
TMS320C6474	3	1000	72 K	43 K	28 K	25 K
TMS320C6472	6	700	101 K	60 K	39 K	35 K

Note that only the processing needed was used in the image formation signal chain to arrive at these benchmarks. It was assumed that all parameters like the background signal, resampling array, the phase compensation vector have already been computed and are supplied in tables in memory. In practical systems, these parameters are estimated through some sort of calibration process. The calibration can involve some auto-focusing algorithm that needs to be run before acquiring the data for imaging. Such auto-calibration mechanism does not need to be real time but needs to be done fast to ensure little delay in image acquisition. These mechanisms are also application dependent and tend to be proprietary to each system vendor.

5 Conclusion

This application report describes how the APIs available in the MED-STK can be used to efficiently implement a signal processing chain for optical coherence tomography-based systems. The results indicate that TI's C64x+ based multi-core devices like the TMS320C6472 has the capability to perform scanline reconstruction at a rate of up to 100 Klines per second. The TMS320C6472 device also has high bandwidth I/O like 2x serial rapid I/O (SRIO) with capability to bring in and out the data at a rate of 5 Gbps. All these capabilities exist at sub 10 W power. Therefore, this device is ideally suited to perform OCT signal processing at much lower power and cost compared to existing systems. If an application requires lower line rate, the OEM has the option to choose a different device with the same core architecture. For higher line rates, multiples of these devices can be used in parallel. TI's next generation of devices provide increased performance at competitive levels of power consumption while continuing to meet the requirements of OCT systems in the future.

6 References

1. M. Brezinski, Optical Coherence Tomography, Elsevier, 2006.
2. S. R. Dooley, R. W. Stewart and T. S. Durrani, *Fast On-Line B-Spline Interpolation*, IEEE Electronics Letters, Vol. 35, No. 14, pp. 1130-1131, July 1999.
3. *TMS320C64x+ DSP Little-Endian DSP Library Programmer's Reference* (SPRUEB8)
4. D. L. Marks, A. L. Oldenburg, J. J. Reynolds and S. A. Boppart, *Autofocus Algorithm for Dispersion Correction in Optical Coherence Tomography*, Applied Optics, pp. 3038-3046, Vol. 42, No. 16, Jun. 2003.
5. M. Wojtkowski, V. J. Srinivasan, T. H. Ko, J. G. Fujimoto, A. Kowalczyk, and J. S. Duker, *Ultrahigh-Resolution, High-Speed, Fourier Domain Optical Coherence Tomography and Methods for Dispersion Compensation*, Optics Express, pp. 2404-2422, Vol. 12, No. 11, May 2004.
6. *Using TI's Embedded Processor Software Toolkit for Medical Imaging* (MED-STK) (SPRABB8)