Application Report
SPRA886 – March 2003

Image Processing Examples Using the TMS320C62x Image/Video Processing Library (IMGLIB)

Chris Chung
Oliver Sohm

TMS320C6000 Software Applications

ABSTRACT

The TMS320C62x™ image/video processing library (IMGLIB) provides a set of C-callable, assembly-optimized functions commonly used in imaging applications. While IMGLIB can be used to develop practical real-time applications with its high performance and ease of use, there are several important factors to consider in a system environment for attaining optimal performance mainly due to the data-intensive nature of imaging applications. This application report presents the usage and performance of several IMGLIB functions to help users utilize IMGLIB in their system development, and also presents performance analysis with three kinds of memory scenarios to help understand potential overhead related to memory hierarchy.

Contents

List of Figures

List of Tables

1 Introduction

Imaging applications are often considered as data-intensive, as well as compute-intensive, because they typically process large amount of data in real time, requiring efficient data transfer mechanisms as well as high computing power. TMS320C62x is an advanced, very long instruction word (VLIW) processor, suitable for imaging applications with its high computing power and large on-chip memory. While TMS320C620x provides direct memory access (DMA) to efficiently transfer data to/from off-chip memory, TMS320C621x also provides cache as well as enhanced direct memory access (EDMA). To help TMS320C62x users shorten the time-to-market in system development, Texas Instruments provides a set of assembly-optimized key imaging functions, named imaging/video processing library (IMGLIB). Each function in the IMGLIB is designed to produce the best performance possible by optimally utilizing available resources and avoiding potential resource conflicts. Therefore, when developing a system utilizing IMGLIB, it is important to understand potential overhead related to memory hierarchy, in order to estimate and improve the actual performance of a system being developed.

Figure 1 shows the memory hierarchy of C620x and related potential overhead. For example, without considering compulsory DMA/cache overhead, DMA operations and/or cache misses can occur when the program is bigger than the size of the program memory (PM) and/or cache. Similarly, when the data do not fit in the data memory (DM), DMA operations are required to transfer code/data between DM and off-chip memory.

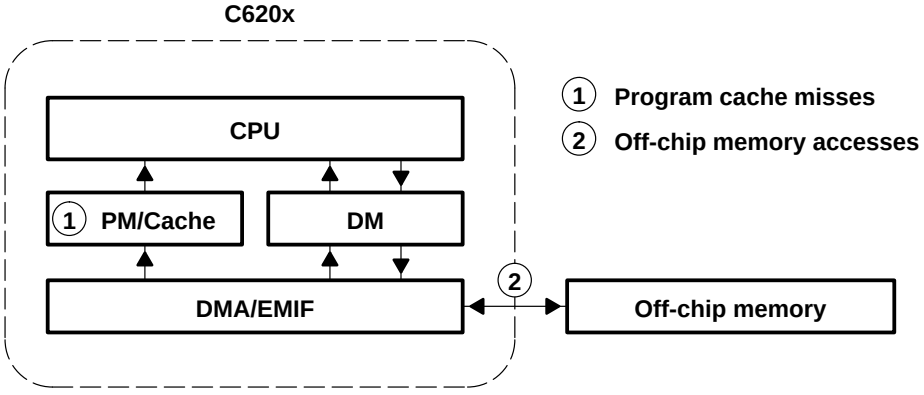


Figure 1. C620x Memory Hierarchy and Potential Overhead

Figure 2 shows the memory hierarchy of C621x and related potential overhead. For example, when the program is bigger than the size of the level-one program cache (L1P), L1P cache misses can occur, stalling the central processing unit (CPU) until the required code is fetched. Similarly, when the data do not fit in the level-one data cache (L1D), L1D cache misses will stall the CPU.

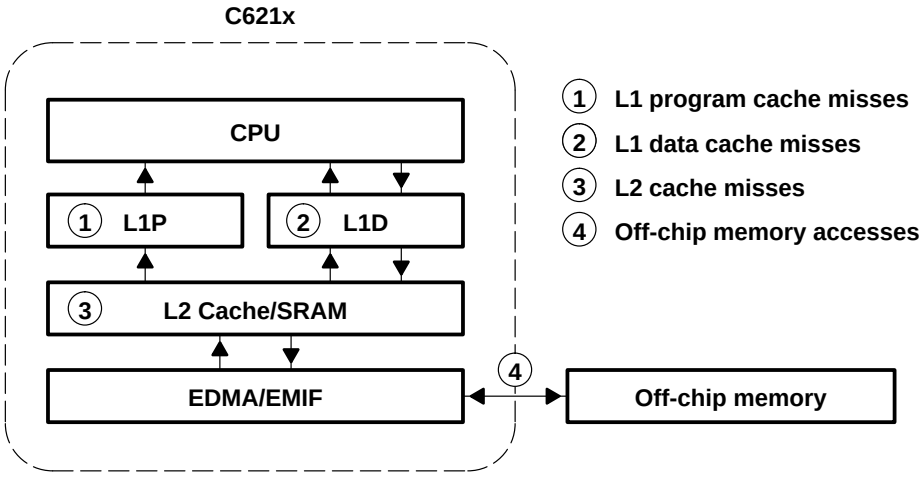


Figure 2. C621x Memory Hierarchy and Potential Overhead

All L1P and L1D misses are serviced by the level-two cache/static random-access memory (L2 cache/SRAM). When L2 cache is used, L2 misses will incur off-chip memory accesses via EDMA. When L2 SRAM is used, EDMA is required to transfer code/data between L2 SRAM and off-chip memory if the code and data do not fit in the L2 SRAM. The data transfer with EDMA is typically more effective than that with L2 cache due to its nature of longer burst transactions, reducing memory access latency overhead. However, the EDMA transfer involves more programming effort because data transfers and synchronization have to be manually managed. TMS320C621x provides both cache and DMA mechanisms to allow users to choose the right mechanism, depending on situations.

This application report presents the usage and performance of several IMGLIB functions to help users utilize IMGLIB in system development. In addition, performance analysis with three kinds of memory scenarios is presented, to help understand potential overhead related to memory hierarchy.

2 Benchmarking

2.1 Emulation/Simulation Setup

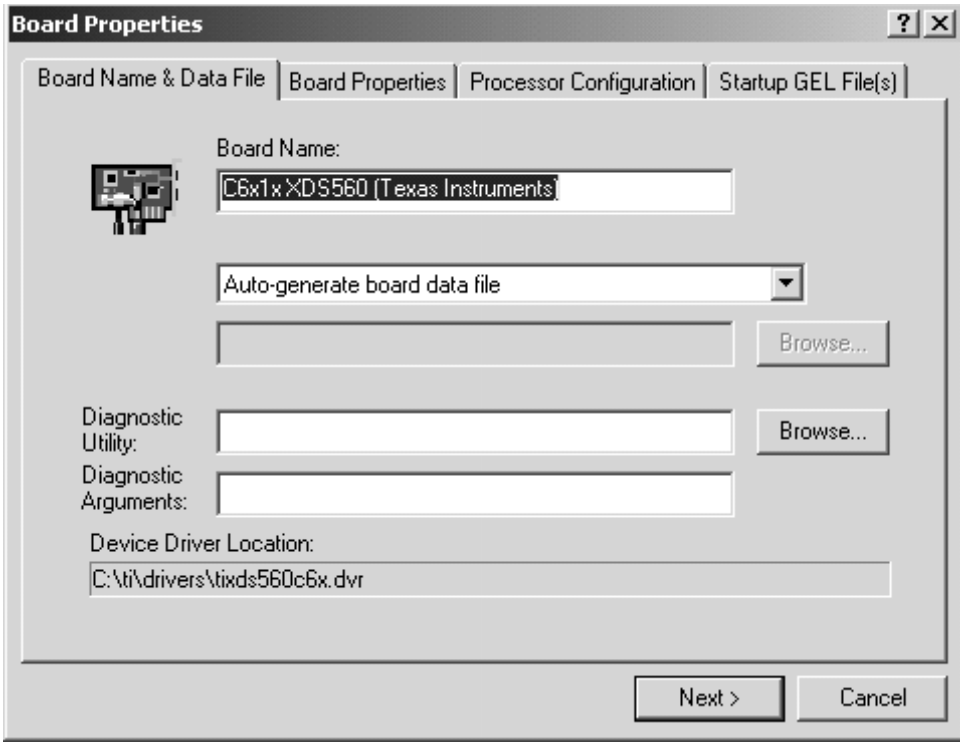
A TMS320C6711 DSP starter kit (DSK) is used in this application report to measure cycle counts for IMGLIB examples. Table 1 lists key features of the C6711 DSK, which are important factors in performance analysis and optimization. More details on the C6711 internal memory structure and operations can be found in *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609).

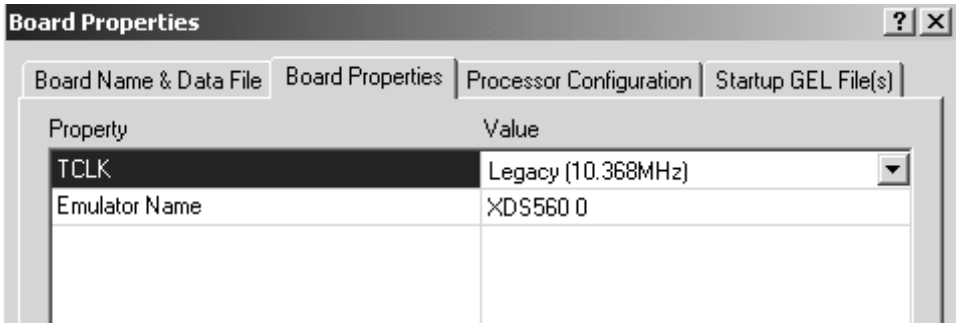
Table 1. C6711 DSK Key Features

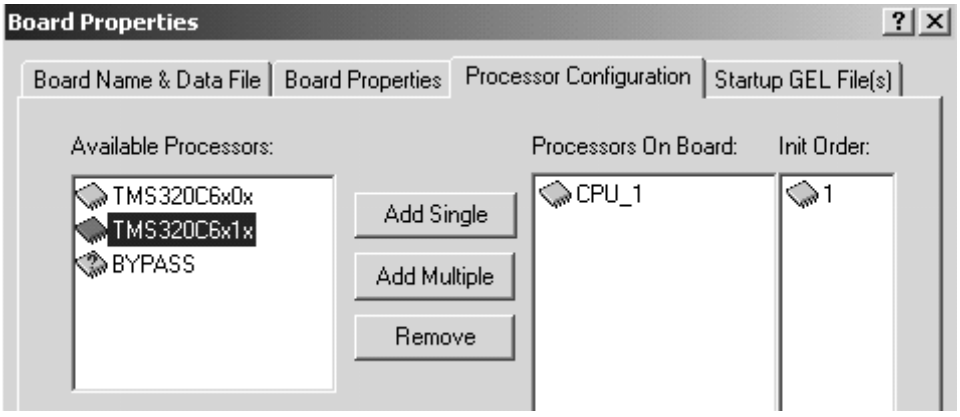
	Item	Description
C6711	Clock frequency	150 MHz
	L1P	4K-byte, direct-mapped, 64-byte cache line
	L1D	4K-byte, 2-way set associative, 32-byte cache line, 64-bit wide, dual-ported
	L2 SRAM	5-cycle L1P miss penalty, 4-cycle L1D miss penalty, up to 64K bytes, four 64-bit banks
	L2 cache	5-cycle L1P miss penalty, 4-cycle L1D miss penalty, up to 64K bytes, 1/2/3/4-way set associative, 128-byte cache line, four 64-bit banks
	L2 to L1D read path	128 bits
	L1D to L2 write buffer	32-bit, 4-entry, L2 can process a write request every 2 cycles
	EMIF	32-bit bus
SDRAM	Clock frequency	100 MHz
	Bus width	32 bits (two 16-bit modules)
	Page size	1K byte

The C6711 DSK is connected to a PC through an XDS560 board, and a Code Composer Studio™ Integrated Development Environment (IDE) configuration, based on "_C6x1x XDS560 Emulator Address 0", is used as follows. If you use other types of interfaces, e.g., XDS510 or parallel port, select the right configuration.

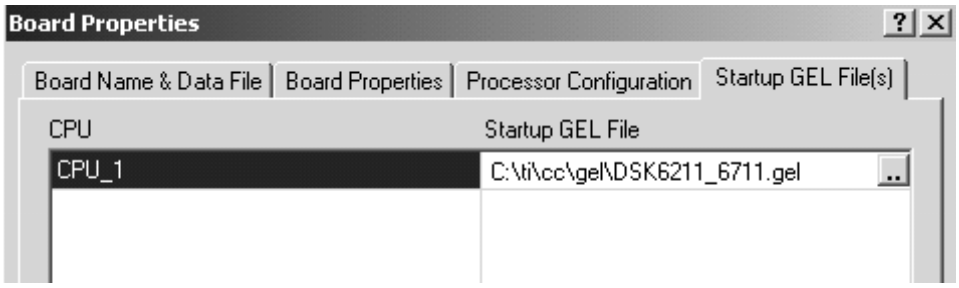
Code Composer Studio is a trademark of Texas Instruments.







Be sure to select the right General Extension Language (GEL) file for the C6711 DSK.



If you use simulation, select “C6711 Sim Ltl Endian.” The cycle counts obtained from simulation might not be accurate, especially with off–chip memory accesses.

Software version numbers used in this application report are as follows:

- Code Composer Studio: version 2.1
- C62 IMGLIB: version 1.02b

2.2 Cycle Count Measurement

The built-in timer in C6711 is used here to measure cycle counts for IMGLIB examples. The following sample code shows how to set up the timer and measure cycle counts with Chip Support Library (CSL).

```
hTimer = TIMER_open(TIMER_DEVANY,0);   /* open a timer */

/*--------------------------------------------------------------------*/
/* Configure the timer. 1 count corresponds to 4 CPU cycles in C62 */
/*--------------------------------------------------------------------*/
                      /* control     period      initial value     */
TIMER_configArgs(hTimer, 0x000002C0, 0xFFFFFFFF, 0x00000000);
/* ------------------------------------------------------------------ */
/*  Compute the overhead of calling the timer.                        */
/* ------------------------------------------------------------------ */
start    = TIMER_getCount(hTimer); /* to remove L1P miss overhead  */
start    = TIMER_getCount(hTimer);
stop     = TIMER_getCount(hTimer);
overhead = stop – start;

start = TIMER_getCount(hTimer);
/* ------------------------------------------------------------------ */
/*  Call a function here.                                             */
/* ------------------------------------------------------------------ */
diff = (TIMER_getCount(hTimer) – start) – overhead;
TIMER_close(hTimer);
printf(”%d cycles \n”, diff*4);
```

The maximum resolution of the timer is 4 CPU cycles, since the input clock to the timer is fixed to the CPU clock divided by four. The function call overhead for TIMER_getCount() is roughly measured and compensated. Additional information on the timer registers can be found in the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

2.3 Example Scenarios and Expected Performance

Three kinds of scenarios are presented in this report, considering potential overhead related to memory hierarchy:

1. When data are in L1D (C621x) or data memory (C620x)
2. When data are in L2 SRAM (C621x)
3. When data are in off-chip memory (both C621x and C620x)

In these examples, L1P miss overhead can be ignored because the examples are small enough to fit into L1P. However, L1P misses have to be carefully managed in the case where L1P thrashing overhead is significant.

2.3.1 Scenario 1: Data in L1D (C621x) or Data Memory (C620x)

This scenario is shown to validate the formula cycle counts listed in the *TMS320C62x Image/Video Processing Library Programmer's Reference* (SPRU400). Note that the formula cycle count assumes flat memory, not considering any overhead related to memory hierarchy. Figure 3 shows a linker command file used for this scenario. For more information on linker commands, refer to the *TMS320C6000 Optimizing Compiler User's Guide* (SPRU187). Information on C6000 memory maps can be found in the *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186).

Since all code and data are initially stored in L2SRAM in C621x, the example function is called twice, to eliminate L1P/L1D miss overhead, and a cycle count for the second call is measured. Although cycle counts close to the formula cycle counts can be achieved in this scenario, additional stall cycles can occur. For example, the write buffer can hold up to four transactions, and L2 cache/SRAM can process a transaction every 2 cycles. Therefore, more than one write-miss transaction in every two cycles can stall the CPU. Bank conflicts in C620x or access conflicts in C621x can also incur additional stall cycles. In C620x, the L1D is organized as multiple banks, and multiple transactions can be handled at the same time, unless they access same bank. In C621x, the L1D is dual-ported, allowing multiple transactions without bank conflicts. However, when two transactions access same word data, additional stall cycle will occur (i.e., data access conflict).

```
MEMORY
{
    L2SRAM:   o = 00000000h   l = 00010000h  /* 64 kbytes */
}
SECTIONS
{
    .cinit      >       L2SRAM
    .text       >       L2SRAM
    .stack      >       L2SRAM
    .bss        >       L2SRAM
    .const      >       L2SRAM
    .data       >       L2SRAM
    .far        >       L2SRAM
    .switch     >       L2SRAM
    .sysmem     >       L2SRAM
    .tables     >       L2SRAM
    .cio        >       L2SRAM
}
```

Figure 3. Linker Command File for Scenarios 1 and 2

2.3.2 Scenario 2: Data in L2 SRAM (C621x Only)

In this scenario, L1D miss overhead needs to be considered. The linker command file for Scenario 1 (shown in Figure 3) is used for this scenario. Table 2 lists expected stall cycles related to L1D read and/or write transactions. When there are read transactions only, the number of stall cycles is the number of L1D read misses times L1D miss penalty (i.e., 4 cycles). In case of write transactions only, there is no stall unless the write buffer is full. The write buffer is 32-bit wide and allows up to four outstanding misses.

When there are both read and write transactions, the L1D read miss penalty can increase because any write transaction in the write buffer is flushed before a read miss is serviced to maintain data coherency.

Table 2. Stall Cycles Related to L1D

Transaction	Number of Stall Cycles
Read transaction only	Number of L1D read misses * L1D miss penalty
Write transaction only	No stall cycle unless the write buffer is full
Read and write transactions	Number of L1D read misses * (L1D miss penalty + additional cycles for write buffer flush)

2.3.3 Scenario 3: Data in Off-Chip Memory (Both C620x and C621x)

In C620x, DMA is always used to transfer data between L2 SRAM and off-chip memory. In C621x, L2 cache as well as EDMA can be used to access the data in off-chip memory. EDMA is typically advantageous over cache in terms of performance for the following three reasons. First, the overhead of access latency with EDMA is less than that with cache, since an EDMA transfer can be much longer than a cache line transfer. Second, computation and data transfers can be tightly overlapped with EDMA, which often results in significant performance improvement. Third, the L2 write-allocate policy can result in more data transfers than needed (i.e., load/allocate/writeback instead of load/store).

With DMA (EDMA), the overall processing time depends on the ratio between the compute time and the data transfer time, as categorized into two conditions listed in Table 3. In the compute-bound condition where the compute time is greater than the data transfer time, the overall processing time is determined by the compute time. Note that the compute time is defined as the processing time without the overhead of off-chip memory accesses. The data transfer time is defined as the time to transfer data to/from off-chip memory, which consists of memory access latency and burst transfer time.

Table 3. Compute-Bound vs. Memory-Bound

Condition	Description	Overall Processing Time
Compute-bound	Compute time > data transfer time	Compute time + time for the first load and last store transfers
Memory-bound	Compute time < data transfer time	Data transfer time + DMA (EDMA) management overhead

To reduce the access latency overhead in memory accesses, the burst transfer must be long enough. However, too long burst transfers can cause negative effect in compute-bound cases because time for the first load and last store transfers can not be hidden behind the compute-time. In the memory-bound condition where the compute time is less than the data transfer time, the overall processing time is determined by the data transfer time; thus further improving the compute time does not contribute to higher performance.

From an experiment on the C6711DSK, the cycle count of 29,500 was measured in transferring 64K bytes of data between on–chip and off–chip memory with DMA. This cycle count will be to analyze EDMA examples in Section 3. Additional stall cycles can occur due to L2 access conflicts between L1D and EDMA. For example, the L2 access conflict can lengthen the data transfer time in C6711 because L1D has always a higher priority.

Figure 4 shows a linker command file used for this scenario.

```
MEMORY
{
    L2SRAM:   o = 00000000h   l = 00010000h  /* 64 kbytes */
    CE0:      o = 80000000h   l = 00100000h  /* 1 Mbytes */
}
SECTIONS
{
    .cinit      >       L2SRAM
    .text       >       L2SRAM
    .stack      >       L2SRAM
    .bss        >       L2SRAM
    .const      >       L2SRAM
    .data       >       L2SRAM
    .far        >       L2SRAM
    .switch     >       L2SRAM
    .sysmem     >       L2SRAM
    .tables     >       L2SRAM
    .cio        >       L2SRAM
    .imgbuf     >       CE0  /* User created data section for off-chip memory */
}
```

Figure 4. Linker Command File for Scenario 3

The following statements are used to allocate image arrays to a user-defined section (*.imgbuf*) in off-chip memory.

```
#pragma DATA_SECTION(in_image,".imgbuf")

#pragma DATA_SECTION(out_image,".imgbuf")
```

Figure 5 shows a double buffering code used for DMA (EDMA) transfers, which utilizes the DAT module in CSL. The double buffering code is simplified for easier explanation, thus it can only handle the case where the total data size is a multiple of the buffer size times two. It uses two sets of input and output buffers, called *InBuffA/OutBuffA* and *InBuffB/OutBuffB*. With the two sets of buffers, the CPU can process data with one set of buffers, while DMA (EDMA) transfers data in another set of buffers.

The DAT_wait() is used to wait for a transfer to complete. The DAT_copy() issues a data transfer that happens in the background of CPU processing. The first two blocks of input data are transferred to input buffers before the double buffering loop begins. Once the transfers have completed, you can process the data in InBuffA and store the results to OutBuffA. The data in OutBuffA is sent to off-chip memory, and the next DAT_copy() begins copying the next input block to InBuffA for future processing. While these two transfers are occurring, you process the data in InBuffB and store the results to OutBuffB. When it is done, the result in OutBuffB is sent to off-chip memory.

```
number_of_transfers = total_data_size / buffer_size;
/*--------------- Initial transfers ----------------------------*/
id_InBuffA  = DAT_copy(in_image, InBuffA, buffer_size);
id_InBuffB  = DAT_copy(in_image + buffer_size, InBuffB, buffer_size);
/* Begin Double Buffering */
for(i=0; i < number_of_transfers; i+=2)
{
    DAT_wait(id_InBuffA);   /* wait for transfers to complete */
    DAT_wait(id_OutBuffA);
    /* ------------------------------------------------------------ */
    /* Process the data in InBuffA and store the results to OutBuffA */
    /* ------------------------------------------------------------ */
    Process( InBuffA, OutBuffA, buffer_size );
    id_OutBuffA = DAT_copy(OutBuffA, out_image + (i* buffer_size), buffer_size);
    if( i < number_of_transfers-2 )
        id_InBuffA  = DAT_copy(in_image + ((i+2)* buffer_size), InBuffA,
                      buffer_size);
    DAT_wait(id_InBuffB);
    DAT_wait(id_OutBuffB);

    /* ------------------------------------------------------------ */
    /* Process the data in InBuffB and store the results to OutBuffB */
    /* ------------------------------------------------------------ */
    Process( InBuffB, OutBuffB, buffer_size );
    id_OutBuffB = DAT_copy(OutBuffB, out_image + (i+1)* buffer_size],
                  buffer_size);
    if( i < number_of_transfers-2 )
        id_InBuffB  = DAT_copy(in_image + ((i+3)* buffer_size), InBuffB,
                      buffer_size);
}
```

Figure 5. Simplified Double Buffering Code

2.4 Data Alignment

Due to the structure of internal memory/cache, some IMGLIB functions require input/output memory arrays to be aligned to a specific boundary. While this restriction does not apply to C621x devices that employ dual-ported internal memory/cache, it must be carefully managed in C620x for attaining optimal performance. For example, the following statement is used to allocate the array (*input*) to a 4-byte boundary.

#pragma DATA_ALIGN (input, 4)

The C62x compiler automatically aligns arrays of type *double* to an 8-byte boundary, while others are aligned to a 4-byte boundary if they are not declared in a *struct* statement. When dynamic memory allocation is used, the allocated memory is aligned to an 8-byte boundary regardless of types. More information on data alignment rules by the compiler can be found in *TMS320C6000 Optimizing Compiler User's Guide* (SPRU187).

The structure of internal memory/cache on the C62x gemeration varies from device to device. Therefore, refer to the appropriate device data sheet to determine the structure of a particular device.

3 Examples

This section presents the usage and performance of few IMGLIB functions: histogram, threshold, dithering, and correlation. To minimize the variation in cycle count measurement, be sure to select the *Reset* menu (under *Debug* in *Code Composer Studio*) before running an example. The cycle counts listed in this section were measured in *Release* mode.

3.1 Histogram

Image histogram counts the number of occurrences of pixel intensity in an image, which is widely used in image analysis and enhancement. The *IMG_histogram* function, which computes the histogram on an 8-bit image, is defined as:

```
void IMG_histogram  (unsigned char * restrict in_data, int n, short accumu-
late, unsigned short * restrict t_hist, unsigned short * restrict hist)
```

The maximum number of pixels that can be counted in each bin is 65535. Note that a temporary array, *t_hist*, must have 1024 16-bit entries and be initialized to zero. For example, with EDMA, this function is called for each block of data. Therefore, *t_hist* must be cleared before each call. The input array (*in_data*) must be aligned to a 4-byte boundary, and the number of input data (*n*) must be a multiple of 8. Table 4 lists IMG_histogram benchmarks.

Table 4. IMG_histogram Benchmarks

	Number of Cycles				
		Data In On-Chip Memory		Data In Off-Chip Memory (150-MHz CPU, 100-MHz SDRAM)	
Number of Data (N)	Formula (9/8) * N + 560	Scenario 1 (L1D)	Scenario 2 (L2 SRAM)	Scenario 3 (EDMA)	Transfer Cycles
1,024	1,712	1,736	1,868†	–	–
65,536	74,288	–	82,480‡	99,056§	29,500

† Without further cache optimization
‡ Estimated from the result of 1K byte of data in Scenario 2
§ With 8K-byte EDMA buffers

The cycle count for Scenario 1 is close to the formula cycle count because no cache miss occurs. For Scenario 2, there will be L1D miss overhead (i.e., 4 cycles) for every 32 bytes of input data. For example, the expected number of cycles with 1K byte of data is 1840 cycles [1840 = 1712 (i.e., formula cycles) + 1024 (i.e., number of data in bytes) / 32 (i.e., L1D cache line) * 4 (i.e., L1D miss overhead with L2 SRAM)], which is close to the measured cycle count of 1868.

Since 8K-byte EDMA buffers are used, there will be 8 function calls for 64K–byte of data in Scenario 3. Therefore, the formula cycle count becomes 78,208 cycles (78,208 = ((9/8)*8,192+560)*8), which corresponds to 86,400 compute cycles [86,400 (= 78,208 + 65,536/32 * 4)] considering L1D read miss penalty. Since this compute cycle count (i.e., 86,400) is much larger than the transfer cycles (i.e., 29,500 from section 2.3.3) in this example, this function is considered as compute-bound; thus expected performance includes cycles for the first and last block transfers (i.e., 3,688 = 29,500 / 8). Another overhead to consider is the time to clear *t_hist* eight times instead of one, which increases the overall processing time since this function is compute-bound.

3.2 Threshold

Image threshold has many different uses in image/video processing systems, including converting grayscale images to binary images for morphological processing, and converting image formats suitable for segmentation. The IMGLIB contains four threshold functions:

- IMG_thr_gt2max: pixels greater than the threshold are set to 255.
- IMG_thr_gt2thr: pixels greater than the threshold are set to the threshold.
- IMG_thr_le2min: pixels less than or equal to the threshold are set to 0.
- IMG_thr_le2thr: pixels less than or equal to the threshold are set to the threshold.

The four functions are defined as:

```
void IMG_thr_gt2max  (const unsigned char * restrict in_data, unsigned char *
restrict out_data, short cols, short rows, unsigned char threshold)

void IMG_thr_gt2thr  (const unsigned char * restrict in_data, unsigned char *
restrict out_data, short cols, short rows, unsigned char threshold)

void IMG_thr_le2min  (const unsigned char * restrict in_data, unsigned char *
restrict out_data, short cols, short rows, unsigned char threshold)

void IMG_thr_le2thr  (const unsigned char * restrict in_data, unsigned char *
restrict out_data, short cols, short rows, unsigned char threshold)
```

Note that the number of data is specified in two-dimension variables, *cols* and *rows*. The input (*in_data*) and output (*out_data*) arrays must be aligned to a 4-byte boundary and must not be overlapped. The number of input data (*cols* * *rows*) must be at least 16 and a multiple of 16. Stack needs to be aligned to an 8-byte boundary. Table 5 lists IMG_thr_gt2max benchmarks.

Table 5. IMG_thr_gt2max Benchmarks

	Number of Cycles				
		Data in On-Chip Memory		Data in Off-Chip Memory (150-MHz CPU, 100-MHz SDRAM)	
Number of Data (N)	Formula (9/16) * N + 24	Scenario 1 (L1D)	Scenario 2 (L2 SRAM)	Scenario 3 (EDMA)	Transfer Cycles
1,024	600	628	888†	–	–
65,536	36,888	–	53,488‡	78,944§	59,000

† Without further cache optimization
‡ Estimated from the result of 1K-bytes of data in Scenario 2
§ With 4K-byte EDMA buffers

The cycle count for Scenario 1 is close to the formula cycle count because no cache miss occurred. For Scenario 2, there will be L1D write miss overhead as well as L1D read miss overhead for every 32 bytes of data. For example, the expected number of cycles with 1K byte of data is 728 cycles [728 = 600 (i.e., formula cycles) + 1,024 (i.e., number of data in bytes) / 32 (i.e., L1D cache line) * 4 (i.e., L1D read miss overhead with L2 SRAM)], which shows about 18% error compared to the measured cycle count of 888. This is because L1D read miss stall cycle count increases (1.22 times) when the write buffer is not empty.

Since 4K-byte EDMA buffers are used, there will be 16 function calls for 64K–byte of data in Scenario 3. Therefore, the formula cycle count becomes 37,248 cycles (37,248 = ((9 / 16) * 4,096 + 24) * 16), which corresponds to 55,437 estimated compute cycles (55,437 = (37,248 + 65536 / 32*4) * 1.22) for Scenario 2. Since the estimated compute cycle is less than the transfer cycles (i.e., 59,000 = 29,500 * 2) in this example, this function is considered as memory-bound. Therefore, the expected cycle count is 59,000 cycles, which has 19,944 cycles of difference from the measure one (i.e., 78,944). Most of these additional stall cycles are caused by the L2 access conflicts between L1D and EDMA as discussed in Section 2.3.3.

3.3 Dithering

Image dithering is commonly used in printing applications. The C62 IMGLIB dithering function (IMG_errdif_bin) implements the Floyd-Steinberg error diffusion algorithm, which is defined as:

```
void IMG_errdif_bin (unsigned char * restrict errdif_data, int cols, int rows,
short * restrict err_buf, unsigned char thresh)
```

The err_buf[ ] arrary must be initialized to 0 prior to the first call. Each pixel in the errdif_data[ ] is compared against a user-specified threshold (thresh). Pixels larger than the threshold are set to 255, while other pixels are set to 0. The error value for a pixel is propagated to the neighboring pixels using the Floyd-Steinberg filter. The number of columns (*cols*) must be at least 4. Table 6 lists IMG_errdif_bin benchmarks.

Table 6. IMG_errdif_bin Benchmarks

	Number of Cycles				
		Data In On-Chip Memory		Data In Off-Chip Memory (150-MHz CPU, 100-MHz SDRAM)	
Number of Data (N = COLS * ROWS)	Formula (4*COLS+14)* ROWS+21	Scenario 1 (L1D)	Scenario 2 (L2 SRAM)	Scenario 3 (EDMA)	Transfer Cycles
1,024 = 32 * 32	4,565	4,588	4,724†	–	–
65,536 = 256 * 256	265,749	–	273,941‡	290,880§	59,000

† Without further cache optimization
‡ Estimated cycles from 1K byte of data in Scenario 2
§ With 4K-byte EDMA buffers

The cycle count for Scenario 1 is close to the formula cycle count due to no cache misses. For Scenario 2, there will be L1D miss overhead (i.e., 4 cycles) for every 32 bytes of input data. For example, the expected number of cycles with 1K byte of data is 4,716 cycles [4,716 = 4,588 (i.e., cycles for Scenario 1) + 1,024 (i.e., number of data in bytes) / 32 (i.e., L1D cache line) * 4 (i.e., L1D miss overhead with L2 SRAM)], which is close to the measured cycle count of 4,724. There is no additional stall cycle related to write buffer full, since the input memory is used as the output memory (i.e., in-place computation).

Since the cycle count for Scenario 2 (i.e., 273,941) is larger than the transfer cycles (i.e., 59,000) in this example, this function is considered as compute-bound with EDMA; thus, expected cycle count includes cycles for the first and last block transfers, as well as the cycles of Scenario 2.

3.4 Correlation

Image correlation is used for image matching. The IMGLIB correlation function (IMG_corr_3x3) computes correlation of an image with a 3x3 mask. The output will have the highest value at the best-matched input image location. The IMG_corr_3x3 is defined as:

```
void IMG_corr_3x3  (const unsigned char * restrict in_data, int * restrict
out_data, const unsigned char mask[3][3], int x_dim, int n_out )
```

This function needs to be called once for each row. However, it may be invoked for multiple rows at a time when the number of output (*n_out*) is a multiple of the width (*x_dim*). In this case, two outputs at the end of each row will have meaningless values, thus care must be taken when interpreting the results. All arrays (*in_data*, *out_data* and *mask*) must not be overlapped. The number of outputs (*n_out*) must be a multiple of 2. Table 7 lists IMG_corr_3x3 benchmarks.

Table 7. IMG_corr_3x3 Benchmarks

	Number of Cycles				
		Data In On-Chip Memory		Data In Off-Chip Memory (150-MHz CPU, 100-MHz SDRAM)	
Number of Data (N)	Formula 4.5 * N + 35	Scenario 1 (L1D)	Scenario 2 (L2 SRAM)	Scenario 3 (EDMA)	Transfer Cycles
1,024	4,636	4,676	4,920†	–	–
65,536	294,947	–	310,338‡	323,684§	151,188

† Without further cache optimization
‡ Estimated cycles from 1K byte of data in Scenario 2
§ With 2K-byte input and 8K-byte output EDMA buffers

In Scenarios 1 and 2, this example is similar to the image threshold in section 3.2, since it has separate input and output arrays.

The IMG_corr_3x3 function requires N+2 rows of input data to compute N rows of output results. Due to the nature of block processing with EDMA, input image blocks are overlapped by 2 rows. Therefore, the size of input data becomes 73,728 bytes [73,728 = 65,536 + (65,536 / 4,096) * (2 * 256)] and the total size of data transfers is 335,872 bytes (335,872 = 73,728 + 65,536 * 4), which corresponds to 151,188 cycles (151,188 = 29,500 * 335,872 / 65,536). Since the estimated cycle count for Scenario 2 (i.e., 310,338) is more than the transfer cycles (i.e., 151,188) in this example, this function is considered as compute-bound; thus expected cycle count includes cycles for the first and last block transfers, as well as the cycles of Scenario 2.

4 References

1. *TMS320C621x/C671x DSP Two-Level Internal Memory Reference Guide* (SPRU609).
2. *TMS320C6000 Peripherals Reference Guide* (SPRU190).
3. *TMS320C62x Image/Video Processing Library Programmer's Reference* (SPRU400).
4. *TMS320C6000 Optimizing Compiler User's Guide* (SPRU187).
5. *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186).
6. *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401).
7. Rafael C. Gonzalez and Richard E. Woods, *Digital Image Processing*, Addison-Wesley Publishing Company, New York, 1992.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third–party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265