Application Report
SPNA165–September 2012

A Tutorial on Optimizing Vision Algorithms on TI DSPs

Senthil Kumar Yogamani

ABSTRACT

Computer vision algorithms are computationally intensive; tailoring them for a resource constrained embedded processor like DSP is challenging. The performance improvements achieved by optimization can be huge. For example, optimization of OpenCV's Pedestrian Detection on the C674x DSP improved the performance by a factor of > 200x. This application report is targeted towards engineers who develop or optimize vision algorithms on TI DSPs.

Contents

List of Figures

List of Tables

1 Introduction

The design of optimal algorithms is important to achieve real-time performance and to fully leverage the DSP architecture parallelism. Some of the most influential algorithms, such as FFT and quick-sort, are efficient algorithms to improve speed. This document focuses on optimizing image processing and computer vision algorithms, and discusses commonly used optimization tricks and techniques. It also focuses on the basic details of the DSP required for optimization. For a more detailed description, see *Introduction to TMS320C6000 DSP Optimization* (SPRABF2).

2 DSP Architecture

2.1 SIMD

The C6000 DSPs achieve parallelism by very long instruction word (VLIW) and single instruction, multiple data (SIMD). SIMD is exhibited in the form of packed arithmetic. The basic size of the register is 32-bit, which can hold 4 bytes, two shorts or one integer. Different instructions interpret the data differently add(int), add2(short), add4(char). Byte data exploits a lot of parallelism and the integer does the least. It is important to choose the smallest data type that is required.

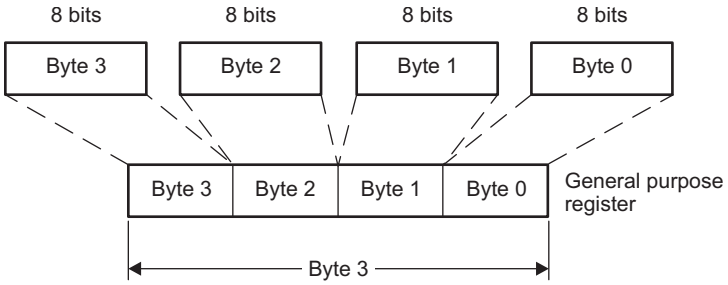

Figure 1. Packed Arithmetic

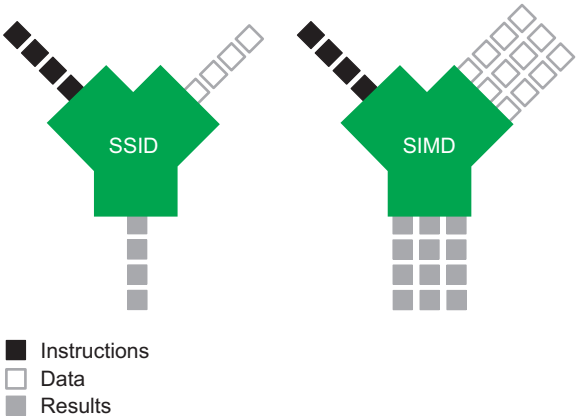

Figure 2. SIMD Processing

In the case of SIMD, the same operation is repeated on multiple data. Vectorization is the process compiler used to achieve optimal SIMD mapping. The compiler does not automatically perform this procedure (see the following examples: Histogram(IMGLIB) and FFT(DSPLIB)). Typically, the SIMD iteration level is performed.

```
for I=1:n
C[I] = A[I] + B[I];
for I=1:n/4
_mem4(C[I]) = _add4(A[I],B[I]);
```

2.2 Pipelining

Pipelining is a form of parallelism where a sequential series of operations are compared. Here is a simple example of instruction execution pipeline that is present in most processors. VLIW processors exhibit the same at an instruction level. The term very long instruction is used because of multiple instructions corresponding to different iterations getting executed at the same cycle.

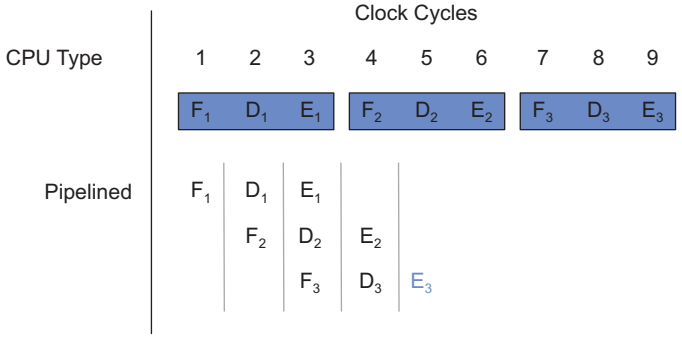

Figure 3. FDE Pipelining

In the instruction level pipelining shown in Figure 4, the architecture has two units: M for multiply and A for add and sub. The instructions are staggered and parallelized as shown in Figure 3. To exploit the parallelism, the operations must be orthogonal to fit across the parallel units.

The following are necessary conditions for efficient pipelining:

- Repeated operations on a sequence of data
- Complementary operations so that different units can be used

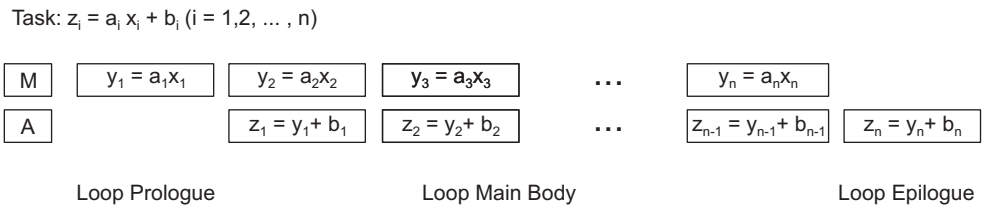

Figure 4. Instruction Level Pipelining

Most signal processing algorithms satisfy this criteria and instruction level parallelism (ILP) is quite common in DSP architectures. ILP hardware is simpler as there is no redundancy like SIMD, where all functionality is replicated across the units. This does not imply that SIMD is better than VLIW. SIMD requires that operations be the same and in that sense VLIW provides more flexibility. Think of an algorithm that works better on VLIW than SIMD.

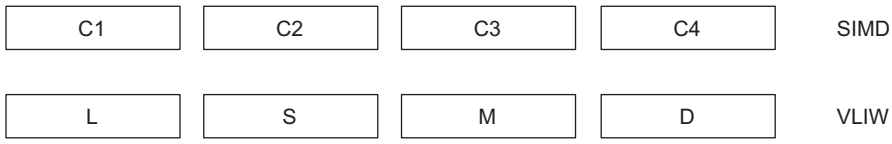

Figure 5. Comparison VLIW Versus SIMD Hardware

2.3 DSP Architecture Details

Here is a more detailed architecture abstraction of DSP. It has four parallel VLIW units: L,S,M and D each performing a specialized functionality. It is replicated twice as called by sides A and B; each side is tightly coupled to its own set of 16 32- bit registers. The functionality of each unit is described in Table 1. Some operations are more likely to occur in practice relative to others, for example, addition and subtraction occur more frequently so this is replicated across L, S and D units. Similar logical operations are replicated in L and S. Each instruction will be executed in one of these eight units. The compiler will try to find an optimal instruction mapping on these units. It has more options for adds and less options for LD, ST, and multiply.

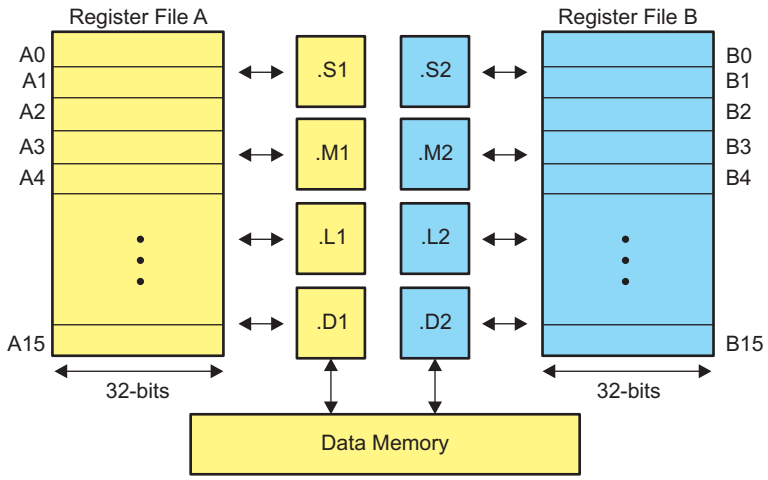

Figure 6. C6x DSP Architecture

Table 1. DSP Functional Units

C64x+ Functional Units			
L	**S**	**D**	**M**
Integer Adder	Integer Adder	Integer Adder	Integer Multiplier
Logical	Logical	Load-Store	
Integer Comparison	Shifting		
Bit Counting	Bit Manipulation		
	Constant		
	Branch and Control		
	Dual 16-Bit Math		

Table 2. Instruction Mapping to Units

	Functional Units			
Instruction	**.L Units**	**.M Units**	**.S Units**	**.D Units**
ABS	√			
ABS2	√			
ADD	√		√	√
ADDAB				√
ADDAD				√
ADDAH				√
ADDAW				√
ADDK			√	
ADDKPC			√	
ADDSUB	√			

Table 2. Instruction Mapping to Units (continued)

	Functional Units			
Instruction	**.L Units**	**.M Units**	**.S Units**	**.D Units**
ADDSUB2	√			
ADDU	√			
ADD2	√		√	√

3 Optimized DSP Libraries

There is a lot of software infrastructure and libraries optimized for DSP. These libraries have been successfully used by several customers, which saves a lot of development effort. They can also be used for learning tips and tricks of optimization. Computational intensive blocks are identified and optimized via software pipelining, C6x intrinsics, and SIMD processing. These optimized kernels are memory agnostic and require efficient memory management using DMA, which is discussed in a later section. All libraries are free of charge and royalty. Except for VLIB, all others are provided in source form.

3.1 Vision Library (VLIB)

VLIB consists of 65 optimized kernels provided as object code. Customers can build their own applications adding their own secret sauce using these low-level building blocks. Figure 7 shows a possible application that can be implemented by VLIB kernels. It comes with detailed application programming interface (API) documentation that explains the functionality and API of all the kernels, (see the *Vision Library (VLIB) Application Programming Interface Reference Guide* (SPRUG00)) and the test and example functions along with the Code Composer Studio™ projects for several system-on-chips (SoCs). The bit-exact PC version of the library as well as corresponding Matlab-Simulink blocks are also provided. The following contains the list of VLIB functions sorted according to category.

- Background modeling and subtraction
  - Luminance extraction from YUV:422
  - Exponentially-weighted running mean and variance
  - Uniformly-weighted running mean and variance
  - Statistical background subtraction
  - Mixture of Gaussians background modeling and subtraction
  - Morphological operations (erosion and dilation)
  - Connected components labeling
- Feature extraction
  - Harris corner store (7x7)
  - Hough transform for lines
  - Histogram computation for integer scalars
  - Histogram computation for multi-dimensional vectors
  - Weighted histogram computation for integer scalars
  - Weighted histogram computation for multi-dimensional vectors
  - Legendre moments
  - Canny edge detection
    - Smoothing
    - Gradient computation
    - Non-maximum suppression
    - Hysteresis

- Low-level pixel processing
  - Color conversion YUV:422 interleaved to
    - YUV planar
    - RGB
    - LAB
    - HSI
  - Integral image
  - Image pyramid (2x2 block averaging)
  - Non-maximum suppression (3x3, 5x5, and 7x7)
  - Gradient image pyramid (5-tap)
  - Gaussian image pyramid (5-tap)
  - First-order recursive IIR filters (horizontal and vertical)
  - SAD-based disparity for stereo
- Tracking, recognition, and so forth
  - Lucas-Kanade feature tracking (7x7)
  - Kalman filtering
  - Nelder-Mead simplex optimization
  - Bhattacharya distance

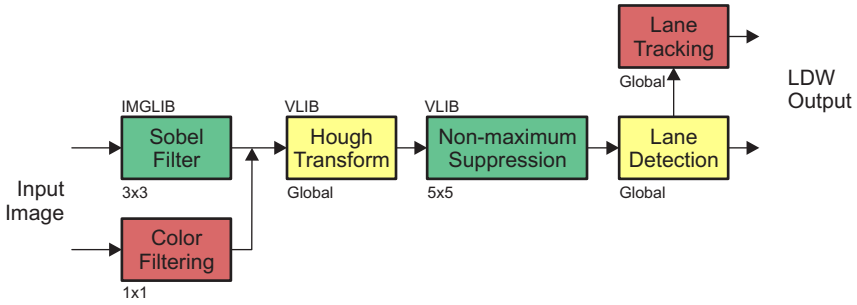

Figure 7. LDW Implementation Using VLIB

3.2 Image and Video Processing Library (IMGLIB)

The image library (IMGLIB) contains commonly used image and video processing routines filtering, histograms, and so forth. The rich set of software routines included in the IMGLIB is organized into three different functional categories as follows: compression and decompression, image analysis, and picture filtering and format conversions. In addition, a set of 22 low-level kernels are provided which perform simple image operations such as addition, subtraction, multiplication, and so forth and are intended to be used as a starting point for developing more complex kernels. All kernels are provided in source form.

- IMGLIB Image Analysis Functions Overview
  - Boundary and Perimeter Functions
  - Dilation and Erosion Operation Functions
  - Edge Detection Function
  - Histogram Function
  - Image Threshold Function
- IMGLIB Picture Filtering Functions Overview
  - Color Space Conversion Functions
  - Convolution Function
  - Correlation Functions

  - Error Diffusion Functions
  - Median Filtering Function
  - Pixel Expand Functions
- Compression and Decompression Functions Overview
  - Forward and Inverse DCT Functions
  - High Performance Motion Estimation Functions
  - MPEG-2 Variable Length Decoding Functions
  - Quantization Functions
  - Wavelet Processing Functions

3.3 DSPLIB, MathLIB and IQMath

The DSPLIB consists of 1D signal processing functions such as filtering, FFT, matrix operations and math operations.

The MathLIB is optimized floating-point math functions such as sqrt, log, exp and trigonometric.

IQmath contains optimized math functions in Q-point fixed point representation using LUTs. As this is fixed point, it is faster than MathLIB. The MathLIB is optimized floating-point math functions such as sqrt, log, exp and trigonometric for fixed-point and floating-point processors.

4 Profiling and Identifying Hot Spots

4.1 Concept of Loop and Kernel

Image processing applications typically consists of several loops. Loops are the intensive portions of the code that are focused on here. The set of operations inside a loop are called kernels. The optimization steps can be split as loop optimization and kernel optimization. Nested loops are common as well; it is important to remember only the innermost loop pipelines. There is also pipe up and pipe down overheads. Figure 8 has a visualization of this concept where the operations (kernels) are repeated over several windows.

```
//FUNCTION 1
FOR J = 1: M
     FOR I = 1: N  ----------------------> Only innermost loop pipelines on the DSP
     {
          OPERATIONS;
     }

//FUNCTION 2
FOR K = 1: P
     FOR L = 1: Q
     {
          OPERATIONS;
     }
```

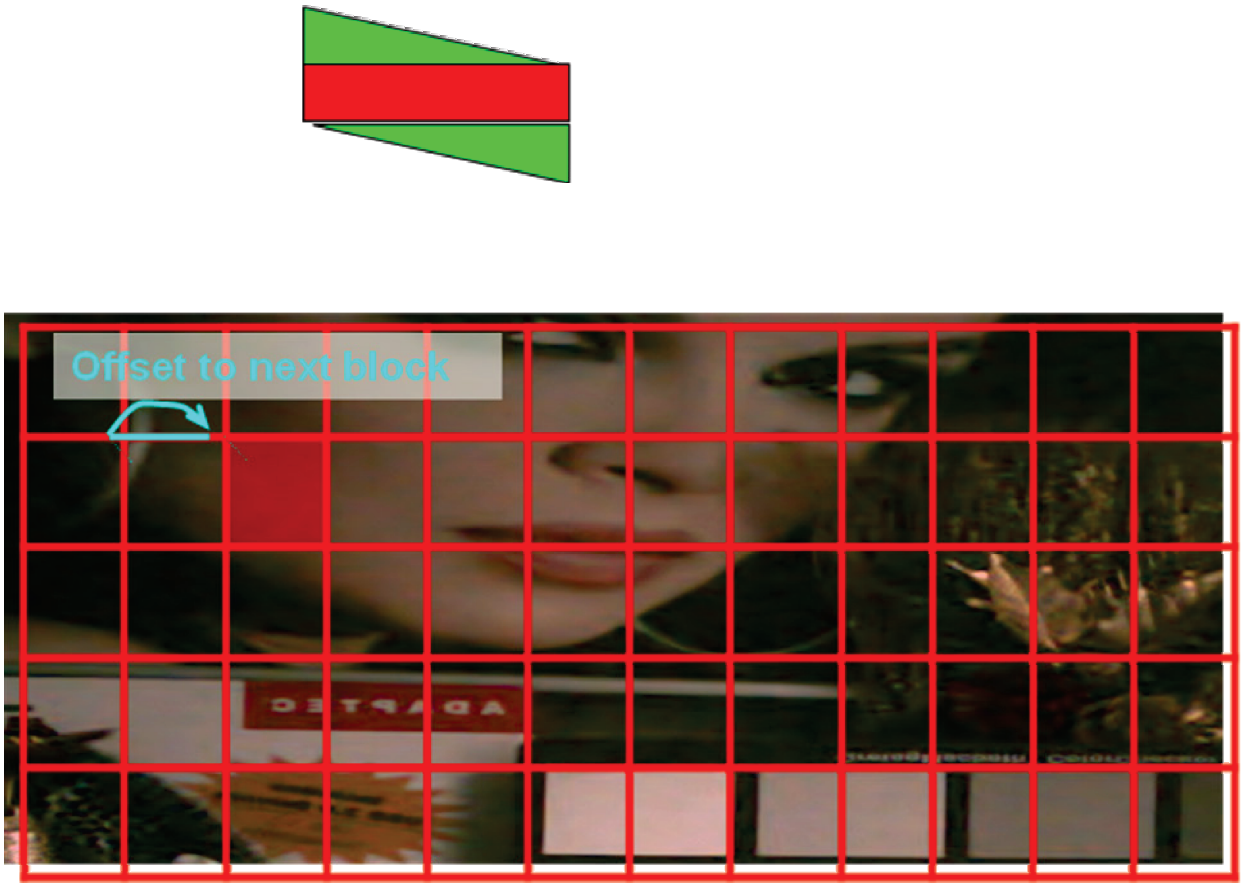

Figure 8. Loop and Kernel Illustration

4.2 Preparing Estimates

Preparation of estimates is often useful for finding the bottlenecks and also to get a rough idea of the level of achievable optimization. It is basically mapping of operations in a kernel to the eight units. The number of cycles can be looked up in the Instruction set guide. Most instructions take 1 cycle, whereas, more complex operations, like mpy, may take more cycles. There are several other things to consider like delay slot, cross paths, and so forth. Delay slot refers to the number of cycles a hardware unit is blocked until it can execute another operation. Cross path delays happen when an A side register is accessed by a B side unit. Without considering these, the estimate can act as a lower bound for actual performance.

A simple example is provided here for illustration. This function requires two loads to load m and n, 1 mpy and 1 loop branching. Table 3 shows how these instructions are mapped to units. Note that ADD and SUB can map to any of the units L, S and D, and appropriate units that are free can be chosen. Finally, you can see which unit is occupied the most and that becomes the bottleneck.

```
for (I=0; I < count; I++)
  {
    product = m[i] * n[i];
    sum += product;
  }
```

Table 3. Kernel Instructions Mapping Onto Computation Units

Instructions	Unit	Number Available (per cycle)	Number Required
LDH, LDH	.D	2	2
MPY	.M	2	1
B	.S	2	1
ADD, SUB	.L (.L, .D, .S)	2(2-6)	2

4.3 Benchmarking

Benchmarking is often the first step in optimization. The '-O3' flag would suggest the compiler to optimize aggressively and it should be added in the build flags to leverage the optimization by the compiler. It is good to have an idea how much improvement can be achieved. Sometimes the current performance might be good enough. A recommended benchmarking scheme is provided here. The simulator ignores memory overheads and only models the computation units. It is useful to measure this number and then look at the memory and system overheads. Comparison with a rough number of operations provides the amount of parallelization. The parallelization factor can be up to 32 (4 because of SIMD and 8 because of VLIW). The performance estimates discussed above can optionally be used to validate the optimization.

Table 4. Benchmarking Table

Algorithm	Number of Ops, Iterations Scalar Core Performance	DSP SIM Benchmark	Parallelization Factor (1-64)	EVM Benchmark	System Overhead Factor
Sobel Filter	X	Y	X and Y	Z	Z and Y
Edge Histogram					
...					
...					

The time stamp counter can be used to measure the execution time of a section of code. The time stamp counter is a free-running 64-bit counter that is normally incremented during each CPU cycle and, therefore, it is more accurate than the operating system services. The time stamp counter is accessed through the Time Stamp Counter (Low) (TSCL) and Time Stamp Counter (High) (TSCH) read-only registers. The TSCL register returns the 32 LSBs of the time stamp counter, the TSCH register returns the 32 MSBs of it. The order is important: to get a consistent 64-bit value, you have to read TSCL register before the TSCH register. As the counter is initially disabled after reset, you have to first enable it by writing to the TSCL register. The actual value does not matter, it is ignored anyway. After that, counting begins and can only be stopped by resetting or powering down the CPU. The following C code illustrates how to use the timing registers. This works on the simulator and the actual hardware.

```
#include <c6x.h> // _itoll, TSCH, TSCL
uint64_t start_time, end_time;
start_time = _itoll(TSCH, TSCL);
/* your code section to profile */
end_time = _itoll(TSCH, TSCL);
printf("Your code section took: %lld cycles\n", end_time - start_time);
```

5 Memory Optimization

5.1 Memory Hierarchy

DSP architecture uses a hierarchical memory with two on-chip memories L1 and L2 and an external memory like DDR. There is a trade-off between cost and speed, so the faster expensive memories are smaller in size. Figure 9 illustrates a typical hierarchical memory in a C64x DSP. Data typically resides in the external memory that is 6 times slower than the CPU, but ideally the data can be read at the same speed as the CPU because of L1. There are two modes of achieving the same in the hardware and they are discussed below.

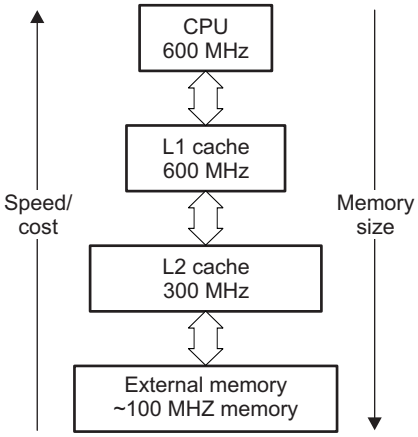

Figure 9. Memory Hierarchy in C64x DSP

The internal memories L1 and L2 can be either configured as cache or addressable internal memory. Table 5 lists the sizes of each memory and possible cache configurations. For more details, see the *TMS320C6000 DSP Cache User's Guide* (SPRU656).

Table 5. L1 and L2 Cache Characteristics

L1D Cache Characteristics	
Characteristics	**C674x DSP**
Organization	2-way set associative
Protocol	Read allocate, write back
CPU Access Time	1 cycle
Capacity	4K, 8K, 16K, or 32K bytes
Line Size	64 bytes
Replacement Strategy	Least recently used (LRU)
Write Buffer	4 x 128-bit entries
External Memory Caches	Configurable
L1P Cache Characteristics	
Organization	Direct mapped
Protocol	Read allocate
CPU Access Time	1 cycle
Capacity	4K, 8K, 16K, or 32K bytes
Line Size	32 bytes
External Memory Caches	Always cached
L2 Cache Characteristics	
Organization	4-way set associative
Protocol	Read and write allocate, Writeback
Capacity	4K, 8K, 16K, or 32K bytes
Line Size	128 bytes

Table 5. L1 and L2 Cache Characteristics (continued)

L1D Cache Characteristics	
Characteristics	**C674x DSP**
Replacement Strategy	Least recently used (LRU)
External Memory Caches	Always cached

5.2 Cache Optimization

The cache controller hardware handles most of the aspects; the only thing you need to take care of is configuring the size of the cache. It is hard to recommend an optical cache size without considering all the aspects of an application. Ideally, you would need all the internal memory as cache but sometimes you would need some internal memory for commonly accessed data like LUTs, and so forth. The sizes of cache can be set by MMRs, as shown below. This can be added in the main C source file. For more details, see the *TMS320C674x DSP Megamodule Reference Guide* (SPRUFK5). There are various trace and event trigger capabilities available on various targets that provides cache hits, misses, and stall information. For more details, see the *TMS320C6000 DSP Cache User's Guide* (SPRU656).

```
#define L1PCFG  *(unsigned int*)0x01840020
#define L1DCFG  *(unsigned int*)0x01840040
#define L2CFG   *(unsigned int*)0x01840000
#define MAR128  *(unsigned int*)0x01848200
L1PCFG |= 0x00000004;
L1DCFG |= 0x00000004;
L2CFG  |= 0x00000003;
MAR128 |= 0x00000001;
```

5.3 EDMA

The advantage of cache is that it is very easy to setup. But EDMA typically offers much better performance typically around 4X - 6X. But integrating EDMA requires design of block-based algorithm and the understanding of EDMA drivers. EDMA is an independent hardware that can copy data from DDR to local memory of DSP while DSP is performing computations, thus, hiding DDR overheads. Figure 10 illustrates how this is done efficiently in a ping-pong buffering scheme. The input and output buffers are split into ping and pong namely. While data is copied to ping buffer in0 by DMA, DSP executes functions on in1 data. Once this is done, the processing is done on pong buffer in1, and data is brought in by DMA to in0.

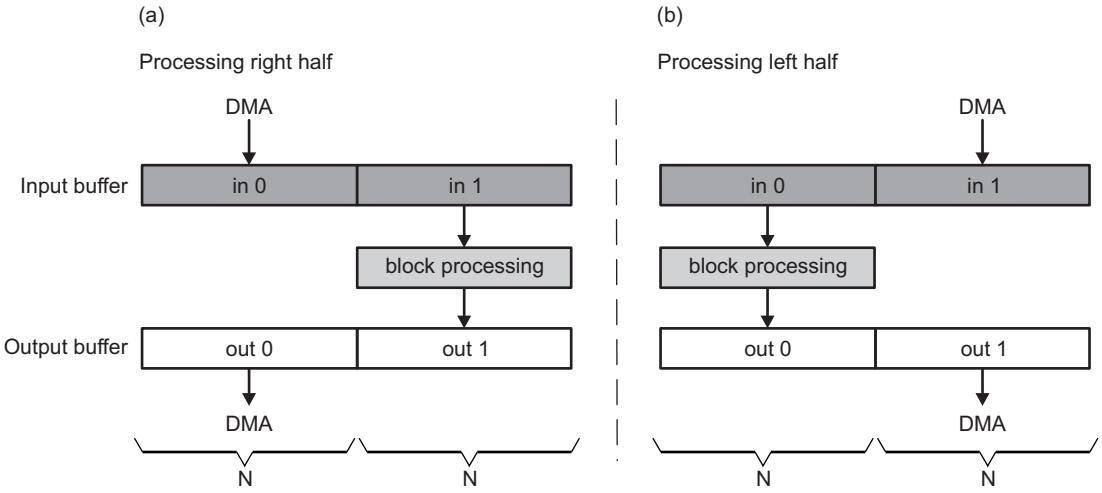

Figure 10. Ping-Pong Buffering

6 Kernel Level Optimization

This section contains techniques for optimization of the operations in the kernel.

6.1 Operation Simplification

The general idea of this rule is to minimize the number of operations in an expression. Multiplications are more expensive than additions. The compiler does not attempt such simplifications.

For example: ac + ad + bc + bd → (a + b)(c + d)

Horner's rule is a specific popular method for minimizing the number of operations in polynomials. It recursively pulls out common factors to reduce the number of multiplications from O (n^2) to O(n) (where n is the order of the polynomial). It also provides better numerical stability.

For example: d + c*x^2 + b*x^4 + a*x^6 → d + x^2*(c+x^2*(b +a*x^2))

In case of expressions especially the ones involving comparisons, it is often possible to move complex operations to known constants instead of variables. In the example shown below, the operation sqrt is moved from a variable 'var' to a constant 'const' so that the operation on 'const' can be pre-computed.

For example: val = sqrt(var); val > const → var>const^2

In some expressions, it might be possible to remove some intermediate computations or simplify the expression. In the example below, M + N >0 can be simplified to a+b>0 without needing to compute the sqrt operation.

For example: M = a/sqrt(x+y); N = b/sqrt(x+y); M + N >0

a/sqrt(x+y) + b/sqrt(x+y) > 0 → a+b >0

6.2 Calculations Reuse Across Iterations

Typically for windowed pixel operations, there is a lot of overlap between successive computations. In Figure 11, a sum is computed over a moving WxW window. This kind of operation is present in Harris Score and Stereo Computation. Block A corresponds to the data for the first iteration and block B corresponds to the next iteration. It is clear that there is a lot of overlap in the computation. Similarly, there is an overlap in the vertical direction. Instead of WxW operations per output pixel, a more efficient approach by re-using the previous iterations would reduce to four operations. Similar re-use is possible even in a weighted sum operation or a non-linear operation like median filter.

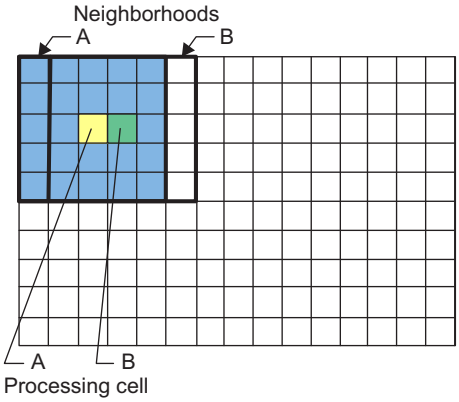

Figure 11. Illustration of Redundant Operations Across Iterations

6.3 Avoiding Control Code

Conditional code flow prevents the code to be pipelined. Therefore, it is better to avoid conditional code inside a loop wherever possible. Fortunately all instructions of the DSP can be conditional and simple if-cases are fine. But more complex conditional structures like nested if-else, switch-case, break, and function calls break the pipeline. In this case, the compiler provides feedback in the assembly file “loop contains control code”. It is advised to re-write nested if-else and switch case in a more linear way. A common problem is to use function calls for standard C functions like sqrt, sin, and so forth. It is easy to miss that they are function calls and they break pipelining. In this case, these functions have to be made inline.

6.4 Fixed-Point Design – IQmath

Fixed point algorithms are typically faster than an equivalent floating point version even on a floating-point processor. On a floating-point processor like C674x, here is the non-pipelined number of cycles for each data type. Therefore, the performance improvement can be around 4-8X. In case of a fixed-point processor the improvements can be around 30X.

- 8-bit fixed-point data – 4 operations in 1 cycle
- 16-bit fixed-point data – 2 operations in 1 cycle
- 32-bit fixed-point data – 1 operation in 1 cycle
- 32-bit floating point (SP) – 1 operation in 4 cycles
- 64-bit floating point (DP) – 1 operation in 7 cycles

Floating-point numbers can be represented in fixed point by using Q-point arithmetic **[ref????]**. Q-point has a lower resolution and dynamic range compared to floating point, but it might be sufficient in some use cases. If not, floating point has to be used. For example, Q8.7 refers to a fractional fixed-point number with 8 integer bits and 7 fractional bits. TI provides an IQmath library that contains Q-point arithmetic functions for floating-point math algorithms like sin, cos, mpy, div, and so forth.

- Division: x/3 → (x/3*2^32/2^32) → x* [2^32/3]>>32
- Multiply: x*0.05 → Qc = round(2^8*0.05); (x*Qc)>>8

6.5 Look-up Table (LUT)

Complex arithmetic functions like sine, sqrt and division can be replaced by a LUT to reduce computations. The disadvantages are that it requires additional memory. If the argument is float, then it is not possible to create a large LUT, but the float can be quantized and a relatively smaller LUT can be created. Sometimes it is a trade-off between accuracy and performance. LUT can also be used for pre-computing a set of operations that are known before run-time. The example below that is part of HOG features illustrates this:

```
Imgs_orien(Imgs_orien >= 160) = 9;

Imgs_orien(Imgs_orien >=  40) = 3;
Imgs_orien(Imgs_orien >=  20) = 2;

Orientation[360]=[1,1,1,…2,2,2,…3,3,3…]
```

6.6 Array of Structures (AOS) vs Structure of Arrays (SOA)

Array of structures (AOS) is a commonly used data structure in image processing. For example, most of the data structures in OpenCV are AOS. But SOA are typically more efficient on a DSP for the following reasons:

- It allows better SIMD because the data elements are sequential in memory
- It is easy to align the array pointers
- It is easier to manage memory when only some elements of the array are needed

```
AOS:                          SOA:
struct                        struct
{                             {
int x;                        int *x;
int y;                        int *y;
char edge;                    char *edge;
}POS[100];                    }POS;
AOS → x1y1e1x2y2e2….
SOA → x1x2……y1y2…..e1e2….
```

6.7 Pointer Aliasing

Pointer aliasing refers to the usage of different pointers for the same memory location. In image processing, it is common to use pointer aliases for different rows. For instance, consider a 3x3 kernel, pRow1, pRow2 and pRow3 can be used to access 3 rows. Instead of pRow2, pRow1+ width can be used to avoid aliasing. The reason for doing this is that the following DSP parallelizes the algorithm by running different iterations of the loop at the same time (pipelining). So, pRow3 of one iteration and pRow2 of another iteration can access the same memory location and create a conflict. There is a flag (-mt) that tells the compiler to assume there is no pointer aliasing so that it will optimize more aggressively.

6.8 Other General Programming Guidelines

- Some of the C++ constructs like templates overloading are often inefficient for optimal code generation for the DSP. Performance degradation of around 5X can arise due to such C++ constructs; it is recommended to avoid them in the case of intensive loops.
- Loads and stores are faster typically when the addresses are aligned to a 32-bit address boundary because of the architecture aspects. It is good to align the array addresses using #pragma DATA_ALIGN.
- It is recommended to use the minimum required data-type as the performance because it is better as the data-size goes down. Standard software, like OpenCV and Matlab, use float or double values liberally even where integer data type would suffice without loss of accuracy.
- Although it is a good programming practice to write generic code, specific code can lead to better optimization. For example, instead of writing a generic 3x3 filter and passing coefficients as parameters for a Sobel filter, it is better to incorporate the co-efficients and exploit sparsity and symmetry.

7 Loop Transformations

Techniques to modify loops to enable better pipelining are discussed in this section.

7.1 Loop Merging

It is a common technique in image processing to combine two loops for the row and column into one. This technique is employed in several IMGLIB kernels. Most of the time the kernel is uniform across all the iterations. So, it becomes easy to merge the loops. Sometimes, there is some redundancy at the corners where invalid computations are done. However, it is more efficient to do these computations instead of having two loops. There are several advantages of this technique. There is a loop overhead in the inner most loop and this is multiplied by the number of times the outer loop is executed. So, it becomes significant and it is eliminated when the loops are merged. Also the pointer calculations become simpler and usually are handled by hardware in the latter case using auto-increment.

```
for I=1:row                       for I=1:row*col
     for J=1:col                  {
     {
     …                            }
     }
```

7.2 Collapsing Small Inner Loops

Shown below on the left is a common implementation of kernels in image processing. The block_width and block_height are typically small, of the order of 3(sobel 3x3 filter) to 7(harris corner). As the innermost loops are very small, it does not pipeline efficiently. In this case, it is efficient to remove the loop and write the operations explicitly. This brings the col loop as the innermost loop and it is more efficient.

```
for I=1:row                                  for I=1:row
     for J=1:col                             for J=1:col
     for r=1:block_width                     {
     for c=1:block_heigh
          { ….}                              }
```

7.3 Loop Unrolling

It is also useful sometimes for achieving better load balancing. For example, the kernel might have 1 add and 1 multiply. As there are two units, one of the units is left unused. A more efficient approach is to combine the two successive iterations so that both A and B units are loaded. The compiler does this automatically most of the time and it can also be done manually using #unroll pragmas.

```
for I=1:row                          for I=1:row
    for J=1:col                      for J=1:col/2
            A(i,j) = B(i,j)*C(i,j);          A(i,j) = B(i,j)*C(i,j);
            A(i,j) = A(i,j)+ 23;             A(i,j) = A(i,j)+ 23;
                                             A(i,j+1) = B(i,j+1)*C(i,j+1);
                                             A(i,j+1) = A(i,j+1)+ 23;
```

7.4 Loop Fusion

Sometimes, it is more efficient to combine two loops together because the epilog and prolog occur only once for both the loops together instead of occurring once for each loop, as shown in Figure 12. This combination can also bring in new possibilities of simplifying operations across the two loops. This method is particularly useful when the kernels are small. In this case, the pipeline overhead becomes significant and the set of operations might not have an efficient mapping to all the DSP units.

```
for (I = 0; I < 100; I++)
a[i] = b[i]+c;
for (I = 0; I < 100; I++)
d[i] = a[i+1]+e;

for (I = 0; I < 100; I++)
{
a[i] = b[i]+c;
d[i] = a[i+1]+e;
}
```

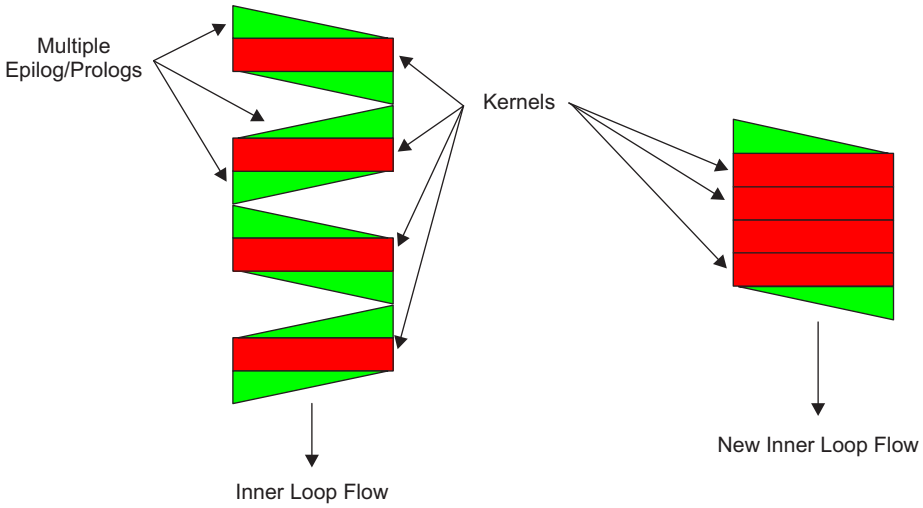

Figure 12. Illustration of Loop Merging

7.5 Loop Fission

Loop fission is the opposite of the loop fusion. From previous optimization, it might seem that loop fission reduces performance, however, it is useful sometimes. Sometimes, the kernel might be very large and run out of the registers required for the operation, which is called register pressure in compiler terminology. If this happens, the loop pipeline breaks and is notified in the loop pipeline information. Then, the loop has to be split into two. In some cases, the loop might have dependencies that could lead to inefficient pipelining. In the example given below, the value of A depends on its value in the previous iteration; this is called dependency. It might be more efficient to move the portions of code with dependency to another loop where the other portions can be pipelined more efficiently.

```
FOR I = 1: N                                  FOR I = 1: N
     FOR J = 1: M                               FOR J = 1: M
         A(I,J+1) = A(I,J) + C                      B(I+1,J) = B(I,J) + D
         B(I+1,J) = B(I,J) + D
                                                       FOR I = 1: N
                                                         For J = 1: M
                                                            A(I,J+1) = A(I,J) + C
```

7.6 Loop Fission Nested Loop Interchange

Sometimes it might be useful to change the order of nesting. In the first example below, interchange helps in removing the dependency. In the second example, the interchange results in better cache performance because two arrays are accessed sequentially compared to the one in the former version.

```
for I=1:m                                     for j=1:n
    for j=1:n                                     for I=1:m
        A(i,j+1) = A(i,j) + B                         A(i,j+1) = A(i,j) + B

for I=1:n                                     for k=1:n
    for j=1:n                                     for I=1:n
        for k=1:n                                     for j=1:n
            A(i,j) = A(i,j) + B(i,k)*C(k,j)               A(i,j) = A(i,j) + B(i,k)*C(k,j)
```

7.7 Combination of Loop Techniques

The loop techniques are discussed individually with simple examples. It is important to note that the real scenario is very complex and requires a combination of several techniques, and usually experience trial and error. It is also important to keep the SIMD optimization in mind while performing the transformations because it might inhibit SIMD. Although it is difficult to consider all combinations, usually with experience one becomes more intuitive on how to apply these techniques.

```
FOR  I = 1: N
   FOR  J = 1: M
       A(I,J) = A(I,J) + X
       B(I+1,J) = A(I,J) + B(I,J)
       C(I,J+1) = A(I,J) + C(I,J)
       D(I+1,J) = B(I+1,J) + C(I,J) + D(I,J)
```

8 DSP Specific Optimizations

After performing algorithmic optimization, the last step is to perform DSP architecture specific optimization. This can be done by examining the generated ASM file. Each loop has its own software pipeline information (SPLOOP).

8.1 SIMD

Check the SPLOOP to see if the appropriate SIMD instructions, like _add4 and _mem8, are automatically inserted by the compiler. If not, the appropriate SIMD intrinsics have to be manually inserted in C code.

8.2 Special Instructions

DSP has some special instructions that can do multiple operations in a single instruction. The commonly used ones are Subabs, Addsub, Avg, Dotp4, and Min and max. For instance, dotp4 computes dot-product between 4 bytes. Otherwise, to accomplish the same, it requires more than six instructions. The compiler does not usually recognize this and it has to manually be inserted as an intrinsic.

8.3 Load Balancing

The software pipeline information shows the distribution of the instruction in the A and B side of the L, S, D, and M. In this case, the D unit has the maximum instructions mapped to it, so it is the bottleneck. To improve the performance, try to see if there is a possibility to move the instructions to another unit. For example, the common way of swapping two variables is using a temp variable and load and stores using D unit. But, if D unit is the bottleneck, it can be implemented in an alternate way using XOR, in which case the instructions are mapped to the L unit.

```
define swap (x, y)   |    define swap (x, y)
  temp := x          |      X := X XOR Y
  x := y             |      Y := Y XOR X
  y := temp          |      X := X XOR Y
```

Another example would be the computation of y = 2*x, as shown below.

```
y = 2*x → M unit
y = x+x → L or S unit
y = x <<1 →L unit
```

```
;*   SOFTWARE PIPELINE INFORMATION
;*
;*      Loop source line                 : 5
;*      Loop opening brace source line   : 5
;*      Loop closing brace source line   : 6
;*      Known Minimum Trip Count         : 1
;*      Known Max Trip Count Factor      : 1
;*      Loop Carried Dependency Bound(^) : 7
;*      Unpartitioned Resource Bound     : 2
;*      Partitioned Resource Bound(*)    : 2
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                     0        0
;*      .S units                     0        1
;*      .D units                     2*       1
;*      .M units                     0        0
;*      .X cross paths               1        0
;*      .T address paths             2*       1
;*      Long read paths              0        0
;*      Long write paths             0        0
;*      Logical  ops (.LS)           0        0     (.L or .S unit)
;*      Addition ops (.LSD)          1        0     (.L or .S or .D unit)
;*      Bound(.L .S .LS)             0        1
;*      Bound(.L .S .D .LS .LSD)     1        1
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 7  Schedule found with 1 iterations in parallel
...
;*        SINGLE SCHEDULED ITERATION
;*
;*        C25:
;*   0               LDW     .D1T1   *A4++,A3          ; |6|  ^
;*     ||            LDW     .D2T2   *B4++,B5          ; |6|  ^
;*   1       [ B0]   BDEC    .S2     C24,B0            ; |5|
;*   2               NOP             3
;*   5               ADD     .L1X    B5,A3,A3          ; |6|  ^
;*   6               STW     .D1T1   A3,*A5++          ; |6|  ^
;*   7               ; BRANCHCC OCCURS {C25}           ; |5|
```

Figure 13. Generated ASM File

9 Summary

The summary of the optimization techniques are shown below:

1. Set all of the compiler optimization flags and the profile using the latest version of cgtools. Sometimes the optimization performed by the compiler might be good enough.
2. Enable cache to improve memory overheads. Use EDMA to improve performance further.
3. Look for an existing function in TI's optimized libraries like fastRTS, DSPLIB, IMGLIB and VLIB and use it.
4. Understand the whole algorithm. A high-level visualization of the computations and data flow will help in identify bottlenecks.
5. Perform C level optimization using kernel and loop level optimization techniques.
6. Look at the compiler feedback in the ASM file and use intrinsic level optimization for SIMD and load re-balancing.
7. Steps 5 and 6 might impact or conflict with each other; typically through several iterations an optimal combination is found.

10 References

- *Introduction to TMS320C6000 DSP Optimization* (SPRABF2)
- *Vision Library (VLIB) Application Programming Interface Reference Guide* (SPRUG00)
- *Canny Edge Detection Implementation on TMS320C64x/64x+ Using VLIB* (SPRAB78)
- *Hand-Tuning Loops and Control Code on the TMS320C6000* (SPRA666)
- Real-Time DSP Implementation of Pedestrian Detection Algorithm Using HOG Features - International Conference on Intelligent Transportation Systems Telecommunications, 2012.
- *TMS320C6000 Optimizing C Compiler Tutorial* (SPRU425)
- *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide* (SPRU732)
- Textbook - *Performance Optimization of Numerically intensive Codes*
- *TMS320C674x DSP Megamodule Reference Guide* (SPRUFK5)
- Video Lecture: *Optimizing Your C Code for Performance on TMS320C6000 DSPs*: http://learningmedia.ti.com/public/media/DSP8-58/01/21972/index.html
- *TMS320C6000 DSP Optimization Workshop*: http://processors.wiki.ti.com/index.php/TMS320C6000_DSP_Optimization_Workshop
- *How to Write Fast Numerical Code*: http://www.inf.ethz.ch/personal/markusp/teaching/263-2300-ETH-spring12/course.html
- *Programming the EDMA3 Using the Low-Level Driver (LLD)*: http://processors.wiki.ti.com/index.php/Programming_the_EDMA3_using_the_Low-Level_Driver_%28LLD%29
- *TMS320C6000 DSP Cache User's Guide* (SPRU656)

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions of TI 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components which meet ISO/TS16949 requirements, mainly for automotive use. Components which have not been so designated are neither designed nor intended for automotive use; and TI will not be responsible for any failure of such components to meet such requirements.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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