Video Filtering Considerations for Media Processors

By David Katz and Rick Gentile, Blackfin Applications Group, Analog Devices, Inc.

Until recently, designers needing to perform video or image analysis in real time, a typical requirement of medical, industrial and military systems, had to resort to expensive specialized processors. With the advent of fixed-point, high-performance embedded media processors, however, it has become possible to do image processing economically in real time. To develop truly efficient algorithms, it is essential for designers to take advantage of the architectural features these processors provide. This article discusses how digital image filtering algorithms can leverage the multimedia-friendly features of an embedded media processor's architecture. The Blackfin processor's features and instruction set are used as a reference point, but the same concepts apply to high-performance media processors in general.

Although the clock speeds of fixed-point processors now reach beyond 300 MHz, this speed increase alone doesn't guarantee the ability to accommodate real-time video filtering. Just as important are multimedia-geared architectural features and video-specific instructions.

Most video applications need to deal with 8-bit data, since individual pixel components (whether RGB or YUV) are usually byte quantities. Therefore, 8-bit video ALUs and byte-based address generation can make a huge difference in manipulating pixels. This is a nontrivial point, because DSPs typically operate on 16-bit or 32-bit boundaries.

Another complementary feature is a flexible data register file. In traditional fixed-point DSPs, word sizes are usually fixed. However, there is an advantage to having data registers that can be treated as either a 32-bit word (e.g., R0) or 2 16-bit words (R0.L and R0.H, for the low and high halves, respectively). The utility of this structure will become apparent below.

Dedicated single-cycle instructions can be very convenient for providing efficient multimedia coding algorithms. A good example of this is a "Sum of Absolute Differences" instruction that can add up differences between several pixel sets simultaneously, indicating how much a scene has changed between frames.

Two-dimensional Image Convolution

Since a video stream is really an image sequence moving at a specified rate, image filters need to operate fast enough to keep up with the succession of input images. Thus, it is imperative that image filter kernels be optimized for execution in the lowest possible number of processor cycles. This can be illustrated by examining a simple image filter set based on two-dimensional convolution.

Convolution is one of the fundamental operations in image processing. In a two-dimensional convolution, the calculation performed for a given pixel is a weighted sum of intensity values from pixels in its immediate neighborhood. Since the neighborhood of a mask is centered on a given pixel, the mask usually has odd dimensions. The mask size is typically small relative to the image, and a 3x3 mask is a common choice, because it is computationally reasonable on a per-pixel basis, but large enough to detect edges in an image.

The basic structure of the 3x3 kernel is shown in Figure 1a. As an example, the output of the convolution process for a pixel at row 20, column 10 in an image would be:

Out(20,10)=A(19,9)+B*(19,10)+C*(19,11)+D*(20,9)+E*(20,10)+F*(20,11)+G*(21,9)+H*(21,10)+I*(21,11)*

A	B	C
D	E	F
G	H	I

(a) 3x3 Convolution Kernel

0	0	0
0	1	0
0	0	0

(b) Delta

-1	0	1
-1	0	1
-1	0	1

1	1	1
0	0	0
-1	-1	-1

(c) Edge Detection

1/8	1/8	1/8
1/8	0	1/8
1/8	1/8	1/8

(d) Smoothing

-1/8	-1/8	-1/8
-1/8	1	-1/8
-1/8	-1/8	-1/8

(e) Edge Enhancement

Figure 1: The 3x3 convolution mask and how it can be used

It is important to choose coefficients in a way that aids computation. For instance, scale factors that are powers of 2 (including fractions) are preferred because multiplications can then be replaced by simple shift operations.

Figures 1b-1e show several useful 3x3 kernels, each of which is explained briefly below.

The Delta Function shown in Figure 1b is among the simplest image manipulations, passing the current pixel through without modification.

Figure 1c shows 2 popular forms of an edge detection mask. The first one detects vertical edges, while the second one detects horizontal edges. High output values correspond to higher degrees of edge presence.

The kernel in Figure 1d is a smoothing filter. It performs an average of the 8 surrounding pixels and places the result at the current pixel location. This has the result of “smoothing,” or low-pass filtering, the image.

The filter in Figure 1e is known as an “unsharp masking” operator. It can be considered as producing an edge-enhanced image by subtracting from the current pixel a smoothed version of itself (constructed by averaging the 8 surrounding pixels).

Performing Image Convolution on an Embedded Media Processor

Let’s take a closer look at the two-dimensional convolution process. The high-level algorithm can be described by the following steps:

1. Place the center of the mask over an element of the input matrix.
2. Multiply each pixel in the mask neighborhood by the corresponding filter mask element.
3. Sum each of the multiplies into a single result.
4. Place each sum in a location corresponding to the center of the mask in the output matrix

Figure 2 shows three matrices: an input matrix f(x,y), a 3x3 mask matrix h(x,y), and an output matrix g(x,y).

$$g(x,y) = \sum_{i=-1}^{i=1} \sum_{k=-1}^{k=1} h(i,k) f(x-i, y-k)$$

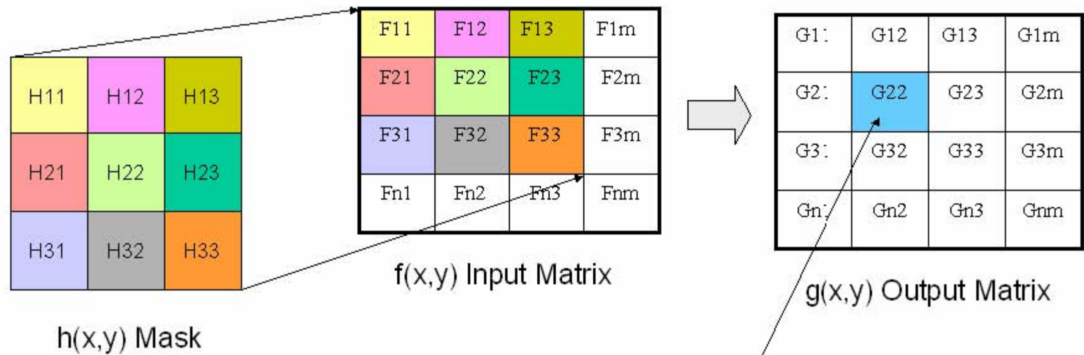

Figure 2: Operation of the 3x3 convolution mask on an image

After each output point is computed, the mask is moved to the right by one element. On the image edges, the algorithm wraps around to the first element in the next row. For example, when the mask is centered on element F2m, the H23 element of the mask matrix is multiplied by element F31 of the input matrix. As a result, the usable section of the output matrix is reduced by 1 element along each edge of the image.

Let's pause for a moment to consider the demands such a filter places on a processor: For a VGA image (640x480 pixels/frame) at 30 frames/sec, there are 9.2 Mpixels/sec. Now consider if the 9 multiplies and 8 accumulates need to be done serially: that's (9+8)*9.2 = 156 MIPS! If the accumulates are done in parallel with the multiplies, the load will be reduced to 9*9.2 = 83 MIPS. The following example will show how an additional 2x cycle savings can be achieved.

Efficient 2D Convolution: An example

For the code described in the following sections, the focus will be on the "inner" loop where all of the multiply/accumulate (MAC) operations are performed. This example will demonstrate that by aligning the input data properly, both MAC units can be used in a single processor cycle to process two output points at a time. During this same cycle, multiple data fetches occur in parallel with the MAC operation.

The critical section of this application is the inner loop, which is shown in Figure 3. Each line within the inner loop is executed in a single instruction. The input data is represented in 16-bit quantities. The start of the input matrix must be aligned on a 32-bit boundary. This ensures that two consecutive points from the input matrix can be read in with a single 32-bit read. Prior to entering this loop, the first value of the input matrix is stored in R0.H and the second value (F12) is stored in R0.L, as shown in the first 2 operations during Cycle 1 in Figure 3. The register R1.L also was loaded prior to entering the inner loop. It contains the value of the first element in the mask matrix (H11).

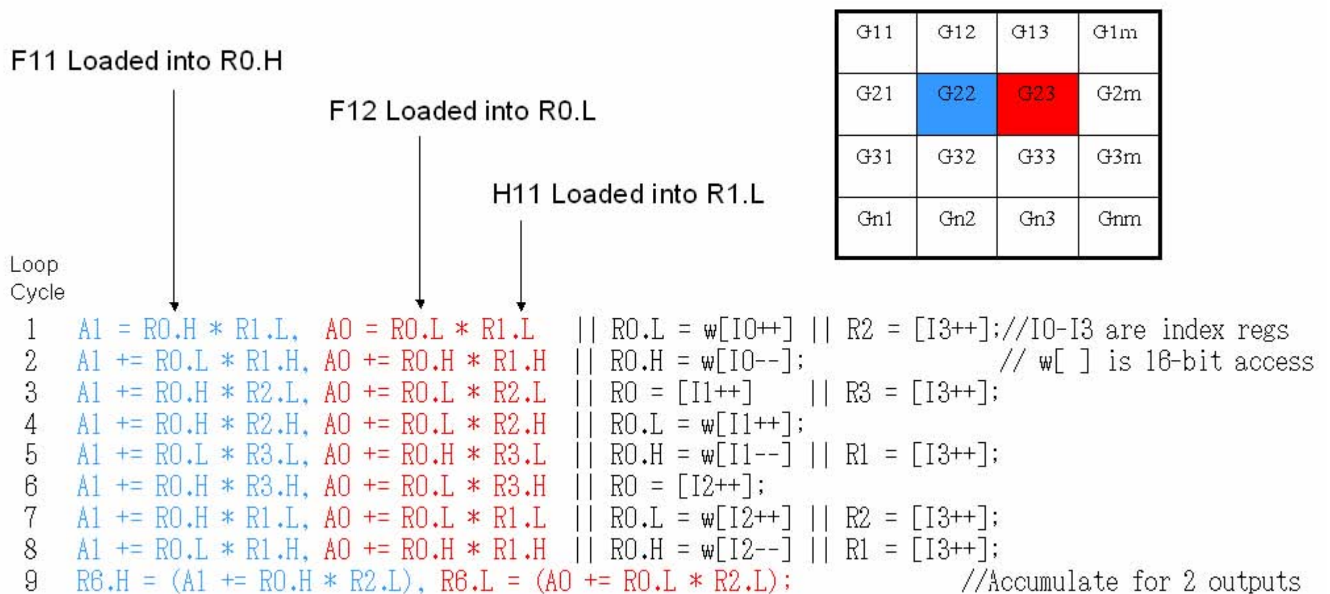

Figure 3: Efficient Implementation of 3x3 Convolution on a DSP

As described earlier, there are nine multiplies and eight accumulates required to obtain each element of the output matrix. However, because of the dual MAC operations, 2 output elements are available at the completion of each inner loop. Thus, F11*H11 and F12*H11 are available in the accumulators at the end of the first instruction. Each instruction in the inner loop moves to the next mask value. Results are summed in separate accumulators. The final outputs of the inner loop are loaded into R6.

Not only are multiple arithmetic operations occurring each cycle, but also load/store operations take place in parallel to achieve even greater efficiency. Again using Cycle #1 as an example, the next input element (F13) is read into R0.L and is available for use in a MAC on the very next instruction. Similarly, R2 is loaded with the next set of mask values. These values are used in subsequent MAC operations in the inner loop.

Conclusion

As image filtering goes, two-dimensional convolution with a 3x3 mask is relatively straightforward to implement. However, the study presented above demonstrates how selecting a processor designed for real-time image processing and understanding its architectural components can increase algorithm efficiency and reduce cycle time, in this case by a factor of 4. This understanding, in turn, can provide a strong foundation for implementing more complex image processing functionality on the same platform.