Image Processing 8

8.1 OVERVIEW

Image processing often involves computation on large matrices (of data values) that represent digitized images. Each element of the array represents a pixel of the image; its location in the array corresponds to its location in the image, and its value determines the color or shading of the pixel.

The largest two-dimensional matrix that a full complement of ADSP-2100 data memory will hold contains 16384 elements (128 rows by 128 columns). If your application requires a larger array, you must use extended memory addressing. For more information on a hardware implementation of this type of memory architecture, contact Analog Devices' DSP Applications Group.

The CNTR register of the ADSP-2100 is a 14-bit unsigned register that can track loop iteration; a loop can be set to execute up to 16383 times. If more loop iterations are needed, they can be obtained by nesting loops. If the number of iterations needed can not be achieved easily through loop nesting, the loop can be programmed explicitly. In this method, the AF register is preloaded with the number of loops, and at the end of the loop it is decremented. A conditional jump tests the AF value after the decrement; if it is not zero, a jump to the top of the loop is executed, and if it is zero, no jump occurs, and the loop is exited. A loop using this method can be iterated up to 65535 times. Each loop execution incurs a two-cycle overhead penalty, however.

8.2 TWO-DIMENSIONAL CONVOLUTION

Two-dimensional convolution has a variety of applications in image processing. One common application is finding an object in an image using two-dimensional edge detection. To determine the orientation of the edge, an input window is convolved with several different templates. A direction is chosen based on the template that yields the maximum convolution value. Another application is pixel smoothing, which can be accomplished with two-dimensional convolution by varying the convolution coefficients. A smoothed pixel value is based on the weighted average of the unsmoothed value and those of the eight neighboring pixels.

When implementing the convolution, instead of giving the address of the pixel being convolved, the address of the upper left hand corner of the convolving window is given and the convolution window will not extend outside the input window. Due to this fact, the output window will be smaller (by two pixels per row and column) than the input window. Input and coefficient matrices stored by rows allow you to take advantage of the one-cycle address modification capability of the ADSP-2100 using modify (M) registers. When the pointer to the input window reaches the end of each coefficient row, it is updated with a different modify register that causes the pointer to move to the beginning of the next row of the convolution window. When the multiplication is complete, still another modify register moves the index register to the point at which the next convolution will begin.

The routine in Listing 8.1 uses two loops to allow for large input windows. Because the CNTR register is 14-bits wide, a single loop would restrict the input matrix to 16383 elements. The two loops also may be easily modified for extended memory addressing.

Before calling the routine you must store the address of the input matrix in I0, the coefficient matrix in I4, and the output matrix in I1. The length registers L0 and L1 should be set to zero. L4 should be set to nine to use the circular buffer modulo addressing with the pointer to the convolution matrix, I4, because the matrix is used repeatedly. When performing the convolution on an MxN matrix, CNTR should be set to M–2, the number of output rows, and M1 should be set to N–2, the number of output columns. M1 is also used to move the I0 pointer from one row to the next, which is necessary because the nine values of the input matrix used in the convolution are not contiguous in memory. M2 is set to –(2N +1) to move the I0 pointer back to the beginning of the convolution window, and M3 is set to two to move the I0 pointer to the next input row. M0 and M4 are set to one for sequential fetches from data memory and program memory.

The routine begins by reading the first data and coefficient values. The *in_row* loop is executed once for each row of the output matrix. The *in_col* loop executes once for each column of the output matrix. Inside this loop, one multiply/data fetch multifunction instruction is executed for each element of the convolution window.

The last instruction of the *in_col* loop saves the output data point. A MODIFY instruction that moves the input matrix pointer, I0, to the beginning of the next input row finishes the *in_row* loop.

```
.MODULE Two_Dimensional_Convolution;

{           3  3
   G(x,y) = Σ  Σ [H(i,j) × F(x+i, y+j)]
           i=1 j=1

   Calling Parameters
        I0 —> F(x,y), MxN Input Matrix                          L0 = 0
        I1 —> G(x,y), (M-2)x(N-2) Output Matrix                 L1 = 0
        I4 —> H(i,j), 3x3 Coefficient Matrix (circular)         L4 = 9
        CNTR = M-2          M3 = 2
        M0 = 1              M4 = 1
        M1 = N-2            M2 = -(2 × N + 1)

   Return Values
        G(x,y) Filled

   Altered Registers
        MX0,MY0,MR,I0,I1,I4

   Computation Time
        ((10 × (N-2) + 4) × (M-2)) + 3 + 10 cycles
}

.ENTRY  conv;

conv:   MX0=DM(I0,M0), MY0=PM(I4,M4);           {Get first data and coeff}
        DO in_row UNTIL CE;                     {Loop M-2 times}
           CNTR=M1;
           DO in_col UNTIL CE;                  {Loop N-2 times}
              MR=MX0*MY0 (SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
              MR=MR+MX0*MY0 (SS), MX0=DM(I0,M1), MY0=PM(I4,M4);
              MR=MR+MX0*MY0 (SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
              MR=MR+MX0*MY0 (SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
              MR=MR+MX0*MY0 (SS), MX0=DM(I0,M1), MY0=PM(I4,M4);
              MR=MR+MX0*MY0 (SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
              MR=MR+MX0*MY0 (SS), MX0=DM(I0,M0), MY0=PM(I4,M4);
              MR=MR+MX0*MY0 (SS), MX0=DM(I0,M2), MY0=PM(I4,M4);
              MR=MR+MX0*MY0 (RND), MX0=DM(I0,M0), MY0=PM(I4,M4);
in_col:       DM(I1,M0)=MR1;            {Save convolution value}
in_row:    MODIFY(I0,M0);               {Point to next input row}

       RTS;
.ENDMOD;
```

Listing 8.1 Two-Dimensional Convolution

8 Image Processing

8.3 SINGLE-PRECISION MATRIX MULTIPLY

Matrix multiplication is commonly used to translate or rotate an image. The routine presented in this section multiplies two input matrices: X, an RxS (R rows, S columns) matrix stored in data memory, and Y, an SxT (S rows, T columns) matrix stored in program memory. The output Z, an RxT (R rows, T columns) matrix, is written to data memory.

Before calling the routine, which is shown in Listing 8.2, the values must be set up as follows. The starting address of X must be in I2 and the starting address of Y in I5. The starting address of the result buffer Z must be in I1. All matrices are stored by rows; that is, each successive memory location contains the next sequential row element, and the last element of a row is followed by the first element of the next row. M0 and M4 must be set to one. M5 must contain the value of T (number of columns in Y), and M1 the value of S (number of columns in X). The CNTR register must contain the value of R (number of rows in X), and SE must contain the value necessary to shift the result of each multiplication into the desired format. For example, SE would be set to zero to obtain a matrix of 1.31 values from the multiplication of two matrices of 1.15 values.

The *row_loop* loop is executed once for each row of the result matrix (R times). Before the *column_loop* loop is entered, CNTR is set to the value of T and I5 is reset to point to the beginning of the Y matrix. The *column_loop* loop is executed once for each column of the result matrix (T times). The *element_loop* loop is executed to compute one element of the output matrix.

Before the element of the output matrix is computed, CNTR is set to S for the number of multiplies necessary, I0 is set to the first element of the current X matrix row, and I4 is set to the first element of the current Y matrix column. The MR register is cleared, the first element of the current X row is loaded into MX0, and the first element of the current Y column is loaded into MY0.

After an element of the output matrix is computed, it is adjusted in the shifter to maintain data integrity. For example, if each matrix is composed of 4.12 elements, each output element must be shifted to the left three bits to form a 4.12 value before being stored, if the output matrix is also to be composed of 4.12 elements. The magnitude of the shift is controlled by the value that was preloaded in the SE register. During the first shift operation, the multiword instruction also updates the pointer to the next column of the Y matrix.

The last instruction in the *column_loop* loop stores the value of the output element in memory. The *row_loop* loop finishes by modifying the pointer to the current row of the X matrix.

The time required to complete the multiplication depends on the size of the input matrices. The routine requires one cycle for the return instruction and one cycle to start the *row_loop* loop, which is executed R times. The *row_loop* loop requires four cycles in addition to the cycles used in *column_loop*, which is executed T times. The *column_loop* loop contains eight cycles overhead plus the *element_loop* loop, which executes in S cycles. Two more cycles are required for data reads from program memory before the code is completely contained in cache memory. The total execution time is therefore $((S + 8) \times T + 4) \times R + 4$ cycles.

```
.MODULE matmul;

{
   Single-Precision Matrix Multiplication

                S
        Z(i,j) = Σ [X(i,k) × Y(k,j)]   i = 0 to R; j = 0 to T
               k=0

        X is an RxS matrix
        Y is an SxT matrix
        Z is an RxT matrix

   Calling Parameters
        I1 –> Z buffer in data memory              L1 = 0
        I2 –> X, stored by rows in data memory     L2 = 0
        I6 –> Y, stored by rows in program memory   L6 = 0
        M0 = 1      M1 = S
        M4 = 1      M5 = T
        L0,L4,L5 = 0
        SE = Appropriate scale value
        CNTR = R

   Return Values
        Z Buffer filled by rows

   Altered Registers
        I0,I1,I2,I4,I5,MR,MX0,MY0,SR

   Computation Time
        ((S + 8) × T + 4) × R + 2 + 2 cycles
}
```

(listing continues on next page)

```
.ENTRY  spmm;

spmm:   DO row_loop UNTIL CE;
           I5=I6;                                   {I5 = start of Y}
           CNTR=M5;
           DO column_loop UNTIL CE;
              I0=I2;                                {Set I0 to current X row}
              I4=I5;                                {Set I4 to current Y col}
              CNTR=M1;
              MR=0, MX0=DM(I0,M0), MY0=PM(I4,M5);   {Get 1st data}
              DO element_loop UNTIL CE;
element_loop:    MR=MR+MX0*MY0 (SS), MX0=DM(I0,M0), MY0=PM(I4,M5);
              SR=ASHIFT MR1 (HI), MY0=DM(I5,M4);    {Update I5}
              SR=SR OR LSHIFT MR0 (LO);             {Finish shift}
column_loop:  DM(I1,M0)=SR1;                        {Save output}
row_loop:  MODIFY(I2,M1);                           {Update I2 to next X row}
        RTS;
.ENDMOD;
```

Listing 8.2 Single-Precision Matrix Multiply

8.4 HISTOGRAM

A histogram describes the frequency of occurrences of a particular value in a matrix (or, more generally, any set of data). A common image-processing application is determining the number of times a particular gray scale (pixel value) occurs in a digitized two-dimensional image.

To perform a histogram on a range of data, you must know the number of unique values each datum can have. The histogram contains one location for each value in which it records the occurrences of that value. For example, if each pixel in an image is represented by an 8-bit value, each pixel can take on 256 (28) different values. Its histogram has 256 different locations, one for each value.

The histogram routine is shown in Listing 8.3. The address of the input array is read from the address stored in I0; L0 and L4 must both be set to zero. The modify register M1 must be set to one, and M5 and M0 must be set to zero. All locations of the output array, whose address is stored in I4, must be initialized to zero before the routine is called.

The routine checks the value of each element in the input data array and increments the appropriate location for the value in the output array. The routine begins by reading in the first value of the input data array. This value is used to modify the index register, which initially points to the beginning of the output data array. The second input value is then read, and the CNTR is set to the number of remaining input points (the total number less two).

Each pass through the *histo_loop* loop calculates the modify value for the output buffer pointer (I4), reads in and increments the current counter, inputs the next value, and modifies the I4 register to point to the next location as it saves the new counter value. The instructions following the *histo_loop* loop read in and increment the counters for the last two data values.

No boundary checking is performed on the input data, so the calling routine must ensure that the output histogram has enough locations to accommodate the maximum possible range of the input data.

```
.MODULE Histogram_subroutine;

.CONST  N_less_2=2046;        {number of pixels - 2}

{
   Calculates histogram of input data

   Calling Parameters
        I0 –> Input buffer in data memory       L0 = 0
        I4 –> Histogram buffer in program memory L4 = 0
             (I4 initialized to first location;
              All buffer locations initialized to 0)
        M0 = 0
        M1 = 1
        M5 = 0

   Return Values
        Histogram buffer filled with results

   Altered Registers
        AX0,AX1,AY0,AF,AR,I0,I4,M4

   Computation Time
        7 + (4 × (Input buffer length - 2) + 2) + 12 cycles
}
```

(listing continues on next page)

```
.ENTRY  histo;

histo:  AX1=DM(I0,M1);                          {Get 1st data value}
        M4=AX1;
        MODIFY(I4,M4);                          {Point to its counter}
        AX0=DM(I0,M0);                          {Get 2nd data value}
        AF=PASS AX1;                            {Pass to y operand}

        CNTR=N_less_2;

        DO histo_loop UNTIL CE;
             AR=AX0-AF, AX1=DM(I0,M1), AY0=PM(I4,M5);
                              {Calc index, reread data, locate cntr}
             M4=AR;           {Transfer index to modify register}
             AR=AY0+1, AX0=DM(I0,M0);           {Incr cntr, load next data}
histo_loop:  PM(I4,M4)=AR, AF=PASS AX1;
             {Save updated cntr, point to next, pass to y}

   AR=AX0-AF, AY0=PM(I4,M5);                    {Calc index, locate cntr}
   M4=AR;                                       {Transfer index to modify register}
   AR=AY0+1;                                    {Increment cntr}
   PM(I4,M4)=AR;                                {Save updated cntr, point to next}
   AY0=PM(I4,M5);                               {Locate cntr}
   AR=AY0+1;                                    {Incr cntr}
   PM(I4,M5)=AR;                                {Store updated cntr}
   RTS;

.ENDMOD;
```

Listing 8.3 Histogram

8.5 REFERENCES

Offen, R.J. 1985. VLSI Image Processing. New York: McGraw-Hill Book Company.

Oppenheim, A.V. ed. 1978. Applications of Digital Signal Processing. Englewood Cliffs, N.J.: Prentice-Hall, Inc.