RASTER IMAGE PROCESSING ON THE TMS320C6X VLIW DSP

Prof. Brian L. Evans

in collaboration with

Niranjan Damera-Venkata and

Wade Schwartzkopf

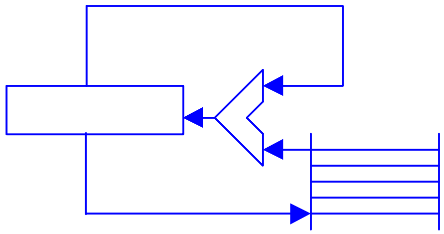
Embedded Signal Processing Laboratory

The University of Texas at Austin

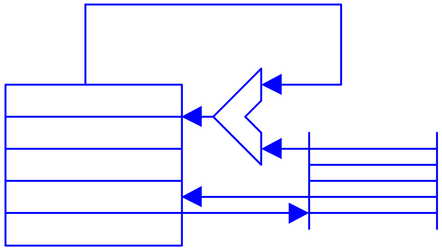
Austin, TX 78712-1084

http://signal.ece.utexas.edu/

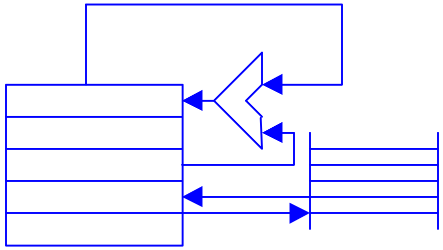
Accumulator architecture



Memory-register architecture



Load-store architecture

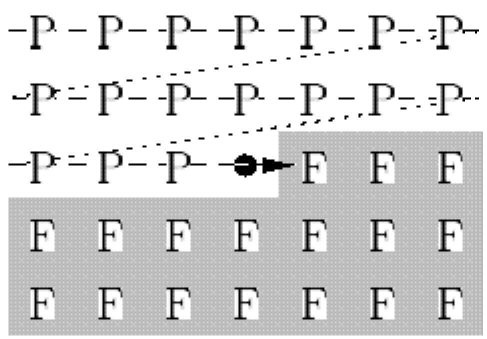


Outline

- Introduction
- Color conversion
- Interpolation
- Halftoning
- C/C++ coding tips
- Conclusion

Introduction

- Raster scan



P = Past
F = Future

- Raster image processing
  - Process one or more rows at a time
  - Pixel operations: color conversion, ordered dither halftoning
  - Local operations: JPEG coding, FIR filtering, interpolation, error diffusion halftoning

Raster Image Processing on the TMS320C6x

- TMS320CC6x works best with 16-bit data
  - Bytes per image pixel: 1 for greyscale, 3 or 4 for color
  - Reduce processor performance or double memory
- Number of 4800-pixel rows (e.g. 8 in. at 600 dpi) of a greyscale image that can fit into memory

Processor	MHz	Data (kbits)	Program (kbits)	Price	Max Rows byte pixels	Max Rows halfwords
C6211 **	167	32+512	32	$25	13	6.5
C6201	200	512	512	$159	13	6.5
C6202	250	1000	2000	$184	26	13.0
C6203	300	4000	3000	n/a	104	52.0

** C6211 has 512 kbits of L2 on-chip cache.
All of it used for the image.

Color Spaces

- RGB: Red Green Blue
  - Additive color
  - CRT displays
- YCrCb: Luminance Chrominance
  - Decouples intensity and color information
  - Digital image/video compression standards (digital TV)
  - Eye less sensitive to chrominance than luminance: subsample Cr/Cb without significant visual degradation
- CMY(K): Cyan Magenta Yellow (Black)
  - Subtractive color
  - Printing and photography
  - Black ink used for improved color gamut, and faster drying and purer rendering for black and greys

ITU RGB to YCrCb Standards

- Nested conversion formulas
  - $Y = C_{red} R + C_{green} G + C_{blue} B$
  - $C_r = (R - Y) / (2 - 2 C_{red})$
  - $C_b = (B - Y) / (2 - 2 C_{blue})$

- 8-bit format
  - Y,R,G,B in [0,255]
  - C_r in [-128,127]
  - C_b in [-128,127]

Recommendation	C_{red}	C_{green}	C_{blue}	C_r range	C_b range
ITU (CCIR) 601-1	0.2989	0.5866	0.1145	[-182,182]	[-144,144]
ITU (CCIR) 709	0.2126	0.7152	0.0722	[-162,162]	[-137,137]
ITU	0.2220	0.7067	0.0713	[-164,164]	[-137,137]

- Y is lossless, but Cr/Cb is clipped to [-128,127]
- Assume that RGB has been gamma corrected
- Rec 601-1 used with TIFF and JPEG standards

ITU YCrCb to RGB Standards

- Nested conversion formulas
  - $R = Y + (2 - 2\, C_{red})\, C_r$
  - $B = Y + (2 - 2\, C_{blue})\, C_b$
  - $G = (Y - C_{blue}\, B - C_{red}\, R) / C_{green}$
- 8-bit format
  - Y,R,G,B in [0,255]
  - C_r in [-128,127]
  - C_b in [-128,127]

Recommendation	C_{red}	C_{green}	C_{blue}	R range	B range
ITU (CCIR) 601-1	0.2989	0.5866	0.1145	[-179,433]	[-227,480]
ITU (CCIR) 709	0.2126	0.7152	0.0722	[-200,455]	[-238,493]
ITU	0.2220	0.7067	0.0713	[-199,453]	[-238,493]

- Range of G is [-134,390] for Rec 601-1
- RGB values are clipped to [0,255] and rounded

http://www.neuro.sfc.keio.ac.jp/~aly/polygon/info/color-space-faq.html
http://www.inforamp.net/~poynton/notes/colour_and_gamma/ColorFAQ.html

RGB/YCrCb Conversion in Floating Point

- Nested formulas

$Y = 0.2989\,R + 0.5866\,G + 0.1145\,B$

$C_r = 0.7132\,(R - Y)$

$C_b = 0.5647\,(B - Y)$

$R = Y + 1.4022\,C_r$

$B = Y + 1.7710\,C_b$

$G = 1.7047\,Y - 0.1952\,B - 0.5647\,R$

- Matrix multiplication

$$\begin{matrix} 0.2989 & 0.5866 & 0.1145 \\ 0.5000 & -0.4183 & -0.0817 \\ -0.1688 & -0.3312 & 0.5000 \end{matrix}$$

$[Y\ C_r\ C_b]^T = \boldsymbol{M}\ [R\ G\ B]^T$

$$\begin{matrix} 1 & 1.4022 & 0 \\ 1 & -0.7145 & -0.3458 \\ 1 & 0 & 1.7710 \end{matrix}$$

$[R\ G\ B]^T = \boldsymbol{M}\ [Y\ C_r\ C_b]^T$

- Round and clip each quantity to eight bits

RGB/YCrCb Conversion in Fixed Point

- Multiplication by direction calculation
  - Quantize coefficients
  - Put coefficients in registers and stream pixels
  - Highly accurate under extended precision accumulation
  - Well-matched to DSPs and graphics cards
- Multiplication by table lookup
  - Precalculate multiplications (floating point times byte)
  - Store in 5 (9) 256-byte tables for nested (matrix) formulas means 5 (9) times increase in memory bandwidth and poor cache performance
  - Do not need extended precision accumulators
  - Well-matched to ASICs and microcontrollers
- Additions: 4 (6) for nested (matrix) formulas

RGB/YCrCb Conversion on 16-bit DSP

- Move 8-bit color quantity to upper 8 of 16 bits
- Use 16 x 16 multiplication, 32-bit accumulation
- Nested formulas (coefficients scaled by 2^{15}-1)

$Y = 9794\,R + 19221\,G + 3752\,B$

$C_r = 23369\,(R - Y)$

$C_b = 18504\,(B - Y)$

$R = Y + 2\,(22973\,C_r)$

$B = Y + 2\,(29015\,C_b)$

$G = 2\,(27929)\,Y - 6396\,B - 18504\,R$

- Matrix multiplication (coefficients scaled by 2^{15}-1)

$$\begin{matrix} 9794 & 19221 & 3752 \\ 16384 & -13706 & -2677 \\ -5531 & -10852 & 16384 \end{matrix}$$

$[Y\ C_r\ C_b]^T = \boldsymbol{M}\ [R\ G\ B]^T$

$$\begin{matrix} 32767 & 2\,(22973) & 0 \\ 32767 & -23412 & -11331 \\ 32767 & 0 & 2\,(29015) \end{matrix}$$

$[R\ G\ B]^T = \boldsymbol{M}\ [Y\ C_r\ C_b]^T$

RGB to CMY and CMYK

- RGB to CMY (ideal case)
  - $C = 255 - R$
  - $M = 255 - G$
  - $Y = 255 - B$

Division takes 1 or 2 instructions per bit of precision in result

- CMY to CMYK
  - $K = \min(C, M, Y)$
  - $C = 255\,(C - K) / (255 - K)$
  - $M = 255\,(M - K) / (255 - K)$
  - $Y = 255\,(Y - K) / (255 - K)$
  - 2-D lookup tables

RGB to CMYK

$K = 255 - \max(R, G, B)$

$C = 255\,(m - R) / m$

$M = 255\,(m - G) / m$

$Y = 255\,(m - B) / m$

$m = \max(R, G, B)$

- R, G, B, C, M, Y, and K have a range of [0,255]
- Useful in printers and copiers

Matrix Computation Example

```
; Texas Instruments, INC.
;
; MATRIX VECTOR MULTIPLY
;
; ftp://ftp.ti.com/pub/tms320bbs/c67xfiles/mvm.asm
;
; DESCRIPTION
; A[][] * B[] = C[]
;
; ARGUMENTS PASSED
; a[]     -> A4
; b[]     -> B4
; c[]     -> A6
; rows    -> B6
; columns -> A8
;
; CYCLES
; (n + 20)*m + 1 (m = # of rows, n = # of columns)
```

Matrix Computation Example (cont.)

```
          *** begin piplining inner loop

          SUB   .L1X    rows,1,ocntr
||        ADD   .L2     bptr,4,btmp
||        LDW   .D1T1   *aptr++(4),aa0    ;1  load a[i] from memory
||        LDW   .D2T2   *bptr,bb0         ;1  load b[i] from memory
||        SUB   .S2X    colms,1,lcntr     ;   load cntr = comumns - 1

oloop:

   [lcntr] LDW  .D1T1   *aptr++(4),aa0    ;2  if(lcntr) load a[i] from memory
|| [lcntr] LDW  .D2T2   *btmp++(4),bb0    ;2  if(lcntr) load b[i] from memory
|| [lcntr] SUB  .L2     lcntr,1,lcntr     ;2  if(lcntr) lcntr -= 1
||         SUB  .S1     colms,2,icntr     ;
||         ZERO .L1     sum0              ;   zero the running sum

   [lcntr] LDW  .D1T1   *aptr++(4),aa0    ;3  if(lcntr) load a[i] from memory
|| [lcntr] LDW  .D2T2   *btmp++(4),bb0    ;3  if(lcntr) load b[i] from memory

|| [lcntr] SUB  .L2     lcntr,1,lcntr     ;3  if(lcntr) lcntr -= 1

   [lcntr] LDW  .D1T1   *aptr++(4),aa0    ;4  if(lcntr) load a[i] from memory
|| [lcntr] LDW  .D2T2   *btmp++(4),bb0    ;4  if(lcntr) load b[i] from memory
|| [lcntr] SUB  .L2     lcntr,1,lcntr     ;4  if(lcntr) lcntr -= 1
```

Matrix Computation Example (cont.)

```
   [lcntr]  LDW   .D1T1   *aptr++(4),aa0   ;5  if(lcntr) load a[i] from memory
|| [lcntr]  LDW   .D2T2   *btmp++(4),bb0   ;5  if(lcntr) load b[i] from memory
|| [lcntr]  SUB   .L2     lcntr,1,lcntr    ;5  if(lcntr) lcntr -= 1
||          B     .S2     iloop            ;1  branch to iloop

   [lcntr]  LDW   .D1T1   *aptr++(4),aa0   ;6  if(lcntr) load a[i] from memory
|| [lcntr]  LDW   .D2T2   *btmp++(4),bb0   ;6  if(lcntr) load b[i] from memory
|| [lcntr]  SUB   .L2     lcntr,1,lcntr    ;6  if(lcntr) lcntr -= 1
|| [icntr]  SUB   .L1     icntr,1,icntr    ;6  if(icntr) icntr -= 1
||          MPYSP .M1X    aa0,bb0,mult0    ;1  mult0 = a[i]*b[i]
|| [icntr]  B     .S2     iloop            ;2  if(icntr) branch to iloop

   [lcntr]  LDW   .D1T1   *aptr++(4),aa0   ;7  if(lcntr) load a[i] from memory
|| [lcntr]  LDW   .D2T2   *btmp++(4),bb0   ;7  if(lcntr) load b[i] from memory
|| [lcntr]  SUB   .L2     lcntr,1,lcntr    ;7  if(lcntr) lcntr -= 1
|| [icntr]  SUB   .L1     icntr,1,icntr    ;7  if(icntr) icntr -= 1
||          MPYSP .M1X    aa0,bb0,mult0    ;2  mult0 = a[i]*b[i]
|| [icntr]  B     .S2     iloop            ;3  if(icntr) branch to iloop
```

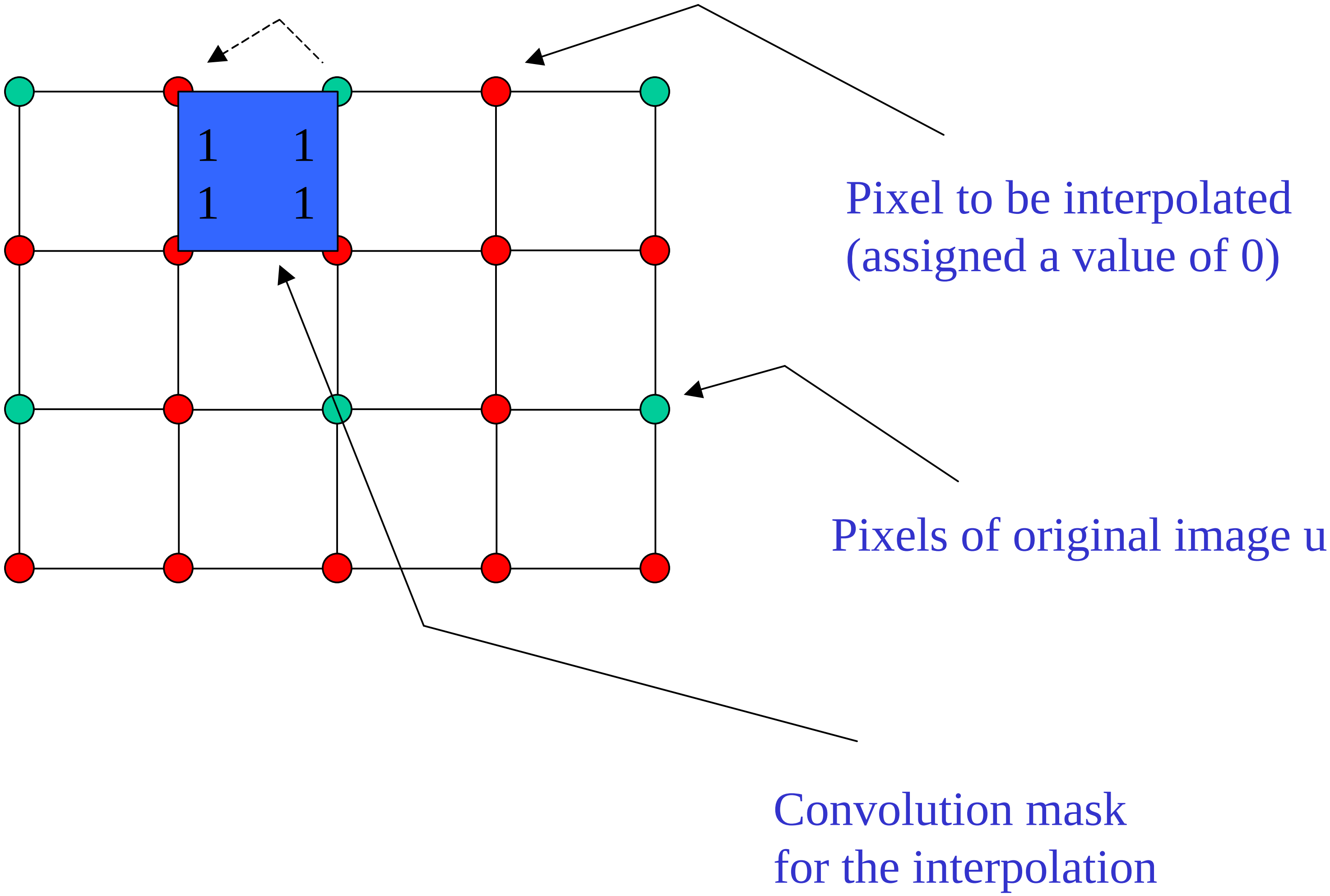
Matrix Computation Example (cont.)

```
   [lcntr]  LDW   .D1T1    *aptr++(4),aa0    ;8  if(lcntr) load a[i] from memory
|| [lcntr]  LDW   .D2T2    *btmp++(4),bb0    ;8  if(lcntr) load b[i] from memory
|| [lcntr]  SUB   .L2      lcntr,1,lcntr     ;8  if(lcntr) lcntr -= 1
|| [icntr]  SUB   .L1      icntr,1,icntr     ;8  if(icntr) icntr -= 1
||          MPYSP .M1X     aa0,bb0,mult0     ;3  mult0 = a[i]*b[i]
|| [icntr]  B     .S2      iloop             ;4  if(icntr) branch to iloop

   [lcntr]  LDW   .D1T1    *aptr++(4),aa0    ;9  if(lcntr) load a[i] from memory
|| [lcntr]  LDW   .D2T2    *btmp++(4),bb0    ;9  if(lcntr) load b[i] from memory
|| [lcntr]  SUB   .L2      lcntr,1,lcntr     ;9  if(lcntr) lcntr -= 1
|| [icntr]  SUB   .L1      icntr,1,icntr     ;9  if(icntr) icntr -= 1
||          MPYSP .M1X     aa0,bb0,mult0     ;4  mult0 = a[i]*b[i]
|| [icntr]  B     .S2      iloop             ;5  if(icntr) branch to iloop

iloop:

   [lcntr]  LDW   .D1T1    *aptr++(4),aa0    ;10  if(lcntr) load a[i] from memory
|| [lcntr]  LDW   .D2T2    *btmp++(4),bb0    ;10  if(lcntr) load b[i] from memory
|| [lcntr]  SUB   .L2      lcntr,1,lcntr     ;10  if(lcntr) lcntr -= 1
|| [icntr]  SUB   .S1      icntr,1,icntr     ;10  if(icntr) icntr -= 1
||          MPYSP .M1X     aa0,bb0,mult0     ;5   mult0 = a[i]*b[i]
||          ADDSP .L1      mult0,sum0,sum0   ;1   sum0 = sum0+mult0
|| [icntr]  B     .S2      iloop             ;6   if(icntr) branch to iloop
```

Matrix Computation Example (cont.)

```
***************** add up the running sums ***

            MV     .D1     sum0,temp1          ;    temp1 = sum0
            ADDSP  .L1     sum0,temp1,temp2    ;    temp2 = temp1 + sum0 (2nd sum0)
            MV     .D1     sum0,temp1          ;    temp1 = sum0 (the 3rd sum0)
            ADDSP  .L1     sum0,temp1,temp3    ;    temp3 = temp1 + sum0 (4th sum0)
            NOP            2                   ;    wait for temp3
   [ocntr]  B      .S2     oloop               ;    if(ocntr) branch to oloop
            ADDSP  .L1     temp2,temp3,sum0    ;    sum0 = temp2 + temp3
***
   [ocntr]  MV     .D2     bptr,btmp           ;    reset *b to beginning of b

            SUB    .S1     colms,2,icntr       ;    inner cntr = columns - 2
||          SUB    .S2X    colms,1,lcntr       ;    load cntr = comumns - 1

            LDW    .D1T1   *aptr++(4),aa0      ;1   load a[i] from memory
||          LDW    .D2T2   *btmp++(4),bb0      ;1   load b[i] from memory

            STW    .D1     sum0,*cptr++(4)     ;    c[i] = sum0
|| [ocntr]  SUB    .L1     ocntr,1,ocntr       ;    if(ocntr) ocntr -= 1
```

Nearest Neighbor Interpolation



Nearest Neighbor Interpolation

- Interpolation by pixel replication
  - Computationally simple
  - Aliasing

```
/* v is the zoomed (interpolated) version of u */
v[m,n]=u[round(m/2),round(n/2)]
```

- May be implemented as 2-D FIR filter by **H**
  - Alternate pixels may be skipped

$$\mathbf{H} = \begin{bmatrix} \boxed{1} & 1 \\ 1 & 1 \end{bmatrix}$$

Bilinear Interpolation

- Interpolate rows then columns (or vice-versa)
  - Increased complexity
  - Reduced aliasing

```
/* v is the zoomed (interpolated) version of u */
v1[m,2n]   = u[m,n]
v1[m,2n+1] = a1*u[m,n]+a2*u[m,n+1]
v[2m,n]    = v1[m,n]
v[2m+1,n]  = b1*v1[m,n]+b2*v1[m+1,n]
```

- May be implemented as a 2-D FIR filter by **H** followed by a shift

$$\mathbf{H} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & \boxed{4} & 2 \\ 1 & 2 & 1 \end{bmatrix} \gg 4$$

2-D FIR Filter

- Difference equation

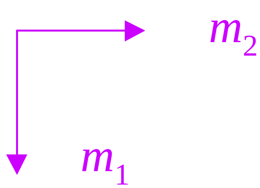
$$y(n) = 2\ x(n_1,n_2) + 3\ x(n_1\text{-}1,n_2) + x(n_1,n_2\text{-}1) + x(n_1\text{-}1,n_2\text{-}1)$$

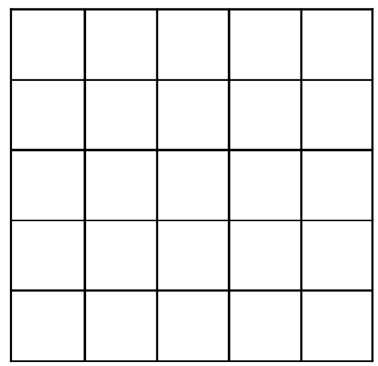
- Flow graph

$$y(n_1,n_2) = \sum_{m_1=0}^{M_1-1} \sum_{m_2=0}^{M_2-1} a(m_1,m_2)\ x(n_1-m_1,n_2-m_2)$$

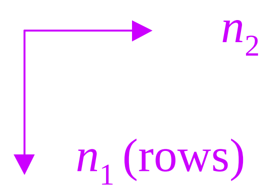
0	0	0
0	**2**	**1**
0	**3**	**1**

$a(m_1,m_2)$





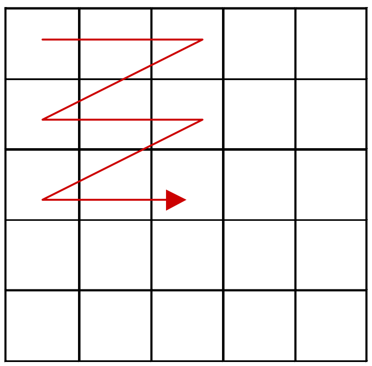
$x(n_1,n_2)$



- Vector dot product plus keep M_1 rows in memory and circularly buffer input

2-D Filter Implementations

- Store M_1 x M_2 filter coefficients in sequential memory (vector) of length $M = M_1 M_2$
- For each output, form vector from N_1 x N_2 image
  1. M_1 separate dot products of length M_2 as bytes
  2. Form image vector by raster scanning image as bytes
  3. Form image vector by raster scanning image as words



Raster scan

	Implementation 1	Implementation 2	Implementation 3
Throughput (samples/cycle)	*1*	*2*	*1.5*
Data read at one time (bytes)	*1*	*1*	*2*

2-D FIR Implementation #1 on C6x

```
; registers: A5=&a(0,0) B5=&x(n1,n2) B7=M A9=M2 B8=N2
fir2d1 MV    .D1   A9,A2       ; inner product length
||     SUB   .D2   B8,B7,B10   ; offset to next row
||     CMPLT.L1    B7,A9,A1    ; A1=no more rows to do
||     ZERO  .S1   A4          ; initialize accumulator
||     SUB   .S2   B7,A9,B7    ; number of taps left
fir1   LDBU  .D1  *A5++,A6     ; load a(m1,m2), zero fill
||     LDBU  .D2  *B5++,B6     ; load x(n1-m1,n2-m2)
||     MPYU  .M1X A6,B6,A3     ; A3=a(m1,m2) x(n1-m1,n2-m2)
||     ADD   .L1   A3,A4,A4    ; y(n1,n2) += A3
||[A2] SUB   .S1   A2,1,A2     ; decrement loop counter
||[A2] B     .S2   fir1        ; if A2 != 0, then branch
       MV    .D1   A9,A2       ; inner product length
||     CMPLT.L1    B7,A9,A1    ; A1=no more rows to do
||     ADD   .L2   B5,B10,B5   ; advance to next image row
||[!A1]B      .S1   fir1       ; outer loop
||     SUB   .S2   B7,A9,B7    ; count number of taps left
; A4=y(n1,n2)
```

2-D FIR Implementation #2 on C6x

```
; registers: A5=&a(0,0) B5=&x(n1,n2) A2=M B7=M2 B8=N2
fir2d2 SUB   .D2   B8,B7,B9   ; byte offset between rows
||     ZERO  .L1   A4         ; initialize accumulator
||     SUB   .L2   B7,1,B7    ; B7 = numFilCols - 1
||     ZERO  .S2   B2         ; offset into image data

fir2   LDBU  .D1  *A5++,A6    ; load a(m1,m2), zero fill
||     LDBU  .D2  *B6[B2],B6  ; load x(n1-m1,n2-m2)
||     MPYU  .M1X A6,B6,A3    ; A3=a(m1,m2) x(n1-m1,n2-m2)
||     ADD   .L1   A3,A4,A4   ; y(n1,n2) += A3
||     CMPLT.L2   B2,B7,B1    ; need to go to next row?
||     ADD   .S2   B2,1,B2    ; incr offset into image

[!B1]  ADD   .L2   B2,B9,B2   ; move offset to next row
||[A2] SUB   .S1   A2,1,A2    ; decrement loop counter
||[A2] B     .S2   fir2       ; if A2 != 0, then branch
; A4=y(n1,n2)
```

2-D FIR Implementation #3 on C6x

```
; registers: A5=&a(0,0) B5=&x(n1,n2) A2=M B7=M2 B8=N2
fir2d3 ZERO .D1   A4           ; initialize accumulator #1
||     SUB  .D2   B8,B7,B9     ; index offset between rows
||     ZERO .L2   B2           ; offset into image data
||     MVKH .S1   0xFF,A8      ; mask to get lowest 8 bits
||     SHR  .S2   B7,1,B7      ; divide by 2: 16bit address

       ZERO .D2   B4           ; initialize accumulator #2
||     ZERO .L1   A6           ; current coefficient value
||     ZERO .L2   B6           ; current image value
||     SHR  .S1   A2,1,A2      ; divide by 2: 16bit address
||     SHR  .S2   B9,1,B9      ; divide by 2: 16bit address
```

Initialization

2-D FIR Implementation #3 on C6x (cont.)

```
fir3    LDHU .D1 *A5++,A6     ; load a(m1,m2) a(m1+1,m2+1)
||      LDHU .D2 *B6[B2],B6   ; load two pixels of image x
||      CMPLT.L2  B2,B7,B1    ; need to go to next row?
||      ADD   .S2  B2,1,B2    ; incr offset into image

        AND   .L1 A6,A8,A6    ; extract a(m1,m2)
||      AND   .L2 B6,A8,B6    ; extract x(n1-m1,n2-m2)
||      EXTU .S1 A6,0,8,A9    ; extract a(m1+1,m2+1)
||      EXTU .S2 B6,0,8,B9    ; extract x(n1-m1+1,n2-m2+1)

        MPYHU .M1X A6,B6,A3 ; A3=a(m1,m2) x(n1-m1,n2-m2)
||      MPYHU .M2X A9,B9,B3 ; B3=a*x offset by 1 index
||      ADD   .L1  A3,A4,A4   ; y(n1,n2) += A3
||      ADD   .L2  B3,B4,B4   ; y(n1+1,n2+1) += B3
||[!B1]ADD  .D2  B2,B9,B2   ; move offset to next row
||[A2] SUB  .S1  A2,1,A2    ; decrement loop counter
||[A2] B    .S2  fir3       ; if A2 != 0, then branch
; A4=y(n1,n2) and B4=y(n1+1,n2+1)
```

Main Loop

FIR Filter Implementation on the C6x

```
        MVK   .S1 0x0001,AMR ; modulo block size 2^2
        MVKH  .S1 0x4000,AMR ; modulo addr register B6
        MVK   .S2 2,A2       ; A2 = 2 (four-tap filter)
        ZERO  .L1 A4         ; initialize accumulators
        ZERO  .L2 B4
; initialize pointers A5, B6, and A7
fir     LDW   .D1  *A5++,A0  ; load a(n) and a(n+1)
        LDW   .D2  *B6++,B1  ; load x(n) and x(n+1)
        MPY   .M1X A0,B1,A3  ; A3 = a(n) * x(n)
        MPYH  .M2X A0,B1,B3  ; B3 = a(n+1) * x(n+1)
        ADD   .L1  A3,A4,A4  ; yeven(n) += A3
        ADD   .L2  B3,B4,B4  ; yodd(n) += B3
[A2]    SUB   .S1  A2,1,A2   ; decrement loop counter
[A2]    B     .S2  fir       ; if A2 != 0, then branch
        ADD   .L1  A4,B4,A4  ; Y = Yodd + Yeven
        STH   .D1  A4,*A7    ; *A7 = Y
```

Throughput of two multiply-accumulates per cycle

Ordered Dithering on a TMS320C62x

periodic array of thresholds

1/8	5/8	7/8	3/8
7/8	3/8	1/8	5/8

Throughput of two cycles

```
; remove next two lines if thresholds in linear array
      MVK    .S1 0x0001,AMR       ; modulo block size 2^2
      MVKH   .S1 0x4000,AMR       ; modulo addr reg B6
; initialize A6 and B6
      .trip  100                  ; minimum loop count
dith: LDB    .D1 *A6++,A4         ; read pixel
||    LDB    .D2 *B6++,B4         ; read threshold
||    CMPGTU .L1x A4,B4,A1        ; threshold pixel
||    ZERO   .S1  A5              ; 0 if <= threshold
  [A1] MVK   .S1  255,A5          ; 255 if > threshold
||    STB    .D1  A5,*A6++        ; store result
||[B0] SUB   .L2  B0,1,B0         ; decrement counter
||[B0] B     .S2  dith            ; branch if not zero
```

More Efficient Ordered Dithering on the C6x

```
        MVK    .S1 0x00ff,A8         ; white pixel #1
||      MVK    .S2 0x0001,AMR        ; modulo block size 2^2
        SHL    .S1 A8,8,A9           ; white pixel #2
||      MVKH   .S2 0x4000,AMR        ; modulo addr reg. B6
        SHL    .S1 A8,16,A10         ; white pixel #3
||      SHL    .S2 A8,24,B9          ; white pixel #4

; initialize
;   A2 number of pixels divided by 4
;   A6 pointer to pixels (will be overwritten)
;   B6 pointer to thresholds
dith2: LDW     .D1 *A6,A4            ; read 4 pixels (bytes)
       LDW     .D2 *B6++,B4          ; read 4 thresholds
       EXTU    .S1 A4,24,24,A12      ; extract pixel #2
       EXTU    .S2 B4,24,24,B12      ; extract threshold #2
       ZERO    .L1 A5                ; store output in A5
       CMPLTU  .L2 A12,B12,B0        ; B0 = (A12 < B12)
```

Throughput of 1.25 pixels

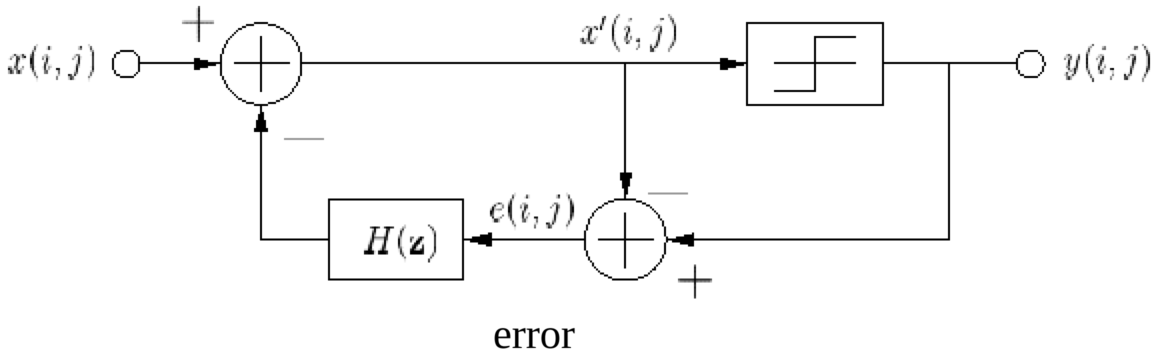
Initialization

More Efficient Ordered Dithering on the C6x

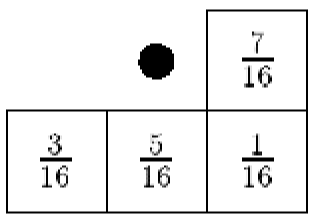
```
        EXTU   .S1 A4,16,24,A13     ; extract pixel #2
        EXTU   .S2 B4,16,24,B13     ; extract threshold #2
[!B0]   OR     .L1 A5,A8,A5         ; output of pixel 1
        CMPLTU .L2 A13,B13,B1       ; B1 = (A13 < B13)

        EXTU   .S1 A4,8,24,A14      ; extract pixel #3
        EXTU   .S2 B4,8,24,B14      ; extract threshold #3
[!B1]   OR     .L1 A5,A9,A5         ; output of pixels 1-2
        CMPLTU .L2 A14,B14,B2       ; B2 = (A14 < B14)

        EXTU   .S1 A4,0,24,A15      ; extract pixel #4
        EXTU   .S2 B4,0,24,B15      ; extract threshold #4
[!B2]   OR     .L2 A5,B9,B5         ; output of pixels 1-3
        CMPLTU .L1 A15,B15,A1       ; B2 = (A15 < B15)

[!A1]   OR     .S1 B5,A11,A5        ; output of pixels 1-4
        STW    .D1 A5,*A6++         ; store results
[A2]    SUB    .L1 A2,1,A2          ; decrement loop count
[A2]    B      .L2 dith2            ; if A2 != 0, branch
```

Floyd-Steinberg Error Diffusion

- Noise-shaped feedback coder (2-D sigma delta)



- Error filter $H(z)$



Floyd-Steinberg Error Diffusion

- C implementation color/grayscale error diffusion
- Replacing multiplications with adds and shifts
  - 3*error = (error << 2) - error
  - 5*error = (error << 2) + error
  - 7*error = (error << 3) - error
  - Can reuse (error << 2) calculation
- Replace division by 16 with adds and shifts
  - *n* >> 4 does not give right answer for negative n
  - Add offset of 2^4-1 = 15 for negative *n*: (*n* + 15) >> 4
  - Alternative is to work with | error |
- Combine nested for loops into one for loop that can be pipelined by the C6x tools

C/C++ Coding Tips

- Local variables
  - Define only when and where needed to assist compiler in mapping variables to registers (especially on C6x)
  - Give initial values to avoid uninitialized read errors
  - Choose names to indicate purpose and data type
  - In C, may only be defined at start of new environment
  - In C++, may be defined anywhere
  - Function arguments as local variables (may be updated)
- Reading strings from files using fgets
  - Reads N characters or newline, whichever comes first
  - Does not guarantee that newline is read
  - Does not guarantee that string is null terminated
- Define as many constants as possible

C/C++ Coding Tips

```c
int fileHasLine(FILE *filePtr,
                     char *searchStr) {
  char bufStr[128], *strPtr;
  int foundFlag;
  foundFlag = 0;
  while ( ! feof(filePtr) ) {
    strPtr = fgets(bufStr, 127, filePtr);
    if (strPtr &&
        strcmp(bufStr,searchStr) == 0) {
      foundFlag = 1;
      break;
    }
  }
  return(foundFlag);
}
```

Not robust

```c
#define BUFLEN 128
```

Robust

```c
int fileHasLine(FILE *filePtr,
           const char *searchStr) {
  int foundFlag = FALSE;
  while ( ! feof(filePtr) ) {
    char bufStr[BUFLEN];
    int bufStrLen = 0;
    char *strPtr =
      fgets(bufStr, BUFLEN-1, filePtr);
    bufStr[BUFLEN-1] = '\0';
    bufStrLen = strlen(bufStr);
    if ( bufStr[bufStrLen-1] == '\n' )
      bufStr[bufStrLen - 1] = '\0';
    if (strPtr &&
        strcmp(bufStr,searchStr) == 0) {
      foundFlag = TRUE;
      break;
    }
  }
  return(foundFlag);
}
```

Differences in blue

C/C++ Coding Tips

- Allocating dynamic memory
  - Function malloc allocates but does not initialize values: use calloc (allocate/initialize) or memset (initialize)
  - In C++, new operator calls malloc and then calls the constructor for each created object
  - On failure, malloc and new return 0: when new fails, _new_handler is called if set (set by set_new_handler)
- Deallocating dynamic memory
  - Function free crashes if passed a null pointer
  - In C++, delete operator first calls destructor of object(s) and then calls free: delete ignores null pointers
  - Use delete [] arrayPtr to deallocate an array
  - Zero pointer after deallocating it to prevent redeletion
  - Deallocate a pointer before reassigning it

C/C++ Coding Tips

```
Filter::Filter() {
  buf = 0;
}
Filter::AllocateBuffer(int n) {
  buf = new int [n];
}
Filter::DeallocateBuffer() {
  if (buf) delete buf;
}
Filter::~Filter() {
  DeallocateBuffer();
}
```

Not robust

```
Filter::AllocateBuffer(int n) {
  DeallocateBuffer();
  buf = new int [n];
  if (buf == 0) {
     cerr << “allocation failed”;
     exit(0);
  }
  memset(buf, 0, n* sizeof(int));
}
Filter::DeallocateBuffer() {
  delete [] buf;
  buf = 0;
}
```

Robust (keep constructor and destructor)

C/C++ Coding Tips

- Static string length

```
#define  KEYSTR   "MarketShare"
#define  STATIC_STRLEN(s)  (sizeof(s) - 1)
strncmp(strBuf, KEYSTR, STATIC_STRLEN(KEYSTR)) == 0)
```

- Dynamic string length

```
#define  IS_STRING_NULL(s)       (! *(s))
#define  IS_STRING_NOT_NULL  (*(s))
```

```
char* robustGetString(char* bufStr, int bufLen, FILE* filePtr) {
   char *strPtr = fgets(bufStr, bufLen-1, filePtr);
   bufStr[bufLen-1] = '\0';
   bufStrLen = strlen(bufStr);
    if ( bufStr[bufStrLen-1] == '\n' ) bufStr[bufStrLen - 1] = '\0';
   return(strPtr);
}
```

Conclusion

- Printer pipeline
  - RGB to YCrCb conversion
  - JPEG compression and decompression
  - Document segmentation and enhancement
  - YCrCb to RGB to CMYK conversion
  - Interpolation (e.g nearest neighbor or bilinear)
  - Halftoning (e.g. ordered dither or error diffusion)
- Split embedded software systems
  - C++ for non-real-time tasks: GUIs and file input/output
  - C for low-level image processing operations
  - ANSI C can be cross-compiled onto DSPs
  - Program C code to work with blocks or rows because embedded processors have little on-chip memory

Conclusion

- Web resources
  - comp.dsp newsgroup: FAQ www.bdti.com/faq/dsp_faq.html
  - embedded processors and systems: www.eg3.com
  - on-line courses and DSP boards: www.techonline.com
  - software development: www.ece.utexas.edu/~bevans/talks/software_development
  - TI color laser printer xStream technology www.ti.com/sc/docs/dsps/xstream/index.htm
- References

  - B. L. Evans, "Software Development in the Unix Environment". http://www.ece.utexas.edu/~bevans/talks/software_development/
  - B. L. Evans, "EE379K-17 Real-Time DSP Laboratory," UT Austin. http://www.ece.utexas.edu/~bevans/courses/realtime/
  - B. L. Evans, "EE382C Embedded Software Systems," UT Austin. http://www.ece.utexas.edu/~bevans/courses/ee382c/