VLIB IMPLEMENTATION ON EVE CO-PROCESSOR

TEXAS INSTRUMENTS INC.

Table of Contents

NOTE: It should be noted that some APIs in this document contains performance and code size information – These are only theoretical estimates, Please refer to the data sheet for the actual performance and the code size.

1. L1 distance

Description

L1 Distance, also called city block distance, is a measure of the distance between two vectors. This function accepts as input two vectors, p and q, of size N. It returns the L1 distance, L1D, between p and q as a 32-bit unsigned integer as in the formula below. Refer vcop_vec_array_l1_distance function in package for more details

$$L1D = \sum_{i=1}^{N} | p(i) - q(i) |$$

Usage

Input vectors are of the type short and output value is a single integer value.

Constraints

1. array_len must be a multiple of 16
2. The VCOP kernel provides 16 sum value instead of single value, user need to sum up these 16 values in ARP32 after calling this kernel

Performance Considerations

Command Decode Time: 2 * vector core command length = 2 * 11 = 22 cycles.
Parameter Fetch Time: 12 + ceiling ((num_param_words/8) = 12 + 1 = 13 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + (array_len)*2/16
Total Time: (array_len)*2/16 + 51 overhead cycles.

Code size = 44 bytes

Techniques

- De-interleaved loads are used to load 16 elements in one instruction.
- VSAD instruction is leveraged to do subtraction, abs and accumulation in one instruction.

2. 2D Gradient Filtering

Description

For each pixel in the image, the 2nd step in Canny edge detection extracts the horizontal and vertical 1st order gradients along with an approximation of the gradient magnitude. Gradients are 2D vectors which point in the direction of the greatest rate of change, in this case, in intensity. This function extracts the 2D gradient vector coordinates as well as magnitude.

The first order 3×3 gradient filter calculates the first derivative in both the horizontal and vertical directions, Gx and Gy, respectively. So for the image pixel I(x,y), we calculate the gradients and the magnitude as shown below.

$$Gx = I(x+1,y) - I(x-1,y)$$
$$Gy = I(x,y+1) - I(x,y-1)$$
$$Gmag = (|Gx| + |Gy|)$$

Usage

Input is an 8 bit image and 3 short output images are produced. The final output is of size width * height.
There is a separate kernel which uses gradient and edgemap as input and returns a list of gradientX and gradient Y at the points where an edge is present and gives output in packed format of (GradX, GradY) along with the list of edge pixels

API

```
void vcop_gradients_xy_list
(
   __vptr_uint8   pIn,
   __vptr_uint32  pUpperLeftXY,
   __vptr_uint8   pEdgeMap,
   __vptr_uint32  xSequence_C,
   __vptr_int32   pGradXY,
   __vptr_uint32  pEdgeListXY,
   __vptr_uint16  pListSize,
   unsigned short width,
   unsigned short height,
   unsigned short pitchInData,
   unsigned short pitchEdgeMap
)
```

Field	Data Type	Input/ Output	Description

Field	Data Type	Input/ Output	Description
pIn	*__vptr_uint8*	Input	Pointer the the input gray scale buffer (8bit data). Size of this buffer should be width * height
pUpperLeftXY	*__vptr_uint32*	Input	This is pointer to the buffer which contains the upper left corner coordinates Size of this array should be sizeof(uint32_t)
pEdgeMap	*__vptr_uint8*	Input	Pointer to the binary image containing 1 at locations where edges are present. Size of this buffer should be width * height
xSequence_C	*__vptr_uint32*	Input	Pointer to pre-calculated sequence from 0 to width -1 left shifted by 16. Size of this buffer should width * sizeof(uint32_t)
pGradXY	`__vptr_int32`	Output	Pointer to the buffer output gradient buffer containing gradient in both X and Y direction in packed format for all the pixels for which edgeMap = 1. Output is stored like (gx<<16) | gy Size of this buffer should be (width * height * sizeof(int32_t))
pEdgeListXY	`__vptr_uint32`	Output	Pointer to the buffer which will contain the x and y coordinates stored in packed format in a 32 bit container. (x<<16) | y. Size of this buffer should be width * height * sizeof(uint32_t)).
pListSize	`__vptr_uint16`	Output	Pointer to the buffer which will contain the size of the list of edges. * Size of this buffer should be sizeof(uint16_t) * 8
width	`uint16_t`	Input	Width of the input buffer
height	`uint16_t`	Input	Height of the input buffer
pitchInData	`uint16_t`	Input	Pitch of the input buffer
pitchEdgeMap	`uint16_t`	Input	Pitch of the edgeMap buffer. The (0,0) pixel of this buffer should correspond to (1,1) pixel in image buffer

Constraints

1. The total number of output pixels (height*width) must be multiple of 16
2. Output pointers Gx, Gy and Gmag must be word aligned.

Performance Considerations

Command Decode Time: 2 * vector core command length = 2 * 20 = 40 cycles.
Parameter Fetch Time: 12 + ceiling ((num_param_words/8) = 12 + 2 = 14 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + (w * h)*5/16
Total Time: (w * h)*5/16 + 70 overhead cycles.

Code size = 80 bytes

Techniques

- De-interleaved loads are used to load 16 elements in one cycle.
- The available two compute units are used for each operation in parallel.
- Interleaved stores are used to store 16 elements in one cycle.

```
void vcop_vec_gradients_xy_and_magnitude_cn
(
    unsigned char pIn[],
    short pGradX[],
    short pGradY[],
    short pMag[],
    unsigned short width,
    unsigned short height
)
{
    int i4;
    unsigned int inT, inL, inR, inB;

    for (i4 = 0; i4 < width*height; i4++)
    {
        inT = pIn[i4+1];
        inL = pIn[i4+width];
        inR = pIn[i4+width+2];
        inB = pIn[i4+2*width+1];

        pGradX[i4] =  inR - inL;
        pGradY[i4] =  inB - inT;
        pMag[i4]   =  abs(pGradX[i4]) + abs(pGradY[i4]);
    }
}
```

Figure 2-1 Natural C code for 2D Gradient Filtering

```
void vcop_vec_gradients_xy_and_magnitude
(
    __vptr_uint8   pIn,
    __vptr_int16   pGradX,
    __vptr_int16   pGradY,
    __vptr_int16   pMag,
    unsigned short width,
    unsigned short height
)
{
    __vector VinT1,VinT2;               //Top pixel
    __vector VinL1,VinL2;               //Left pixel
    __vector VinR1,VinR2;               //Right pixel
    __vector VinB1,VinB2;               //Bottom pixel
    __vector VgX_1,VgX_2;               //Gx
    __vector VgY_1,VgY_2;               //Gy
    __vector Vabs_gX_1,Vabs_gX_2;       //abs of Gx
    __vector Vabs_gY_1,Vabs_gY_2;       //abs of Gy
    __vector Vmag1,Vmag2;               //Mag

    for (int I1 = 0; I1 < width*height/(2*VCOP_SIMD_WIDTH); I1++)
    {
        __agen Addr1,Addr2;

        Addr1 = I1*VECTORSZ*2;
        Addr2 = I1*VECTORSZ*4;

        (VinT1,VinT2) = (pIn+1)         [Addr1].deinterleave();
        (VinL1,VinL2) = (pIn+width)     [Addr1].deinterleave();
        (VinR1,VinR2) = (pIn+width+2)   [Addr1].deinterleave();
        (VinB1,VinB2) = (pIn+2*width+1)[Addr1].deinterleave();
```

```
            VgX_1 = VinR1 - VinL1;
            VgY_1 = VinB1 - VinT1;

            VgX_2 = VinR2 - VinL2;
            VgY_2 = VinB2 - VinT2;

            Vabs_gX_1 = abs(VgX_1);
            Vabs_gY_1 = abs(VgY_1);

            Vabs_gX_2 = abs(VgX_2);
            Vabs_gY_2 = abs(VgY_2);

            Vmag1     =  Vabs_gX_1 + Vabs_gY_1;
            Vmag2     =  Vabs_gX_2 + Vabs_gY_2;

            pGradX[Addr2].interleave() = (VgX_1,VgX_2);
            pGradY[Addr2].interleave() = (VgY_1,VgY_2);
            pMag[Addr2].interleave()   = (Vmag1,Vmag2);
        }
    }
```

Figure 2-2 VCOP_Kernel C code for 2D Gradient Filtering

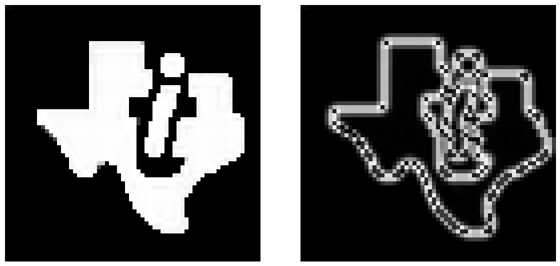


Figure 2-3 2D Gradient Filtering – Input and Output Gradient Image

3. Normalized Gradient

Description

These set for kernel together finds normalized gradient. The input to these kernel is gradient in X and Y direction. Normalized gradient is calculated as follows :

$$\vec{g} = \langle gX, gY \rangle$$

$$gM = \|\vec{g}\| = \sqrt{gX^2 + gY^2}$$

$$\hat{g} = \frac{\vec{g}}{\|\vec{g}\|} = \frac{\langle gX, gY \rangle}{\sqrt{gX^2 + gY^2}}$$

The above equation involves calculating square root. For this we are using following Lookup Table approach which is explained in Section 6.5 in EVE programmer's Guide.

Usage

Calculation of normalized gradient is split into 3 kernels :

vcop_gradients_xy_mag_lut_index_calulcation : This kernel calculates the index to be used for lookup in reciprocal square root table. It also outputs (int)log4(a)

vcop_reciprocal_sqrt_lookup : This kernel uses the location from the previous kernel and does a 8 way table look up to get the value at that location.

vcop_gradients_xy_unit_vecs : This kernel uses the above two outputs and calculates the normalized gradients

API

```
void vcop_gradients_xy_mag_lut_index_calulcation
(
  __vptr_int16  gradXY,
  __vptr_uint16 lutIdxPtr,
  __vptr_int8   log4aPtr,
  unsigned short width,
  unsigned short height,
  unsigned short pitch
)
```

Field	*Data Type*	*Input/ Output*	*Description*
pGradXY	*__vptr_int16*	*Input*	*Pointer to the buffer output gradient buffer containing gradient in both X and Y direction in packed format for all the pixels for which edgeMap = 1. Output is stored like (gx<<16) | gy Size of this buffer should be (width * height * sizeof(int32_t))*
lutIdxPtr	*__vptr_uint16*	*Output*	*This is pointer to the buffer which will contain the value of (int)log4(a). This will be used to calculate the square root. Size of this array should be width * height * sizeof(uint8_t)*
log4aPtr	*__vptr_int8*	*Output*	*Pointer to the buffer which will contain the size of the list of edges. Size of this buffer should be sizeof(uint16_t) * 8*
width	*uint16_t*	*Input*	*Width of the input buffer*
height	*uint16_t*	*Input*	*Height of the input buffer*
pitch	*uint16_t*	*Input*	*Pitch of the input buffer*

void vcop_reciprocal_sqrt_lookup(
__vptr_uint16 lutIdxPtr,
__vptr_uint8 reciSqrtLut,
__vptr_uint8 reciSqrtLutOutput,
unsigned short listSize)

Field	*Data Type*	*Input/ Output*	*Description*
lutIdxPtr	*__vptr_uint16*	*Input*	*This is pointer to the buffer which will contain the index of the reciprocal sqaure root table Size of this array should be width * height * sizeof(int16_t)*
reciSqrtLut	*__vptr_uint8*	*Input*	*This is pointer to the 8 way lookup table. Size of this should be 768 * 8*
reciSqrtLutOutput	*__vptr_uint8*	*Output*	*This is pointer to the buffer which will contain the value after the lookup from the given table. Size of this array should be width * height * sizeof(uint8_t)*
listSize	*uint16_t*	*Input*	*Size of the list for which lookup is required*

```
void vcop_gradients_xy_unit_vecs
(
  __vptr_int16  gradXY,
  __vptr_int8   log4aPtr,
  __vptr_uint8  reciprocalLutOutput,
  __vptr_int16  unitXYptr,
   unsigned short listSize
)
```

Field	*Data Type*	*Input/ Output*	*Description*
gradXY	*__vptr_int16*	*Input*	*Pointer to the gradient buffer containing gradient in both X and Y direction in packed format for all the pixels for which edgeMap = 1. Gradients are stored as (gx<<16) | gy. Size of this buffer should be (width * height * sizeof(int32_t))*
log4aPtr	*__vptr_int8*	*Input*	*This is pointer to the buffer which contain the value of (int)log4(a). This will be used to calculate the square root. Size of this array should be width * height * sizeof(uint8_t)*
reciSqrtLutOutput	*__vptr_uint8*	*Input*	*This is pointer to the buffer which will contain the value after the lookup from the given table. Size of this array should be width * height * sizeof(uint8_t)*
unitXYptr	*__vptr_int16*	*Output*	*This is pointer to the buffer which will contain the normalized gradients in X and Y direction in packed format as (Ux<<16) | Uy;Size of this array should be width * height * sizeof(u32nt8_t*
listSize	*uint16_t*	*Input*	*Size of the list for which lookup is required*

Constraints

1. NONE

Performance Considerations

vcop_gradients_xy_mag_lut_index_calulcation :
This loop is compute bound so buffer placement is not relevant

vcop_reciprocal_sqrt_lookup
Following is the buffer placement assumed for optimal performance of this kernel

```
lutIdxPtr       -> A Copy
reciSqrtLut     -> C Copy
reciSqrtLutOutput -> B Copy
```

vcop_gradients_xy_unit_vecs

```
gradXY    -> C Copy
log4aPtr     -> A Copy
reciprocalLutOutput -> B Copy
unitXYptr -> C Copy
```

4. Dilation

Description

Dilation is an elementary morphological operation. By itself, dilation expands objects in an image and is commonly used to connect neighboring objects before the connected components analysis. In conjunction with erosion, it is used to build other morphological operations, such as opening closing, top hat, bottom hat and morph diff.

These functions use either bit-packed (each pixel is represented by a bit) binary images or grayscale images. To detail binary images, if Pi is the pixel of image at location i along the width (horizontal direction), then the image will be in memory in the following format at ascending Bit locations: P7 P6 ..... P0 P15 P14 .... P8 P23 P22 …. P16 P31 P30 …. P24 ……

If 'in' is the input image, 'out' is the output image and 'mask' is the mask, the results are calculated using the equation below:

For Binary data:

out(u, v) = OR **(** **in**(u+i, v+j) AND **mask**(n+i ,m+j) **)**

For Grayscale data:

out(u, v) = MAX **(** **in**(u+i, v+j) AND **mask**(n+i ,m+j) **)**

In the above equation, the *logical summation* OR/MAX is done over
i = -(se_width-1/2) to (se_width-1/2) and
j = -(se_height-1/2) to (se_height-1/2), where se_width and se_height are the dimensions of the structuring element/mask and
(n,m) is the centre of the mask.

In addition to a general dilation function, there are two specific functions using the commonly used masks like rectangle and cross which offer a speedup over the generic dilation mask function.

Binary Data

API

All routines are C-callable and can be called as:

1. **Dilate Mask Kernel**

void vcop_vec_bin_image_dilate_mask
(
__vptr_uint32 pIn,
__vptr_uint32 mask0,
__vptr_uint32 mask1,
__vptr_uint32 mask2,

```
    __vptr_uint32 out,
    int         cols,
    int         pitch,
    int         height
)
```

- in : 32-bit packed input binary image
- out : 32-bit packed output binary image
- mask0 : Column 0 of 3*3 dilation kernel
- mask1 : Column 1 of 3*3 dilation kernel
- mask2 : Column 2 of 3*3 dilation kernel
- cols : Number of columns (bits) in the binary image
- pitch : Pitch of the binary image in terms of bits
- height : Number of rows in the binary image

- Returns : None or void.

2. Dilate Square Kernel

```
void vcop_vec_bin_image_dilate_square
(
    __vptr_uint32 pIn,
    __vptr_uint32 out,
    int         cols,
    int         pitch,
    int         height
)
```

- in : 32-bit packed input binary image
- out : 32-bit packed output binary image
- cols : Number of columns (bits) in the binary image
- pitch : Pitch of the binary image in terms of bits
- height : Number of rows in the binary image

- Returns : None or void.

3. Dilate Cross Kernel

```
void vcop_vec_bin_image_dilate_cross
(
    __vptr_uint32 pIn,
    __vptr_uint32 out,
    int         cols,
    int         pitch,
    int         height
)
```

- in : 32-bit packed input binary image
- out : 32-bit packed output binary image
- cols : Number of columns (bits) in the binary image
- pitch : Pitch of the binary image in terms of bits
- height : Number of rows in the binary image

- Returns : None or void.

Usage

This routine accepts 8-bit packed input binary image and performs dilation using a general/square/cross 3x3 kernel and writes as a 8-bit packed output binary image. Each binary image byte will have left most pixel 0 at LSB and right most pixel 7 at MSB of the byte. Therefore the first 4 bytes of the image in memory will be:
P7 P6 ... P0 P15 P14 ... P8 P23 P22 ... P16 P31 P30 ... P24
where Pi is the pixel at location i of binary image.

Constraints

- The pitch in the input binary image should be a multiple of 32. This is because word npt read (of pIn) has to be word aligned.
- For Generic Kernel, the mask should be in WBUF.

Performance Considerations

- Performance of Dilation Generic Kernel will be: 18/256 cycles per pixel after an overhead of 84 cycles.
- Performance of Dilation Cross Kernel will be: 6/256 cycles per pixel after an overhead of 82 cycles.
- Performance of Dilation Square Kernel will be: 6/256 cycles per pixel after an overhead of 80 cycles.

Techniques

- SHFOR is used heavily to perform shifts and ORs in a single cycle
- We use the same accumulator registers for efficient SHFOR pipelining which has a delay slot of 1 cycle.
- In Dilation Mask Kernel, we create an extra loop to do the 3x3 operations as 3 1x3 operations because of register pressure.

Grayscale Data

API

All routines are C-callable and can be called as:

1. **Dilate Mask Kernel**

```
void vcop_grayscale_dilate_mask
(
    unsigned short      blk_w,
    unsigned short      line_ofst,
    unsigned short      blk_h,
    __vptr_uint8        data_ptr,
    unsigned short      se_w,
    unsigned short      se_h,
    __vptr_uint8        se_pt,
    __vptr_uint8        output_ptr
)
```

- blk_w : Number of columns in the grayscale image
- line_ofst : Pitch of the image
- blk_h : Number of rows in the grayscale image
- data_ptr : input grayscale image
- se_w : width of structuring element
- se_h : height of structuring element
- se_pt : pointer to structuring element
- output_ptr : output grayscale image

- Returns : None or void.

2. **Dilate Rectangle Kernel**

```
void vcop_grayscale_dilate_rect
(
    unsigned short      blk_w,
    unsigned short      line_ofst,
    unsigned short      blk_h,
    __vptr_uint8        data_ptr,
    unsigned short      se_w,
    unsigned short      se_h,
    __vptr_uint8        scratch_ptr,
    __vptr_uint8        output_ptr
)
```

- blk_w : Number of columns in the grayscale image

- line_ofst : Pitch of the image
- blk_h : Number of rows in the grayscale image
- data_ptr : input grayscale image
- se_w : width of structuring element
- se_h : height of structuring element
- scratch_ptr: pointer to scratch buffer
- output_ptr : output grayscale image

- Returns : None or void.

3. Dilate Cross Kernel

```
void vcop_grayscale_dilate_cross
(
    unsigned short      blk_w,
    unsigned short      line_ofst,
    unsigned short      blk_h,
    __vptr_uint8        data_ptr,
    unsigned short      se_w,
    unsigned short      se_h,
    unsigned short      cross_se_row,
    unsigned short      cross_se_col,
    __vptr_uint8        scratch_ptr,
    __vptr_uint8        output_ptr
)
```

- blk_w : Number of columns in the grayscale image
- line_ofst : Pitch of the image
- blk_h : Number of rows in the grayscale image
- data_ptr : input grayscale image
- se_w : width of structuring element
- se_h : height of structuring element
- cross_se_row: row number of cross structuring element
- cross_se_col : column number of cross structuring element
- scratch_ptr: pointer to scratch buffer
- output_ptr : output grayscale image

- Returns : None or void.

Usage

All dilation kernels accepts input data in an array of unsigned char in "data_ptr", of width "blk_w" and height "blk_h" with each line having a line pitch of "line_ofst" elements, the SE size - width "se_w" and height of "se_h", filtering the input with the SE according to the variant and writing the result in an output array output_ptr of width "blk_w" elements per line, where each line has a line pitch of "line_ofst" and "blk_h" such lines.

Constraints

- For Generic Kernel, the mask should be in WBUF.

Performance Considerations

- For Dilation Rectangle and Cross Kernel, the scratch buffer should be in WBUF for optimum performance.
- For all Dilation kernels, the input and output buffers should be in different data buffers for optimum performance.

Techniques

- Operating on 16 elements at a time using deinterleave load.

5. Erosion

Description

Erosion, along with dilation, is an elementary morphological operation. By itself, erosion shrinks objects in an image and is commonly used to remove noise before further analysis. In conjunction with dilation, it is used to build other morphological operations, such as opening closing, top hat, bottom hat and morph diff.

These functions use either bit-packed (each pixel is represented by a bit) binary images or grayscale images. To detail binary images, if Pi is the pixel of image at location i along the width (horizontal direction), then the image will be in memory in the following format at ascending Bit locations: P7 P6 ..... P0 P15 P14 .... P8 P23 P22 …. P16 P31 P30 …. P24 ……

If 'in' is the input image, 'out' is the output image and 'mask' is the mask, the results are calculated using the equation below:

For Binary data:

out(u, v) = AND **(** **in**(u+i, v+j) AND **mask**(n+i ,m+j) **)**

For Grayscale data:

out(u, v) = MIN **(** **in**(u+i, v+j) AND **mask**(n+i ,m+j) **)**

In the above equation, the *logical product* AND/MAX is done over
i = -(se_width-1/2) to (se_width-1/2) and
j = -(se_height-1/2) to (se_height-1/2), where se_width and se_height are the dimensions of the structuring element/mask and
(n,m) is the centre of the mask.

In addition to a general erosion function, there are two specific functions using the commonly used masks like rectangle and cross which offer a speedup over the generic erosion mask function.

Binary Data

API

All routines are C-callable and can be called as:

4. Erosion Mask Kernel

```
void vcop_vec_bin_image_erode_mask
(
    __vptr_uint32 pIn,
    __vptr_uint32 mask0,
    __vptr_uint32 mask1,
    __vptr_uint32 mask2,
    __vptr_uint32 out,
    int          cols,
    int          pitch,
    int          height
)
```

- in : 32-bit packed input binary image
- out : 32-bit packed output binary image
- mask0 : Column 0 of 3*3 erosion kernel
- mask1 : Column 1 of 3*3 erosion kernel
- mask2 : Column 2 of 3*3 erosion kernel
- cols : Number of columns (bits) in the binary image
- pitch : Pitch of the binary image in terms of bits
- height : Number of rows in the binary image

- Returns : None or void.

5. Erosion Square Kernel

```
    void vcop_vec_bin_image_ erode_square
    (
        __vptr_uint32 pIn,
        __vptr_uint32 out,
        int          cols,
        int          pitch,
        int          height
    )
```

- in : 32-bit packed input binary image
- out : 32-bit packed output binary image
- cols : Number of columns (bits) in the binary image
- pitch : Pitch of the binary image in terms of bits
- height : Number of rows in the binary image

- Returns : None or void.

6. Erosion Cross Kernel

```
    void vcop_vec_bin_image_ erode_cross
    (
        __vptr_uint32 pIn,
        __vptr_uint32 out,
        int          cols,
        int          pitch,
        int          height
```

)

- in : 32-bit packed input binary image
- out : 32-bit packed output binary image
- cols : Number of columns (bits) in the binary image
- pitch : Pitch of the binary image in terms of bits
- height : Number of rows in the binary image

- Returns : None or void.

Usage

This routine accepts 8-bit packed input binary image and performs erosion using a general/square/cross 3x3 kernel and writes as a 8-bit packed output binary image. Each binary image byte will have left most pixel 0 at LSB and right most pixel 7 at MSB of the byte. Therefore the first 4 bytes of the image in memory will be:
P7 P6 ... P0 P15 P14 ... P8 P23 P22 ... P16 P31 P30 ... P24
where Pi is the pixel at location i of binary image.

Constraints

- The pitch in the input binary image should be a multiple of 32. This is because word npt read (of pIn) has to be word aligned.
- For Generic Kernel, the mask should be in WBUF.

Performance Considerations

- Performance of Erosion Generic Kernel will be: 18/256 cycles per pixel after an overhead of 84 cycles.
- Performance of Erosion Cross Kernel will be: 7/256 cycles per pixel after an overhead of 84 cycles.
- Performance of Erosion Square Kernel will be: 10/256 cycles per pixel after an overhead of 96 cycles.

Techniques

- In Erosion Mask Kernel, we create an extra loop to do the 3x3 operations as 3 1x3 operations because of register pressure.

Grayscale Data

API

All routines are C-callable and can be called as:

4. **Erosion Mask Kernel**

```
void vcop_grayscale_ erode_mask
(
    unsigned short      blk_w,
    unsigned short      line_ofst,
    unsigned short      blk_h,
    __vptr_uint8       data_ptr,
    unsigned short      se_w,
    unsigned short      se_h,
    __vptr_uint8       se_pt,
    __vptr_uint8       output_ptr
)
```

- blk_w : Number of columns in the grayscale image
- line_ofst : Pitch of the image
- blk_h : Number of rows in the grayscale image
- data_ptr : input grayscale image
- se_w : width of structuring element
- se_h : height of structuring element
- se_pt : pointer to structuring element
- output_ptr : output grayscale image

- Returns : None or void.

5. **Erosion Rectangle Kernel**

```
void vcop_grayscale_ erode_rect
(
    unsigned short      blk_w,
    unsigned short      line_ofst,
    unsigned short      blk_h,
    __vptr_uint8       data_ptr,
    unsigned short      se_w,
    unsigned short      se_h,
    __vptr_uint8       scratch_ptr,
    __vptr_uint8       output_ptr
)
```

- blk_w : Number of columns in the grayscale image

- line_ofst : Pitch of the image
- blk_h : Number of rows in the grayscale image
- data_ptr : input grayscale image
- se_w : width of structuring element
- se_h : height of structuring element
- scratch_ptr: pointer to scratch buffer
- output_ptr : output grayscale image

- Returns : None or void.

6. Erosion Cross Kernel

```
void vcop_grayscale_ erode_cross
(
    unsigned short      blk_w,
    unsigned short      line_ofst,
    unsigned short      blk_h,
    __vptr_uint8        data_ptr,
    unsigned short      se_w,
    unsigned short      se_h,
    unsigned short      cross_se_row,
    unsigned short      cross_se_col,
    __vptr_uint8        scratch_ptr,
    __vptr_uint8        output_ptr
)
```

- blk_w : Number of columns in the grayscale image
- line_ofst : Pitch of the image
- blk_h : Number of rows in the grayscale image
- data_ptr : input grayscale image
- se_w : width of structuring element
- se_h : height of structuring element
- cross_se_row: row number of cross structuring element
- cross_se_col : column number of cross structuring element
- scratch_ptr: pointer to scratch buffer
- output_ptr : output grayscale image

- Returns : None or void.

Usage

All erosion kernels accepts input data in an array of unsigned char in "data_ptr", of width "blk_w" and height "blk_h" with each line having a line pitch of "line_ofst" elements, the SE size - width "se_w" and height of "se_h", filtering the input with the SE according to the variant and writing the result in an output array output_ptr of width "blk_w" elements per line, where each line has a line pitch of "line_ofst" and "blk_h" such lines.

Constraints

- For Generic Kernel, the mask should be in WBUF.

Performance Considerations

- For Erosion Rectangle and Cross Kernel, the scratch buffer should be in WBUF for optimum performance.
- For all Erosion kernels, the input and output buffers should be in different data buffers for optimum performance.

Techniques

- Operating on 16 elements at a time using deinterleave load.

6. Exponentially-Weighted Running Mean

Description

A background subtraction algorithm might consist of:
1. Computing a representative statistic of the luma component for each pixel in a video.
2. Labeling deviations from this statistic as foreground. One such statistic is the exponentially-weighted (EW) running mean.

This function updates the exponential running mean of the luma component of a video. If the foreground mask bit is set, indicating there is obstruction by a foreground object, the running mean will not be updated.

The following update equation is used for those pixels where the foreground mask is zero:

$$\text{updatedMean} = (1 - \text{weight}) \times \text{previousMean} + \text{weight} \times \text{newestData}$$

Usage

Inputs are 16-bit runningMean estimate from previous image, 8-bit current image and 32-bit packed previous estimate of foreground. The output is updated in the same runningMean array. The input and output sizes are the same as the number of pixels.

Constraints

1. Number of pixels should be a multiple of 8.

Performance Considerations

Command Decode Time: 2 * vector core command length = 2 * 21 = 42 cycles.
Parameter Fetch Time: 12 + ceiling ((num_param_words/8) = 12 + 2 = 14 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + (frameSize) /2
Total Time: (frameSize) /8 + 72 overhead cycles.

Code size = 84 bytes

Techniques

- Unpack instruction and predicated stores were leveraged.

7. Exponentially-Weighted Running Variance

Description

A background subtraction algorithm might consist of:
1. Computing a representative statistic of the luma component for each pixel in a video.
2. Labeling deviations from this statistic as foreground. One such statistic is the exponentially-weighted (EW) running variance.

This function updates the exponential running variance of the luma component of a video. If the foreground mask bit is set, indicating there is obstruction by a foreground object, the running variance will not be updated.

The following update equation is used for those pixels where the foreground mask is zero:

$$\text{updatedVar} = (1 - \text{weight}) \times \text{previousVar} + \text{weight} \times (\text{newestData} - \text{previousMean})^2$$

Usage

Inputs are 16-bit runningVar estimate from previous image, 16-bit runningMean estimate from current image, 8-bit current image and 32-bit packed previous estimate of foreground. The output is updated in the same runningVar array. The input and output sizes are the same as the number of pixels.

Constraints

1. Number of pixels should be a multiple of 8.

Performance Considerations

Command Decode Time: 2 * vector core command length = 2 * 27 = 54 cycles.
Parameter Fetch Time: 12 + ceiling ((num_param_words/8) = 12 + 2 = 14 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + (frameSize)*9/8
Total Time: (frameSize) /8 + 84 overhead cycles.

Code size = 108 bytes

Techniques

- Unpack instruction and predicated stores were leveraged.

8. Gaussian 5x5 Pyramid Kernel – 8-bit and 16-bit

Description

Gaussian image pyramid is a data structure consisting of the original image at level 0, 2x2 sub-sampled image at Level 1, further 2x2 sub-sampled image at Level 2, etc. It is commonly used in detection and tracking applications to reduce the amount of processing.

This function can be used to calculate the next level of a pyramid. Given a pointer to a rectangular region of interest described by W (input data width), P (input data pitch), and H (input data height), this kernel returns (W-4)/2 x (H-3)/2 values. For example, if H=5, it will calculate a single row of results. The anti-aliasing filter used at each step is a binomial approximation to the 5x5 Gaussian filter given by:

$$\begin{vmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{vmatrix} / 256$$

Usage

There are two APIs to perform horizontal and vertical filtering respectively.An intermediate buffer pB is used to hold the results from horizontal filtering and it is passed on to the vertical filtering. The input is an 8-bit or 16-bit image and output image is half the size of the input image. The function has to be run multiple times to generate multiple levels of the pyramid.

Constraints

1. Number of columns should be a multiple of 8.

Performance Considerations

8-bit & 16-bit versions:
Command Decode Time: 2 * vector core command length = 2 * 32 = 64 cycles.
Parameter Fetch Time: 12 + ceiling ((num_param_words/8) = 12 + 3 = 15 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + (pixelCount) *9.5/8
Total Time: (pixelCount) *9.5/8 + 95 overhead cycles.

Code size = 128 bytes

Techniques

- Multiply and Accumulate pipeline is leveraged.
- Store with truncation of bits is used.

9. Non-Maximum Suppression – Canny Edge Detection

Description

As the third stage in Canny Edge Detection, non-maximum suppression identifies potential edge pixels. It suppresses all pixels whose edge strength is not a local maximum along the gradient direction. The function creates an 8-bit edge map labeling each pixel location as a non-Edge (0) or possible-edge (1) or an edge (255).

This function creates an 8-bit edge map that labels each pixel either as a non-edge (0) or a possible-edge (1). For each pixel location, the gradient direction is established. Two virtual points, say at a and b lying along the gradient direction on either side of the current location c are interpolated using the gradient magnitudes from surrounding neighbors. Locations that achieve a local maximum are regarded as possible edges, such as, Gmag(c) > Gmag(a) AND Gmag(c) >= Gmag(b); otherwise, these points are declared non-edges.

Usage

The API uses the 16-bit Gx, Gy and Gmag as input arrays and produces an 8-bit image containing values of 1 in locations of possible maximas suprresing false maximas and 255 at location where there is a strong edge. This functionality is split into three kernels :

vcop_canny_bin_indexing :

This kernel does the binning of edges into 4 cases:

Case 1: Edges lying between -22.5 to +22.5 : output Index < 4
Case 2: Edges lying between +22.5 to +67.5 : output Index == 4
Case 3: Edges lying between +67.5 to +112.5 : output Index > 5
Case 4: Edges lying between +112.5 to +157.5 : output Index == 5
This kernel uses following three conditions
Condition 1 : abs(gy) > gx * tan(22.5)
Condition 1 : abs(gy) > gx * tan(67.5)
Condition 3 : Gx^Gy < 0

Following is the truth tables implemented in this kernel:

	Condition1	Condition2	Condition3
Case 1	0	0	0
Case 1	0	0	1
Case 1	0	1	0
Case 1	0	1	1
Case 2	1	0	0
Case 4	1	0	1
Case 3	1	1	0
Case 3	1	1	1

vcop_canny_nms_max_cases :

This kernel finds the maximum of all the pixel for all 4 cases along the Direction of edge. 4 cases are described as :

Case 1: Edges lying between -22.5 to +22.5 : output Index < 4
Case 2: Edges lying between +22.5 to +67.5 : output Index == 4
Case 3: Edges lying between +67.5 to +112.5 : output Index > 5
Case 4: Edges lying between +112.5 to +157.5 : output Index == 5

vcop_canny_nms_double_thresholding :

This kernel uses the previous two kernels to do NMS and also apply double threshold to give an image whose output is 0 for pixels which are below low threshold 1 for pixels which are above low threshold and below High Threshold 255 for pixels which are above High Threshold

Techniques

- Select instruction is leveraged multiple times.

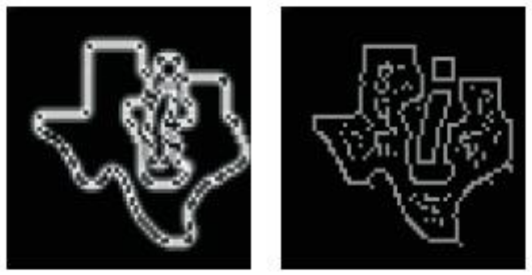


Figure 10-1 Input Gradient Image (Gmag) and Non-maximum Suppressed output image

10. Normal Flow (16-bit)

Description

Normal flow computes, for every pixel in the image, motion vectors parallel to the gradient direction at each pixel. Normal flow vectors, averaged over an image region, can provide useful information regarding the direction and magnitude of motion.
This function takes as input the x and y gradients, the gradient magnitude, and pixel-wise image difference and computes the normal flow vectors in the x and y directions. The fuctions uses the look-up table approach to compute the division of the magnitude of gradient values.

Usage

The API uses the 16-bit imDiff, Emag, Ex, Ey, T and LUT as input arrays and produces 16-bit arrays contianing normal flow values in the X and Y direction.

Constraints

1. The LUT (look-up table) array should hold values such that LUT[n] = X, where X is the value of 1/n represented in SQ0.15 format.
2. The threshold T, on gradient magnitude ensures that only those pixels with gradient magnitude greater than T will be processed. Normal flow values for pixels that do not pass the threshold will be 0.

Performance Considerations

Look-up Table:
8 lookup/cycle since we are using 8 table 1 point look up.

Compute Loop:
Command Decode Time: 2 * vector core command length = 2 * 22 = 44 cycles.
Parameter Fetch Time: 12 + ceiling (num_param_words/8) = 12+2 = 14 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + (width*height)/8
Total Time: (width*height)/8 + 74 overhead cycles.

Code size = 116 bytes

Techniques

- Use of LUT pipeline to compute the reciprocal of the magnitude of gradient values.
- Use of deinterleave load instruction.
- Use of predication to check for the threshold condition.

11. Gradient 5x5 Pyramid Kernel (8-bit)

Description

Gradient image pyramid is a data structire consisting of the original image at level 0, 2x2 subsampled gradient images at level 1, further 2x2 subsampled gradient images at level 2, etc. It is commoly used in detection and tracking, as well as in image fusion applications, in order to reduce the amount of processing.
The two functions for gradient pyramid are used for horizontal and vertical gradient filtering, respectively. These functions can be used to calculate the next level of a pyramid. Given a pointer to a rectangular region of interest described by W (input data width), P (input data pitch), and H (input data height), each of these kernels returns (W-4)/2 x (H-3)/2 values. For example, if H=5, each will calculate a single row of results. The filters used at each step are:

$$HS = \begin{matrix} -1 & -2 & 0 & 2 & 1 \\ -4 & -8 & 0 & 8 & 4 \\ -6 & -12 & 0 & 12 & 6 \\ -4 & -8 & 0 & 8 & 6 \\ -1 & -2 & 0 & 2 & 1 \end{matrix} \quad \text{(horizontal)}$$

$$VS = \begin{matrix} -1 & -4 & -6 & -4 & -1 \\ -2 & -8 & -12 & -8 & -2 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 8 & 12 & 8 & 2 \\ 1 & 4 & 6 & 4 & 1 \end{matrix} \quad \text{(vertical)}$$

After the filtering step, the intermediate results are rounded and scaled to values 0-255 (the output value of 128 indicates no gradient) as shown in equations below:

Gh = ((conv2(A,H5) + 64) >> 7) + 128;
Gv = ((conv2(A,V5) + 64) >> 7) + 128;

Usage

The API uses the 8-bit pIn input array, width, pitch, cols and rows parameters as inputs and produces 8-bit filtered output in pOut.

Constraints

1. Image width must be a multiple of 8.

Performance Considerations

Horizontal:

Command Decode Time: 2 * vector core command length = 2 * 38 = 76 cycles.
Parameter Fetch Time: 12 + ceiling ((num_param_words/8) = 12 + 3 = 15 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + 1.75*(pixelCount)
Total Time: 1.75*(pixelCount) + 107 overhead cycles.

Code size = 152 bytes

Vertical:
Command Decode Time: 2 * vector core command length = 2 * 37 = 74 cycles.
Parameter Fetch Time: 12 + ceiling ((num_param_words/8) = 12 + 3 = 15 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + 0.562*(pixelCount)
Total Time: 1. 749*(pixelCount) + 105 overhead cycles.

Code size = 148 bytes

Techniques

- The 5x5 gradient filer is implemented as horizontal and vertical filters because of kernel separability.
- VMAD and VADIF were leveraged optimally.

12. Non-Maxima Suppression

Description

Vision algorithms such as Harris Corner detection produce an intensity map or voting space for which the local maxima or peaks need to be found.

This function compares the value of each input pixel against its neighbors. For an output pixel to be "on" (numerical value=255), the input pixel value must be both:

- Greater than or equal to its neighbors' values
- Greater than the minimum threshold

If the above conditions are not met simultaneously, the output will be 0.

Usage

There are two versions this function, Non-maxima suppression (NMS) for 16-bit signed input and NMS for 32-bit signed input data. Both kernels can operate on any generic neighborhood size of mxn pixels.

Constraints

1. Width of the image must be a multiple of 8. There are no restrictions on height, m or n.

Performance Considerations

16-bit input version:
Command Decode Time: 2 * vector core command length = 2 * 61 = 122 cycles.
Parameter Fetch Time: 12 + ceiling ((num_param_words/8) = 12 + 5 = 17 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + 0.875*(pixelCount)
Total Time: 0.875*(pixelCount) + 155 overhead cycles.

Code size = 244 bytes

32-bit input version:
Command Decode Time: 2 * vector core command length = 2 * 51 = 102 cycles.
Parameter Fetch Time: 12 + ceiling ((num_param_words/8) = 12 + 4 = 16 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + 0.719*(pixelCount)
Total Time: 0.719*(pixelCount) + 134 overhead cycles.

Code size = 204 bytes

Techniques

- Separable implementation is used for computing window maxima.
- De-interleaved loads and interleaved stores were leveraged during horizontal filtering
- Processes two rows at a time to ensure both functional units are equally loaded during vertical filtering and supression.
- Predicated Stores was used to save an extra cycle.

13. Double Thresholding

Description

Vision algorithms such as Canny edge detection uses double threholding. This funciton accepts an edge map, with each location labeled with values of either 0 (non-edge) or 127 (possible-edge). It searches for locations where the magnitude is at or above the high threshold. Values in the edge map are modified from possible-edge (127) to edge (255).

This function compares the value of the edge map against low threshold and high threshold. If the current value of the edgemap is a possible edge (127) and the value is greater than low and high threshold, then adds 128 to it to make it an edge (255).

Usage

The API uses the 16-bit pInMag, edgeMap, width, pitch, height, loThresh and hiThresh as inputs and produces 16-bit arrays contianing edgemap_out values.

Performance Considerations

Compute Loop:
Command Decode Time: 2 * vector core command length = 2 * 10 = 20 cycles.
Parameter Fetch Time: 12 + ceiling (num_param_words/8) = 12+2 = 14 cycles.
Loop Execution Time: 16 cycles (pipeline ramp up/down) + (width*height)/8
Total Time: (width*height)/8 + 74 overhead cycles.

Code size = 68 bytes

Techniques

- Use of predication to check for the threshold condition.

14. Harris Corner Score

Description

Various vision algorithms operate by identifying salient image points and processing their neighborhoods. The Harris Score is a popular measure of saliency. It tends to find corner-like image textures, which are relatively easy to match between different views or to track in a video sequence. Computes the Harris corner score for each pixel in a luma image. As input, the function takes the horizontal and vertical gradients of the image. This gives flexibility to the user in selecting the scale for gradient computations.

For each pixel, the following equations together compute the 2×2 gradient covariance matrix M, where the summations are over n×n pixel neighborhoods:

$$M(1,1) = \mathrm{sum}(\mathrm{gradX})^2$$
$$M(1,2) = M(2,1) = \mathrm{sum}(\mathrm{gradX} \times \mathrm{gradY})$$
$$M(2,2) = \mathrm{sum}(\mathrm{gradY})^2$$

The cornerness score is defined as det(M) – k × trace(M)^2, where k is a tunable sensitivity parameter, typically around 0.04.
There is another score defined for the purpose of detecting edges along with the corners and the score is defined as trace(M). Lets call it as method B for Harris Score calculation.

For the original Harris Score defined as det(M) – k × trace(M)^2, the underlying score computation is of 64-bit precision. To save memory, an approximation of the binary log of this value is stored in the output. Order relationships between different score values is maintain in this format, up to quantization limits. Such a format is superior compared to simple rounding in terms of quantization errors as the below format preserves maximum number of bits from the original score.

Harris Score 16 bit only supports positive scores and if actual score negative it will make it zero. 32bit harris score supports both positive and negative score

The exact format of Harris score is as follows:

Harris Score 16-bit format(Only +ve score) :

Exponent (6 bit)	Mantissa (10 bit)

Harris Score 32-bit format(Both +ve and –ve score):

Exponent (6 bit)	Mantissa (26 bit)

Table below explains the compression scheme used for 16bit Harris score :
TOTAL_BITS = 47
MANTISSA_BITS = 10

Input Range (TOTAL_BITS = 47 bits)	**Output Range (16bits)**	**Translation**
[-pow(2,TOTAL_BITS-1), -pow(2, MANTISSA_BITS)]	0	outputvalue = 0
[-pow(2, MANTISSA_BITS) + 1, -1]	0	outputvalue = 0
[0, pow(2, MANTISSA_BITS)-1]	[0, pow(2, MANTISSA_BITS) -1]	outputvalue = inputvalue

[pow(2, MANTISSA_BITS), pow(2, TOTAL_BITS-1) - 1]	[pow(2, MANTISSA_BITS), pow(2, TOTAL_BITS-1) - 1]	outputvalue = ((exp << MANTISSA_BITS) | (inputvalue >> shift))
		shift = Number of significant bits - MANTISSA_BITS
		exp = shift

Table below explains the compression scheme used for 32bit Harris score :
TOTAL_BITS = 47
MANTISSA_BITS = 26

Input Range (TOTAL_BITS = 47 bits)	**Output Range (16bits)**	**Translation**

[-pow(2,TOTAL_BITS-1), -pow(2, MANTISSA_BITS)]	[-pow(2,TOTAL_BITS-1), -pow(2, MANTISSA_BITS)]	outputvalue = ((exp << MANTISSA_BITS) | (inputvalue >> shift))
		shift = Number of significant bits - MANTISSA_BITS
		exp = -(shift +1)

[-pow(2, MANTISSA_BITS) + 1, -1]	[-pow(2, MANTISSA_BITS) + 1, -1]	outputvalue = inputvalue
[0, pow(2, MANTISSA_BITS)-1]	[0, pow(2, MANTISSA_BITS) -1]	outputvalue = inputvalue

[pow(2, MANTISSA_BITS), pow(2, TOTAL_BITS-1) - 1]	[pow(2, MANTISSA_BITS), pow(2, TOTAL_BITS-1) - 1]	outputvalue = ((exp << MANTISSA_BITS) | (inputvalue >> shift))
		shift = Number of significant bits - MANTISSA_BITS
		exp = shift

The exponent corresponds to the location of the left most significant leading bit (one for positive numbers and zero for negative numbers) in the score. Mantissa contains the relevant bits (10 bits for 16-bit Harris score and 26 bits for 32-bit Harris Score) starting from the left most significant leading bit.

For example, a score of 571717286 (= 0010 0010 0001 0011 1011 0110 1010 0110b), will be encoded by the 32-bit kernel as = (30-26)* 2^{26} + (571717286) >> 4 = 304167786.
For example, a score of 571717286 (= 0010 0010 0001 0011 1011 0110 1010 0110b), will be encoded by the 32-bit kernel as = -(30-26 + 1)* 2^{26} +((-571717286) >> 4 &) = -304167787.

Usage

The vcop_harrisScore_7x7() API uses 16-bit gradX, gradY as inputs and produces 16-bit compressed Harris Score values whereas the vcop_harrisScore_u32_7x7() API produces a 32-bit Harris score for the same input.
If user want to use score defined in method B as trace(M), then there is a separate API for it with name vcop_harrisScore_32_methodB. This routine takes 16bit gradX and gradY as input and produces a 32 bit Harris Score.

API

```
void vcop_harrisScore_32_methodB
(
   __vptr_int16 gradX,
   __vptr_int16 gradY,
   __vptr_uint32 scratchXX,
   __vptr_uint32 scratchYY,
   unsigned short inBlockWidth,
   unsigned short inBlockHeight,
   unsigned short srcPitch,
   unsigned short dstPitch,
   unsigned char  windowSize,
   __vptr_uint32  outm
)
```

Field	Data Type	Input/ Output	Description
gradX	*__vptr_int16*	Input	Pointer to the gradient in X direction. Size of this buffer should be inBlockWidth * inBlockHeight * sizeof(uint16_t)
gradY	*__vptr_int16*	Input	Pointer to the gradient in Y direction. Size of this buffer should be inBlockWidth * inBlockHeight * sizeof(uint16_t)
scratchXX	*__vptr_uint32*	Scratch	This is pointer to an intermediate scratch buffer. Size of this buffer should be (inBlockWidth * (inBlockHeight + 1) * uint32_t)

Field	Data Type	Input/ Output	Description
scratchYY	*__vptr_uint32*	Scratch	This is pointer to an intermediate scratch buffer. Size of this buffer should be (inBlockWidth * (inBlockHeight + 1) * uint32_t)
inBlockWidth	`uint16_t`	Input	Width of the input block
inBlockHeight	`uint16_t`	Input	Height of the input block.
srcPitch	`uint16_t`	Input	Pitch of the input block
dstPitch	`uint16_t`	Input	Pitch for the output score
windowSize	`uint16_t`	Input	Size of the window. Should be an odd number

15. Bhattacharya Distance

Description

Bhattacharya distance is a popular measure of the similarity between two discrete probability distribution functions.

If p and q are two discrete probability distributions containing N elements each, the Bhattacharya Distance measure is computed as

$$\left(1 - \sum_{i=1}^{N} \sqrt{p(i) \times q(i)}\right)^{1/2}$$

Usage

The API uses the 16-bit X, Y and N as inputs and produces the distance (UBD = unsigned Bhattacharya Distance) as output.

Performance Considerations

Approximate Performance = **13.68** cycles/pixel + overheads

Techniques

- Uses Table look up technique to compute square root of the probability distributions.

16. Hough transform for Lines

Description

Hough Transform for line is very commonly used feature extraction technique in many image processing and computer vision applications. It is typically used after edge detection to determine the most dominant lines in an edge image. Hough Transform groups edge points into line candidates by performing an explicit voting in the parameter space.

Usage

The Hough Transform algorithm uses an accumulator array called the Hough Space to detect the existence of lines in an image. This accumulator array spans the complete (rho, theta) space. This particular kernel only gives the voted rho array for a given theta. User is supposed to call this kernel for multiple theta to get the full (rho, theta) space.

API

```
void vcop_hough_for_lines
(
  __vptr_uint16        pEdgeMapList,
  __vptr_int16         pCosSinThetaMulNormQ15,
  __vptr_uint16        intermIndexArray,
  __vptr_uint16        votedRhoArray8Copy,
  unsigned short       listSize,
  unsigned short       rhoMaxLength
)
```

Field	**Data Type**	**Input/ Output**	**Description**
pEdgeMapList	*__vptr_uint16*	Input	Pointer to the edge list which is in packed format with x coordinate followed by y. Both x and 6 are 16 bit quantity. Size of this buffer should be listSize * 2 * sizeof(uint16_t)
pCosSinThetaMulNormQ15	*__vptr_int16*	Input	This is pointer to the buffer which contains precalculated values of cos(theta) * normFactor followed by sin(theta) * normactor which are signed quantity in Q15 format. Where normactor = (rhoMaxLength / (2 * diameter) diameter = sqrt(imgWidth^2 + imgHeight ^2) which can be approximated to sqrt(2) * max (imgWidth, imgHeight). Size of this array should be 2 * sizeof(uint16_t)

Field	Data Type	Input/ Output	Description
intermIndexArray	*__vptr_uint16*	Scratch	This is pointer to an intermediate scratch buffer which contains the rho values calculated for each edge point in the list. Size of this buffer should be (listsize * uint16_t)
votedRhoArray8Copy	*__vptr_uint16*	output	Pointer to the buffer which will store the 8 copies of voted rho (per theta). Size of this buffer should be (rhoMaxLength * 2 * 8)
listSize	`uint16_t`	Input	Size of edge list in terms on number of edges. Should be multiple of 16
rhoMaxLength	`uint16_t`	Input	Maximum value which rho could take.

```
void vcop_merge_voted_rho_array
(
  __vptr_uint16   votedRhoArray8Copy,
  __vptr_uint16   votedRhoArray,
  __vptr_uint16   interimTransposeBuf1,
  __vptr_uint16   interimTransposeBuf2,
  __vptr_uint16   offsetArray,
  unsigned short  rhoMaxLength
)
```

Field	Data Type	Input/ Output	Description
votedRhoArray8Copy	*__vptr_uint16*	Input	Pointer to the buffer which contain the 8 copies of voted rho (per theta). Size of this buffer should be (rhoMaxLength * (uint16_t) * 8)
votedRhoArray	*__vptr_uint16*	Input	Pointer to the buffer containing which voted rho (per theta) that needs to get updated. Size of this buffer should be (rhoMaxLength * sizeof(uint16_t)))

Field	Data Type	Input/ Output	Description
offsetArray	*__vptr_uint16*	Input	This is pointer to the buffer which contains precalculated offsets for scatter store. The offsets are chosen such that after scatter store all enteries should go in a different bank. It is observed that if we chose any odd number (in terms of words) greater than 8 then it will automatically result into scatter store enteries to go into all different banks. This offsets can be found by first caluclating the stride considering the array is of rhoMaxLength. We have split the intermediate transpose buffers into 2 hence for rhoMaxLength array total bytes = rhoMaxLength * sizeof(uint16_t). Bytes per intermediate transpose buffer will be rhoMaxLength * sizeof(uint16_t)/2 . Now first convert it to numer of words = (rhoMaxLength * sizeof(uint16_t)) / 2)/ 4. Now we can choose next odd word number lets say it is transposeStrideInWords interimTransposeStride = transposeStrideInWords * 4. OffsetArray should contain interimTransposeStride for all 8 elements (interimTransposeStride * i) where i 0,1....7 Size of this array should be 8 * sizeof(uint16_t)
interimTransposeBuf1	*__vptr_uint16*	Scratch	This is pointer to an intermediate scratch buffer to store the transpose for each edge point in the list. Size of this buffer should be (interimTransposeStride * 8)
interimTransposeBuf2	*__vptr_uint16*	Scratch	This is pointer to an intermediate scratch buffer to store the transpose for each edge point in the list. Size of this buffer should be (interimTransposeStride * 8)
rhoMaxLength	`uint16_t`	Input	Maximum value which rho could take. Should be multiple of 16

Performance Considerations

To get the best performance, the buffer placement is very important – below table summarize the suggested placement of buffers

void vcop_hough_for_lines

Parameter	**Memory area**	**size (bytes)**
pEdgeMapList	VCOP_IBUFLA	listSize * sizeof(uint32_t)
pCosSinThetaMulNormQ15	VCOP_WMEM	2* sizeof(uint16_t)
intermIndexArray	VCOP_IBUFHA	listSize * sizeof(uint16_t)
votedRhoArray8Copy	VCOP_WMEM	8 * rhoMaxLength * sizeof(uint16_t)

vcop_merge_voted_rho_array

Parameter	**Memory area**	**size (bytes)**
votedRhoArray8Copy	VCOP_WMEM	8 * rhoMaxLength * sizeof(uint16_t)
votedRhoArray	VCOP_IBUFHA	rhoMaxLength * sizeof(uint16_t)
interimTransposeBuf1	VCOP_IBUFHA	transposeStride * 8
interimTransposeBuf2	VCOP_WMEM	transposeStride * 8
offsetArray	VCOP_WMEM	8 * sizeof(uint16_t)

Where
transposeStride = (((rhoMaxLength * sizeof(uint16_t) / 2) / 4) + 1) * 4;

Constraints

1. List size should be multiple of 16.

Techniques

- 8 WAY histogram update is used

17. Hough for Circles

Description

Hough transform for circles is commonly used to detect circular edges in an image, for applications such as circular traffic sign recognition. The component consists of multiple kernels designed to suit the data flow requirements for a Hough transform based Circle detection applet at frame level. Since the internal memory is limited, the Hough space in general will be divided into multiple smaller blocks. The kernels allows user to compute the index into which each points in an input block needs to vote into the corresponding Hough space block. The voting into the relevant Hough space and detection of circles from the final Hough space are provided as separate kernels.

Usage

The following kernels are present for implementing Hough for Circle frame level applet

- vcop_hough_circle_compute_idx:
  The input to this kernel (vcop_hough_circle_compute_idx) is normalized gradient in X and Y direction in packed format. The lower 16 bits should contain Y component and upper 16 bit should contain X component. The X and Y co-ordinate corresponding to each unit gradient vector should be provided in packed format – with Y co-ordinate in lower 16 bits and X co-ordinate in upper 16 bits. For the radius of circle which is of interest, we compute potential center locations for each gradient pixel in the input list. The center coordinate is converted to one dimensional index to the Hough space based on the pitch of the Hough space buffer.

- vcop_hough_circle_init_hough_space
  Since the Histogram engine in EVE only updates the histogram memory, user needs to initialize this memory with zeroes to start with. This kernel fill sup the Hough space memory with zeroes to prepare for voting with vcop_hough_circle_vote_to_hough_space kernel.

- vcop_hough_circle_vote_to_hough_space
  The Hough space indices provided by vcop_hough_circle_compute_idx kernel is used to vote to the corresponding Hough space buffer. The Hoguh space buffer has 8-bit data type and hence during histogram operation we saturate the Hough space to 255.

- vcop_hough_for_circle_detect
  This kernel takes the Hough space as input and generates a list of Centre co-ordinates and Hough space score for points in Hough space that have higher votes than a user specified threshold. The total number of circles detected is also returned in an output buffer.

Constraints

- The vcop_hough_for_circle_detect kernel expects that the input Hough space has a pitch that is multiple of 8. The width of Hough space need not be a multiple of 8. But user should make sure that the entries in Hough space between Hough space width and Hough space pitch are all zeroes so that no spurious detection occurs. This could be done by setting the correct settings for houghSpaceSaturateX and houghSpaceSaturateY in the vcop_hough_circle_compute_idx kernel.

Techniques

- De-interleaved load for better usage of input data bandwidth available on VCOP
- Collated store of the detected circle information (Center X & Y co-ordinates and Hough space score)
- One way Histogram for voting into Hough space

18. FAST9

Description

FAST9 is a feature detector which tries to detect features in a given frame based on a thresholding decision. The thresholding decision is performed by comparing the center pixel (pixel under consdieration) against 16 neighbors and checking for consecutive thresholding decision. If any 9 or more consecutive thresholds evaluate to true, the center pixel is considered a feature, otherwise it's marked as not being a feature.

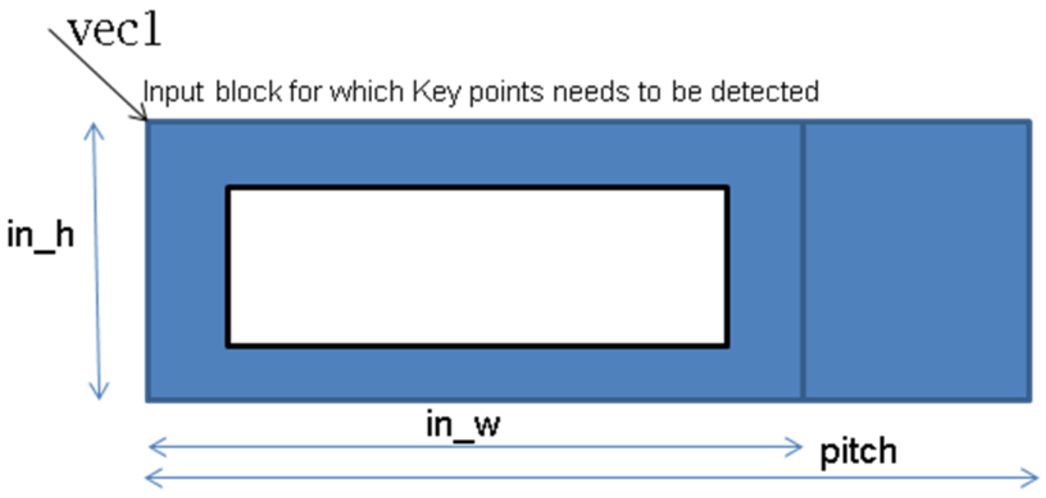
Usage

The FAST9 algoirthm requires (block width + 6) and (block height + 6) of data to generate block width and block height results consecutively. The padding is necessary as the center pixel requires a 3 pixel border in each direction for the algorithm to work. Thus a 7 x7 block is required for processing one pixel.

API

```
void vcop_fast9
(
   unsigned char *   vec1,
   unsigned char *   out_0_ptr,
   unsigned char *   out_1_ptr,
   unsigned short *  out_2_ptr,
   unsigned char *   Out,
   signed char   Thr,
   unsigned int pitch,
                                        unsigned int in_w,
                                        unsigned int in_h
                                      );
```

Vec1	→ Points to the location of the first pixel of the block
Out_0_ptr – out_2_ptr	→ Scratch buffers
Out	→ Primary output, decision of feature detection
Thr	→ Positive thresholds for comparing the center pixel against neighbors
pitch	→Input block width (should be minimum compute width + 6)
in_w	→ compute width
in_h	→ compute height



Performance Considerations

The VCOP cycle count for detecting (in_w x in_h) features is = 230 + 4.875 * in_w * in_h VCOP cycles.

Code Size = 3854 bytes

To get the best performance, the buffer placement is very important – below table summarize the suggested placement of buffers

Parameter	purpose	size (bytes)	memory area
Vec1	pointer to block as mentioned in above figure	Pitch * (in_h+6)	IBUFL
out_0_ptr	scratch space	2*in_w*in_h	IBUFH
out_1_ptr	scratch space	2*in_w*in_h	WBUF
out_2_ptr	scratch space	4*in_w*in_h	IBUFH
Out	Output indicating it to be a feature point or not	in_w*in_h	IBUFL

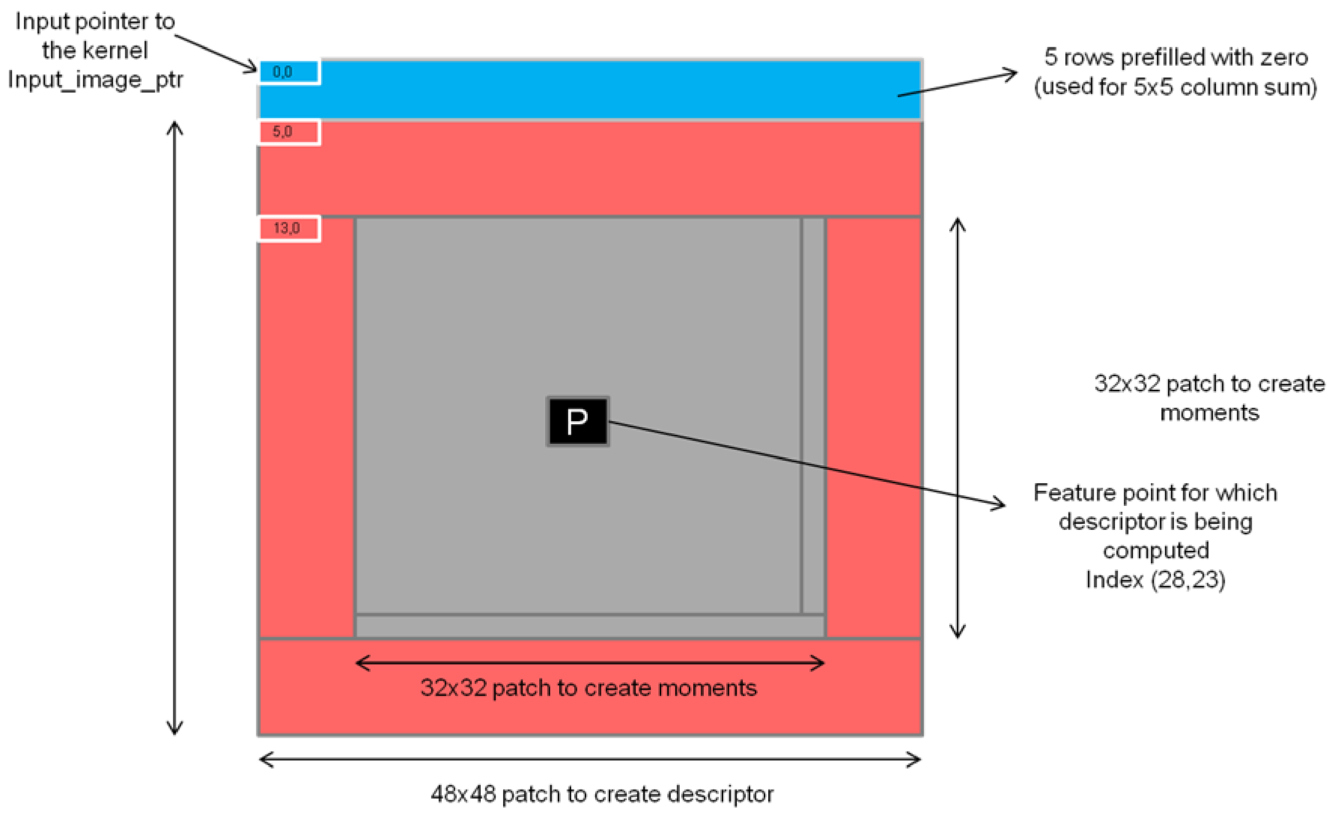
19. Compute_rBrief

Description

rBrief is a feature descriptor calculator. It generates a 256 bit feature vector for each feature which acts as a unique signature for the feature. The rBrief descriptor can then be used for tracking a feature across frames by calculating hamming distance or by other means. The rBrief feature descriptor is considered to be invariant to translation as well as rotation motion across frames.

Usage

The rBrief algorithm processes only one feature at a time. This kernel needs 53x48 block around the key point as shown in the below diagram. Top 5 rows in 53x48 block has to be prefilled by zero. This block is used by kernel as read only so the prefill of zero can be done only once, if the kernel is being called multiple times. The rBrief kernel first computes the horizontal and vertical moments using a 32x32 patch around the feature point (32x32 is a subset of the 48x48 patch). Using the horizontal and vertical moments, the orientation of the feature is detected. The orientation is then used to create a list of 256 src and dst pair locations within the 48x48 patch around the feature. The kernel also computes 5x5 sum for each pixel in 48x48 patch. The 5x5 sum computed is used for the 256 pair comparisons and provides a 256 bit feature descriptor, wherein each bit corresponds to the boolean result of one of the comparisons.


Input pointer to the kernel Input_image_ptr
0,0
5,0
13,0
5 rows prefilled with zero (used for 5x5 column sum)
32x32 patch to create moments
P
Feature point for which descriptor is being computed Index (28,23)
32x32 patch to create moments
48x48 patch to create descriptor

API

```
void vcop_compute_rBrief
(
    __vptr_int8    moments_col_mask,
    __vptr_int16   moments_col_sum,
    __vptr_uint8   moments_row_mask,
    __vptr_int16   moments_row_sum,
    __vptr_int16   moments_m10,
    __vptr_int16   moments_m01,
    __vptr_uint16  arctan_xthr,
    __vptr_uint8   arctan_pack_decision,
    __vptr_int16   cos_array_ptr,
    __vptr_int16   sin_array_ptr,
    __vptr_uint16  offset_ptr,
    __vptr_int16   cos_ptr,
    __vptr_int16   sin_ptr,
    __vptr_uint8  input_image_ptr,
    __vptr_int16 col_sum_ptr,
    __vptr_int16 row_col_sum_ct_ptr,
    __vptr_int8  in_src_dst_x_ptr,
    __vptr_int8  in_src_dst_y_ptr,
    __vptr_int8  rot_src_dst_ptr_x,
    __vptr_int8  rot_src_dst_ptr_y,
    __vptr_uint16  rot_src_lin_ptr,
    __vptr_uint16  rot_dst_lin_ptr,
    __vptr_int16  tlu_src_ptr,
    __vptr_int16  tlu_dst_ptr,
    __vptr_uint8  true_descriptor_optr
);
```

mometns_col_mask	→	Mask defining a circle within a 32x32 patch (Preloaded as per test bench)
moments_col_sum	→	Scratch for storing column sums for moments
moments_row_mask	→	Transpose of the mask defining the 32x32 patch (Preloaded as per test bench)
moments_row_sum	→	Scratch for storing row sums for moments
moments_m10	→	Scratch for storing m10 moment
moments_m01	→	Scratch for storing m01 moment
arctan_xthr	→	Preloaded threshold value for calculating arctan
arctan_pack_decision	→	Scratch for calculating arctan
cos_array_ptr	→	Preloaded cosine table
sin_array_ptr	→	Preloaded sine table
offset_ptr	→	Preloaded offset for 48x48 transpose
cos_ptr	→	Scratch for selected cosine table
sin_ptr	→	Scratch for selected sine table
input_image_ptr	→	Pointer to start of 48x53 patch including zero padded regions
col_sum_ptr	→	Scratch for storing col sums
row_col_sum_ct_ptr	→	Scratch for storing of 5x5 sums
in_src_dst_x_ptr	→	Preloaded X-coordinate of 256x2 pairs
in_src_dst_y_ptr	→	Preloaded Y-coordinate of 256x2 pairs
rot_src_dst_ptr_x	→	Scratch for rotated X-coordinate of 256 src pts
rot_src_dst_ptr_y	→	Scratch for rotated Y-coordinate of 256 src pts
rot_src_lin_ptr	→	Scratch for rotated linearized src 256 points
rot_dst_lin_ptr	→	Scratch for rotated linearized dst 256 points
tlu_src_ptr	→	Scratch for 256 5x5 src sums
tlu_dst_ptr	→	Scratch for 256 5x5 dst sums
true_descriptor_otpr	→	Primary output, 256 bit brief descriptor

Performance Considerations

To utilize the EVE for optimum performance, the rBrief kernel works on generating the brief descriptor of one feature at a time. The performance (VCOP cycle count) for one feature = 2200 cycles.

Code Size = 2878 bytes

To get the best performance, the buffer placement is very important – below table summarize the suggested placement of buffers

parameter	purpose	size (bytes)	memory area
moments_col_mask	Mask defining a circle within a 32x32 patch	32*32	WBUF
moments_col_sum	scratch space	2*32	IBUFH
moments_row_mask	Transpose of the mask defining the 32x32 patch	32*32	WBUF
moments_row_sum	scratch space	2*32	IBUFH
moments_m10	Scratch for storing m10 moment	2	IBUFL
moments_m01	Scratch for storing m01 moment	2	WBUF
arctan_xthr	Preloaded threshold value for calculating arctan	2*16	IBUFH
arctan_pack_decision	Scratch for calculating arctan	16	IBUFH
cos_array_ptr	Preloaded cosine table	2*33	IBUFL
sin_array_ptr	Preloaded sine table	2*33	WBUF
offset_ptr	Preloaded offset for 48x48 transpose	2*8	WBUF
cos_ptr	Scratch for selected cosine table	2*32	WBUF
sin_ptr	Scratch for selected sine table	2*32	IBUFL
input_image_ptr	Pointer to start of 48x53 patch including zero padded regions	53*48	IBUFL
col_sum_ptr	Scratch for storing col sums	2*53*50	IBUFH
row_col_sum_ct_ptr	Scratch for storing of 5x5 sums (two copies)	2*48*48*2	WBUF
in_src_dst_x_ptr	Preloaded X-coordinate of 256x2 pairs	2*256	IBUFH
in_src_dst_y_ptr	Preloaded Y-coordinate of 256x2 pairs	2*256	IBUFH
rot_src_dst_ptr_x	Scratch for rotated X-coordinate of 256 src pts	2*256	IBUFL
rot_src_dst_ptr_y	Scratch for rotated Y-coordinate of 256 src pts	2*256	WBUF
rot_src_lin_ptr	Scratch for rotated linearized src 256 points	2*256	IBUFL
rot_dst_lin_ptr	Scratch for rotated linearized dst 256 points	2*256	IBUFH
tlu_src_ptr	Scratch for 256 5x5 src sums	2*256	IBUFH
tlu_dst_ptr	Scratch for 256 5x5 dst sums	2*256	IBUFL
true_descriptor_otpr	Primary output, 256 bit brief descriptor	32	IBUFH

20. Hamming Distance

Description

Hamming distance between two strings of given length is the number of bit locations at which the two strings are different. In other words, in case of binary strings, the minimum number of bit substitutions that is needed to make one string equal to the other string. Hamming distance is widely used in many disciplines such as information theory, coding theory, cryptography and computer vision applications. In computer vision domain, it is particularly used for aiding in feature correspondence or matching wherein the two features are matched if the hamming distance between these two features is the least.

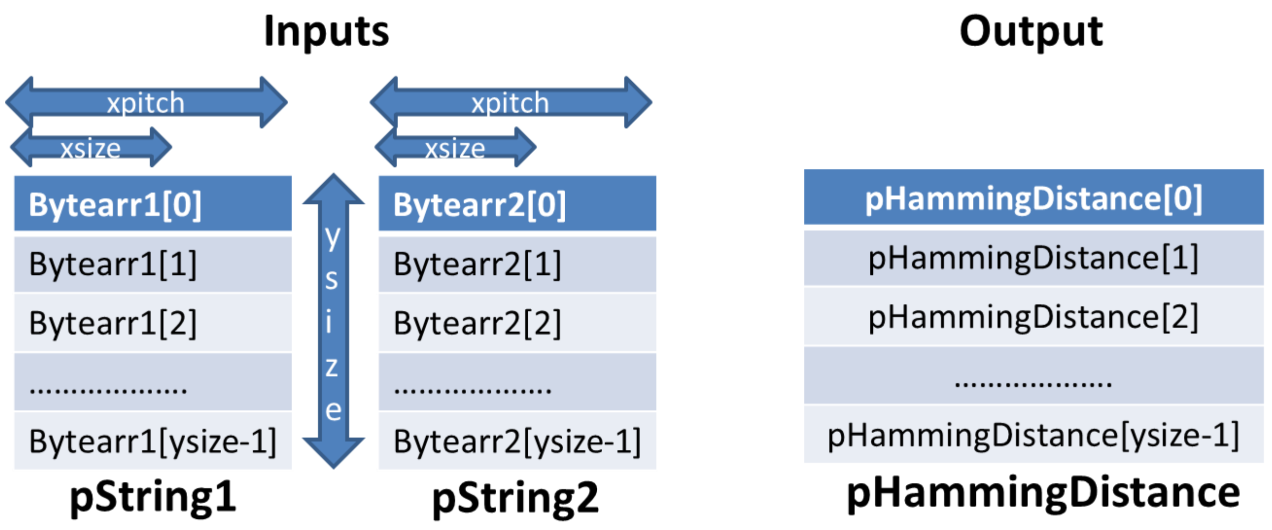
Hamming distance represents the number of ones in a "pString1 XOR pString2" operation wherein pString1 and pString2 denotes the two strings or feature descriptors of length "size". The output pHammingDistance can be denoted as follows wherein "bit_count" indicates the number of ones in a given binary string.

pHammingDistance = bit_count(pString1 XOR pString2)

API

Following input and output parameters are expected to be passed to the various kernels implemented (`vcop_hamming_distance`, `vcop_hamming_distance_multiple_32` and `vcop_hamming_distance_size_lt_32`).

- pString1 – Pointer to the first byte array string with length as (xpitch*ysize) bytes
- pString2 – Pointer to the second byte array string with length as (xpitch*ysize) bytes
- xsize – Denotes the size of each byte array element in pString1 or pString2 in bytes
- ysize – Denotes the number of byte array elements in pString1 or pString2
- mode – Denotes the hamming distance computation mode
    - Supports the following two modes of operation, please see illustration below
        - mode = 0: Many-to-One
        - mode = 1: One-to-One
- xpitch – Denotes the pitch of each byte array element in pString1 or pString2
- pHammingDistance – Pointer to the hamming distance output array
- pScratch – Scratch buffer needed to store the hamming distance vector output of 8 words between for each pair of byte array elements of pString1 and pString2
    - Requires a size of (ysize*8)*4 bytes



mode = 0:

- pHammingDistance[i] = bit_count(Bytearr1[i] XOR Bytearr2[0])

mode =1:

- pHammingDistance[i] = bit_count(Bytearr1[i] XOR Bytearr2[i])

where i = 0, 1, 2, ... ,(ysize-1)

Hamming distance kernel implementation on EVE does vector loads of 8 words each to process (8*4) bytes of input elements in a single iteration of the loop. Since the VCOP hardware requires the loop counter to be greater than zero, the kernel incorporates handling of corner cases based on the length i.e., "xsize" of the two byte array elements being processed. Here is the summary of the kernel functions to compute hamming distance. The following functions are supported:

- `vcop_hamming_distance` - Used for computing hamming distance when "xsize" > 32 & non- multiple of 32
- `vcop_hamming_distance_multiple_32` - Used for computing hamming distance when the "xsize" is an exact multiple of 32
- `vcop_hamming_distance_size_lt_32` - Used for computing hamming distance when the "xsize" is less than 32

Usage

Hamming distance kernel takes two input byte array pointer strings of data type unsigned char and length "ysize*xpitch" and computes an unsigned integer output array of size (ysize*4) bytes.

Recommendation: It is recommended to use feature size which is a multiple of 32 bytes to avoid overhead and thus achieve optimal performance. The larger the size of byte array element, xsize, the better the performance. It should be noted that the size of byte array element, xsize should be a positive integer but will have performance impact if the size is less than 32 bytes as summarized below.

Constraints

1. The total length of the input strings should be greater than 0 and less than or equal to 4096 bytes. That is, (ysize*xpitch) <= 4096 bytes
2. Requires a scratch buffer of length 8 words to be passed for storing intermediate outputs.
3. Each output array element would be saturated to 32 bit unsigned word in case of overflow.

Techniques

- Use integer data type for input strings so as to enable n-point load of 8 words as opposed to 8 bytes load using char data type.
    - Enables us to consume more data per iteration
- Use loop unrolling to have multiple sets of data to compute.
    - Enables us to hide the delay slot introduced by count bits instruction of VCOP
- Allocate one large scratch buffer to store two intermediate outputs with an offset programmed within the loop
    - Allows us to have only one loop instead of two loops (loop overheads can be avoided) to add all the intermediate outputs to generate vector output of hamming distance
    - Loop 3 will use one point loads to add the individual words of vector output to get final hamming distance

21. Feature Matching

Description

Feature matching is used for finding correspondence between interest points between two frames – either from different views of the same scene (as in stereo) of subsequent frames from the same camera. Establishing correspondence is a primary step to many applications such as depth map estimation, Image stitching, three –dimensional reconstruction etc.

These are set of kernels used for implementing a Hamming Distance based brute force feature descriptor matching applet. These kernels are currently part of the Hamming Distance kernel folder within VLIB. The core processing requirement of the feature matching applet is to compute Hamming distances between each pair of feature descriptors – one taken from the first input list and other from the second input list. A reliable match is declared only if the minimum Hamming distance is lesser than a threshold and also if the second minima is far enough from the first. This is ensured by checking that

```
minDist0 <= (1 - matchConfidence) * minDist1
```

where, `minDist0` is the minimum Hamming distance, `minDist1` is the second minimum hamming distance and `matchConfidence` is a user specified setting based on how reliable the user wants the feature matching to be.

API

The following kernels are part of this collection:

- vcop_featureMatching_32, vcop_featureMatching_lt_32, vcop_featureMatching_gt_32: These kernels compute hamming distances between two input lists of feature descriptors - pString1 and pString2. The 3 variants are similar to the Hamming Distance kernels except for the fact that here the Hamming distance output also has feature index embedded in the lower 16-bits and the actual hamming distance measure is packed in the upper 16-bits. Also instead of many-to-one mode where Hamming distances between all the feature descriptors in string 2 is computed with the first feature descriptor of string 1, here from performance considerations the kernel implements a many to 16 computation of Hamming distances. The kernel vcop_featureMatching_32 is to be used when feature descriptor is of 32 bytes. Kernel vcop_featureMatching_lt_32 is for feature descriptors that are smaller than 32 bytes and vcop_featureMatching_gt_32 is for feature descriptors that are larger than 32 bytes.

- vcop_featureMatch_initialize
  This kernel initializes the list of minimum hamming distances - pMinDist0 for the minima and pMinDist1 for the second minima to very high value (0xFFFF FFFF).

- vcop_findTwoBestMatches: This kernel takes in the list of Hamming distances and provides the least hamming distance entry in pMinDis0 and the second minima in pMinDist1. The kernel works on the many-to-16 Hamming distance output from the feature matching kernel.

- vcop_pickConfidentMatches
  This kernel picks the reliable feature matches when the two best matches are provided to it. Reliable matches are one which has Hamming distance measure that is less than the user specified threshold and the best two hamming distances satisfy the ration test criteria mentioned above.

Techniques

- De-interleaved load and skip store.
- Transposed store using offset_np1() mode to make use of SIMD computes in horizontal direction.

22. Block Statistics

Description

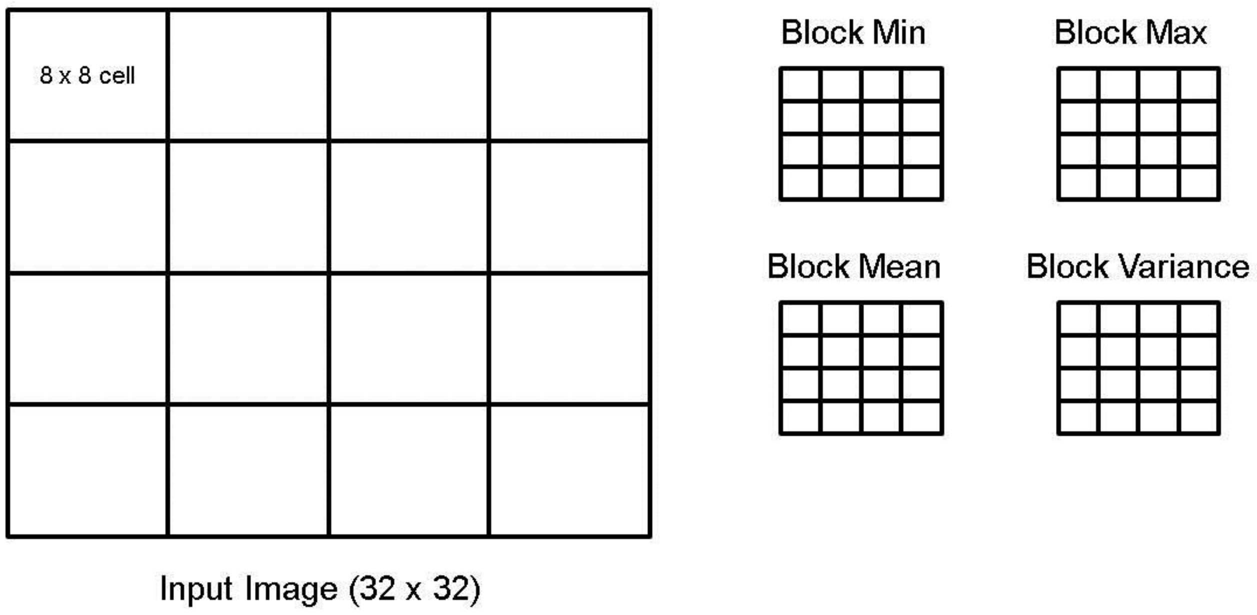
This routine accepts an 8-bit grayscale input image of size blockWidth by blockHeight with a stride of blockStride. The image is divided into non-overlapping blocks of statBlockWidth by statBlockHeight. The kernel computes block statistics over these non-overlapping blocks. The following statistics are computed:

1. Minima(min_B)
2. Maxima(max_B)
3. Mean(mean_B)
4. Variance(variance_A)

The kernel doesn't perform averaging during mean and variance computations. Hence mean output reported is actually N*mean and variance is N^2*variance where N is the number of samples within a block (N = statBlockWidth*statBlockHeight). User has to divide the outputs by N and N^2 respectively to arrive at mean and variance.

For a given image frame or ROI of size M x N and block size of m x n, the ROI is divided into closely packed non-overlapping blocks/cells. There will be Mo x No such blocks where

- Mo = floor(M/m)
- No = floor(N/n)
- Example scenario for an image of size 32 x 32 with 8 x 8 cell size is shown below



API

This routine is C-callable and can be called as:

```
void vcop_block_statistics
(
   __vptr_uint8       im_A,
   unsigned short     blockStride,
   unsigned short     blockWidth,
   unsigned short    blockHeight,
   unsigned short    statBlockWidth,
   unsigned short    statBlockHeight,
   __vptr_uint8      scratch_C,
   __vptr_uint16     scratchSum_A,
   __vptr_uint32     scratchSumSq_B,
   __vptr_uint32     scratchSumSq_C,
   __vptr_uint16     scratchSumSq_C_lo,
   __vptr_uint16     scratchSumSq_C_hi,
   __vptr_uint8      min_B,
   __vptr_uint8      max_B,
   __vptr_uint16     mean_B,
   __vptr_uint32     variance_A
)
```

- im_A : 8-bit grayscale image block. This buffer should contain atleast blockHeight*blockStride bytes.
- blockStride : Stride of the input image block.
- blockWidth : Width of the input image block.
- blockHeight : Height of the input image block.
- statBlockWidth : Width over which block statistics needs to be computed.
- statBlockHeight : Height over which block statistics needs to be computed.
- scratch_C : Scratch buffer for storing row-wise minima and row-wise maxima. User need to allocate a minimum of 8*ceil(blockWidth/8)*36 bytes. Only one half of this buffer will be effectively used.
- scratchSum_A : Scratch buffer for storing row-wise sum of image pixels. User need to allocate a minimum of 8*ceil(blockWidth/8)*36 bytes. Only one half of this buffer will be effectively used.
- scratchSumSq_B : Scratch buffer for holding row-wise sum of squares of image pixels. User need to allocate at least 8*ceil(blockWidth/8)*36 bytes.
- scratchSumSq_C : Scratch buffer for holding block sum of squares of image pixels. User need to allocate at least 8*36 bytes.
- scratchSumSq_C_lo : Address of 16-bit LSB of scratchSumSq_C
- scratchSumSq_C_hi : Address of 16-bit MSB of scratchSumSq_C
- min_B : Block minimum output. The buffer requires a minimum of 8*36 bytes. The output will be present in an 8*8 region with a stride of 36 bytes.
- max_B : Block maximum output. The buffer requires a minimum of 8*36 bytes. The output will be present in an 8*8 region with a stride of 36 bytes.
- mean_B : Block mean output. The kernel outputs N*mean where N is the number of samples in the block. The buffer requires a minimum of 8*36 bytes. The output will be present in an 8*8 region with a stride of 36 bytes.
- variance_A : Block variance output. The kernel outputs N^2*variance where N

is the number of samples in the block.The buffer requires a minimum of 8*36 bytes.

- Returns : None or void.

Constraints

- Number of pixels in a block (blockWidth x blockHeight) <= 256
- Number of blocks vertically (blockHeight/statBlockHeight) is <= 8
- Number of blocks horizontally (blockWidth/statBlockWidth) is <= 8
- **Assumptions**
    - Input:
        - An 8-bit grayscale input image: (uint8_t)
    - Outputs:
        - Block Minima: (uint8_t)
        - Block Maxima: (uint8_t)
        - Block Mean: (uint16_t). Instead of outputting mean, the kernel outputs N*mean.
        - Block Variance: (uint32_t). Instead of outputting variance, the kernel provides N^2*variance.
    - Assumptions:
        - Number of pixels within a cell is <= 256 is used to decide the precision of mean and variance

Performance Considerations

For a compute block of width M x N and cell size of m x n:

- Total VCOP cycles = 124 + FLOOR(N/n,1)*n*CEILING(M/8, 1)*2+ FLOOR(M/m, 1)*m*CEILING(FLOOR(N/m, 1)/8,1)*2+ FLOOR(N/n,1)*5 cycles
- 124 cycles is overhead which includes command decode, parameter fetch and pipe up

Code Size = 1544 bytes

Techniques

- Utilizes (np+1) transpose stores to enable SIMD processing
- Uses 32 bit x 16 bit Extended Precision Multiplication for achieving better accuracy and optimal performance

23. Median Filter

Description

This kernel implements functional units required for implementing median filtering for large filter kernels. Histogram sort approach is a quite popular technique with O(n) complexity for an m-by-n kernel. Functions are provided here for computing block histograms and for selection of any

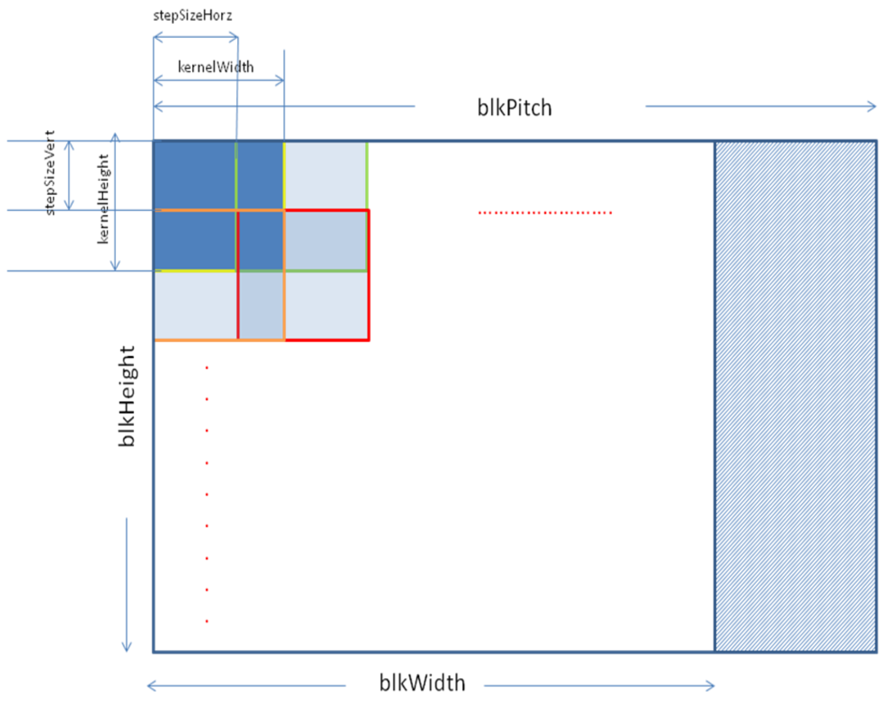
order statistic from the obtained histogram. These kernels can be employed for generating any order statistics.
The "vcop_update_block_histogram_8c" kernel takes a 2D block as input and provides histogram for the block. The "vcop_select_kth_smallest_from_hist" kernel takes an input histogram array and returns the median value from it.

The "vcop_update_block_histogram_8c" kernel makes use of the HISTOGRAM engine within VCOP to sort the input image block. It utilizes 8 channel histogram feature of VCOP to update 4 histogram votes in 1 VCOP cycle. Eight channel histogram requires block width to be a multiple of 8. To cater to generic kernel width we use a technique employing weighted histogram capability of the HISTOGRAM engine. We divide the kernel into two regions - one which is a clean multiple of 8 and remaining region which is of width < 8. The histogram of the first region is voted into the histogram buffer using a weight vector of all ones. The second region is voted into histogram memory with a weight vector of (kernelWidth%8) ones and rest all zeros. The 8 channel histogram consists internally of 8 copies of histograms and this kernel accumulates the different copies and provides a single histogram as output.

The "vcop_select_kth_smallest_from_hist" kernel selects any generic kth order statistic by doing a hierarchical search – first on a coarse histogram that is derived and subsequently refines using the full histogram.

A wrapper is provided over these kernels to generate median from a block of an input image. This wrapper routine accepts an 8-bit grayscale input image block of size blkWidth by blkHeight with a stride of blkPitch. The image block is divided into overlapping blocks of kernelWidth by kernelHeight. The overlap between successive kernel windows is controlled by stepSizeHorz and stepSizeVert parameters. Median output is computed over the kernel windows, which slides by 'stepSizeHorz' pixel in horizontal and by stepSizeVert in vertical directions.

Below figure shows the input picture format with all control parameters.



API

This routine is C-callable and can be called as:

```
void vcop_large_kernel_median_wrapper_init
(
    unsigned char                *pInput_A,
    unsigned char                *pMedian_B,
    unsigned short                blkPitch,
    unsigned short                blkWidth,
    unsigned short                blkHeight,
    unsigned char                 kernelWidth,
    unsigned char                 kernelHeight,
    unsigned short                stepSizeHorz,
    unsigned short                stepSizeVert,
    short                        *histo_C,
    char                         *scratch_wgt_B,
    short                        *scratch_histo_B,
    short                        *blk_hist_C,
    short                        *coarse_hist_scratch_A,
    unsigned short               *pBlock,
    median_kernel_context        *context
```

- pInput_A : Pointer to start of an 8-bit image block

pMedian_B	: Pointer to median output
blkPitch	: Stride of the input image block
blkWidth	: Width of the image block
blkHeight	: Height of the image block
kernelWidth	: Filtering kernel width over which median is to be computed
kernelHeight	: Filtering kernel height over which median is to be computed
stepSizeHorz	: Step in x direction between suzzessive kernel windows
stepSizeVert	: Step in y direction between suzzessive kernel windows
histo_C	: Buffer for storing the 8 channel histogram
scratch_wgt_B	: Weight to be used during wieghted histogram
scratch_histo_B	: Scratch to be used for accumulation of histogram
blk_hist_C	: Final block histogram in signle channel format
coarse_hist_scratch_A	: 16-bin coarse histogram used for selection algorithm
pBlock	: Pointer to the VCOP parameter block

Constraints

- The maximum image block size (blockHeight*blockWidth) and therefore the kernel size that can be handled is restricted by the internal memory of EVE. The block size has to be less than ~ 16kB (16, 351 bytes to be exact).
- The histogram kernel cannot be used for filtering kernel size (kernelWidth * kernelHeight) less than 8.

Performance Considerations

- vcop_select_kth_smallest_from_hist – 215 cycles (average case, ie. for median values around 128) including generation of coarse histogram.

- vcop_update_block_histogram_8c – ¼ cycles per pixel for 8 channel histogram update + 360 cycles for accumulation of 8 channel histogram into single channel.

Techniques

- Eight channel weighted histogram feature of VCOP histogram engine
- Transpose store feature to enable use of SIMD capability while accumulating eight channel histogram and also to arrive at coarse histogram.

24. Fast9 Score

Description

Two methods of Fast9 score have been implemented.

a. SAD Based

FAST9 score computes a SAD based score computation computed for key points given by the Fast9 key point detection.

$$V = \max\left(\sum_{x \in S_{bright}} | I_{p->x} - I_p | - t, \sum_{x \in S_{dark}} | I_{p->x} - I_p | - t \right)$$

Where,

p	Key point detected using Fast9 detection
I_p	Intensity value at location indicated by p
$I_{p->x}$	Intensity value at locations of circular ring around p
t	Fast9 threshold

Usage

The Fast9 score algorithm requires an 8x8 patch of image data around each key point, which is detected using Fast9 detection. It computes the SAD based score using the above mentioned formula for each key point.

API

```
vcop_fast9_score_kernel
(
   (unsigned int*)in_temp,
   (unsigned int*)pTemp_buf,
   (unsigned int*)pTemp,
   pTemp,
   num_features,
   Thr,
   lut_index,
   Offset_Out,
   BScore,
   DScore,
   Score,
        (unsigned int *)__pblock_vcop_fast9_score_kernel
      );
```

in_temp	→ Points to the location of the first pixel of the block
pTemp_buf	→ Scratch buffer
pTemp	→ Scratch buffer
num_features	→ Number of features to process
Thr	→ Positive thresholds for comparing the center pixel against neighbors
Lut_index	→Index values to look-up the circular ring pixel values around the Keypoint
Offset_out	→ Scratch buffer
BScore	→ Scratch buffer holding score for bright condition
DScore	→ Scratch buffer holding score for dark condition
Score	→Output score, which is max(Bscore, Dscore)

__pblock_vcop_fast9_score_kernel →Pointer to the param block, used to update the destination pointer address so that Score is contiguously while processing multiple blocks.

Memory Requirements

Parameter	Purpose	size (bytes)	memory area
in_temp	pointer to the location of the first pixel of block	num_features*VCOP_SIMD_WIDTH *VCOP_SIMD_WIDTH	IBUFL
pTemp_buf	scratch space	num_features*VCOP_SIMD_WIDTH*9	IBUFH
pTemp	scratch space	num_features*VCOP_SIMD_WIDTH*8	WBUF
Lut_index	constant index table	VCOP_SIMD_WIDTH*17	IBUFH
Offset_out	scratch space	num_features*4*17	IBUFL
BScore	scratch space	num_features*2	IBUFL
DScore	scratch space	num_features*2	IBUFH
Score	Output	num_features *2	WBUF

b. Threshold based

In this method, the fast9 score is defined as the highest threshold for which the key point still satisfies the fast9 key point condition, i.e, atleast 9 pixels in the circular ring around the key point satisfies either the bright or the dark condition.

$$V = TH_{\max} \; \forall p \in (S_{bright} \,|\, S_{dark})$$

Where,

p	key point detected using Fast9, represented by (X, Y), co-ordinate list
TH_{max}	Maximum threshold for which p is still a key point
S_{bright}	Set containing bright key points. A pixel $p \epsilon S_{bright}$ if $(I_p + t) <= I_{p\text{->}n}$
S_{dark}	Set containing dark key points. A pixel $p \epsilon S_{dark}$ if $I_{p\text{->}n} <= (I_p - t)$
I_p	Intensity value of the key point at location indicated by p
$I_{p\text{->}n}$	Intensity value of the key point at location n which is along the circular ring around key point p

Usage

The Fast9 score algorithm requires an 8x8 patch of image data around each key point, which is detected using Fast9 detection. It computes the threshold based score using the above mentioned formula for each key point.

API

```
vcop_fast9_thresh_score
 (
        (unsigned int*)in_temp,
        (unsigned int*)pTemp_buf,
        (unsigned int*)pTemp,
        pTemp,
        num_features,
        Thr,
        lut_index,
        Offset_Out,
        BScore,
        DScore,
        Score_b,
        Score_d,
        Score,
        (unsigned int *)__pblock_vcop_fast9_thresh_score
  );
```

in_temp	→ Points to the location of the first pixel of the block
pTemp_buf	→ Scratch buffer
pTemp	→ Scratch buffer
num_features	→ Number of features to process
Thr	→ Positive thresholds for comparing the center pixel against neighbors

Lut_index	→Index values to look-up the circular ring pixel values around the Keypoint. We need to look-up 24 indices i.e. the first 8 are looked-up
Offset_out	→ Scratch buffer
BScore	→ Scratch buffer 16 scores – each is max of 9 offset pix, with different start position
DScore	→ Scratch buffer 16 scores – each is min of 9 offset pix, with different start position
Score_B	→ Scratch buffer holding min of 16 BScore
Score _D	→ Scratch buffer holding max of 16 DScore
Score	→ Final score

__pblock_vcop_fast9_score_kernel →Pointer to the param block, used to update the destination pointer address so that Score is contiguously while processing multiple blocks.

Memory Requirements

Parameter	Purpose	size (bytes)	memory area
in_temp	pointer to the location of the first pixel of block	num_features*VCOP_SIMD_WIDTH *VCOP_SIMD_WIDTH	IBUFL
pTemp_buf	scratch space	num_features*VCOP_SIMD_WIDTH*9	IBUFH
pTemp	scratch space	num_features*VCOP_SIMD_WIDTH*8	WBUF
Lut_index	constant index table	VCOP_SIMD_WIDTH*17	IBUFH
Offset_out	scratch space	num_features*4*25	IBUFL
BScore	scratch space	num_features*16	IBUFL
DScore	scratch space	num_features*16	IBUFH
Score_b	scratch space	num_features*2	IBUFL
Score_d	scratch space	num_features*2	IBUFH
Score	Output	num_features *2	WBUF

25. Horizontal Non Maximal Suppression

Description

This kernel does non-maximal suppression along the horizontal direction. Given a list of key point locations (x,y) and the corresponding score, this function checks if a given point has a neighbor and then checks if the score is greater than or equal the neighbor. If not, the score is suppressed. This kernel assumes that the x,y location input is in packed data 32-bit format (16-bit each) and is in raster scan order of x , i.e the data is arranged in buckets of y, with increasing x.

API Usage

```
vcop_horizontal_non_max_suppression
(
    corners,
    num_corners,
    sad_scores,
    nms_x_corners,
    nms_x_score,
    Id
);
```

Corners	→ Input corner list location (x,y)
Num_corners	→ number of corners
Sad_scores	→Input score
Nms_x_corners	→ Output, x,y corners packed with Id of the non-suppressed corners, 0 if the corner is suppressed
Nms_x_score	→ Score output of the non-suppressed corners, 0 if the corner is suppressed
Id	→ Id input

Memory Requirements

Parameter	purpose	size (bytes)	memory area
Corners	Input corner list (x,y)	num_corners*4	IBUFL
Sad_scores	Input Score	num_corners*2	IBUFH
Nms_x_corners	Output packed x,y,Id	(num_corners+1)*4	WBUF
nms_x_score	Output score	num_corners*2	WBUF
Id	scratch space	8	IBUFL

Other considerations

1. The first and the last key point is always non-suppressed since they do not have neighbor values to compare against.
2. This kernel packs the output corners with ID. This ID is used for looking up score values in the vertical non-maximal suppression, after the sorting kernel.
3. A loop to zero pad the output upto 2048 elements is present, since performance of sort is optimal if we consider 2048 elements.

If the need is to apply only horizontal non-maximal suppression, then the packing of ID and the zero pad loop can be discarded to improve performance.

26. Vertical Non Maximal suppression

Description

This kernel does non-maximal suppression along the vertical direction. Given a list of key point locations (x,y) and the corresponding score, this function checks if a given point has a neighbor and then checks if the score is greater than or equal the neighbor. If not, the score is suppressed. This kernel assumes that the x,y location input is in packed data 32-bit format (16-bit each) and is packed along with ID. Also, the kernel expects the data to be in raster scan order of y, i.e buckets of x, with increasing y.

API Usage

```
vcop_vertical_non_max_suppression_kernel
(
    corners,
    num_corners,
    sad_scores,
    nms_id,
    nms_y_score,
    (unsigned short*)nms_y_corners,
    nms_y_corners,
    nms_score,
    Id
);
```

Corners	→ Input corner list location packed with id (x,y,id)
Num_corners	→ number of corners
Sad_scores	→Input score
Nms_id	→ Id extracted from the input – corners
Nms_y_score	→Sad scores re-arranged based on the Id lookup
Nms_y_corners	→ Output, x,y corners of the non-suppressed corners, 0 if the corner is suppressed
Nms_score	→ Score output of the non-suppressed corners, 0 if the corner is suppressed
Id	→ Id input

Memory Requirements

Parameter	purpose	size (bytes)	memory area
Corners	Input corner list (x,y)	num_corners*4	IBUFL
Sad_scores	Input Score	num_corners*2	IBUFH
Nms_id	Scratch space	num_corners*2	IBUFL
nms_y_score	scratch space	num_corners*2	WBUF
Nms_y_corners	Output (x,y)	num_corners*4	IBUFH
nms_score	Output score	num_corners*4	IBUFH
Id	scratch space	8	WBUF

Other considerations

1. The first and the last key point is always non-suppressed since they do not have neighbor values to compare against
2. This kernel packs the output score with ID. This ID is used for looking up score values in the prune big list kernel, after the sorting kernel.
3. A loop to zero pad the output upto 2048 elements is present, since performance of sort is optimal if we consider 2048 elements.

If the need is to apply only horizontal non-maximal suppression, then the packing of ID and the zero pad loop can be discarded to improve performance.

27. Multi-point Harris Score

Description

This kernel computes 16-bit Harris Score for a given set of input image blocks. The kernel processes a 9x9 region from each input image block. The Harris score is computed using the 7x7 X and Y- gradients.

For each input block, the kernel computes 7x7 gradients in X and Y directions. The following equations together compute the 2×2 structure tensor matrix M from the gradX and gradY matrices, where the summations are over 7×7 pixel neighborhoods:

$$M(1,1) = \mathrm{sum}(\mathrm{gradX})\hat{}2$$
$$M(1,2) = M(2,1) = \mathrm{sum}(\mathrm{gradX} \times \mathrm{gradY})$$
$$M(2,2) = \mathrm{sum}(\mathrm{gradY})\hat{}2$$

The cornerness score is defined as det(M) – k × trace(M)^2, where k is a tunable sensitivity parameter, typically around 0.04.

The underlying score computation is of 64-bit precision. To save memory, an approximation of the binary log of this value is stored in the output. Order relationships between different score values is maintain in this format, up to quantization limits. Such a format is superior compared to simple rounding in terms of quantization errors as the below format preserves maximum number of bits from the original score.

The exact format of Harris score is as follows:

Harris Score 16-bit format:

Exponent (6 bit)	Mantissa (10 bit)

The exponent corresponds to the location of the leading one-bit in the score. Mantissa contains the relevant bits (10 bits for 16-bit Harris score) starting from the leading 1-bit. The exponent is 0 if score is less than 2^{10}. If score is greater, (*lmb* -10) is stored in the exponent.

For example, a score of 571717286 (= 0010 0010 0001 0011 1011 0110 1010 0110b), will be encoded by the 16-bit kernel as = (30-10)* 2^{10} + (571717286) >> 20 = 21025.

API

This routine is C-callable and can be called as:

```
void vcop_multipoint_harrisScore_7x7_u16
(
   __vptr_uint8      pImgBlocks_A,
   __vptr_uint16    harrisScore_C,
   unsigned short   numPoints,
   unsigned short   inputStride,
   unsigned short   interBlockOffset,
   unsigned short   sensitivityParam,
   short            start_idx,
   __vptr_uint16    seq_array_C,
   __vptr_int32     vertSumX2_C,
   __vptr_int32     vertSumY2_B,
   __vptr_int32     vertSumXY_C,
   __vptr_int32     Ixx_A,
   __vptr_int32     Iyy_C,
   __vptr_int32     Ixy_A,
   __vptr_uint16    Ixx_l,
   __vptr_uint16    Iyy_l,
   __vptr_uint16    Ixy_l,
   __vptr_int16     Ixx_h,
   __vptr_int16     Iyy_h,
   __vptr_int16     Ixy_h,
   __vptr_uint32    detL_C,
   __vptr_int32     detH_C,
   __vptr_int32     pBlock
)
```

- pImgBlocks_A : Pointer to image blocks. The number of image blocks should be equal to numPoints. Each block in the input image block should be atleast 9x9. The image data is 8-bit.
- harrisScore_C : Output Harris score in upper 16-bits packed with payload information in the lower 16-bits. The Harris score is computed from 7x7 gradient of the input image blocks. The payload information is an increasing sequence of numbers.
- numPoints : Number of points for which harris score has to be computed.
- inputStride : Stride in the input image block buffer.
- interBlockOffset : Offset between start of one block and it's next block.
- sensitivityParam : Tunable sensitivity parameter for computing Harris score.
- start_idx : Starting index of the unique payload information appended
- in the output.
- seq_array_C : This is a predefined sequence of numbers (0 to 7)
- vertSumX2_C : Scratch buffer to hold intermediate row sum of square of x-gradient
- vertSumY2_B : Scratch buffer to hold intermediate row sum of square of y-gradient
- vertSumXY_C : Scratch buffer to hold intermediate row sum of product of x and y-gradients.
- Ixx_A : Scratch buffer to hold sum of square of x-gradients
- Iyy_C : Scratch buffer to hold sum of square of y-gradients

• Ixy_A	: Scratch buffer to hold sum of product of x and y-gradients.
• Ixx_l	: Lower 16-bits of Ixx
• Iyy_l	: Lower 16-bits of Iyy
• Ixy_l	: Lower 16-bits of Ixy
• Ixx_h	: Higher 16-bits of Ixx
• Iyy_h	: Higher 16-bits of Iyy
• Ixy_h	: Higher 16-bits of Ixy
• detL_C	: Lower 32-bits of determinant of the structure tensor
• detH_C	: Upper 32-bits of determinant of the structure tensor
• pBlock	: Pointer to the VCOP parameter block

Constraints

- Only non-negative scores are encoded in the current scheme. All negative scores are saturated to zero.

Techniques

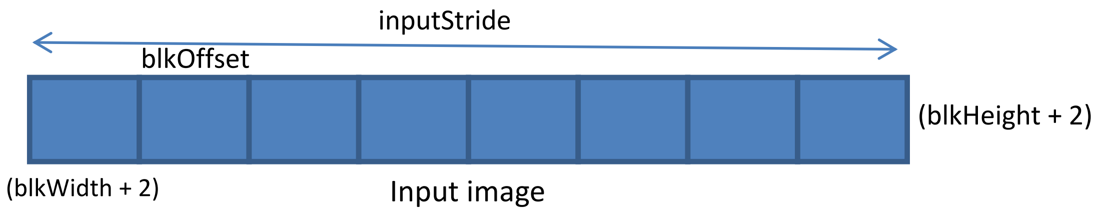
- Transpose store feature to enable use of SIMD capability while computing block sums

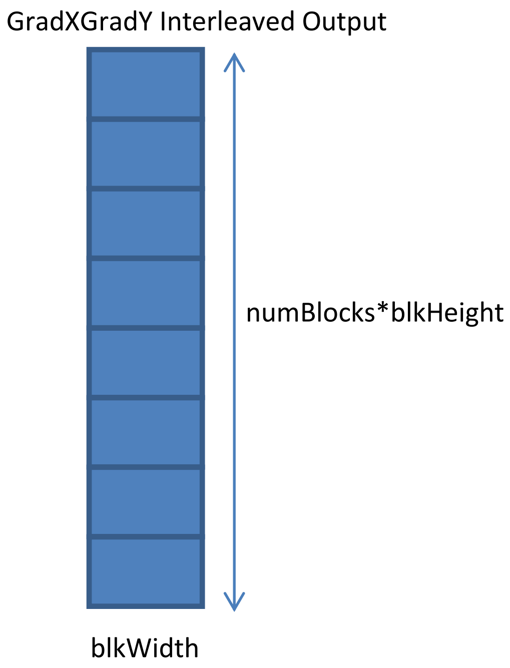
Extended precision multiplication for computing intermediate determinant of Structure tensor in 64-bit precision.

28. Multi-block Gradient XY

Description

This kernel computes the block wise gradient along X and Y directions for each block of 8-bit grayscale image input. The output gradient along X and Y directions is stored in interleaved format. The kernel requires border of 1 pixel in either direction for computing gradients and so the output would be computed from the pixel location which is shifted by 1 line down and l pixel to right w.r.t image input. The kernel takes two more inputs namely blkOffset and numBlocks from user to know the offset between two input blocks to be processed and the number of input blocks for which gradient needs to be computed. Please see the illustration for further details on the input and output images.





API

This routine is C-callable and can be called as:

```
void vcop_multiblock_gradient_xy
(
   __vptr_uint8   pIn_A,
   __vptr_int16   pIntlvGradXY_B,
   unsigned short inputStride,
   unsigned short blkWidth,
   unsigned short blkHeight,
   unsigned short blkOffset,
   unsigned short numBlocks
)
```

- pIn_A : Pointer to input image blocks. The number of image blocks should be equal to numBlocks
- pIntlvGradXY_B : Pointer to output gradient XY which is in interleaved format
- inputStride : Stride of the input image in bytes
- blkWidth : Output block width in pixels wherein each output pixel is Gradient XY interleaved with four bytes
- blkHeight : Output block height in pixels wherein each output pixel is Gradient XY interleaved with four bytes
- blkOffset : Offset between the two input blocks in bytes.
- numBlocks : Number of input blocks for which interleaved gradient XY output.needs to be computed

Constraints

- Input block height and block width should be at least equal to (blkWidth+2) & (blkHeight+2) to account for the border pixels needed for the gradient output

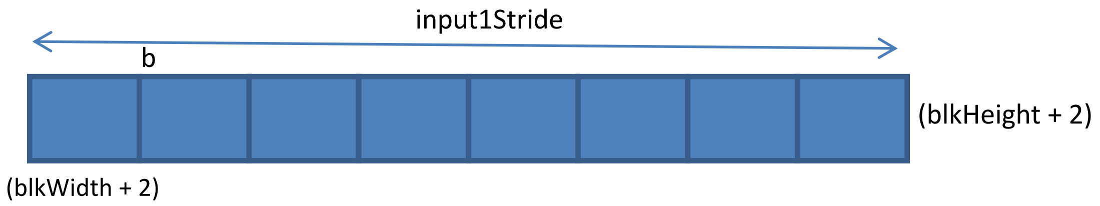
Techniques

- Interleaved store instruction is used.

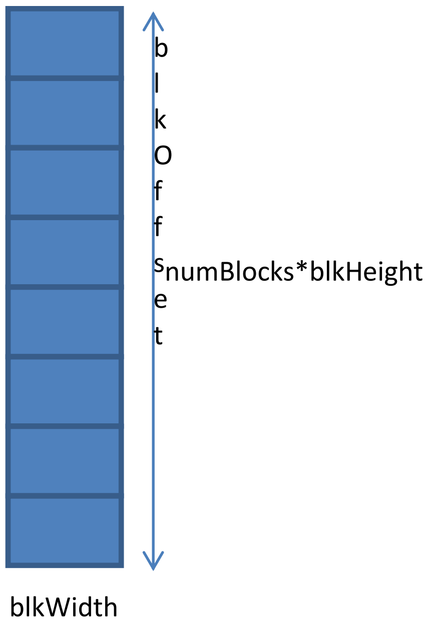
29. Multi-block Bilinear Interpolation

Description

This kernel computes the block wise bilinear interpolated pixels output for each block of 8-bit grayscale image input. The kernel requires at least a border of 1 pixel in either direction for computing bilinear interpolation output and so the output would be computed from the pixel location which is shifted by 1 line down and l pixel to right w.r.t image input. The kernel would take the bilinear weights that need to be applied for pixels within each input block. The second input which is bilinear weights would be of dimension (numKeyPoints x 4) where the factor 4 indicates 4 bilinear weights used for interpolation using 4 neighboring pixels. It also takes two more inputs namely blkOffset and numKeyPoints from user to know the offset between two input blocks to be processed and the number of input blocks for which bilinear interpolated image pixels needs to be computed. Please see the illustration for further details on the input and output images.



Bilinear Interpolated Image Pixel Output



API

This routine is C-callable and can be called as:

```
void vcop_multiblock_bilinear_interp_7x7_u8
(
   __vptr_uint8     pIn_A,
   __vptr_uint16   pInpWts_B,
   __vptr_uint8     pOutBilinearInterpImg_C,
   unsigned short  input1Stride,
   unsigned short  input2Stride,
   unsigned short  outputStride,
   unsigned short  blkWidth,
   unsigned short  blkHeight,
   unsigned short  blkOffset,
   unsigned short  shiftValue,
   unsigned short  numKeyPoints
)
```

• pIn_A	: Pointer to input image blocks. The number of image blocks should be equal to numKeyPoints
• pInpWts_B	: Pointer to bilinear weights that is used for interpolation for each block
• pOutBilinearInterpImg_C	: Pointer to interpolated image pixels for the given
•	input image blocks
• input1Stride	: Stride of the input image in bytes
• input2Stride	: Stride of the bilinear weights input in pixels
• outputStride	: Stride of the bilinear interpolated output
• blkWidth	: Output block width in bytes
• blkHeight	: Output block height in bytes
• blkOffset	: Offset between the two input blocks in bytes.
• shiftValue	: Indicates the amount of shift that need to be applied on bilinear interpolated pixel output
• numKeyPoints	: Number of input blocks for which bilinear interpolated output pixels need to be computed

Constraints

- Input block height and block width should be at least equal to (blkWidth+1) & (blkHeight+1) to account for the border pixels needed for the bilinear interpolation

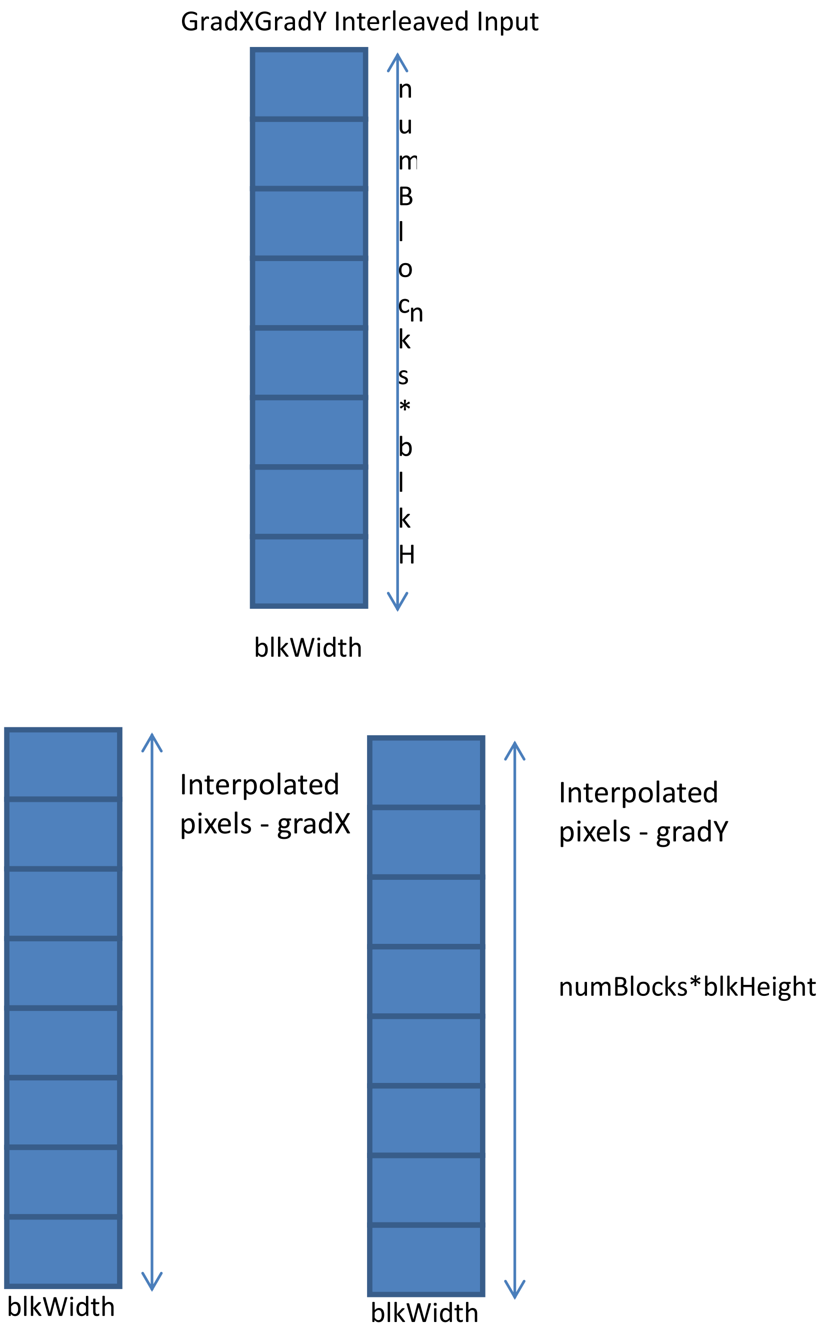
Techniques

- Utilizes the two functional units effectively to obtain optimal performance.

30. Multi-block Gradient XY Bilinear Interpolation

Description

This kernel computes the block wise gradient X and gradient Y bilinear interpolated pixels output for each block of 32-bit interleaved gradient XY input image. The kernel requires at least a border of 1 pixel in either direction for computing bilinear interpolation output and so the output would be computed from the pixel location which is shifted by 1 line down and l pixel to right w.r.t image input.


GradXGradY Interleaved Input
numBlocks*blkH
blkWidth
Interpolated pixels - gradX
blkWidth
Interpolated pixels - gradY
numBlocks*blkHeight
blkWidth

The kernel would take the bilinear weights that need to be applied for pixels within each input block. The second input which is bilinear weights would be of dimension (numKeyPoints x 4) where the factor 4 indicates 4 bilinear weights used for interpolation using 4 neighboring pixels. It also takes two more inputs namely blkOffset and numKeyPoints from user to know the offset between two input blocks to be processed and the number of input blocks for which bilinear interpolated image pixels needs to be computed. It should be noted that the bilinear interpolated gradient outputs for Gradient along X and for Gradient along Y are obtained in two output buffers. Please see the illustration above for further details on the input and output images.

API

This routine is C-callable and can be called as:

```
void vcop_multiblock_bilinear_interp_intlv_7x7_s16
(
   __vptr_int16      pIntlvGradXY_A,
   __vptr_uint16     pInpWts_X,
   __vptr_int16      pGradXBilinearInterpImg_B,
   __vptr_int16      pGradYBilinearInterpImg_C,
   unsigned short    input1Stride,
   unsigned short    input2Stride,
   unsigned short    outputStride,
   unsigned short    blkWidth,
   unsigned short    blkHeight,
   unsigned short    blkOffset,
   unsigned short    shiftValue,
   unsigned short    numKeyPoints
)
```

- pIntlvGradXY_A : Pointer to gradient XY interleaved input image blocks. The number of image blocks should be equal to numKeyPoints
- pInpWts_X : Pointer to bilinear weights that is used for interpolation for each block
- pGradXBilinearInterpImg_B : Pointer to bilinear interpolated gradient X pixels for the given gradient X input blocks
- pGradYBilinearInterpImg _C : Pointer to bilinear interpolated gradient Y pixels for the given gradient Y input blocks
- input1Stride : Stride of the input image in pixels
- input2Stride : Stride of the bilinear weights input in pixels
- outputStride : Stride of the bilinear interpolated gradient X or gradient Y output in pixels
- blkWidth : Output block width in pixels
- blkHeight : Output block height in pixels
- blkOffset : Offset between the two output blocks in bytes.
- shiftValue : Indicates the amount of shift that need to be applied on bilinear interpolated gradient pixel outputs

- numKeyPoints : Number of input blocks for which bilinear interpolated output pixels of gradient X or gradient Y need to be computed

Constraints

- Input block height and block width should be at least equal to (blkWidth+1) & (blkHeight+1) to account for the border pixels needed for the bilinear interpolation

Techniques

- Utilizes the two functional units effectively to obtain optimal performance.
- Utilizes the de-interleave load instructions to concurrently process Gradient X and Gradient Y input pixels to do bilinear interpolation.

31. Structure Tensor Matrix Determinant

Description

This kernel is used to compute the determinant of the structure tensor The following equations together compute the 2×2 structure tensor matrix M from the gradX and gradY matrices, where the summations are over the user defined pixel neighborhoods. This kernel takes the summation of gradients over the given neighborhood as input and outputs the determinant using exponential notation.

$$M(1,1) = \text{sum}(\text{gradX})^\wedge 2$$
$$M(1,2) = M(2,1) = \text{sum}(\text{gradX} \times \text{gradY})$$
$$M(2,2) = \text{sum}(\text{gradY})^\wedge 2$$

The determinant consists of two parts namely 1) number of non-redundant significant bits, d_nrsb and 2) normalized determinant, d_norm. To obtain the 64 bit determinant, d of the structure tensor matrix, one can use the following computation: d = ((1<< (64 - d_nrsb)) | d_norm)

API

This routine is C-callable and can be called as:

```
void vcop_calc_determinant_tensor_matrix
  (
  __vptr_uint16   Ix2L_a,
  __vptr_uint16   Iy2L_b,
  __vptr_uint16   IxyL_c,
  __vptr_int16    Ix2H_a,
  __vptr_int16    Iy2H_b,
  __vptr_int16    IxyH_c,
  __vptr_uint16   d_nrsb_a,
  __vptr_int32    d_norm_b,
  unsigned short  n
  )
```

- Ix2L_a : Pointer to the lower 16 bits of the 32-bit sum(gradX)^2 of each key point
- Iy2L_b : Pointer to the lower 16 bits of the 32-bit sum(gradY)^2 of each key point
- IxyL_c : Pointer to the lower 16 bits of the 32-bit sum(gradX * grad Y)^2 of each key point
- Ix2H_a : Pointer to the upper 16 bits of the 32-bit sum(gradX)^2 of each key point
- Iy2H_b : Pointer to the upper 16 bits of the 32-bit sum(gradY)^2 each key point

IxyH_c	: Pointer to the upper 16 bits of the 32-bit sum(gradX * grad Y)^2 of each key point
d_nrsb_a	: Pointer to the exponential part of determinant of each key point. Indicates the number of non-redundant significant bits
d_norm_b	: Pointer to the normalized determinant of each key point
n	: Number of key points

Constraints

- It computes determinant of only 2x2 structure tensor matrix.

Techniques

- Extended precision multiplication is used along with useful VCOP instructions such as *leading_bit* and *select*.

32. Structure Tensor Matrix Inverse

Description

This kernel is used to compute the inverse of the 2x2 structure tensor matrix. It takes the structure tensor matrix and the determinant as inputs and provides the inverse of the 2x2 structure tensor matrix using exponential and fractional notation in interleaved format for each key point.

API

This routine is C-callable and can be called as:

```
void vcop_calc_inverse_structure_tensor_2x2
(
   __vptr_int32      pTensorArrInp_A,
   __vptr_uint16     pD_nrsb_B,
   __vptr_uint32     pD_norm_C,
   __vptr_int16      pInverseArrOut_A,
   __vptr_int32      pScratchNorm_C,
   __vptr_uint32     pScratchDividend_C,
   unsigned short    inputStride,
   unsigned short    outputStride,
   unsigned short    numFracBits,
   unsigned short    numKeyPoints
)
```

- pTensorArrInp_A : Pointer to the tensor array elements input for each key point with dimension of numKeyPoints x 3; first row denotes the sum(gradX)^2, second row denotes the sum(gradY)^2 & the third row denotes the sum(gradX * gradY)^2 for each key point
- pD_nrsb_B : Pointer to the exponential part of determinant of each key point. Indicates the number of non-redundant significant bits
- pD_norm_C : Pointer to the normalized determinant of each key point
- pInverseArrOut_A : Pointer to the structure tensor matrix inverse output of each keypoint; (sum(gradX)^2/d), (sum(gradY)^2/d), (sum(gradX * gradY)^2/d) for each key point where d is the determinant The above row-wise outputs are in the order specified above The output consists of the fractional and exponential parts in interleaved format. The exponential part is negated output to to aid in performance of LK tracker
- pScratchNorm_C : Pointer to scratch buffer to store intermediate output
- pScratchDividend_C : Pointer to scratch buffer to store absolute of pScratchNorm_C

- inputStride : Stride of the tensor matrix array in pixels
- outputStride : Stride of the inverse matrix output in pixels
- numFracBits : Number of fractional bits to be discarded to handle any Q format
- numKeyPoints : Number of key points

Constraints

- Exponential part of the inverse matrix output is negated to aid in performance of LK tracker

Techniques

- Multiple loops are deployed with scratch buffers to exploit the SIMD computation.
- *leading_bit*, *select* and *apply_sign* VCOP instructions are used as per the computation needs.

33. Lucas Kanade Tracker Specific Kernels

Description

The following kernels are more specific ally tied to the Lucas Kanade Tracker algorithm needs and are not general purpose kernels. Therefore, the API details would not be included for these specific kernels. The kernels have been grouped logically into sub modules w.r.t functionality. Here is a brief description of each of these kernels w.r.t functionality they achieve to aid in Lucas Kanade Tracker algorithm needs.

vcop_weight_computation

This kernel is used to pre-compute the bilinear weights that need to be used during bilinear interpolation of previous frame's image and gradient pixels. It utilizes the previous frame X, Y key points coordinates list which are represented in Q format to compute the bilinear weights for the 2 x 2 pixel neighborhood. This kernel is invoked outside the iterative LK tracker loop for a given set of key points being tracked.

vcop_tensor_matrix_7x7_s16_grad

This kernel is used to compute the elements of structure tensor matrix for a patch window of size 7x7. It utilizes the bilinear interpolated gradient pixel outputs namely gradX and gradY for a patch window of 7x7 centered around each key point or corner in the previous frame to compute the elements of structure tensor matrix. The following equations together compute the 2×2 structure tensor matrix M from the gradX and gradY matrices, where the summations are over 7×7 pixel neighborhoods:

$$M(1,1) = \text{sum}(\text{gradX})\char`^2$$
$$M(1,2) = M(2,1) = \text{sum}(\text{gradX} \times \text{gradY})$$
$$M(2,2) = \text{sum}(\text{gradY})\char`^2$$

vcop_weight_address_bilinear_interpolation

This kernel is primarily used for computing the updated bilinear weights used for interpolation of current frame pixels and for updating the address of the updated 7x7 patch window across each of the key point's estimates. This kernel updates the parameter block of the vcop_foreach_multiblock_bilinear_interp_7x7_u8 with the updated addresses for the 7x7 patch windows of each key point along with weights. It ensures that only the key points who have not met early exit conditions are being updated continuously within the parameter block of vcop_foreach_multiblock_bilinear_interp_7x7_u8.

vcop_foreach_multiblock_bilinear_interp_7x7_u8

This kernel is used for performing bilinear interpolation of the 7x7 patch windows across updated key point's estimates within the current frame. The iterative LK tracker loop would update the key point's estimates with the flow vectors found at the end of each iteration. This requires update of weights and the 7x7 patch windows within the current frame. This kernel utilizes the updated 7x7 patch window and bilinear weights to compute the bilinear interpolated 7x7 patch window of the current frame across each key point. This kernel uses the repeat loop and the repeat loop counter is modified accordingly depending on the number of key points who has not met the early exit condition to obtain optimal performance.

vcop_sum_grad_cross_inter_frame_diff_7x7

This kernel is used for computing the product of the bilinear interpolated gradient and the temporal difference between the 7x7 patch windows of previous and current frames across their respective key point estimates as shown in the equation below. It should be noted that the bilinear interpolated pixels of the previous and current frame 7x7 patch windows are used.

vcop_calc_new_lk_xy

This kernel is used to compute the flow vectors (V_x, V_y) for the key points that are being tracked by the LK tracker in the current frame w.r.t previous frame key point's coordinates. This kernel also handles the early exit scenario and border conditions as well. In the Lucas Kanade tracker algorithm for the given pyramid level, the key points get tracked in an iterative loop wherein the key points position in current frame get updated with the flow vector computed using the equation below.

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \underbrace{\begin{bmatrix} \sum_i w_i I_x(q_i)^2 & \sum_i w_i I_x(q_i) I_y(q_i) \\ \sum_i w_i I_x(q_i) I_y(q_i) & \sum_i w_i I_y(q_i)^2 \end{bmatrix}}_{\text{Structure Tensor, S}}^{-1} \begin{bmatrix} -\sum_i w_i I_x(q_i) I_t(q_i) \\ -\sum_i w_i I_y(q_i) I_t(q_i) \end{bmatrix}$$

Since the EVE implementation is block based, the algorithm would fetch a larger patch window of current frame centered across key point in to internal memory and picks relevant 7x7 patch window across the updated key point position iteratively and thereby computes the flow vector as the per equation mentioned above. It should be noted that unit weights are considered in the current implementation. It should also be noted that the I_x, I_y denote the bilinear interpolated gradient pixel outputs and I_t denotes the temporal difference between the bilinear interpolated 7x7 patch windows in the previous and current frame across their respective key point estimates.

The early exit condition can arise due to any of the following conditions. 1) Flow vectors found in the current iteration is less than the user specified minimum error. 2) Updated key point's estimates are crossing the border of the search window for the current frame image pixels. This kernel ensures that the flow vectors are made zero for those key points who meet any of the above conditions at any iteration.

vcop_copy_new_lk_xy

This kernel is used for copying the tracked key point's coordinates to the output buffer from the scratch buffer used by the iterative LK tracker loop. It should be noted that the scratch buffer which gets updated with tracked key point's coordinates will be initialized with the next set of key point's coordinates in the iterative LK tracker loop by ARP32 to keep it prepared for next compute trigger. Now the issue of synchronization plays a key role if this scratch buffer is connected to output port of DMA Sink node. That is, DMA should complete before the re-initialization of scratch buffer with next set of key points. In order to address this scenario, this kernel is used to copy the output from scratch buffer to the output buffer.

vcop_sad_error_measure_lk

This kernel is used for computing the SAD based error measure for the key points being tracked by the Pyramid LK tracker. It will first compute the bilinear interpolated patch window across the original key point location in previous frame and corresponding tracked point location in current frame. Now, these patch windows are used to compute the SAD based error measure output. It should be noted that this API is invoked only for the last pyramid level corresponding to original resolution after exiting from the iterative LK loop. This kernel also gives out the count of elements which are above the SAD threshold

34. Remap Kernels

Description

This function supports two approaches of Remap – the Tile Approach and the Bounding Box Approach. In case of Tile approach, this function transforms one or more input tiles of pixels by remapping pixels in the input tile to a new position in the output block. In case of Bounding Box approach, this function transforms a single input block or bounding box of pixels by remapping pixels in the input tile to a new position in the output block. The function uses a user-defined backmapping lookup table to find the corresponding input pixel for each output pixel. The lookup table is specified as a 1D table so 2-D coordinates must be converted to 1-D when generating the table. The lookup table consists of both the Integer Offset and the fractional parts of both the X and Y coordinates. The fractional parts are used for the interpolation. The lookup table also consists of Scatter Store Offsets in case of Tile approach. These Offsets are the locations in the Output block to store the looked up and interpolated Input pixels.
Details of the multiple kernels needed to perform Remap are given below:

vcop_bilinearInterpolate8b

This kernel is used to fill the output block from pixels in the input block based on the backmapping lookup table for S8 and U8 formats. The kernel is also used for remapping the YUV 420 SP Luma component. During lookup of input pixel, the kernel looks up 4 pixels (x,y), (x+1,y), (x,y+1) and (x+1,y+1) where (x,y) is the Integer Offset in the lookup table. Using the fractional offset in the lookup table, bilinear interpolation is performed on the 4 pixels to generate the output pixel. The Scatter Store Offset is used to write the interpolated pixels in the output block for tile approach.

vcop_bilinearInterpolate16b

This kernel is used to fill the output block from pixels in the input block based on the backmapping lookup table for S16 and U16 formats. Bilinear Interpolation is done exactly the same way as described for kernel vcop_bilinearInterpolate8b.

vcop_deInterleaveYUV422IBE

This kernel is used to deinterleave the Input YUV 422 IBE block into separate Y (luma), U (chroma) and V (chroma) blocks. Deinterleaving needs to be performed before Interpolating the Luma and Chroma components of YUV 422 IBE.

vcop_deInterleaveYUV422ILE

This kernel is used to deinterleave the Input YUV 422 ILE block into separate Y (luma), U (chroma) and V (chroma) blocks. Deinterleaving needs to be performed before Interpolating the Luma and Chroma components of YUV 422 ILE.

vcop_bilinearInterpolateYUV422Iluma

This kernel is used to fill the output block from pixels in the input block based on the backmapping lookup table for YUV 422 ILE and IBE Luma formats. The YUV 422 Luma content is expected to be separated and de-interleaved. Bilinear Interpolation is done exactly the same way as described for kernel vcop_bilinearInterpolate8b. The difference is in storing the luma output as it needs to be stored every alternate pixel in the output block.

vcop_nnInterpolate8b

This kernel is used to fill the output block from pixels in the input block based on the backmapping lookup table for S8 and U8 formats. The kernel is also used for remapping the YUV 420 SP Luma component. During lookup of input pixel, the fractional offsets along the x and y co-ordinates are rounded and appended to the Integer Offset (x,y) from the lookup table to get the Nearest neighboring pixel in the Input block. This pixel can be either (x,y) itself or (x+1,y) or (x,y+1) or (x+1,y+1) where (x,y) is the Integer Offset in the lookup table. The Scatter Store Offset is used to write the looked up pixels in the output block for tile approach.

vcop_chromaTLUIndexCalc

This kernel is used to compute the Chroma Backmapping lookup table and the Chroma Scatter Store offset from the Luma Backmapping lookup table and Luma Scatter Store Offset, respectively, for YUV 420 SP format. Based on the Interpolation technique to be used for the Chroma component, different approaches are taken for generating the Chroma Backmapping lookup table and Scatter Store Offset. If the Chroma component has to be Nearest Neighbor Interpolated, only the Integer Index is generated after consideration of the fractionals for the chroma lookup table. If the Chroma component has to be Bilinear Interpolated, the Integer Index and the fractional index is generated considering the Chroma block height is half of the Luma block height. The Scatter Store Offsets are generated in either case.

vcop_dsTLUindexAndFrac

This kernel is used to compute the Chroma Backmapping lookup table from the Luma Backmapping lookup table for YUV 422 formats. It needs the chroma components (U and V) to be in separate buffers or deinterleaved. Based on the Interpolation technique to be used for the Chroma component, different approaches are taken for generating the Chroma Backmapping lookup table. If the Chroma component has to be Nearest Neighbor Interpolated, only the Integer Index is generated after consideration of the fractionals for the chroma lookup table. If the Chroma component has to be Bilinear Interpolated, the Integer Index and the fractional index is generated considering the Chroma block stride is half the Luma block stride and the number of chroma U pixels is half the number of Luma pixels.

vcop_bilinearInterpolateYUV420SPchroma

This kernel is used to fill the output block from pixels in the input block based on the generated chroma backmapping lookup table and the Chroma Scatter Store offset for YUV 420 Chroma component. Interpolation is done for both the U and V Chroma components in a single invocation. Bilinear Interpolation is done exactly the same way as described for kernel vcop_bilinearInterpolate8b.

vcop_bilinearInterpolateYUV422Ichroma

This kernel is used to fill the output block from pixels in the input block based on the generated chroma backmapping lookup table for YUV 422 Chroma component. The Luma Scatter Store offset is used to write pixels into the output block for tile approach. The kernel needs to be invoked separately for interpolating the U and V chroma components. Bilinear Interpolation is done exactly the same way as described for kernel vcop_bilinearInterpolate8b.

vcop_nnInterpolate420SPchroma

This kernel is used to fill the output block from pixels in the input block based on the generated chroma backmapping lookup table and the Chroma Scatter Store offset for YUV 420 Chroma component. Interpolation is done for both the U and V Chroma components in a single invocation. Nearest Neighbor Interpolation is done exactly the same way as described for kernel vcop_nnInterpolate8b.

.

vcop_nnInterpolate422Ichroma

This kernel is used to fill the output block from pixels in the input block based on the generated chroma backmapping lookup table for YUV 422 Chroma component. The Luma Scatter Store offset is used to write pixels into the output block. The kernel needs to be invoked separately for interpolating the U and V chroma components. Nearest Neighbor Interpolation is done exactly the same way as described for kernel vcop_nnInterpolate8b.

vcop_memcpy

This kernel is used to copy the Remapped output from work buffer (WBUF) to IBUF. It is invoked only in the case of Tile approach as the Remapped output is generated across multiple iterations of VCOP execution. Hence it is necessary to have the output buffer in WBUF as it can be accessed during ping and pong iterations.

35. Gray-Level Co-occurrence Matrix

Description

This consists of a set of four kernels used for computing gray-level co-occurrence matrix (GLCM) of an 8-bit grayscale image. The gray-level co-occurrence matrix or gray tone spatial dependency matrix for an input image is generally employed for texture analysis. The input for GLCM computation is an input image that has already been binned to 'numLevels' bins. User provides information regarding the direction of analysis in the form of a 2-D offset. The kernels allow for analysis of multiple directions of analysis together. The parameter 'numOffsets' can be used to indicate the number of directions of analysis to be computed together.

vcop_initialize_glcm

This kernel initializes the GLCM output buffer in internal memory (WBUF) to 0. This kernel should be called once per frame. The parameter 'numLevels' dictates the size of each individual GLCM matrices. User needs to program the parameter 'numCopies' such that he takes care of number of histogram channel used for analysis and number of directions ('numOffsets').

vcop_glcm_compute_1c

This kernel votes the input image pixels pair into internal GLCM output in internal memory using 1 channel histogram.

vcop_glcm_compute_8c

This kernel votes the input image pixels pair into internal GLCM output in internal memory using 8 channel histogram.

vcop_accumulate_8c_glcm

In case vcop_glcm_8c kernel is employed user needs to accumulate the 8 channel histogram (internal) to final GLCM matrix.

Constraints

- The internal output GLCM matrix (in WBUF) should fit into 24 KB of internal memory
- Output GLCM data will be clipped to 16 bit.
- Range of offsetX, and offsetY is limited to [-16, 16].

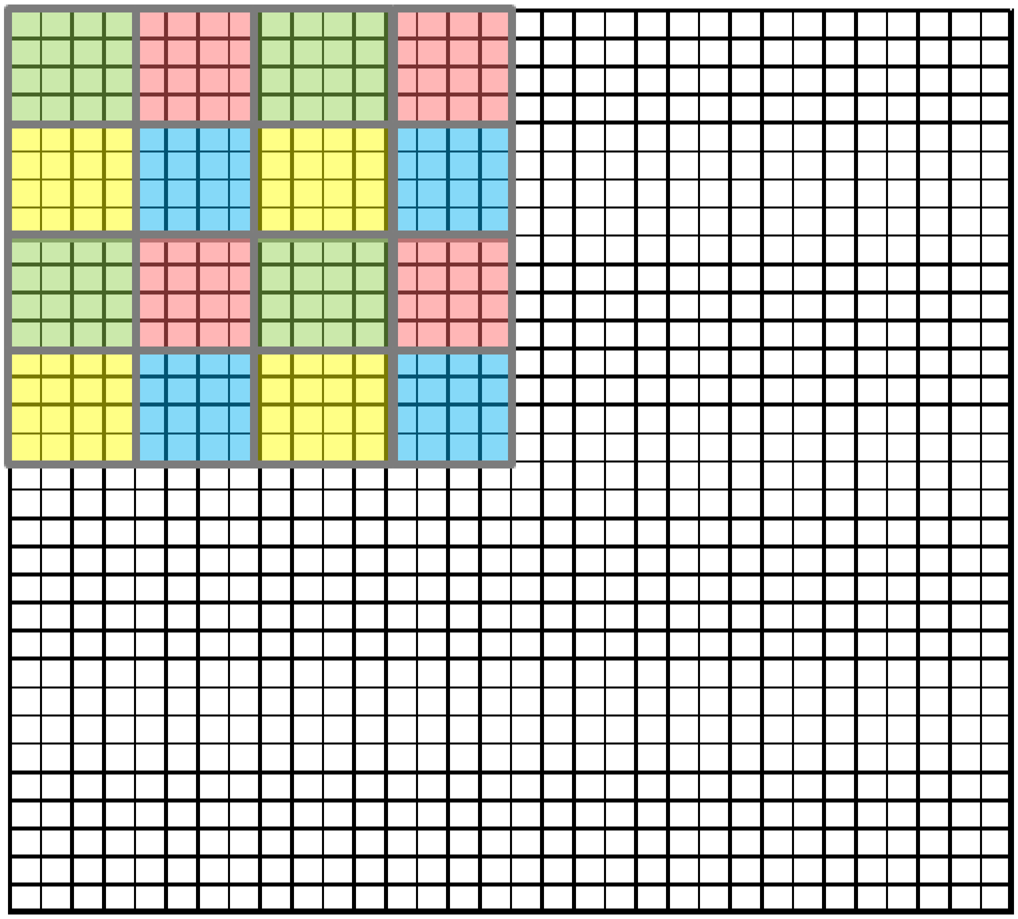
Techniques

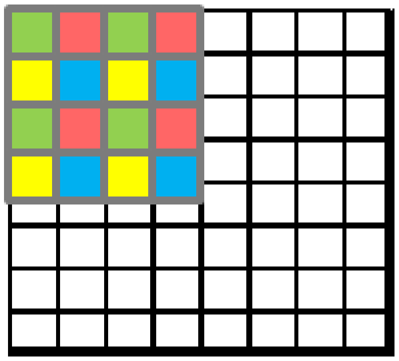
- Generic transpose store for any odd word offset greater than 8.
- 8 channel histogram.

36. Block sum

Description

This kernel implements NxN block sum of input 2D data. The output data elements width will be N time smaller compared input width. Please refer the below picture for more details. In this picture N value is 4. To make use EVE two image buffers and functional units in parallel, this function is written to perform block sum on two input blocks. VLIB has two flavours of this function, one for 8 bit put and another for 16 bit input.





API

This routine is C-callable and can be called as:

```
vcop_nxn_sum_u8(
  __vptr_uint8 inPtr1,
  __vptr_uint8 inPtr2,
  __vptr_uint16 outPtr1,
  __vptr_uint16 outPtr2,
  __vptr_uint32 tempPtr1,
  __vptr_uint32 tempPtr2,
  unsigned short n,
  unsigned short width,
  unsigned short height,
  unsigned short pitch1,
  unsigned short pitch2,
  unsigned short pblock[])
```

• inPtr1	: Pointer to first 8-bit input block
• inPtr2	: Pointer to second 8-bit input block
• outPtr1	: Pointer to first 16-bit output block
• outPtr2	: Pointer to Second 16-bit output block
• tempPtr1	: temporary buffer
• tempPtr2	: temporary buffer
• n	: sum block size
• width	: Processing block width
• height	: Processing block height
• pitch1	: Pitch of first input block
• pitch2	: Pitch of Second input block

Constraints

- The Processing block width and height shall be multiple of 'N'

Performance Considerations

- Buffers related to two input blocks shall be kept in separate buffet for best performance.
- Performance of this function will be : 1/16 + 1/(16*N) + 1/(16*N*N) cycles per pixel

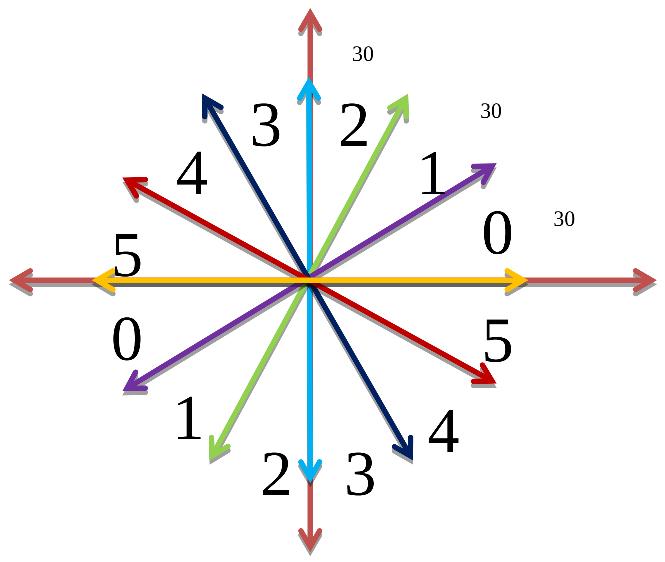
Techniques

- Uses transpose store after the vertical sum to utilized the SIMD efficiently

37. Orientation Binning

Description

This kernel implements Orientation binning VCOP code. This accepts signed gradient X and Gradient Y as inputs. It generates binned orientation for each input pair. The number of bins between 0 -180 degree can be programmed by user. Below picture is example for 6 Bins case



This function used below method to calculate the binning

tan(ϴ) = y/x
sin(ϴ)/cos(ϴ) = y/x
y*cos(ϴ) – x*sin(ϴ) = 0
y*cos(ϴ+d) – x*sin(ϴ+d) = Err

Sign of these terms (which are being subtracted) will same only in quadrant where the x and y are present. So absolute of result value can be minimum in that quadrant.
absolute of "Err" is directly proportional to absolute of "d".
So Err will be evaluated for number of angle values (example, 15,45,75,105,135,165 for 6 bins case) which ever angle is closer to ϴ will produce minimum Err.

$\tan^{-1}(y/x) = \min(abs(yCos(\Theta) - xSin(\Theta)))$
$\Theta \rightarrow$List of binning angles

API

This routine is C-callable and can be called as:

```
unsigned int vcop_orientation_binning (
  __vptr_int16 gradX,
  __vptr_int16 gradY,
  __vptr_uint8 outBin,
  __vptr_int16 wSinTab,
  __vptr_int16 wCosTab,
  unsigned short numBins,
  unsigned short width,
  unsigned short height,
  unsigned short pitch
```

- gradX : Pointer to Horizontal gradient
- gradY : Pointer to Vertical gradient
- outBin : Binned orientation output
- wSinTab : Sin ϴ values in Q16 format for given number of thetas
- wCosTab: Cos ϴ values in Q16 format for given number of thetas
- numBins : Number of bins to be computed
- width : Processing block width
- height : Processing block height
- pitch : Pitch of first input Gradient X and Y

Constraints

- None

Performance Considerations

- Performance of this function will be : **6**/16 x num_bins + 3/16 cycles per pixel

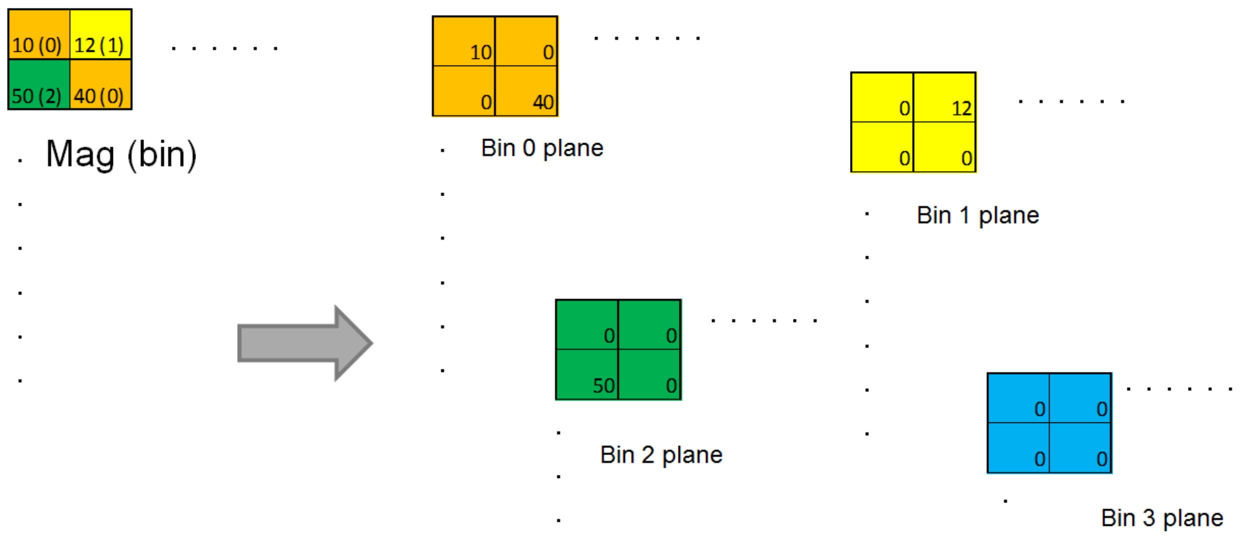
Techniques

- None

38. Location Matrix

Description

This kernel accepts gradient magnitude and gradient orientation bin and generates N number of plans based on the bin values. Below example picture shows a case where we have 4 bin values (0, 1,2,3) . To achieve better performance, this function creates N/2 number of planes in one image buffer and other in another image buffer. Here N is number of bins. VLIB has two flours of this function one for 8 bit magnitude and another for 16 bit magnitude



API

This routine is C-callable and can be called as:

```
vcop_vec_L_matrix_u8_Mag (
  __vptr_uint8 pIn,
  __vptr_uint8 pMag,
  __vptr_uint8 pOut1,
  __vptr_uint8 pOut2,
  __vptr_uint32 pOutClear1,
  __vptr_uint32 pOutClear2,
  unsigned short width,
  unsigned short height,
  unsigned short numBins,
  __vptr_uint8 binIdx,
  unsigned short pblock[])
```

- pIn: Pointer to Input Bin values

- pMag: Pointer to Input gradient magnitude
- pOut1: Pointer to first N/2 Bin Planes
- pOut1: Pointer to Second N/2 Bin Planes
- pOutClear1 : Pointer to first N/2 Bin Planes.
- pOutClear2 : Pointer to Second N/2 Bin Planes
- numBins : Number of bins to be processed
- width : Processing block width
- height : Processing block height
- binIdx : Pointer to list of bin values to be processed

Constraints

- None

Performance Considerations

- Buffers related to two output blocks shall be kept in separate buffet for best performance.
- Performance of 8 bit magnitude function will be : (1/64 + (5/4)/16)* numBins cycles per pixel
- Performance of 16 bit magnitude function will be : (1/32 + (5/4)/16)* numBins cycles per pixel

39. Prune big list

Description

This routine accepts an input list, an array of indices packed with score whihc are sorted based on score and a threshold. It outputs the pruned list of bestN from the input based on the ordering in the index list provided. It also provides a count of elements included in the bestN which are not greater than the threshold. This count can be used later on to offset the pruned output list to exclude those elements which are not greater than the threshold.

Usage

In lot of algorithms, it is required to pick the best N elements from a sorted list and output their (x,y) locations in the image. In such cases, this kernel can be used. The packed (x,y) list is the input list, the score packed with indices is the other input. The LUT pipeline of EVE is used to look up the (x,y) locations from the input list using the indices. A compute loop is used to compare the score values against the threshold to get the count of elements which are not greater than threshold. Since the score list is sorted, we employ early exit once the threshold is less than the score.

API

```
void vcop_prune_big_list
(
  unsigned int   *  inList_A,
  unsigned short *  inSortedIndexList_B,
  unsigned int   *  inSortedIndexList32_B,
  unsigned int   *  outList_C,
  unsigned int   *  nonBestNSize_C,
  unsigned short    threshold,
  unsigned short    bestN
);
```

inList_A	→ Pointer to the input list that has to be pruned
inSortedIndexList_B	→ 16 bit Pointer to the sorted index list
inSortedIndexList32_B	→ 32 bit Pointer to the sorted index list
outList_C	→ Pointer to the pruned output list
nonBestNSize_C	→ Pointer to the count of non bestN elements
threshold	→ Threshold to be considered
bestN	→ Number elements to be output

Performance Considerations

- Buffers related to two input lists and output list shall be kept in separate buffet for best performance.
- Performance function will be: 1 to (1 + 6/16)* cycles per pixel. The second loop performance can be negligible if the early exit condition is satisfied.

To get the best performance, the buffer placement is very important – below table summarize the suggested placement of buffers

Parameter	purpose	memory area
inList_A	Pointer to the input list that has to be pruned	IBUFL
inSortedIndexList_B	Pointer to the sorted index list	IBUFH
outList_C	Pointer to the pruned output list	WBUF
nonBestNSize_C	Pointer to the count of non bestN elements	WBUF

40. Census transform

Description

This kernel computes the census transform of an input block. There are two versions of the kernel: one that accepts 8-bits input and one that accepts 16-bits input.
The function is generic enough to accept any dimensions (winWidth x winHeight) for the support window. For each input pixel, a bit string, composed of the boolean comparisons with each of its neighborhood inside the support window, is produced. A bit '1' indicates that the pixel's value is greater or equal than its neighbor's value, '0' indicates the opposite. The bit string is produced in a raster scan format with bit #0 corresponding to the comparison with the upper left neighbor and the last bit corresponding to the comparison with the lower right neighbor.

In theory, the bit string length should be windWidth * windHeight bits (note that center pixel comparison is included). But bit string lengths that are not multiple of 8, are hard to manipulate because they don't round up to an even number of byte and would need concatenation for post processing. Thus the function produces a more friendly representation by rounding up the bit string to a byte boundary. To achieve this, bits of value 0 are inserted at the end of each bit string until a byte boundary is reached.

Example

Let's have this 3x3 pattern:

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

Applying a 3x3 census transform to the center pixel would produce the following bit string:

Bit #0	Bit #1	Bit #2	Bit #3	Bit #4	Bit #5	Bit #6	Bit #7	Bit #8	Bit #9	Bit #10	Bit #11	Bit #12	Bit #13	Bit #14	Bit #15
1	1	1	1	1	1	1	0	1	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Note that bits #9 to #15 are 0 padding bits inserted to make the bit string align with a byte boundary.

Speed-up with downsampling
It is often desirable to keep a codeword length small in order to speed up processing and keep the amount of memory to store the output low. For instance a 15x15 census window requires floor((15*15+7)/8)= 29 bytes per codeword. The resulting census frame will be 29x larger than the input frame assuming 8-bits input element. Census transform outputs are generally fed to some feature matching algorithm relying on hamming distance and codeword sizes of 1, 2, 4 bytes are more manageable as they typically correspond to a processor's data bandwidth. So a codeword size of 29 bytes is certainly too big to be efficiently handled. As a mean to limit codeword's size, this

function offers the option to downsample a support window in the horizontal or vertical direction through the setting of parameters winHorzStep and winVertStep. The downside of downsampling is of course less information captured in the codeword. Since neighbor pixels are usually highly correlated, the loss of information may be a small, whereas the benefit in processing speed-up and memory saving is quiet significant.

To illustrate the concept, let's have a 5x5 support window:

o		o		o
o		X		o
o		o		o

If winHorzStep= winVertStep= 1, the center pixel 'X' would have to be compared with each of its 25 neighbors, producing a 4 bytes codeword (25 bits aligned to a byte boundary). But if winHorzStep= winVertStep= 2, then only the neighbors represented by a 'o' are used to generate a codeword, which will be 2 bytes long.

Number of bytes per census codeword
In conclusion, the formula that gives the length of each bit string, in number of bytes is as follow:

numBytesPerCensus= (((winWidth + winHorzStep -1)/ winHorzStep)*((winHeight + winVertStep -1)/ winVertStep) + 7)/8 .

So for a 15x15 census window and winHorzStep =2, winVertStep =2, a codeword length would be 8 bytes.

8-bits input API

This routine is C-callable and can be called as:

```
void vcop_census_8bits
(
        __vptr_uint8 pIn8,
        __vptr_uint16 pIn16,
        __vptr_uint8 pOut,
        __vptr_uint8 pScratchBitmask,
        __vptr_uint8 pScratch8,
        __vptr_uint16 pScratch16,
        __vptr_uint16 pOffset,
        __vptr_uint8 pCodeWordMask,
        __vptr_uint8 pRowMask,
        unsigned char winWidth,
        unsigned char winHeight,
        unsigned char winHorzStep,
        unsigned char winVertStep,
        unsigned short computeWidth,
        unsigned short computeHeight,
        unsigned short inStride,
        unsigned short outStride,
        unsigned short scratchStride
)
```

- pIn8: Pointer to the input image block. Each element of the block should be in 8-bits format.
- pIn16: Pointer to the input image block. Has the same pointer value as pIn8, except type is __vptr_uint16. Although seemingly redundant, this argument is required because the kernel-C implementation is not able to typecast pointer and at one point in the code, it needs a pointer to 16-bits to accelerate a copy operation.
- pOut : Pointer to the output block, made of bit strings.
- pScratchBitmask: Pointer to scratch memory of size 2*winWidth*winHeight + 16)*((computeWidth+15)/16)*computeHeight bytes . Preferably, it should be located in IBUFHA.
- pScratch8: Pointer to 4 bytes aligned scratch memory of size MAX(computeWidth*scratchStride*(winWidth*winHeight+7)/8, inStride * (computeHeight + winHeight -1) + 15) bytes. Preferably it should be in WBUF.
  Note:
  ALIGN_8(x) rounds up ‘x’ to next multiple of 8.
  ALIGN_32(x) rounds up ‘x’ to next multiple of 32.
  VCOP_SIMD_WIDTH=8
  VCOP_SIMD_WIDTH2= 16
  ALIGN_SIMD2(x) rounds up ‘x’ to next multiple of 2*SIMD_WIDTH=16

- pScratch16: Same value as pScratch8, except type is __vptr_uint16.
- pOffset: pointer to an array of 8 half-words, initialized by init_census_8bits_params(). Preferably in WBUF.
- pCodeWordMask: pointer to an array of (winWidth*winHeight+7)/8 bytes, initialized by init_census_8bits_params().
- pRowMask pointer to an array of (computeHeight+7)/8 bytes, initialized by init_census_8bits_params().
- winWidth: width of the support window, that defines the neighborhood in which census transform is applied around each pixel.
- winHeight: height of the support window, that defines the neighborhood in which census transform is applied around each pixel.
- winHorzStep: horizontal step between each neighbor position within the support window.
- winVertStep: vertical step between each neighbor position within the support window.
- computeWidth: number of bit strings produced in each output row. Must be multiple of 16.
- computeHeight: number of rows produced. For best performance should be multiple of 8.

- inStride: stride of the input block, in bytes. Must be greater or equal than computeWidth + windWidth – 1.
- outStride: stride of the output block, in bytes. Should be at least equal to computeWidth * (windWidth * windHeight +7)/8. However two extra constraints must be respected: must be multiple of 4, but not multiple of 32.
- scratchStride: initialized by init_census_8bits_params() which should set it to ALIGN_32(computeHeight) + 4 bytes.

Constraints

- pScratch8, pScratch16 must be 4 bytes aligned.
- computeWidth must be multiple of 16.
- computeHeight should be multiple of 8 for best performance.
- inStride >= computeWidth + windWidth – 1.
- outStride >= computeWidth * (windWidth * windHeight +7)/8.
- outStride must be multiple of 4.
- outStride must not be multiple of 32.

Some of the arguments need to be initialized prior to calling vcop_census_8bits(). These arguments are the arrays pointed by pOffset, pCodeWordMask, pRowMask. The function used to initialize them is init_census_8bits_params:

```
int32_t init_census_8bits_params
(
        uint8_t winWidth,
        uint8_t winHeight,
        uint8_t winHorzStep,
        uint8_t winVertStep,
        uint16_t computeWidth,
        uint16_t computeHeight,
        uint16_t outStride,
        uint16_t *pOffset,
        uint8_t sizeOffsetArray,
        uint8_t  *pCodeWordMask,
        uint8_t sizeCodeWordMarkArray,
        uint8_t  *pRowMask,
        uint8_t sizeRowMaskArray,
        uint16_t *pScratchStride);
```

Most of the arguments passed to init_census_8bits_params () are the same as vcop_census_8bits() except for sizeOffsetArray, sizeCodeWordMarkArray, sizeRowMaskArray, which are additional arguments, whose values are size of the different arrays, in bytes.

The function will initialize accordingly the content pointed by pOffset, pCodeWordMask, pRowMask, pScratchStride and can return the following error code:

- 0: no error
- -1: Constraint outStride >= ((winWidth*winHeight+7)/8)*computeWidth not respected.
- -2: Constraint outStride multiple of 4, not respected.
- -3: Constraint outStride not multiple of 32, not respected.
- -4: Constraint of sizeOffsetArray to be 16 bytes, not respected.
- -5: Constraint of sizeCodeWordMarkArray to be (winWidth*winHeight+7)/8 bytes, not respected.
- -6: Constraint of sizeRowMaskArray to be (computeHeight+7)/8 bytes, not respected.

16-bits input API

The API is very similar to the 8-bits version:

```
void vcop_census_16bits
(
        __vptr_uint16 pIn16,
        __vptr_uint8 pOut,
        __vptr_uint8 pScratchBitmask,
        __vptr_uint16 pScratch16,
        __vptr_uint8 pScratch8,
        __vptr_uint16 pOffset,
        __vptr_uint8 pCodeWordMask,
        __vptr_uint8 pRowMask,
        unsigned char winWidth,
        unsigned char winHeight,
        unsigned char winHorzStep,
        unsigned char winVertStep,
        unsigned short computeWidth,
        unsigned short computeHeight,
        unsigned short inStride,
        unsigned short outStride,
        unsigned short scratchStride
)
```

The only differences are that pIn8 and pScratch8 are removed from the parameters. The constraints are the same.

Constraints

- pScratch16, pScratch8, must be 4 bytes aligned.
- computeWidth must be multiple of 16.
- computeHeight should be multiple of 8 for best performance.
- inStride >= computeWidth + windWidth – 1.
- outStride >= computeWidth * (windWidth * windHeight +7)/8.
- outStride must be multiple of 4.
- outStride must not be multiple of 32.

init_census_16bits_params() has the same interface as init_census_8bits_params():

```
int32_t init_census_16bits_params
(
        uint8_t winWidth,
        uint8_t winHeight,
        uint8_t winHorzStep,
        uint8_t winVertStep,
        uint16_t computeWidth,
        uint16_t computeHeight,
        uint16_t outStride,
        uint16_t *pOffset,
        uint8_t sizeOffsetArray,
        uint8_t  *pCodeWordMask,
        uint8_t sizeCodeWordMarkArray,
        uint8_t  *pRowMask,
        uint8_t sizeRowMaskArray,
        uint16_t *scratchStride);
```

Performance Considerations

- For best performance, these pointers should point respectively to:
    - pIn8, pIn16 to IBUFLA or IBUFHA
    - pOut to IBUFLA or IBUFHA
    - pScratchBitMask to IBUFHA
    - pScratch8, pScratch16 to WBUF
    - pOffset, pCodeWordMas and pRowMask to WBUF

- Performance of 8 bit census transform will be : 1/32 + 3 * winHeight * winWidth / 32 + 2*2*ALIGN_8(winWidth * winHeight) / 128 cycles per pixel
- Performance of 16 bit census transform will be : 1/32 + 4 * winHeight * winWidth / 32 + 2*2*ALIGN_8(winWidth * winHeight) / 128 cycles per pixel

41. Normalized cross-correlation

Description

This kernel computes the normalized cross correlation to find areas of an input block that match (similar) to a template image block.

Normalized cross correlation works as follow: given an input image f and a template image t of dimension templateWidth x templateHeight, for every location (u,v) in the input image, it calculates the following value c(u,v)

$$c(u,v) = \frac{\sum_{x,y} (f(x,y) - \bar{f}_{u,v})(t(x-u, y-v) - \bar{t})}{\sqrt{\sum_{x,y} (f(x,y) - \bar{f}_{u,v})^2 \sum_{x,y} (t(x-u, y-v) - \bar{t})^2}}$$

The index x and y iterates through the neighborhood patch of dimensions templateWidth x templateHeight, centered around the position (u,v). $\overline{f_{u,v}}$ is the mean of the aforementioned neighborhood patch from the input image and $\bar{t}$ is the mean of the template image.

The position (umax,vmax) that coincides with the location where the template best matches the area of the input image corresponds to a local maximum for c(u,v).

Note that the term $\sum_{x,y} (t(x-u, y-v) - \bar{t})^2$ is a constant that can be precomputed in advance or even be ignored as it doesn't affect the location of the local maximum of c(u,v).
The template image t can also be normalized by subtracting its mean to produce a new template t'.

Taking into consideration these simplifications, the equation becomes:

$$c(u,v) = \frac{\sum_{x,y} (f(x,y) - \bar{f}_{u,v}) \cdot t'(x-u, y-v)}{K \cdot \sqrt{\sum_{x,y} (f(x,y) - \bar{f}_{u,v})^2}}$$

The constant K can be set to $\sqrt{\sum_{x,y} (t(x-u, y-v) - \bar{t})^2}$ or to 1 if the main goal is to find the local maximum.
Since EVE is a fixed-point processor, it does not directly compute c(u,v). Indeed in order to avoid doing any division, it computes these two partial outputs:

$$num\_cc(u,v) = \sum\nolimits_{x,y} (f(x,y) - \bar{f}_{u,v}) \cdot t'(x-u, y-v)$$

$$denom\_\mathrm{var}(u,v) = \sum\nolimits_{x,y} (f(x,y) - \bar{f}_{u,v})^2$$

num_cc(u,v) is the numerator part of the cross-correlation c(u,v) and denom_var(u,v) is the the denominator part, which almost corresponds to the variance of the neighborhood patch. In order to increase the precision, the kernel produces these two outputs as 32-bits number in Q format. The number of bits reserved for the fractional part is provided as a parameter qShift to the kernel.

In order to obtain the final result, the following post-processing must be performed:

$$c(u,v) = \frac{num\_cc(u,v)}{K \cdot \sqrt{denom\_\mathrm{var}(u,v)}}$$

The reason why the kernel does not perform this final step is that DSP can efficiently implement it thanks to its floating-point capability.

The kernel accepts as input parameter a pointer to the sum array, which must have been previously generated by vcop_slidingSum().

There are three variants of the function:
vcop_ncc(): used when qShift != 0, qShift !=8.
vcop_ncc_qShift8(): used when qShift =8.
vcop_ncc_qShift0(): used when qShift=0

They all have the same prototype, which is the following:

API

```
void vcop_ncc
(
        __vptr_uint8 pOrgImg,
        unsigned short orgImgWidth,
        unsigned short orgImgHeight,
        unsigned short orgImgPitch,
        __vptr_int16   pTempImg,
        unsigned short tempImgWidth,
        unsigned short tempImgHeight,
        unsigned short tempImgPitch,
        __vptr_uint32   pSum,
        __vptr_uint16   pSumL,
        __vptr_uint16   pSumH,
        unsigned short sumStride,
        unsigned char  sizeQshift,
        unsigned char  qShift,
        __vptr_int32   pOutNumCC,
        __vptr_uint32  pOutDenomVar,
        unsigned short outPitch
);
```

- pOrgImg: Pointer to the input image block. Each element of the block should be in unsigned 8-bits format.
- orgImgWidth: ROI width of the input image block.
- orgImgHeight: ROI height of the input image block.
- orgImgPitch: stride of the input block, in number of elements.
- pTempImg: pointer to the normalized 0-mean template image of 16-bits.
- tempImgWidth: width of the template in number of elements.
- tempImgHeight: height of the template in number of rows.
- tempImgPitch: stride of the template in number of elements.
- pSum: points to the output block of 32-bits elements, previously generated by vcop_slidingSum().
- pSumL: points to the 16 LSBs pointed by pSum.
- pSumH: points to the 16 MSBs pointed by pSum.
- sumStride: initialized by init_slidingSum_params() which should set it to ((4*computeWidth + 31) & (~31)) + 4 bytes.
- sizeQshift: it is used when dividing the sum of the pixels by the number of pixels to produce the mean value. A good value for sizeQshift is log2(tempImgWidth * tempImgHeight).
- qShift: is used for formatting the final output into Q format. A good value for qShift is qShift= min(7, (22 - log2(tempImgWidth * tempImgHeight))/2)
- pOutNumCC: output pointer to the signed 32-bits numerator values of the cross-correlation. There are (orgImgWidth – (tempImgWidth – 1)) * (orgImgHeight – (tempImgHeight – 1)) output values produced.
- pOutDenomVar: output pointer to the unsigned 32-bits denominator values of the cross-correlation. There are (orgImgWidth – (tempImgWidth – 1)) * (orgImgHeight – (tempImgHeight – 1)) output values produced.
- outPitch: stride of the output buffer in number of elements.

  For derivation of sizeQshift and qShift, please consult appendix C of the EVE applets user’s guide.

Constraints

- pSum, must be 4 bytes aligned.
- orgImgWidth – (tempImgWidth - 1) must be multiple of 16.

Performance Considerations

- For best performance, these pointers should point respectively to:
  - pOrgImg to IBUFLA
  - pSum, pSumL, pSumH to IBUFHA
  - pOutNumCC to IBUFHA
  - pOutDenomVar to IBUFHA

- Performance will be :
  - vcop_ncc_qShift0(), vcop_ncc_qShift8():
  (4/16 + 0.25*tempImgWidth*tempImgHeight)*ALIGN(output_width)*output_height cycles
  - vcop_ncc():
  (4/16 + 0.5*tempImgWidth*tempImgHeight)*ALIGN(output_width)*output_height cycles

  Where output_width= (orgImgWidth – (tempImgWidth – 1)) and output_height= (orgImgHeight – (tempImgHeight – 1))

  One can observe that both vcop_ncc_qShift0(), vcop_ncc_qShift8() are twice faster than vcop_ncc().