IMAGE & SIGNAL PROCESSING LIBRARY (IMGSIGLIB) IMPLEMENTATION FOR EVE CO-PROCESSOR

TEXAS INSTRUMENTS INC.

Table of Contents

Table of Figures

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. Median3x3

Description

Perform median filtering around a 3 x 3 neighborhood of each data point passed as input. If input[i, j] is the input array and output[i, j] is the output array with i indexing the row and j indexing the column, then:

For (i, j) ∈ [0, h_blk + 2 -1] x [0, w_blk + 2 -1]:

output[i ,j] = median (input[i, j], input[i, j + 1], input[i, j + 2], input[i + 1, j], input[i + 1, j + 1], input[i + 1, j + 2], input[i + 2, j], input[i + 2, j + 1], input[i + 2, j + 2])

In other words, each output[i , j] is the median of the 3 x 3 neighborhood around input[i + 1, j + 1].

The technique to compute the median is as described below and consists of the following three steps.

The rows are sorted as minimum, median and maximum taken three at a time. Once sorted, these results are written out temporarily to an intermediate buffer. The size of this buffer is (3 x (w_blk x (h_blk + 2)).

On the sorted array, max(min), med(med) and min(max) values are computed, going in the horizontal direction, taking three values at a time. This process still keeps the result buffer size to (3 x (w_blk x (h_blk + 2)).

Finally, the median of three values in the vertical direction is computed. The result is a final median over a window of 3 x 3 pixels and the array size is (w_blk x h_blk).

Given below is an example showing the steps explained above.

Input array ➔

3	8	10	0	0
4	6	5	0	0
7	2	9	0	0
0	0	0	0	0
0	0	0	0	0

After step a) ➔

3	2	5	0	0
4	6	9	0	0
7	8	10	0	0
0	0	0	0	0
4	2	5	0	0
7	6	9	0	0
0	0	0	0	0
0	0	0	0	0
7	2	9	0	0

After step b) ➔

3	5	5	x	x
6	6	0	x	x
7	0	0	x	x
0	0	0	x	x
4	2	0	x	x
6	0	0	x	x
0	0	0	x	x
0	0	0	x	x
2	0	0	x	x

After step c) ➔

6	5	0
4	0	0
0	0	0

Thus, the result is obtained after step c) and is an array of size (3 x 3).

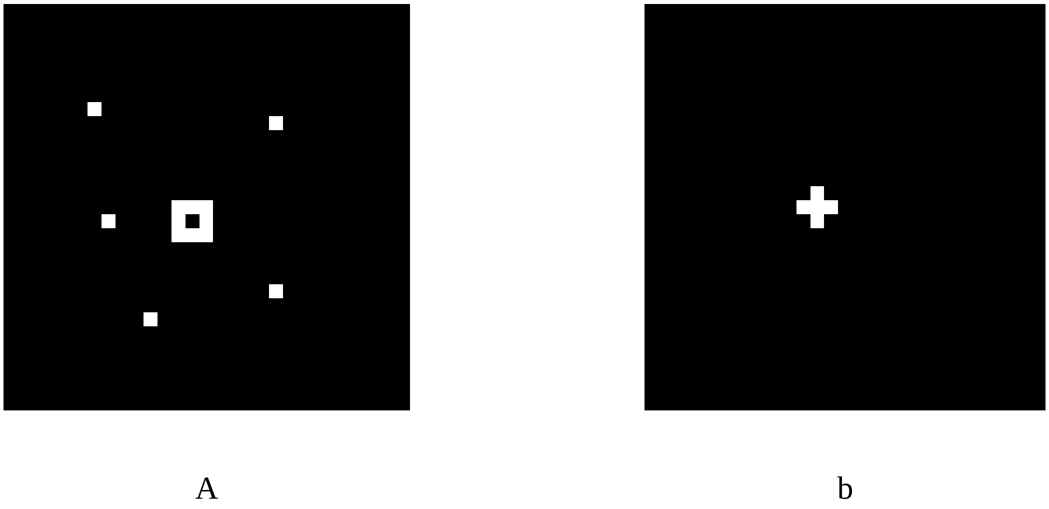


Figure 1a) Input test image b) Median filtered result

The function prototype for the IMGSIGLIB is as follows:

```
void vcop_median3x3_char
(
   __vptr_int8  in,              //  Pointer to an input array of "type_input".
   __vptr_int8  out,             //  Pointer to output array of "type_output".
   __vptr_int8  scratch1,        //  Pointer to intermediate array.
   __vptr_int8  scratch2,        //  Pointer to intermediate array.
   int          w_blk,           //  The block width over which median values are found.
   int          h_blk            //  The block height for median filter.
);

void vcop_median3x3_uchar
(
   __vptr_uint8  in,             //  Pointer to an input array of "type_input".
   __vptr_uint8  out,            //  Pointer to output array of "type_output".
   __vptr_uint8  scratch1,       //  Pointer to intermediate array.
   __vptr_uint8  scratch2,       //  Pointer to intermediate array.
   int          w_blk,           //  The block width over which median values are found.
   int          h_blk            //  The block height for median filter.
);
```

Usage

Both the input and output arrays can be of signed or unsigned char datatypes.

Constraints

1) Intermediate scratch buffers are also used which have sizes of (3 * w_blk * h_blk).
2) w_blk should be a multiple of 16 as de-interleaved loads are used.

Performance Considerations

Estimated cycle count:

Performance = 18/16 cycles/pix

EVE features used

- MINMAX instruction

2. Sobel3x3

Description

The SobelX filter is a 3 X 3 filter and is useful for detecting horizontal edges. The filter structure is as follows:

$$S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

Where the edge detected image is obtained as follows:

$$I_x = S_x * I$$

Where * here denotes the 2-dimensional convolution operation.

Usually, the filter is applied in a separable fashion in the following way:

$$S_{x\_v} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

$$S_{x\_h} = [\ 1 \quad 0 \quad -1\ ]$$

$$I_x = S_{x\_v} * (S_{x\_h} * I)$$

In most implementations of the Sobel filter it is common to use 9 coefficients. However, making use of the address pointer increment feature of EVE vector processor architecture, we can accomplish the same task with 6 coefficients (non-separable implementation) or with 5 coefficients when the separable kernels are used by skipping over zero coefficients when applicable.

Similarly the SobelY filter is a 3 X 3 filter and is useful for detecting horizontal edges. The filter structure is as follows:

$$S_x = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & -1 \end{bmatrix}$$

Where the edge detected image is obtained as follows:

$$I_y = S_y * I$$

Where * here denotes the 2-dimensional convolution operation.

Usually, the filter is applied in a separable fashion in the following way:

$$S_{y\_v} = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$$

$$S_{y\_h} = [\ 1 \quad 2 \quad 1\ ]$$

$$I_y = S_{y\_v} * (S_{y\_h} * I)$$

In most implementations of the Sobel filter it is common to use 9 coefficients. However, making use of the address pointer increment feature of EVE vector processor architecture, we can accomplish the same task with 6 coefficients (non-separable implementation) or with 5 coefficients when the separable kernels are used by skipping over zero coefficients when applicable.

Usage

There are implementation available for SobelX, SobelY and SobelXY as separable filter as those are most optimal implementations. Apart from these there are two more implementation available for thresholding with output to be either in byte edge map or bit packed edge map.

API

```
void vcop_sobelXY_3x3_separable_uchar
(
    __vptr_uint8  inData,
    __vptr_int16  interimDataX,
    __vptr_int16  interimDataY,
    __vptr_int8   outDataX,
    __vptr_int8   outDataY,
    unsigned short computeWidth,
    unsigned short computeHeight,
    unsigned short inputPitch,
    unsigned short outputPitch,
    unsigned short roundShift
)
```

Field	Data Type	Input/ Output	Description
inData	*__vptr_uint8*	Input	Input data pointer. Size of this buffer should be blockWidth * blockHeight * sizeof(uint8_t)
interimDataX	*__vptr_int16*	Scratch	This is pointer to an intermediate scratch buffer to store intermediate X data. Size of this buffer should be (blockHeight * ALIGN_2SIMD(computeWidth) * size(int16_t)
interimDataY	*__vptr_int16*	Scratch	This is pointer to an intermediate scratch buffer to store intermediate Y data. Size of this buffer should be (blockHeight * ALIGN_2SIMD(computeWidth) * size(int16_t)
outDataX	*__vptr_int8*	output	Pointer to the output for X component. Size of this buffer should be (ALIGN_2SIMD(computeWidth) * computeHeight * size(int8_t)
outDataY	*__vptr_int8*	output	Pointer to the output for Y component. Size of this buffer should be (ALIGN_2SIMD(computeWidth) * computeHeight * size(int8_t)
computeWidth	`uint16_t`	Input	Width of the output of this kernel. This is basicaly blockWidth - border
computeHeight	`uint16_t`	Input	Width of the output of this kernel. This is basicaly blockHeight - border
inputPitch	`uint16_t`	Input	Pitch of the input data
outputPitch	`uint16_t`	Input	Pitch of the output data

Field	Data Type	Input/ Output	Description
roundShift	`uint16_t`	Input	Rounding that needs to be applied to the output

```
void vcop_sobelX_3x3_separable_uchar
(
    __vptr_uint8  inData,
    __vptr_int16  interimData,
    __vptr_int8   outData,
    unsigned short computeWidth,
    unsigned short computeHeight,
    unsigned short inputPitch,
    unsigned short outputPitch,
    unsigned short roundShift
)
```

Field	Data Type	Input/ Output	Description
inData	*__vptr_uint8*	Input	Input data pointer. Size of this buffer should be blockWidth * blockHeight * sizeof(uint8_t)
interimData	*__vptr_int16*	Scratch	This is pointer to an intermediate scratch buffer to store intermediate X data. Size of this buffer should be (blockHeight * ALIGN_2SIMD(computeWidth) * size(int16_t)
outData	*__vptr_int8*	output	Pointer to the output for X component. Size of this buffer should be (ALIGN_2SIMD(computeWidth) * computeHeight * size(int8_t)
computeWidth	`uint16_t`	Input	Width of the output of this kernel. This is basicaly blockWidth - border
computeHeight	`uint16_t`	Input	Width of the output of this kernel. This is basicaly blockHeight - border
inputPitch	`uint16_t`	Input	Pitch of the input data
outputPitch	`uint16_t`	Input	Pitch of the output data
roundShift	`uint16_t`	Input	Rounding that needs to be applied to the output

```
void vcop_sobelY_3x3_separable_uchar
(
    __vptr_uint8  inData,
    __vptr_int16  interimData,
    __vptr_int8   outData,
    unsigned short computeWidth,
    unsigned short computeHeight,
    unsigned short inputPitch,
    unsigned short outputPitch,
    unsigned short roundShift
)
```

Field	**Data Type**	**Input/ Output**	**Description**
inData	*__vptr_uint8*	Input	Input data pointer. Size of this buffer should be blockWidth * blockHeight * sizeof(uint8_t)
interimData	*__vptr_int16*	Scratch	This is pointer to an intermediate scratch buffer to store intermediate data. Size of this buffer should be (blockHeight * ALIGN_2SIMD(computeWidth) * size(int16_t)
outData	*__vptr_int8*	output	Pointer to the output for Y component. Size of this buffer should be (ALIGN_2SIMD(computeWidth) * computeHeight * size(int8_t)
computeWidth	`uint16_t`	Input	Width of the output of this kernel. This is basicaly blockWidth - border
computeHeight	`uint16_t`	Input	Width of the output of this kernel. This is basicaly blockHeight - border
inputPitch	`uint16_t`	Input	Pitch of the input data
outputPitch	`uint16_t`	Input	Pitch of the output data
roundShift	`uint16_t`	Input	Rounding that needs to be applied to the output

```
void vcop_sobelXy_3x3_L1_thresholding
(
  __vptr_int8   gradX,
  __vptr_int8   gradY,
  __vptr_uint8  outData,
  unsigned short computeWidth,
  unsigned short computeHeight,
  unsigned short inputPitch,
  unsigned short outputPitch,
  unsigned short  threshold
)
```

Field	Data Type	Input/ Output	Description
gradX	*__vptr_int8*	Input	Gradient X. Size of this buffer should be computeWidth * computeHeight * sizeof(uint8_t)
gradY	*__vptr_int8*	Input	Gradient Y. Size of this buffer should be computeWidth * computeHeight * sizeof(uint8_t)
outData	*__vptr_int8*	output	Pointer to the output of this kenerl containing 255 and places where edges are present and 0 otherwise. Size of this buffer should be (ALIGN_2SIMD(computeWidth) * computeHeight * size(int8_t)
computeWidth	`uint16_t`	Input	Width of the output of this kernel. This is basically blockWidth - border
computeHeight	`uint16_t`	Input	Width of the output of this kernel. This is basically blockHeight - border
inputPitch	`uint16_t`	Input	Pitch of the input data
outputPitch	`uint16_t`	Input	Pitch of the output data
threshold	`uint16_t`	Input	Threshold to be used for thresholding magnitude

```
void vcop_sobelXy_3x3_L1_thresholding_binPack
(
   __vptr_int8   gradX,
   __vptr_int8   gradY,
   __vptr_uint8  outData,
   unsigned short computeWidth,
   unsigned short computeHeight,
   unsigned short inputPitch,
   unsigned short outputPitch,
   unsigned short  threshold
)
```

Field	Data Type	Input/ Output	Description
gradX	*__vptr_int8*	Input	Gradient X. Size of this buffer should be computeWidth * computeHeight * sizeof(uint8_t)
gradY	*__vptr_int8*	Input	Gradient Y. Size of this buffer should be computeWidth * computeHeight * sizeof(uint8_t)
outData	*__vptr_int8*	output	Pointer to the output of this kenerl containing 1 and places where edges are present and 0 otherwise. Size of this buffer should be (ALIGN_2SIMD(computeWidth) * computeHeight * size(int8_t) /8
computeWidth	`uint16_t`	Input	Width of the output of this kernel. This is basically blockWidth - border
computeHeight	`uint16_t`	Input	Width of the output of this kernel. This is basically blockHeight - border
inputPitch	`uint16_t`	Input	Pitch of the input data
outputPitch	`uint16_t`	Input	Pitch of the output data
threshold	`uint16_t`	Input	Threshold to be used for thresholding magnitude

Constraints

1. No Constraints.

EVE features used

1. Deinterleave loads
2. Pack instruction
3. Stores with rounding and saturation

Performance Consideration

1. Performance will be most optimal when computeWidth is multiple of 2 times SIMD width

3. Rotate

Description

Performs rotation by 90, 180 or 270 degree on an input image. This function rotates a submatrix of size compute_width x compute_height of the input data matrix (size input_width x input_height), and writes the rotated submatrix into the output matrix of size output_width x output_height, aligned to the top left corner.

The original VICP library allowed each input data item to be scaled by a scaling factor. This feature has been disabled in the current release and **no** scaling will be carried out on the input data. Subsequently, rounding and saturation are also not needed and this rotate kernel will do neither.

Input and output can be of 8-bit or 16-bit type.

Given below, Figures 2.1 and 2.2, describe the placement of pixels for 90 degree and 270 degree rotations.

EVE has eight memory banks to allow concurrent memory accesses. Due to this, the transpose of any array is written such that each data point is at an offset of 9 words from the previous. As 90 and 270 degree rotates need the transpose feature, they are carried out in two VLOOPS. The first VLOOP writes out blocks of W * (H/32) size to its transposed form using the **‘offset_np1’** qualifier for stores, and the second VLOOP rearranges the pixels as required to obtain the desired rotation.

The kernel to rotate by 180 degrees uses the custom distribution load qualifier **‘dist’**. This allows us to load pixels as p7p6p5…p0 instead of the regular **‘npt’** load while loads p0p1p2…p7. These pixels (loaded as given by **.dist**) are then stored to the correct rows to effectively give a rotation by 180 degrees.

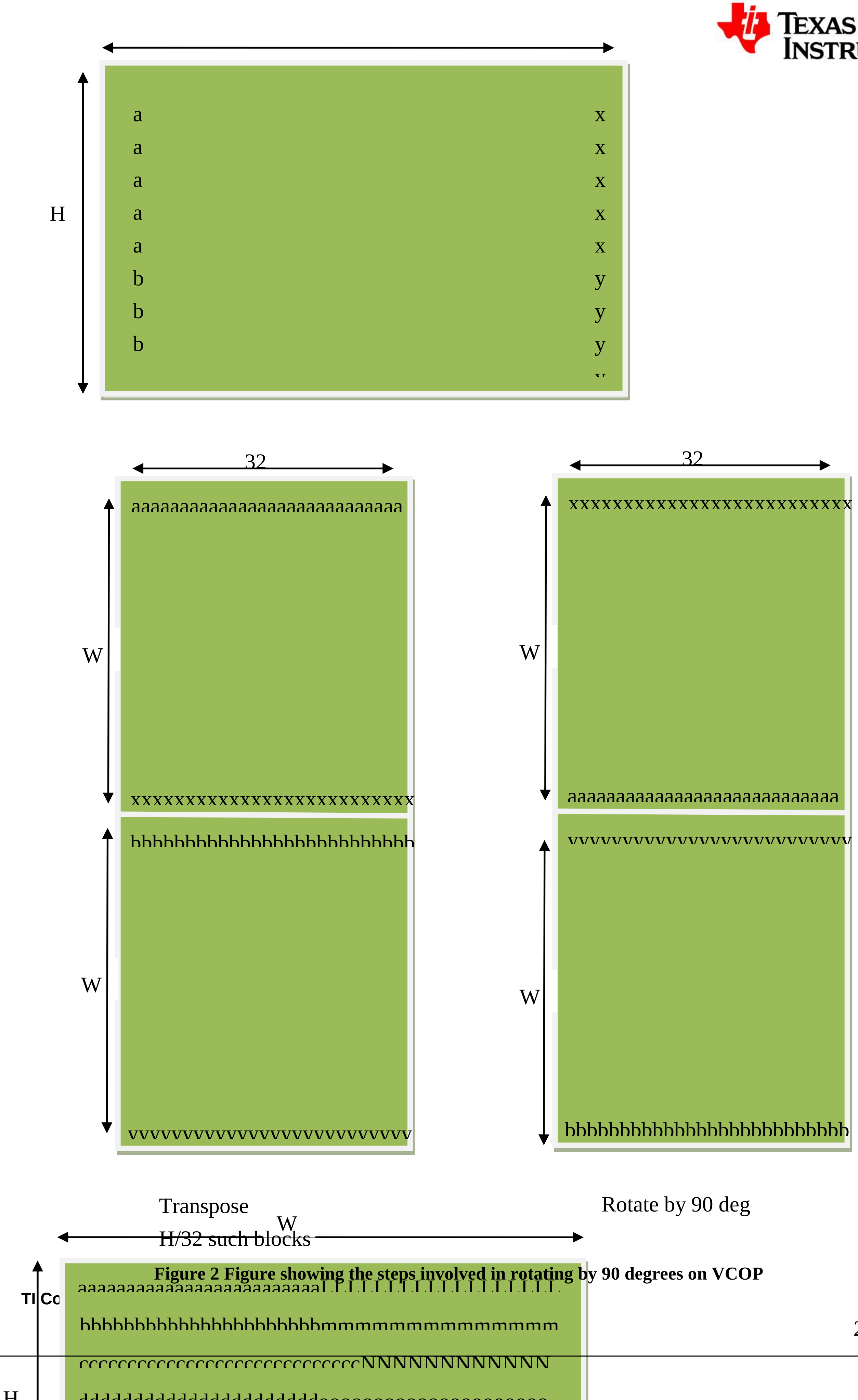

H
a a a a a b b b
x x x x x y y y
32
W
W
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
xxxxxxxxxxxxxxxxxxxxxxxxxxxx
bbbbbbbbbbbbbbbbbbbbbbbbbbbbb
yyyyyyyyyyyyyyyyyyyyyyyyyyyyy
32
W
W
xxxxxxxxxxxxxxxxxxxxxxxxxxxx
aaaaaaaaaaaaaaaaaaaaaaaaaaaaa
yyyyyyyyyyyyyyyyyyyyyyyyyyyyy
bbbbbbbbbbbbbbbbbbbbbbbbbbbbb
Transpose
H/32 such blocks
Rotate by 90 deg
W
H
aaaaaaaaaaaaaaaaaaaaaaaaaaaaLLLLLLLLLLLLLLLLLLL
bbbbbbbbbbbbbbbbbbbbbbbbbbbmmmmmmmmmmmmmmmm
cccccccccccccccccccccccccccccNNNNNNNNNNNNN
dddddddddddddddddddddddddoooooooooooooooooooo

Figure 2 Figure showing the steps involved in rotating by 90 degrees on VCOP

The function prototype for the IMGSIGLIB is as follows:

```
void vcop_rotate_90/180/270
(
    __vptr_uint8  in,          //  Pointer to an input array of "in_type".
    __vptr_uint8  inter,       //  Pointer to intermediate array of "inter_type".
    __vptr_uint8  out,         //  Pointer to output array of "out_type".
    int           img_w,       //  Width of input image in pixels.
    int           img_h,       //  Height of the input image in pixels.
    int           out_w,       //  Width of the output image in pixels.
    int           out_h,       //  Height of the input image in pixels.
    int           blk_w,       //  The compute width over which rotate values are found.
    int           blk_h,       //  The compute height for rotate function.
    int           in_type,     //  0 - BYTE,  1 - SHORT
    int           inter_type,  //  0 - BYTE,  1 - SHORT
    int           out_type,    //  0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
    int           angle,       //  Angle of rotation, can be 90, 180 or 270.
    int           round_shift  //  Amount of rounding and shifting on output data.
);
```

Usage

Input vectors can have signed and unsigned char, short and word formats. Output vector can have signed char, signed short and signed long formats. The final output is of size compute_width * compute_height.

Constraints

1. Compute_height and compute_width must be multiples of 8

Performance Considerations

Estimated cycle count:

Rotate 90/270 degree performance:
(w * h)/4 cycles/pix
Code size = 48 bytes

Rotate 180 degree performance:

(w * h)/8 cycles/pix
Code size = 24 bytes

EVE features used

- Store as transpose (OFFSET_NP1 feature)
- Custom distributed loads (DIST feature)

4. Integral Image

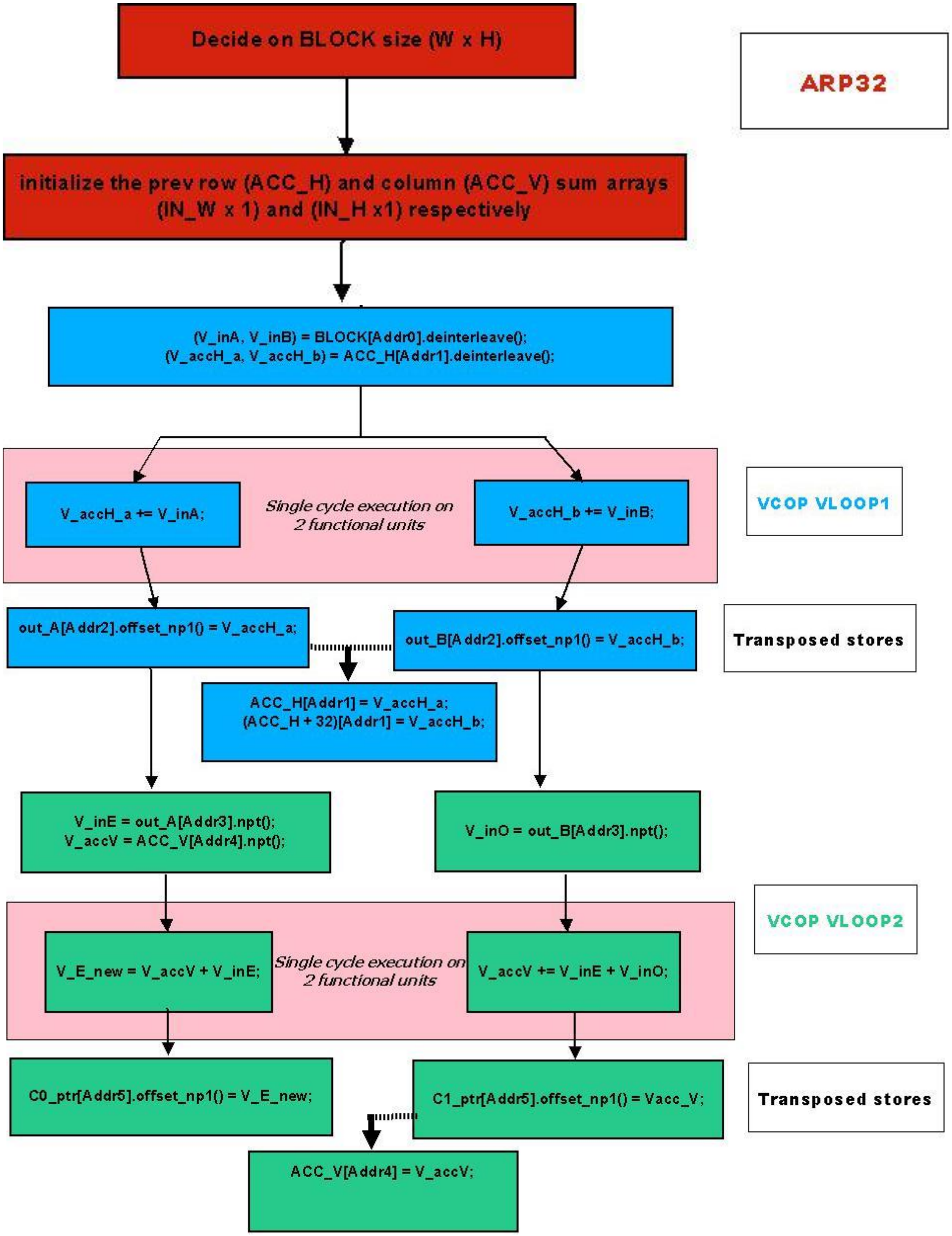
Description

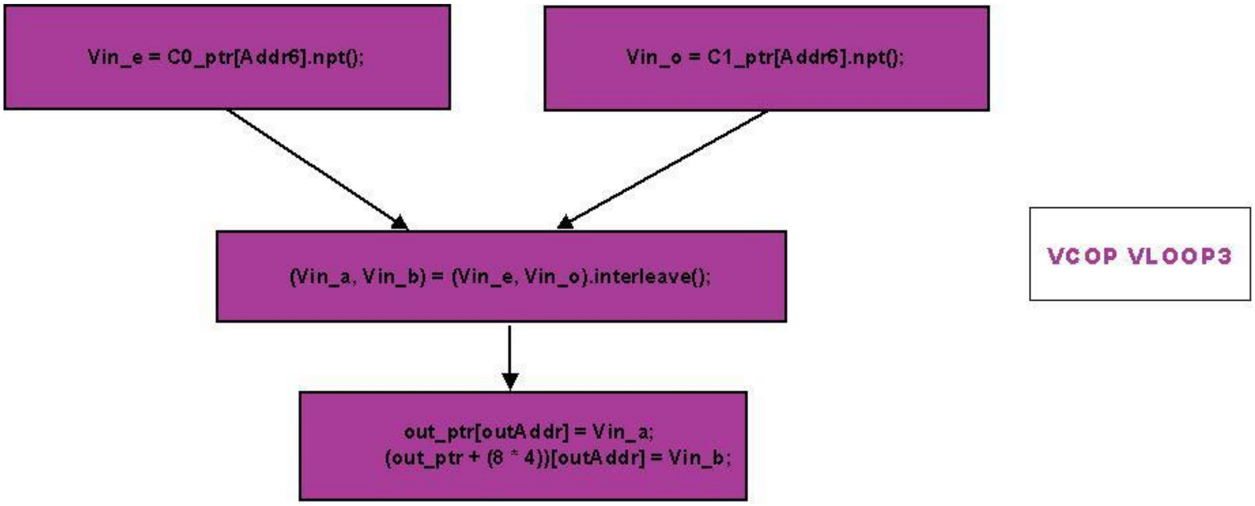
Perform integral imaging (II) function for the entire image. This code assumes a working buffer size of **(blk_w X blk_h)** for byte type input image.

The image is summed incrementally in the vertical direction in a SIMD fashion and the output written in a transposed format. The process is repeated a second time, again adding the rows incrementally in the vertical direction in a SIMD fashion and output of this summation is once again written out in a transposed format. This is the final result of the integral image.

However, the output needs to be reformatted as the EVE transpose output is written such that data is offset by one word while written vertically using the instruction VSTx_OFFST_NP1. This is done so that each word is written to a different bank of memory avoid costly memory contentions. Thus, the reformatting involves realigning the rows to fall at the beginning of every 8 words.

This function obtains the integral on (blk_w x blk_h) blocks of an image, so the end user has to make sure to provide memory locations for the horizontal and vertical accumulators, so as to ensure correct execution of the code for entire image. ACC_H has to have a size of **IN_W** elements of 32-bit words, ACC_V has to have a size of **IN_H** elements of 32-bit words. A flow graph indicating various loops and execution steps for this implementation is given below.


Decide on BLOCK size (W x H)
ARP32
initialize the prev row (ACC_H) and column (ACC_V) sum arrays (IN_W x 1) and (IN_H x1) respectively
(V_inA, V_inB) = BLOCK[Addr0].deinterleave();
(V_accH_a, V_accH_b) = ACC_H[Addr1].deinterleave();
V_accH_a += V_inA;
Single cycle execution on 2 functional units
V_accH_b += V_inB;
VCOP VLOOP1
out_A[Addr2].offset_np1() = V_accH_a;
out_B[Addr2].offset_np1() = V_accH_b;
Transposed stores
ACC_H[Addr1] = V_accH_a;
(ACC_H + 32)[Addr1] = V_accH_b;
V_inE = out_A[Addr3].npt();
V_accV = ACC_V[Addr4].npt();
V_inO = out_B[Addr3].npt();
VCOP VLOOP2
V_E_new = V_accV + V_inE;
Single cycle execution on 2 functional units
V_accV += V_inE + V_inO;
C0_ptr[Addr5].offset_np1() = V_E_new;
C1_ptr[Addr5].offset_np1() = Vacc_V;
Transposed stores
ACC_V[Addr4] = V_accV;



```
Vin_e = C0_ptr[Addr6].npt();
Vin_o = C1_ptr[Addr6].npt();
(Vin_a, Vin_b) = (Vin_e, Vin_o).interleave();
out_ptr[outAddr] = Vin_a;
(out_ptr + (8 * 4))[outAddr] = Vin_b;
```

Figure 4 Figure giving steps for computing Integral Image for an entire image

Usage

This routine computes the integral image of a blk_w x blk_h which is part of a larger image. It writes out the horizontal and vertical accumulated values to 'acc_h_ptr' and 'acc_v_ptr' respectively, to be used as starting accumulated values for the other remaining (blk_w x blk_h) blocks of the image. B0_ptr and B1_ptr are intermediate arrays, used to store results of first VLOOP. C0_ptr and C1_ptr hold the final integral results, but need to be interleaved before being written out as the 'final' II result.

The prototype for this function is given below:

```
void eve_integral_image_char_int_int
(
    unsigned int    blk_w,      // II core kernel height
    unsigned int    blk_h,      // II core kernel width
    unsigned int    in_width,   // Width (or pitch) of the input image.
    __vptr_uint32   acc_h_ptr,  // Pointer to an horz accumulator 1-D array of type int.
    __vptr_uint32   acc_v_ptr,  // Pointer to an vert accumulator 1-D array of type int.
    __vptr_uint8    in_ptr,     // Pointer to an input array of type char.
    __vptr_uint32   B0_ptr,     // Pointer to an even intermediate array of type int.
    __vptr_uint32   B1_ptr,     // Pointer to an odd intermediate array of type int.
    __vptr_uint32   C0_ptr,     // Pointer to an even output array of type int.
    __vptr_uint32   C1_ptr,      // Pointer to an odd output array of type int.
    __vptr_uint32   out_ptr      // Pointer to interleaved output array of type int.
);
```

Constraints

1. This code works on block sizes that are multiples of 16 for 'width' and multiples of 8 for 'height'.
2. ACC_H has to be an array of size IN_W, ACC_V has to be a size of IN_H.
3. Intermediate and final results are always written out as 32-bit values, for optimal implementation sake. However, input can be of either 'char' or 'short' type.

Performance Considerations

Estimated cycle count:

$$\frac{5\times(input\_width*input\_height)}{16}$$

EVE Features used

- Store as transpose (OFFSET_NP1)
- Deinterleaved loads
- 2 functional units

5. Average2x2

Description

The average2x2 code computes the average over each 2x2 window of a block of size BLK_W x BLK_H. This block can be located at any offset within an image. The starting address of the block is passed the kernel from the calling function. Compute width is equal to 8. This value is the minimum SIMD width for gives the kernel useful information on loop iterations to compute the average2x2.

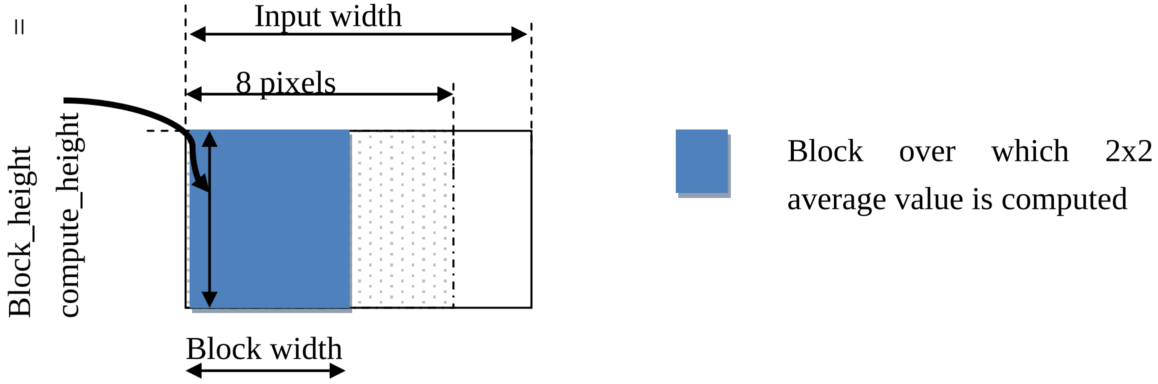


Figure 5 Pictorial description of the block selection for average2x2 for EVE IMGSIGLIB

Shift_val must be selected using the following equation:

shift = $\log_2$(block_width * block_height)

The prototype for the function is:

```
void vcop_avg2x2_uchar
(
   __vptr_uint8  in,                  //  Pointer to an input array of "type_input".
   __vptr_int16 result,               //  Pointer to output array of "type_output".
   int          w_input,              //  Width of the input image in pixels.
```

```
    int         w_in,               //  Width of the input block in pixels.
    int         w_blk,              //  The input width over which average2x2 values are found.
    int         h_blk,              //  The input height for average2x2 filter.
    int         shift_val,          //  (blk_w * blk_h) amount to shift or divide for averaging
    int         type_input,           //  0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
    int         type_output           //  0 - BYTE,  1 - SHORT
);

void vcop_avg2x2_char
(
    __vptr_int8  in,                  //  Pointer to an input array of "type_input".
    __vptr_int16 result,               //  Pointer to output array of "type_output".
    int         w_input,             //  Width of the input image in pixels.
    int         w_in,               //  Width of the input block in pixels.
    int         w_blk,              //  The input width over which average2x2 values are found.
    int         h_blk,              //  The input height for average2x2 filter.
    int         shift_val,          //  (blk_w * blk_h) amount to shift or divide for averaging
    int         type_input,           //  0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
    int         type_output           //  0 - BYTE,  1 - SHORT
);


void vcop_avg2x2_ushort
(
    __vptr_uint16  in,                //  Pointer to an input array of "type_input".
    __vptr_int16 result,               //  Pointer to output array of "type_output".
    int         w_input,             //  Width of the input image in pixels.
    int         w_in,               //  Width of the input block in pixels.
    int         w_blk,              //  The input width over which average2x2 values are found.
    int         h_blk,              //  The input height for average2x2 filter.
    int         shift_val,          //  (blk_w * blk_h) amount to shift or divide for averaging
    int         type_input,           //  0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
    int         type_output           //  0 - BYTE,  1 - SHORT
);
```

```
void vcop_avg2x2_short
(
   __vptr_int16  in,                  //  Pointer to an input array of "type_input".
   __vptr_int16 result,               //  Pointer to output array of "type_output".
   int        w_input,              //  Width of the input image in pixels.
   int        w_in,                 //  Width of the input block in pixels.
   int        w_blk,                //  The input width over which average2x2 values are found.
   int        h_blk,                //  The input height for average2x2 filter.
   int        shift_val,            //  (blk_w * blk_h) amount to shift or divide for averaging
   int        type_input,           //  0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
   int        type_output           //  0 - BYTE,  1 - SHORT
);
```

Usage

Input vectors can have unsigned char, signed char, unsigned short and signed short formats. As the end result is a summation over the height and width of the image, the final output is a scalar value and is the same as the input type. This is because the result is shifted right (if block_width * block_height is a power of 2) or divided (if block_width * block_height is not a power of 2) as part of the 'averaging' step.

The code is executed in two core loops. The first loop loops over the rows and accumulates in a SIMD fashion going over the entire block_height. The result of this accumulation is stored as a column of partial sums using the VST_OFFST_NP1 instruction. The next core loop, loops over this column and does further accumulation, resulting in the final sum.

Constraints

Some of the things to take into consideration when calling this function are:

1. Input can be byte or half word type, signed or unsigned. Output is a setup to be of half word type for this library version.
2. BLK_W <= IN_W, BLK_H <= IN_H.
3. BLK_W must be a multiple of 16.

Performance Considerations

Estimated cycle count:

3/16 cycles/pix

6. Binary Log (bin_log)

Description

Calculate the approximate binary logarithm of an input. On EVE, this is a built in function that can be called in kernel-C using the instruction ‘binlog’.

Usage

The description of binlog is as follows:

```
Vdst = binlog(Vsrc);                          // VBINLOG Vsrc, Vdst
```

Performance Considerations

The instruction helps compute 8 binary log approximations in 2 cycles. Employing both the functional units at time, we can obtain 2/16 (0.125) cyc/pix for pure binary log computations in a single VLOOP.

7. DCT 8x8 Odd-Even (Column)

Description

This function performs the 1-D column DCT on 8x8 data blocks using the odd-even decomposition technique on a 2-D input image. The DCT coefficients are defined **implicitly** and the DCT coefficient pointer points to a dummy array (to maintain compatibility with the earlier VICP prototype for this function). Note that the 1-D column DCT on the 8x8 data blocks is performed with the first point on the 8x8 data block corresponding to the starting address of the image. The input_width and input_height parameters are only used as a guideline to prevent writing the computed DCT into the original image area. The data array is stored row-by-row, input_width data points per row. Resultant output coefficients are stored in **transposed** form assuming further processing by application of the vcop_dct8x8row_int_int row-wise 1-D DCT function.

For ‘short’ type input and ‘short’ type output, the function prototype is:

```
void vcop_dct8x8_OddEven_col_int_int
(
    __vptr_int16   in,              // Pointer to an input array of "type_input".
    __vptr_int16   f_ignore,        // This pointer is ignored, in this implementation
                                   // The DCT coeffs are implicitly defined within the kernel.
    __vptr_int16   out,             // Pointer to output array of "type_output".
    int            w_input,         // Width of the input image in pixels.
    int            h_input,         // Height of the input image in pixels.
    int            w_out,           // Width of the output in pixels.
    int            h_out,           // Height of the output coefficents in pixels.
    int            HBLKS,           // The number of 8x8 blocks of input in horz direction.
    int            VBLKS,           // The number of 8x8 blocks of input in vert direction.
    int            type_input,      //  0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
    int            typecoef_ignore,// This is also ignored, coefficients are always 32-bit int
                                    // and implicitly defined.
    int            type_output,     // 0 - BYTE,  1 - SHORT
    int                rnd_shift          // round and shift amount may be specified.
);
```

Constraints

1. input_width should be greater than or equal to 8 * HBLKS
2. input_height should be greater than or equal to 8 * VBLKS
3. Image data should be a 2-D block

Performance Considerations

Estimated cycle count:

16/64 cyc/pix

EVE features used

- ADDSUB instruction
- 2 functional units help in exploiting symmetry of kernel
- Truncate with arithmetic expression
- Stores with rounding and saturation

8. DCT 8x8 Chen's (Column)

Description

This function performs the 1-D column DCT on 8x8 data blocks using Chen's decomposition technique on a 2-D input image. The DCT coefficients are defined **implicitly** and the DCT coefficient pointer points to a dummy array (to maintain compatibility with the earlier VICP prototype for this function). Note that the 1-D column DCT on the 8x8 data blocks is performed with the first point on the 8x8 data block corresponding to the starting address of the image. The input_width and input_height parameters are only used as a guideline to prevent writing the computed DCT into the original image area. The data array is stored row-by-row, input_width data points per row. Resultant output coefficients are stored in **transposed** form assuming further processing by application of the vcop_dct8x8row_int_int row-wise 1-D DCT function.

For 'short' type input and 'short' type output, the function prototype is:

```
void vcop_dct8x8col_int_int
(
   __vptr_int16   in,             // Pointer to an input array of "type_input".
   __vptr_int16   f_ignore,       // This pointer is ignored, in this implementation
                                  // The DCT coeffs are implicitly defined within the kernel.
   __vptr_int16   out,            // Pointer to output array of "type_output".
   int            w_input,        // Width of the input image in pixels.
   int            h_input,        // Height of the input image in pixels.
   int            w_out,          // Width of the output in pixels.
   int            h_out,          // Height of the output coefficents in pixels.
   int            HBLKS,          // The number of 8x8 blocks of input in horz direction.
   int            VBLKS,          // The number of 8x8 blocks of input in vert direction.
   int            type_input,     //  0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
   int            typecoef_ignore, // This is also ignored, coefficients are always 32-bit int
                                  // and implicitly defined.
   int            type_output,    // 0 - BYTE,  1 - SHORT
```

```
    int                 rnd_shift             // round and shift amount may be specified.
);
```

Constraints

1. input_width should be greater than or equal to 8 * HBLKS
2. input_height should be greater than or equal to 8 * VBLKS
3. Image data should be a 2-D block

Performance Considerations

Estimated cycle count:

13/64 cyc/pix

EVE features used

- ADDSUB instruction
- 2 functional units help in exploiting symmetry of kernel
- Truncate with arithmetic expression
- Stores with rounding and saturation

9. DCT 8x8 Odd-Even (Row)

Description

This function performs the 1-D row-wise DCT on 8x8 data blocks using the odd-even decomposition technique on a 2-D input image. The DCT coefficients are defined **implicitly** and the DCT coefficient pointer points to a dummy array (to maintain compatibility with the earlier VICP prototype for this function). Note that the 1-D row-wise DCT on the 8x8 data blocks is performed with the first point on the 8x8 data block corresponding to the starting address of the image. The input_width and input_height parameters are only used as a guideline to prevent writing the computed DCT into the original image area. Also note that the input data array is stored row-by-row, input_width data points per row. Resultant output coefficients are stored in **transposed** form.

For ‘short’ type input and ‘short’ type output, the function prototype is:

```
void vcop_dct8x8_OddEven_row_int_int
(
   __vptr_int16   in,              // Pointer to an input array of "type_input".
   __vptr_int16   f_ignore,        // This pointer is ignored, in this implementation
                                   // The DCT coeffs are implicitly defined within the kernel.
   __vptr_int16   out,             // Pointer to output array of "type_output".
   int            w_input,         // Width of the input image in pixels.
   int            h_input,         // Height of the input image in pixels.
   int            w_out,           // Width of the output in pixels.
   int            h_out,           // Height of the output coefficents in pixels.
   int            HBLKS,           // The number of 8x8 blocks of input in horz direction.
   int            VBLKS,           // The number of 8x8 blocks of input in vert direction.
   int            type_input,      //  0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
   int            typecoef_ignore,// This is also ignored, coefficients are always 32-bit int
                                   // and implicitly defined.
   int            type_output,     // 0 - BYTE,  1 - SHORT
   int            rnd_shift        // round and shift amount may be specified.
```

);

Constraints

1. input_width should be greater than or equal to 8 * HBLKS
2. input_height should be greater than or equal to 8 * VBLKS
3. Image data should be a 2-D block

Performance Considerations

Estimated cycle count:

16/64 cyc/pix

EVE features used

- ADDSUB instruction
- 2 functional units help in exploiting symmetry of kernel
- Truncate with arithmetic expression
- Stores with rounding and saturation

10. DCT 8x8 Chen's (Row)

Description

This function performs the 1-D row-wise DCT on 8x8 data blocks using Chen's decomposition technique on a 2-D input image. The DCT coefficients are defined **implicitly** and the DCT coefficient pointer points to a dummy array (to maintain compatibility with the earlier VICP prototype for this function). Note that the 1-D row DCT on the 8x8 data blocks is performed with the first point on the 8x8 data block corresponding to the starting address of the image. The input_width and input_height parameters are only used as a guideline to prevent writing the computed DCT into the original image area. The data array is stored row-by-row, input_width data points per row. Resultant output coefficients are stored in **transposed** form.

For 'short' type input and 'short' type output, the function prototype is:

```
void vcop_dct8x8row_int_int
(
   __vptr_int16   in,               // Pointer to an input array of "type_input".
   __vptr_int16   f_ignore,         // This pointer is ignored, in this implementation
                                    // The DCT coeffs are implicitly defined within the kernel.
   __vptr_int16   out,              // Pointer to output array of "type_output".
   int            w_input,          // Width of the input image in pixels.
   int            h_input,          // Height of the input image in pixels.
   int            w_out,            // Width of the output in pixels.
   int            h_out,            // Height of the output coefficents in pixels.
   int            HBLKS,            // The number of 8x8 blocks of input in horz direction.
   int            VBLKS,            // The number of 8x8 blocks of input in vert direction.
   int            type_input,       //  0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
   int            typecoef_ignore,  // This is also ignored, coefficients are always 32-bit int
                                    // and implicitly defined.
   int            type_output,      // 0 - BYTE,  1 - SHORT
   int            rnd_shift         // round and shift amount may be specified.
);
```

Constraints

1. input_width should be greater than or equal to 8 * HBLKS
2. input_height should be greater than or equal to 8 * VBLKS
3. Image data should be a 2-D block

Performance Considerations

Estimated cycle count:

13/64 cyc/pix

EVE features used

- ADDSUB instruction
- 2 functional units help in exploiting symmetry of kernel
- Truncate with arithmetic expression
- Stores with rounding and saturation

11. Median_filter_row

Description

This function performs a median filtering operation along the rows of an input matrix. The size of the median filter can be 3-tap or 5-tap. At each output location, the median of the values in a window of size three or five (depending on filter size) is written to the output. A block of compute_width X compute_height is computed using the median filter and written to the output buffer. The output buffer can be larger the compute window.

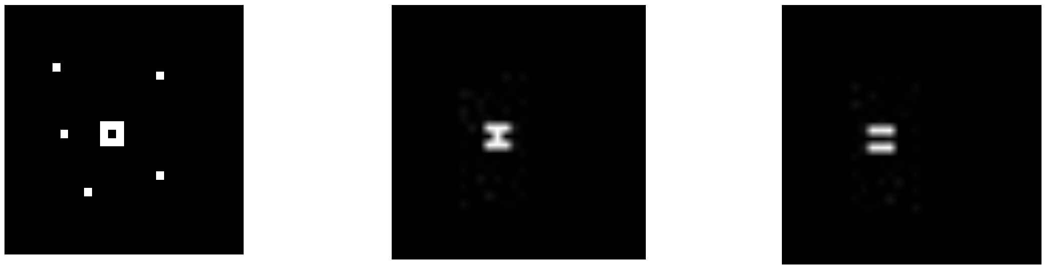
The prototypes for the 3-tap and 5-tap filters with input type of char are as follows:

```
void vcop_median_3tap_filt_row_short
(
    __vptr_int16 input_ptr,     // starting address of input
    __vptr_int16 output_ptr,  //  starting address of output
    Int16 input_width,          // height of input array
    Int16 input_height,         // width of input array
    Int16 output_width,         // height of output array
    Int16 output_height,        // width of output array
    Int16 compute_width,        // height of compute block
    Int16 compute_height,       // width of compute block
    Int16 median_size,          // 3 or 5-tap median filter
    Int16 input_type,           // 0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
    Int16 output_type           // 0 - BYTE,  1 - SHORT
);

void vcop_median_5tap_filt_row_short
(
    __vptr_int16 input_ptr,       // starting address of input
    __vptr_int16 output_ptr,      // starting address of output
    Int16 input_width,            // height of input array
    Int16 input_height,          // width of input array
    Int16 output_width,          // height of output array
    Int16 output_height,         // width of output array
    Int16 compute_width,         // height of compute block
    Int16 compute_height,        // width of compute block
    Int16 median_size,           // 3 or 5-tap median filter
    Int16 input_type,            // 0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
    Int16 output_type            // 0 - BYTE,  1 - SHORT
);
```

Constraints

1. As the data is loaded with the ‘deinterleave’ feature, 16 pixels are loaded at a time. Thus, the input_width must be >= 16
2. output_width >= compute_width.
3. input_width >= compute_width + 2 (3-tap filter), input_width >= compute_width + 4 (5-tap filter)
4. input_height, output_height >= compute_height.



Input image

3-tap filter result

5-tap filter result

Figure 6 Figure showing median row-wise filter results on an image with 3 and 5 tap filtering. Results obtained on EVE

Performance Considerations

Estimated cycle count:
3-tap filter -- 3/16 cyc/pix
5-tap filter – 9/16 cyc/pix

EVE features used

The MINMAX feature is very useful for median computations.

12. Median_filter_col

Description

This function performs a median filtering operation along the columns of an input matrix. The size of the median filter can be 3-tap or 5-tap. At each output location, the median of the values in a window of size three or five (depending on filter size) is written to the output. A block of compute_width X compute_height is computed using the median filter and written to the output buffer. The output buffer can be larger the compute window.

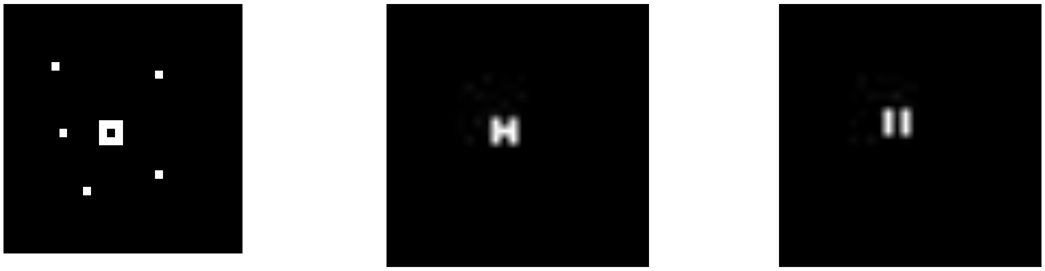
The prototypes for the 3-tap and 5-tap filters with input type of char are as follows:

```
void vcop_median_3tap_filt_col_short
(
    __vptr_int16 input_ptr,     // starting address of input
    __vptr_int16 output_ptr,  //  starting address of output
    Int16 input_width,          // height of input array
    Int16 input_height,         // width of input array
    Int16 output_width,         // height of output array
    Int16 output_height,        // width of output array
    Int16 compute_width,        // height of compute block
    Int16 compute_height,       // width of compute block
    Int16 median_size,          // 3 or 5-tap median filter
    Int16 input_type,           // 0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
    Int16 output_type           // 0 - BYTE,  1 - SHORT
);
```

```
void vcop_median_5tap_filt_col_short
(
    __vptr_int16 input_ptr,       // starting address of input
    __vptr_int16 output_ptr,      // starting address of output
    Int16 input_width,            // height of input array
    Int16 input_height,           // width of input array
    Int16 output_width,          // height of output array
    Int16 output_height,         // width of output array
    Int16 compute_width,         // height of compute block
    Int16 compute_height,        // width of compute block
    Int16 median_size,           // 3 or 5-tap median filter
    Int16 input_type,            // 0 - UBYTE, 1 - BYTE, 2 - USHORT, 3 - SHORT
    Int16 output_type            // 0 - BYTE,  1 - SHORT
);
```

Constraints

1. As the data is loaded with the ‘deinterleave’ feature, 16 pixels are loaded at a time. Thus, the input_width must be >= 16
2. output_width >= compute_width.
3. input_width >= compute_width + 2 (3-tap filter), input_width >= compute_width + 4 (5-tap filter)
4. input_height, output_height >= compute_height.



Input image

3-tap filter result

5-tap filter result

Figure 7 Figure showing median column-wise filter results on an image with 3 and 5 tap filtering. Results obtained on EVE

Performance Considerations

Estimated cycle count:
3-tap filter -- 3/16 cyc/pix
5-tap filter – 9/16 cyc/pix

EVE features used

The MINMAX feature is very useful for median computations.

13. FIR Filter

Description

This function performs 2D FIR filtering operations on a block of data.

The result is in an output block of dimension [w_compute * h_compute]. Horizontal and vertical downsampling factors are supported. The kernel size is [w_coeff * h_coeff].

The prototype for this filter is:

```
void vcop_filter_uchar_char_char
(
   __vptr_uint8   in,              //  Pointer to an input array of "type_input".
   __vptr_int8    coef,            //  Pointer to cofficient array of type 'type_coef'.
   __vptr_int8    res,             //  Pointer to output array of "type_output".
   int            w_input,         //  Width of the input image in pixels.
   int            w_coeff,         //  Width of the coefficients in pixels.
   int            h_coeff,         //   Height of the coefficients in pixels.
   int            w_compute,       //  Compute width in pixels.
   int            h_compute,       //   Compute height in pixels.
   int            dnsmpl_vert,     //   Vertical downsampling value.
   int            dnsmpl_horz,     //   Horizontal downsampling value.
   int            rnd_shift        //   Rounding and shifting amount.
);
```

Constraints

1. This code assumes that 'w_compute' >= 'w_coeff' and 'h_compute' >= 'h_coeff'.
2. This code assumes that 'w_compute' <= 'w_input' and 'h_compute' <= 'h_input'.
3. This code assumes that 'w_compute' <= 'w_out' and 'h_compute' <= 'h_out'.
4. VERTICAL downsampling is applied in this loop, but HORIZONTAL downsampling is carried out on the ARP32.

Performance Considerations

(w_coeff * h_coeff) / VCOP_SIMD_WIDTH

14. Horizontal Upsampling

Description

This function up samples a given input array depending on an up-sampling value specified in the horizontal direction only.

Usage

The prototype for this function is as below:

```
void vcop_HorzUpsample_ushort_short_ushort
(
   __vptr_uint16  in,          // Pointer to an input array of "type_input".
   __vptr_int16   f_coef,      // Pointer to cofficient array of type 'type_coef'.
   __vptr_uint16  out,         // Pointer to output array of "type_output".
   int            w_input,     // Width of the input image in pixels.
   int            ntaps,       // No of coefficients per stage of polyphase.
   int            w_compute,   // Compute width in pixels.
   int            h_compute,   // Compute height in pixels.
   int            w_out,       // Width of the output image in pixels.
   int            U,           // Upsampling value.
   int            type_output, // Don't cares for this function
   int            rnd_shift    // Rounding and shifting amount.
);
```

Performance

NUM_TAPS/UP_SAMPLE cyc/pix

Where NUM_TAPS is given by the number of taps per phase or total_taps/UP_SAMPLE

15. Matrix Multiplication

Description

This kernel carries out multiplication between two 2-D matrices of dimensions (w_in1 * h_in1) and (w_in2 * w_in1) respectively. Result is a dot product between the row of one matrix and the column of the second.

The prototype of the function is as follows:

```
void vcop_mat_mul_uchar_uchar_short
(
   __vptr_uint8   in1_ary,      // Pointer to first input array.
   __vptr_uint8   in2_ary,      // Pointer to second input array.
   __vptr_int16   res,          // Pointer to output array.
   int           w_in1,         // Width of the first input array in pixels.
   int           h_in1,         // Height of the first input array in pixels.
   int           w_in2,         // Width of the second input array in pixels.
   int           rnd_shift      // Rounding and shifting amount.
);
```

Constraints

1. This code assumes that width of first matrix is the same as the height of the second.

Performance Considerations

w_in1 / VCOP_SIMD_WIDTH   cycles/pix

16. Vertical Fractional Resampling using Polyphase Filtering

Description

This kernel helps us carry out fractional resampling using Polyphase filtering on an input image in the vertical direction. The resampling is achieved by using up and down sampling values provided by the user. The core VCOP kernel computes the output values based on a polyphase filtering approach by looping over a sub-set of coefficients for every ‘phase’ of the polyphase filter.

This entire function has four aspects to it. They are described below:

C code for coefficient generation:

This C helper function generates the coefficients depending on the up, down sampling values and the number of total taps for the filter.

The prototype for this function is:

```
short GenResamplingCoeffs
(
   int   U,                                   // Up sampling value
   int   D,                                   // Down sampling value
   int   taps,                               // original number of taps
   short *coeffs,                            // pointer to coefficient array
   unsigned char *sampling_pattern, //Pattern (mask) for predicated stores
   int   *pattern_size                     // Size of the pattern array
);
```

CONSTRAINTS:

1. IN_H >= UP_SAMPLE * (COMP_H/DOWN_SAMPLE -1) + DOWN_SAMPLE + TOTAL_TAPS / UP_SAMPLE
2. COMP_H must be multiple of UP_SAMPLE

A store pattern (mask) generation code to carry out predicated stores on VCOP:

This coefficient generation code also generates the pattern for storing the resampled values. This is needed as we use the collated store feature for VCOP which needs a flag array to indicate whether to store the computed result or skip it. Collated stores also increments the store pointer only if a store actually happened and this feature is also useful for resampling.

Kernel-C code for Polyphase filtering:

The polyphase filter core kernel for VCOP is the implementation which implements only one FIR, keeping both the up and down sampling values. The number of taps for the polyphase is decided upon by the user. The number of phases is

The core itself has two VLOOPs. The first one incorporate the filtering in the vertical direction and writes out resampled output values following a store pattern. This is because collated store has four main characteristics:

a) It uses a masking pattern to either store or not store.
b) It increments the store pointer only if a valid store has occurred.
c) It can store only 8-SIMD values at a time, so only NPT stores can be used with collated stores, not INTERLEVEd stores.
d) It stores in a 1-D fashion, so 8-SIMD outputs are stored at each valid store and the next valid 8-SIMD stores are written at the consecutive memory locations. This means we have to rearrange the output depending on the OUT_H and COMP_W values. This rearrangement is carried out by the second VLOOP and is necessary for correctly visualizing the output.

The prototype for this function is as follows:

```
void vcop_FilterPoly_ushort_short_ushort
(
   __vptr_uint16  in,             // Pointer to an input array of "type_input".
   __vptr_int16   f_coef,         // Pointer to cofficient array of type 'type_coef'.
   __vptr_uint8   smpl_flag,      // Pointer to sampling flag array of type char.
   __vptr_uint16  inter_out,      // Pointer to output array of "type_output".
   __vptr_uint8   out,            // Pointer to reordered output array of "type_outputfinal".
```

```
    int       w_input,         // Width of the input image in pixels.
    int       ntaps,           // No of coefficients per stage of polyphase.
    int       w_compute,       // Compute width in pixels.
    int       h_compute,       // Compute height in pixels.
    int       w_out,           // Width of the output image in pixels.
    int       h_out,           // Height of the output image in pixels.
    int       D,               // Downsampling value.
    int       U,               // Upsampling value.
    int       phases,          // Loop limit giving number of phases for polyphase
    int       rnd_shift        // Rounding and shifting amount.
);
```

CONSTRAINTS

1. COMP_W must be a multiple of 8.

Compiler Aided Memory Allocation (CAMA) description:

This is a feature provided for EVE where we can allocate memory at run time.

void *vcop_malloc(VCOP_MEMHEAP heap, int size)

- Allocate 'size' bytes from a VCOP heap, using heap id values:
VCOP_IBUFLA, VCOP_IBUFHA, VCOP_IBUFLB, VCOP_IBUFHB, VCOP_WMEM

void vcop_free(void *userptr)

- Free memory indicated by pointer. Allocator will automatically determine the corresponding heap

void vcop_setview(VCOP_MEMVIEW view)

- Set allocator according to VCOP_MEMFLAT or VCOP_MEMALIASED view.
Under aliased view, allocator will also manage memory for aliased heaps when memory is allocated from IBUFLA or IBUFHA.
views: VCOP_MEMFLAT (default) or VCOP_MEMALIASED

Performance

((UP_SAMPLE/DOWN_SAMPLE + 1) * ntaps)/8 + (1/8) cyc/pix

EVE features used

1. This function uses the COLLATE store feature of EVE.

17. Sum of Absolute Differences (SAD)

Description

The sum of absolute differences kernel computes the SAD values between an input block of size (blk_w * blk_h) and sub-blocks within a larger reference block of size (ref_w * ref_h). The sub-blocks within the reference block are located at offsets of offset_horz in the horizontal direction and offset_vert in the vertical direction. The input block itself is of size (in_w * in_h) from which sub-blocks of (blk_w * blk_h) are generated.

A total of (steps_horz * steps_vert) SADs are generated for each input sub-block. The minimum of these SADs is the true match for the input sub-block within the reference block.

```
void vcop_SAD_char_int
(
    __vptr_int8   in,                  // Pointer to input array.
    __vptr_int8   ref,                 // Pointer to reference data array.
    __vptr_int16  vert_sad,            // Pointer to SAD in vertical direction.
    __vptr_int16  vert_sad_t,          // Pointer to intermediate SADs transposed.
    __vptr_int16  sad_array,           // Pointer to SADs between one input block and ref blks.
    int           blk_w,               // Block width over which matching is done.
    int           blk_h,               // Block height over which matching is done.
    int           in_w,                // Width of the input region being matched.
    int           in_h,                // Height of the input region being matched.
    int           ref_w,               // Width of the reference region being matched.
    int           ref_h,               // Height of the reference region being matched.
    int           offset_horz,         // Horizontal offset for pixels to be skipped before next match.
    int           offset_vert,         // Horizontal offset for pixels to be skipped before next match.
    int           steps_horz,          // Total block matches in horizontal direction.
    int           steps_vert           // Total block matches in vertical direction.
);
```

Constraints

1. blk_w has to be a multiple of 8
2. in_w >= blk_w
3. steps_horz should be a multiple of 16

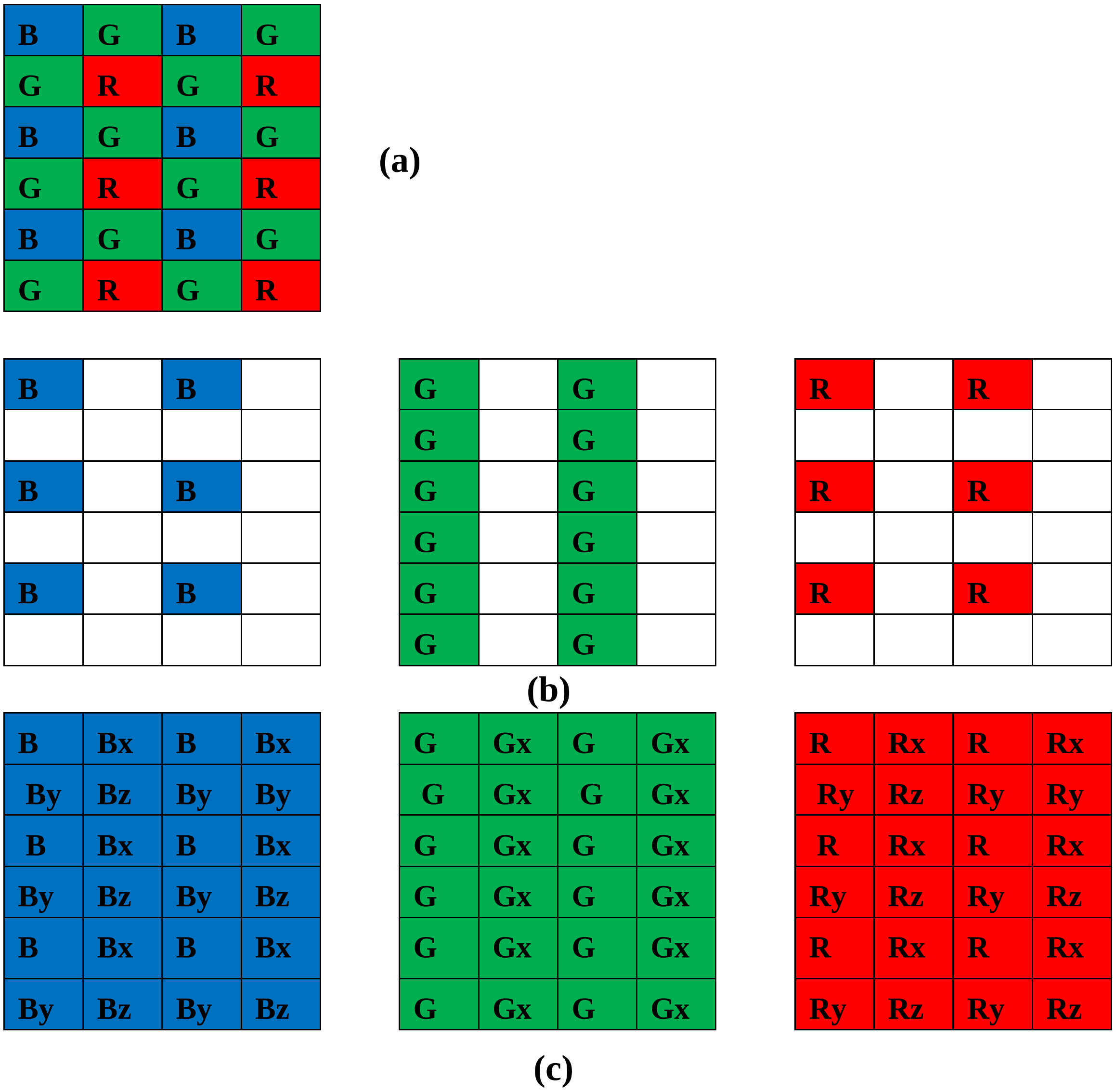
Performance Considerations

(steps_vert * steps_horz) * (blk_w * blk_h)/8
+ (steps_vert * steps_horz * blk_w) / VCOP_SIMD_WIDTH
+ (steps_vert * steps_horz * blk_w)/(TRANSPOSE_WIDTH * 2)

18. Bayer CFA Interpolation using Averaging

Description

This function performs an interpolation of a CFA image into a full-resolution RGB-image. The CFA-pattern is assumed to be a Bayer pattern. The output is composed of three planes, R, G, B of dimensions OUT_W X OUT_H elements.



B	G	B	G
G	R	G	R
B	G	B	G
G	R	G	R
B	G	B	G
G	R	G	R

(a)

B		B	
B		B	
B		B	

G		G	
G		G	
G		G	
G		G	
G		G	
G		G	

R		R	
R		R	
R		R	

(b)

B	Bx	B	Bx
By	Bz	By	By
B	Bx	B	Bx
By	Bz	By	Bz
B	Bx	B	Bx
By	Bz	By	Bz

G	Gx	G	Gx
G	Gx	G	Gx
G	Gx	G	Gx
G	Gx	G	Gx
G	Gx	G	Gx
G	Gx	G	Gx

R	Rx	R	Rx
Ry	Rz	Ry	Ry
R	Rx	R	Rx
Ry	Rz	Ry	Rz
R	Rx	R	Rx
Ry	Rz	Ry	Rz

(c)

Figure 8 Figure (a) gives the Bayer CFA pattern, (b) gives the de-interleaved (de-mosaic) planes and (c) gives the interpolation pattern for Interpolation using averaging

As shown in Figure 12, (a) is the input Bayer pattern as captured in the CFA. The first step (b) is to de-interleave this data to create separate R, G and B planes. The next step (c) is to compute intermediate pixel values by interpolation to obtain the full resolution image. This can be done in three ways, the first to simply replicate pixels as the same value as the next available one. The second way is to average pixels immediately adjacent to the missing one and the third is to carry out FIR filtering to compute the intermediate pixel values.

Usage

The prototype for this function is as follows:

```
void vcop_BayerCFA_interpolate_char
(
    __vptr_uint8   CFA_char,  // Input array
    unsigned int   in_w,           // Input width indicating the pitch to get to next row of inputs
    unsigned int   blk_w,        // Compute block width
    unsigned int   blk_h,         // Compute block height
    __vptr_uint8   R_char,      // Red plane of blk_w X blk_h resolution
    __vptr_uint8   G_char,      // Green plane of blk_w X blk_h resolution
    __vptr_uint8   B_char       // Blue plane of blk_w X blk_h resolution
);
```

```
void vcop_BayerCFA_interpolate_short
(
    __vptr_uint16  CFA_short, // Input array
    unsigned int     in_w,          // Input width indicating the pitch to get to next row of inputs
    unsigned int     blk_w,        // Compute block width
    unsigned int     blk_h,         // Compute block height
    __vptr_uint16   R_short,     // Red plane of blk_w X blk_h resolution
    __vptr_uint16   G_short,     // Green plane of blk_w X blk_h resolution
    __vptr_uint16   B_short      // Blue plane of blk_w X blk_h resolution
);
```

Constraints

1. **blk_w** must be a multiple of 16.

Performance

BayerCFA_interpolate_char: (2/16 * 1/3) + 13/16 = 0.85 cyc/pix
BayerCFA_interpolate_short: (2/16 * 1/3) + 13/16 = 0.85 cyc/pix

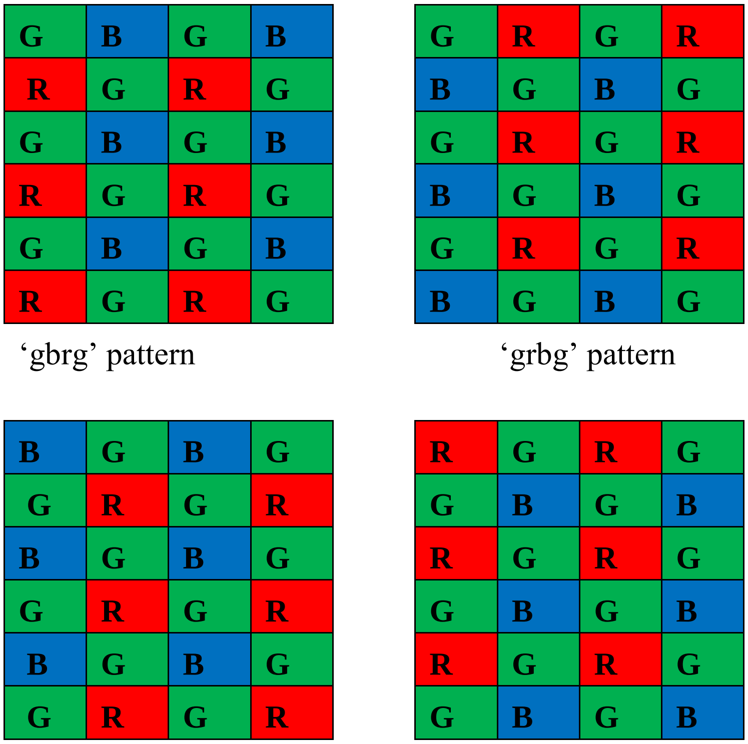
19. Bayer CFA Interpolation using Adams-Hamilton's Demosaicing Algorithm

Description

This function performs an interpolation of a CFA image into a full-resolution 16-bits RGB-image. The CFA-pattern is assumed to be a Bayer pattern. The output is composed of three planes, R, G, B of dimensions OUT_W X OUT_H elements.

The advantage of this function versus the previous one is higher image quality since the function implements Adams-Hamilton demosaicing algorithm. The drawback is slower execution time.

Also the function can handle all the 4 Bayer patterns illustrated below:



G	B	G	B
R	G	R	G
G	B	G	B
R	G	R	G
G	B	G	B
R	G	R	G

'gbrg' pattern

G	R	G	R
B	G	B	G
G	R	G	R
B	G	B	G
G	R	G	R
B	G	B	G

'grbg' pattern

B	G	B	G
G	R	G	R
B	G	B	G
G	R	G	R
B	G	B	G
G	R	G	R

'bggr' pattern

R	G	R	G
G	B	G	B
R	G	R	G
G	B	G	B
R	G	R	G
G	B	G	B

'rggb' pattern

Figure 9 The 4 Bayer patterns

Since the processing is equivalent to applying a 7x7 FIR filter, to generate an output block of blockWidth x blockHeight pixels, the input array must contain (blockWidth + 6) x (blockHeight

+ 6) pixels.

Usage

The prototype for this function is as follows:

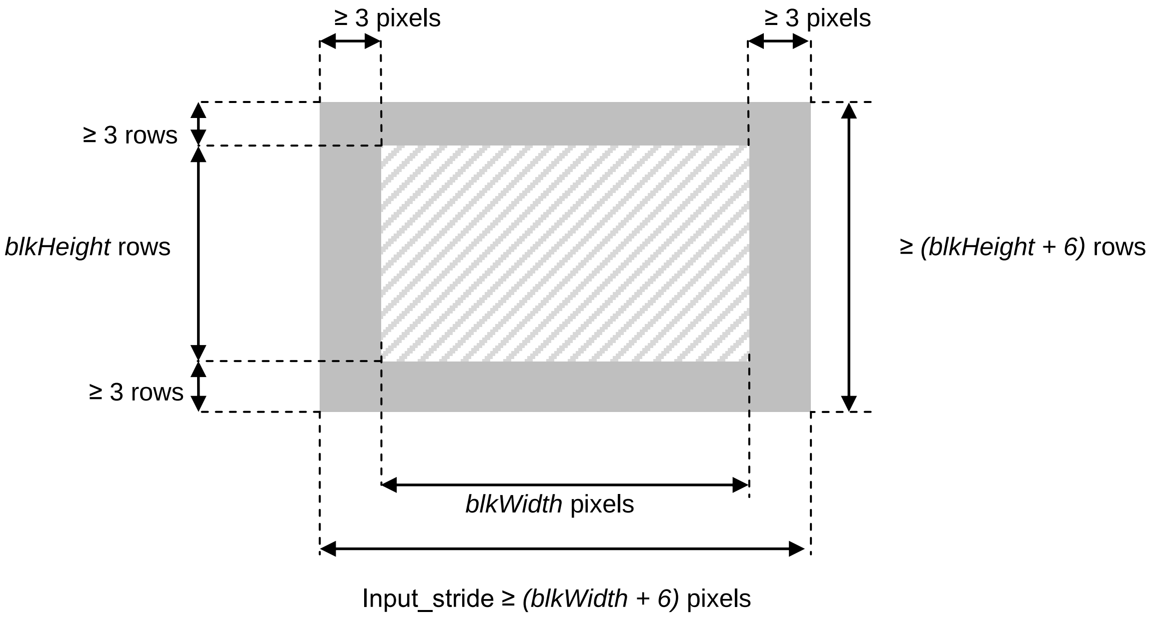
For ‘gbrg’ or ‘grbg’ pattern, use:

```
void vcop_raw2rgb_CGGC_i16u_o16u
(
    __vptr_uint16   pInBuff,  // Input array containing the ‘bggr’ or ‘rggb’ CFA patterns of
                                                // input_stride x (blkHeight + 6) pixels
    __vptr_uint16   pRbuff,   // Output Red plane of outputRB_stride x blkHeight 16-bits pixels
    __vptr_uint16   pGbuff,// Output Green plane of outputG_stride x (blkHeight+2) 16-bits pixels
    __vptr_uint16   pBbuff ,// Output Blue plane of outputRB_stride x blkHeight 16-bits pixels
    __vptr_uint16   pScratch,  // Scratch array of size 2 * outputRB_stride * blkHeight bytes
    unsigned short blkWidth, // Number of horizontal pixels processed in the input array
    unsigned short blkHeight, // Number of row processed in the input array
    unsigned short input_stride, // Stride of the input array in number of pixels
    unsigned short outputG_stride, // Stride of the output green plane in number of pixels
    unsigned short outputRB_stride // Stride of the output red or blue plane in number of pixels
);
```

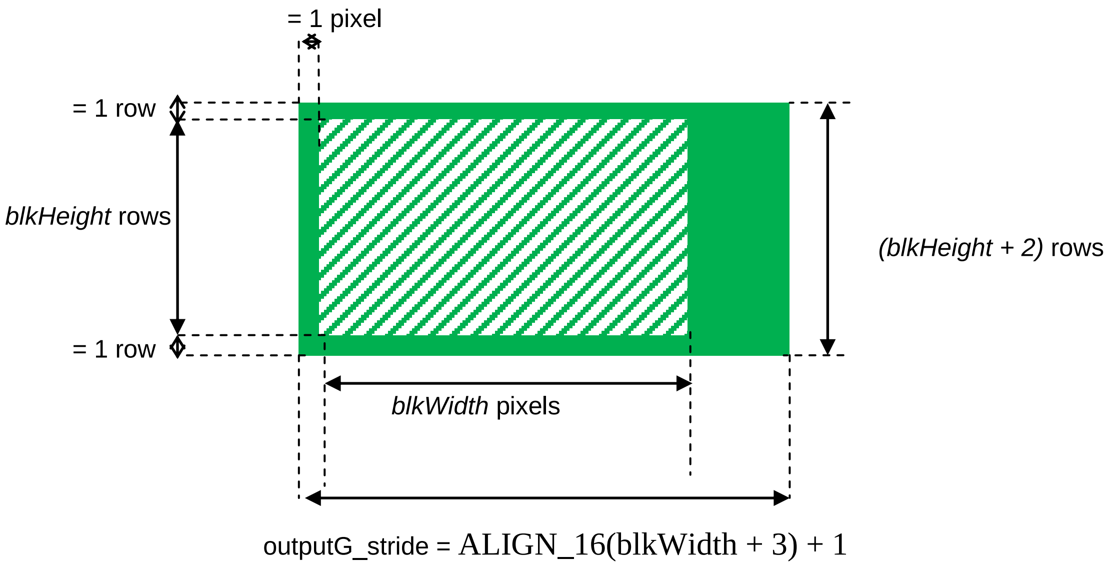
For ‘bggr’ or ‘rggb’ pattern, use:

```
void vcop_raw2rgb_GCCG_i16u_o16u
(
    __vptr_uint16   pInBuff,  // Input array containing the ‘bggr’ or ‘rggb’ CFA patterns of
                                                // input_stride x (blkHeight + 6) pixels
    __vptr_uint16   pRbuff,   // Output Red plane of outputRB_stride x blkHeight 16-bits pixels
    __vptr_uint16   pGbuff,// Output Green plane of outputG_stride x (blkHeight+2) 16-bits pixels
    __vptr_uint16   pBbuff ,// Output Blue plane of outputRB_stride x blkHeight 16-bits pixels
    __vptr_uint16   pScratch,  // Scratch array of size 2 * outputRB_stride * blkHeight bytes
    unsigned short blkWidth, // Number of horizontal pixels processed in the input array
    unsigned short blkHeight, // Number of row processed in the input array
    unsigned short input_stride, // Stride of the input array in number of pixels
    unsigned short outputG_stride, // Stride of the output green plane in number of pixels
    unsigned short outputRB_stride // Stride of the output red or blue plane in number of pixels
);
```

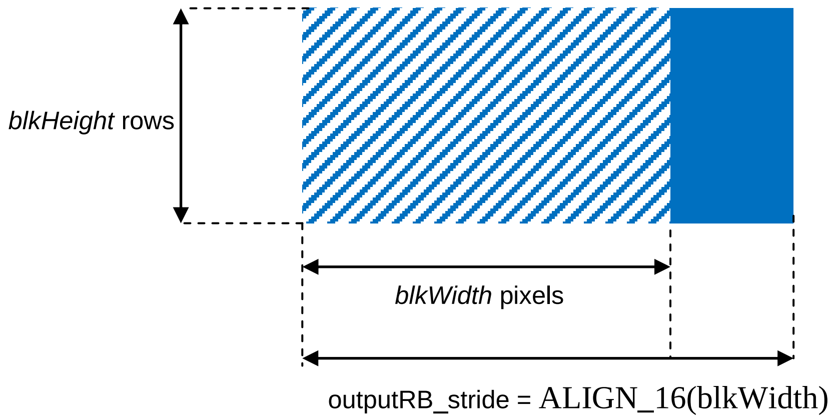
The layout of the input Bayer pattern array is as follow:



The layout of the output Green array is as follow:



The layout of the output Blue or Red array is as follow:



Observe that in the case of the green array, the region of interest of blkWidth x blkHeight pixels starts at offset (1,1), whereas for red and blue array, it starts at (0,0).

Constraints

1. blkWidth must be a multiple of 16.
2. input_stride >= blkWidth + 6
3. outputG_stride= ALIGN_16(blkWidth + 3) + 1
4. outputRB_stride= ALIGN_16(blkWidth)

Performance

2.8 cyc/pix

20. Bayer CFA Horizontal Upsampling

Description

This function takes a Bayer CFA array and upsamples it in the horizontal direction by the up sampling amount indicated by UP_SAMPLE. The coefficients are generated by a helper function provided, whose prototype is give below:

```
short GenResamplingCoeffs
(
   int   U,                // Up sampling value
   int   D,                // Down sampling value
   int   taps,             // original number of taps
   short *coeffs,
   unsigned char *sampling_pattern,
   int   *pattern_size
);
```

This function generates the coefficients which are then used in the main VCOP kernel file for upsampling in the horizontal direction.

Usage

The VCOP function prototype is as follows:

```
void vcop_BayerCFA_HorzUpsample_ushort_short_ushort
(
   __vptr_uint16  in,          // Pointer to an input array of "type_input".
   __vptr_int16   f_coef,      // Pointer to cofficient array of type 'type_coef'.
   __vptr_uint16  out,         // Pointer to output array of "type_output".
   int          w_input,       // Width of the input image in pixels.
   int          ntaps,         // No of coefficients per stage of polyphase.
   int          w_compute,     // Compute width in pixels.
```

```
    int           h_compute,    // Compute height in pixels.
    int           w_out,        // Width of the output image in pixels.
    int           U,            // Upsampling value.
    int           type_output,  // Don’t care for this implementation
    int           rnd_shift     // Rounding and shifting amount.
) ;
```

Constraints

1. Appropriate padding of the input array is assumed depending on the up-sampling value

Performance

The performance of this function is:

NUM_TAPS/UP_SAMPLE cyc/pix

Where NUM_TAPS is equal to the number of taps for each phase, or total_filter_taps/UP_SAMPLE value

EVE features used

1. This function uses the ‘skip’ feature while storing.

21. Bayer CFA Interpolation using FIR Filtering

Description

This function computes intermediate pixels of the demosaiced R. G and B planes by FIR filtering operation with an interpolation factor L of 2. The FIR filtering approach is similar to the FIR Filter described in section 25. As it is a straightforward implementation of the FIR, it is not implemented again here.

22. Bayer CFA Vertical Upsampling

Description

In this function, vertical up sampling of the input CFA array is carried out in a similar fashion as described in Section 27 titled Vertical Fractional Resampling using Polyphase Filtering. The code can be constructed in a similar fashion as that of Section 27 and so will not be implemented in this version of the IMGSIGLIB.

23. Bayer CFA Add

Description

In this function, Bayer CFA elements are added to obtain certain statistics. This addition is carried out such that for a given BLK_H and BLK_W, all the R, Ga, Gb and B pixels in a CFA tile are summed respectively to give four R, Ga, Gb and B values. A Bayer CFA tile is as follows:

B	Ga
Gb	R

This sort of implementation is for statistics collection and is pretty straightforward to implement on VCOP using deinterleaved loads and the summing operation. Thus, this function is not implemented in this version of IMGSIGLIB for EVE.

24. Array Inner Product

Description

The original VICP document describes this kernel as *an inner product between a 4-D data array and a 4-D coefficient array, resulting in a 2-D output array. Each pair of data inner array and coefficient inner array are multiplied point-by-point, and then these product arrays are summed together on the outer 2 dimensions.*

On EVE, this can be achieved by first doing an **Array_op_distributed** using **multiplication** operator, writing to a temporary array and a second loop that iterates over all 2-D blocks and accumulates them. As this sort of behavior is uncommon, this kernel is not implemented here.

25. Accumulate2D Array-op

Description

In the original VICP documentation, it was mentioned that Accumulate2D_array_op *performs point-by-point operation (add, subtract, multiply, absolute difference, and, or, xor, minimum, maximum) on arrays, with accumulation.*

On EVE, this can be achieved by a combination of two functions, **Array_op_distrubuted** writing to a temporary array and a second loop that iterates over all 2-D blocks and accumulates them. As this sort of behavior is uncommon, this kernel is not implemented here.

26. Accumulate2D Array-op Scalar

Description

In the original VICP documentation, it was mentioned that Accumulate2D_array_op *performs point-by-point operation (add, subtract, multiply, absolute difference, and, or, xor, minimum, maximum) on an array by a scalar value, with accumulation.*

On EVE, this can be achieved by a combination of two functions, **Array_op_scalar** writing to a temporary array and a second loop that iterates over all 2-D blocks and accumulates them. As this sort of behavior is uncommon, this kernel is not implemented here.

27. Filter-op

Description

In the original VICP documentation, it was mentioned that Filter_op is *a generic 2-D filtering using ‘+’, ‘-‘, ‘| - |, ‘min’, ‘max’, logical operators.*

On EVE, this can be achieved by using **Filter** kernel and changing the operations in the core kernel to match that specified by Filter_op. As this sort of behavior is not commonly used, this kernel is not implemented here.

28. Filter-op_distribute

Description

In the original VICP documentation, it was mentioned that Filter_op_distribute *performs filtering on a 4-D data array with a 2-D coefficient array, producing a 4-D output array*.

As this sort of behavior is not commonly used, this kernel is not implemented here.

29. RGB16 to RGB Planar data

Description

This kernel separates R, G and B data from RGB555 and RGB565 packed 16-bit data fields. This kernel accepts a pointer to a 16-bit array of size *npixels* and writes out the data as 8-bit R, G and B arrays.

The RGB555 data is packed as 5-bits of R, G and B pixel values respectively with the 16th bit ignored.

RGB565 data is packed as 5-bits of R, 6-bits of G and 5-bits of B pixel values thus utilizing all 16-bits.

Usage

```
void vcop_rgb555_rgb
(
   __vptr_uint16   rgb16,
    unsigned int   npixels,
    __vptr_uint8   r,
    __vptr_uint8   g,
    __vptr_uint8   b
);

void vcop_rgb565_rgb
(
   __vptr_uint16   rgb16,
    unsigned int   npixels,
    __vptr_uint8   r,
    __vptr_uint8   g,
    __vptr_uint8   b
);
```

Constraints

1. *npixels* should be a multiple of 16.

Performance Considerations

3/16 cyc/pix for all three planes

30. RGB Planar to RGB16 data

Description

This kernel takes separate R, G and B data and packs it into 16-bit data of either RGB555 or RGB565 format. This kernel accepts a pointer to a 8-bit array of size *(3 * npixels)* and writes out the result as RGB555 or RGB565 data of array size *(npixels)*.

The RGB555 data is packed as 5-bits of R, G and B pixel values respectively with the 16th bit ignored.

RGB565 data is packed as 5-bits of R, 6-bits of G and 5-bits of B pixel values thus utilizing all 16-bits.

Usage

```
void vcop_rgb_rgb555
(
    __vptr_uint32  in_rgb,
    __vptr_int8    rgb_mask,
    unsigned int   npixels,
    __vptr_uint16  rgb555
);

void vcop_rgb_rgb565
(
    __vptr_uint32  in_rgb,
    __vptr_int8    rgb_mask,
    unsigned int   npixels,
    __vptr_uint16  rgb565
);
```

Constraints

1. *npixels* should be a multiple of 8.

Performance Considerations

4/8 cyc/pix for all three planes

31. Alpha Blending for YUV 420 NV12

Description

Perform alpha blending of two YUV420 NV12 images, using alpha plane. Alpha blending is carried out on the YUV420 NV12 source data and output is an image in same format.

Usage

```
void vcop_alpha_blend_yuv420nv12
(
   __vptr_uint8   in_img1_A,
   __vptr_uint8   in_img2_B,
   __vptr_uint8   alphaFrame_A,
   __vptr_uint8   out_B,
   unsigned short width,
   unsigned short height,
   unsigned short in_img1_stride,
   unsigned short in_img2_stride,
   unsigned short out_stride
);
```

Constraints

1. Width should be multiple of 16 to get optimal performance.
2. alphaFrame should be of size width*height

Performance Considerations

0.375 cycle/pix + 92 cycle VCOP overheads

32. Alpha Blending for YUV 422 Interleaved

Description

Perform alpha blending of two YUV422 interleaved images, using alpha plane. Alpha blending is carried out on the YUV422 interleaved source data that can be in the format of UYVY/YUYV where each pixel data is 16-bit.

Usage

```
void vcop_alpha_blend_yuv422i
(
    __vptr_uint8   in_img1,
    __vptr_uint8   in_img2,
    __vptr_uint8   alphaFrame,
    __vptr_uint8   out,
    unsigned short width,
    unsigned short height,
    unsigned short in_img1_stride,
    unsigned short in_img2_stride,
    unsigned short out_stride
);
```

Constraints

1. Width should be multiple of 8 to get optimal performance.
2. alphaFrame should be of size width*height

Performance Considerations

0.5 cycle/pix + 64 cycle VCOP overheads

33. YUV 420 NV12 to 422 UYVY Format Conversion

Description

Perform format conversion of a YUV420 NV12 image to YUV 422 UYVY image. The missing UV samples in the YUV 422 UYVY output are filled up by up-sampling by the UV in input by 2 (replication).

Usage

```
void vcop_yuv_420nv12_to_422uyvy
(
    __vptr_uint8   in_img_A,
    __vptr_uint8   out_B,
    unsigned short width,
    unsigned short height,
    unsigned short in_stride,
    unsigned short out_stride
);
```

Constraints

1. Width should be multiple of 8 to get optimal performance.

Performance Considerations

0.140625 cycles/pixel + 56 cycle VCOP overheads

34. YUV 422 UYVY to 420 NV12 Format Conversion

Description

Perform format conversion of a YUV420 NV12 image to YUV 422 UYVY image. The output UV values are obtained by averaging the even and odd row samples (with rounding) from YUV 422 UYVY input.

Usage

```
void vcop_yuv_422uyvy_to_420nv12
(
   __vptr_uint8   in_img_A,
   __vptr_uint8   out_B,
   unsigned short width,
   unsigned short height,
   unsigned short in_stride,
   unsigned short out_stride
);
```

Constraints

1. Width should be multiple of 8 to get optimal performance.

Performance Considerations

0.1875 cycles/pixel + 54 cycle VCOP overheads

35. YCbCr444Deinterleave (8-bit or 16-bit pixels sorted into 3 separate arrays)

Description

This kernel separates the three color components of YCbCr images and writes them to individual arrays. The input pixels are arranged in the 444 sampling format.

Usage

The prototype for the function is:

```
void vcop_YCbCr444_Deinterleave444_char
(
    __vptr_uint32  YCbCr_char,
    __vptr_int8    YCbCrmask,
    unsigned int   npixels,
    __vptr_uint8   Y_char,
    __vptr_uint8   Cb_char,
    __vptr_uint8   Cr_char
);

void vcop_YCbCr444_Deinterleave444_short
(
    __vptr_uint32  YCbCr_short,
    __vptr_int8    YCbCrmask,
    unsigned int   npixels,
    __vptr_uint16  Y_short,
    __vptr_uint16  Cb_short,
    __vptr_uint16  Cr_short
);
```

The ***YCbCrmask*** arrays give the values that indicate shift amounts for the Y, Cb and Cr should be shifted by to separate them.

Performance

YCbCr444_Deinterleave444_char: 3/8 cyc/pix
YCbCr444_Deinterleave444_short: 3/4 cyc/pix

36. YCbCr422Deinterleave (8-bit or 16-bit pixels sorted into 3 separate arrays)

Description

This kernel separates the three color components of YCbCr images and writes them to individual arrays. The input pixels are arranged in the 422 sampling format.

Usage

The prototype for the function is:

```
void vcop_YCbCr422_Deinterleave422_char
(
    __vptr_uint8   YCbYCr_char,
    unsigned int   npixels,
    __vptr_uint8   Y_char,
    __vptr_uint8   Cb_char,
    __vptr_uint8   Cr_char
);

void vcop_YCbCr422_Deinterleave422_short
(
    __vptr_uint16  YCbYCr_short,
    unsigned int   npixels,
    __vptr_uint16  Y_short,
    __vptr_uint16  Cb_short,
    __vptr_uint16  Cr_short
);
```

Performance

YCbCr422_Deinterleave422_char: 1/8 cyc/pix

YCbCr422_Deinterleave422_short: 1/8 cyc/pix

37. YCbCr444toYCbCr422

Description

This kernel separates the three color components of YCbCr images, arranged in the 444 format, and writes them to individual arrays while downsampling the chroma components to write them out in 422 format. The conversion is carried out by averaging two consecutive chroma samples.

Usage

The prototype for the function is:

```
void vcop_YCbCr444_Downsample422_char
(
    __vptr_uint32  YCbCr_char,
    __vptr_int8    YCbCrmask,
    unsigned int   npixels,
    __vptr_uint8   Y_char,
    __vptr_uint8   Cb_char,
    __vptr_uint8   Cr_char
);

void vcop_YCbCr444_Downsample422_short
(
    __vptr_uint32  YCbCr_short,
    __vptr_int8    YCbCrmask,
    unsigned int   npixels,
    __vptr_uint16  Y_short,
    __vptr_uint16  Cb_short,
    __vptr_uint16  Cr_short
);
```

The ***YCbCrmask*** arrays give the values that indicate shift amounts for the Y, Cb and Cr should be shifted by to separate them.

Performance

YCbCr444_Downsample422_char: 5/8 cyc/pix
YCbCr444_Downsample422_short: 5/4 cyc/pix

38. YCbCr444toYCbCr420

Description

The conversion from YCbCr444 to YCbCr420 can be carried out in a similar fashion to that described in section 46 above.

39. YCbCr422toYCbCr420

Description

The conversion from YCbCr422 to YCbCr420 can be carried out in a similar fashion to that described in section 46 above.

40. YCbCrInterleave (YCbCrPack)

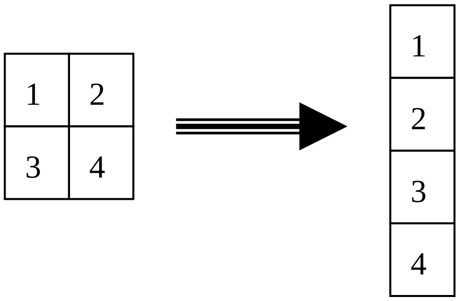
Description

YCbCr pack can be carried out using the ‘interleave’ function that is part of the VCOP instruction set.

41. Block2Sequence

Description

This function reorganizes the data into a block sequential fashion.



This function can be written easily using LD and ST instructions of VCOP. The other way more useful way to think of this problem is to use the DMA to read in separate blocks and write them out in the output format after relevant processing by VCOP.

42. Color Space Conversion

Description

This function converts RGB to interleaved YUV, by applying coefficients weighting red, blue and green, to luma. Similarly we apply a weighted product of red, blue and green to U and V.

```
Y = ( (  66 * R + 129 * G +  25 * B + 128) >> 8) +  16
U = ( ( -38 * R -  74 * G + 112 * B + 128) >> 8) + 128
V = ( ( 112 * R -  94 * G -  18 * B + 128) >> 8) + 128
```

The values from this equation, will have to be clamped between 0..255.

Usage

```
void rgb_to_yuv
(
    __vptr_uint8    iPtrR,
    __vptr_uint8    iPtrG,
    __vptr_uint8    iPtrB,
    unsigned short  width,
    short           coefs_r,
    short           coefs_g,
    short           coefs_b,
    __vptr_int16    coefs,
    __vptr_uint8    iPtrOut
);
```

The function "rgb_to_yuv" accepts red, green and blue pixels in arrays "iPtrR", "iPtrG" and "iPtrB" into an interleaved array of YUYV pixels stored in "iPtrOut" by applying a 3x3 array of coefficients to convert from RGB to YUV space.

Performance Considerations

3/8 cyc/pix

VCOP features used

This function is intended to highlight the value of predicated stores to memory.   It also uses the circular buffering feature of VCOP.

43. Sum

Description

This can be easily accomplished by following the steps of Block_Average or Integral_Image functions described earlier. The optimal method of implementing this is to first add the rows for “compute_height” worth of pixels, write the result out as a transpose and have a second loop to add “compute_width” worth of pixels in the vertical order.

44. Intensity Scaling

Description

The routine accepts a gray-scale input image (inImg) of size blockWidth by blockHieght and bins the input gray-scale pixels into bins as specified by 'scalingLUT'. The output at 'outImg' is of size blockWidth by blockHeight. The kernel uses 8-way table look up feature of VCOP. The scaling LUT need to be provided with replication for enabling 8-way look up table. A utility function (prepare_lut) is provided with the kernel to provide a default LUT that uniformly quantizes the input pixels between ‘loPixelVal’ and ‘hiPixelVal’ into ‘numLevels’ bins.

Usage

```
void vcop_intensity_scaling
(
   __vptr_uint8       inImg_A,
   __vptr_uint8       scalingLUT_C,
   __vptr_uint8       outImg_B,
   unsigned short     blockWidth,
   unsigned short     blockHeight,
   unsigned short     blockStride
);
```

Constraints

2. Width should be multiple of 16 to get optimal performance.
3. alphaFrame should be of size width*height

EVE features used

- 8 way look-up table

45. YUV Padding

Description

This kernel accepts a line of data and generates 2D block outputs by repeating the line vertically. To make use of two functional units and two image buffers, this function operates on two sets of data in parallel. VLIB has two flours of this function one for 8 bit data and another for 16 bit data. 16 bit kernel can be used for interleaved UV padding

X0	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	X11	X12	X13	X14	X15	X16

X0	X0	X0	X0	X0	X0	X0	X0
X1	X1	X1	X1	X1	X1	X1	X1
X2	X2	X2	X2	X2	X2	X2	X2
X3	X3	X3	X3	X3	X3	X3	X3
X4	X4	X4	X4	X4	X4	X4	X4
X5	X5	X5	X5	X5	X5	X5	X5
X6	X6	X6	X6	X6	X6	X6	X6
X7	X7	X7	X7	X7	X7	X7	X7
X8	X8	X8	X8	X8	X8	X8	X8
X9	X9	X9	X9	X9	X9	X9	X9
X10	X10	X10	X10	X10	X10	X10	X10
X11	X11	X11	X11	X11	X11	X11	X11
X12	X12	X12	X12	X12	X12	X12	X12
X13	X13	X13	X13	X13	X13	X13	X13
X14	X14	X14	X14	X14	X14	X14	X14
X15	X15	X15	X15	X15	X15	X15	X15
X16	X16	X16	X16	X16	X16	X16	X16

API

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

```
unsigned int vcop_yuv_left_right_padding_u8(
   __vptr_uint8 in_left_A,
   __vptr_uint8 in_right_B,
   __vptr_uint8 out_left_A,
   __vptr_uint8 out_right_B,
   unsigned short width,
   unsigned short height,
   unsigned short out_stride,
   unsigned short pblock[])
```

- in_left_A: Pointer to First input line
- in_right_B: Pointer to Second input line
- out_left_A: Pointer to First 2D output block
- out_right_B: Pointer to Second 2D output block
- width : Processing block width
- height : Processing block height
- out_stride : Stride of output blocks

Constraints

- None

Performance Considerations

- Buffers related to two output blocks shall be kept in separate buffet for best performance.
- Performance of this function will be : 1/32 cycles per output pixel

46. Software Image Signal Processor (Soft ISP) kernels

Description

The software image signal processor (Soft ISP) consists of collection of kernels to be used for implementing an ISP for RCCC sensor. The following kernels are provided:

vcop_decompand_piecewise_linear

This routine accepts a 16-bit companded RAW input image and outputs a linearized RAW image after decompanding to a 16-bit linear data. The operation assumes piecewise decompanding with two knee points.

vcop_soft_isp_extract_r

This routine accepts a 16-bit RAW input image in RCCC format and extracts R pixels. The R pixels are downscaled to 8-bit.

vcop_black_clamp_c_balance

This routine accepts a 16-bit RAW input image and outputs a RAW image after dark current subtraction/black clamp and C imbalance correction (for an RCCC image).

vcop_rccc_to_cccc

This routine accepts a 16-bit RAW input image in RCCC format and performs CFA interpolation to fill out the missing C samples at R locations. The kernel implements an edge/line aware CFA interpolation scheme.

vcop_gbce_simple

This routine accepts a 16-bit RAW input image in RCCC format and performs global brightness and contrast enhancement (GBCE). The kernel expects a 12-bit GBCE tone curve to be provided at pGbceToneCurve. The tone curve is expected to be provided as a 4-way LUT. The input pixel value is shifted first to ensure that the bit depth is less than or equal to 12 bit before looking-up in the tone curve.

vcop_gbce_interp

This kernel performs global tone mapping for input images with bit depth between 12 and 16. This routine accepts a 16-bit RAW input image in RCCC format and performs global brightness and contrast enhancement (GBCE). A 12-bit LUT is provided for performing GBCE. In this method the input pixel value is used to find out the two nearest entries in the LUT provided and an interpolated value is reported as the output.

vcop_stats_collector_dense

This is the c-reference for the vcop_stats_collector_dense kernel which collects certain statistics for assisting Auto Exposure algorithm in deciding the exposure settings for the sensor. In this function the statistics is collected from every pixel within a window.

Constraints

- Block width and block heights should be multiple of 2. This is expected by nature of the RCCC pixel format.
- Statistics Block width should be a multiple of 16 and block height should be a multiple of 2.

EVE features used

- 4 way look-up table

47. Contrast Stretching

Description

The routine accepts a gray-scale input image (inputImage) of size blkWidth by blkHeight and uses min and max value of histogram to stretch the intensity levels to the full range of intensity. The output at outputImage of size blkWidth by blkHeight. Following kernels are used for this feature :

vcop_histogram_8c_word:
This routine accepts an input image of blkWidth and blkHeight size and gives 8 copies of histogram using 8 way histogram update feature.

vcop_contrast_stretching :
This routines accepts an input image of blkWidth and blkHeight size and gives an output image of same size by applying the scaling factor derived from minVal and scaleFactor16

vcop_histogram_8c_word_sum :
This routine accepts an eight copy histogram and returns a single histogram by summing all the 8 copies.

Usage

```
void vcop_histogram_8c_word
(
   __vptr_uint8  inputImage,
   unsigned short blkWidth,
   unsigned short blkHeight,
   unsigned short inPitch,
   __vptr_uint32 histogram8Copy
)

void vcop_contrast_stretching
(
  __vptr_uint8  inputImage,
  __vptr_uint8 outputImage,
  unsigned short blkWidth,
  unsigned short blkHeight,
  unsigned short inPitch,
  unsigned short outPitch,
  unsigned char  minVal,
  unsigned short scaleFactorQ16
)
void vcop_histogram_8c_word_sum
(
    __vptr_uint32  histogram8Copy,
    __vptr_uint32  transposeBuff,
    __vptr_uint32  histogram
)
```

Constraints

Width should be multiple of 16.

EVE features used

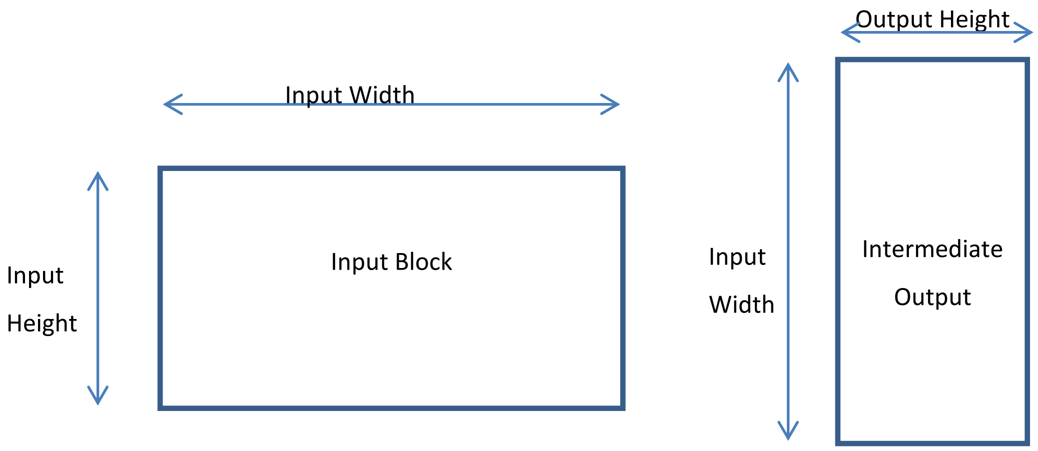
- 8 way histogram update
- Transpose Store

48. YUV Scalar

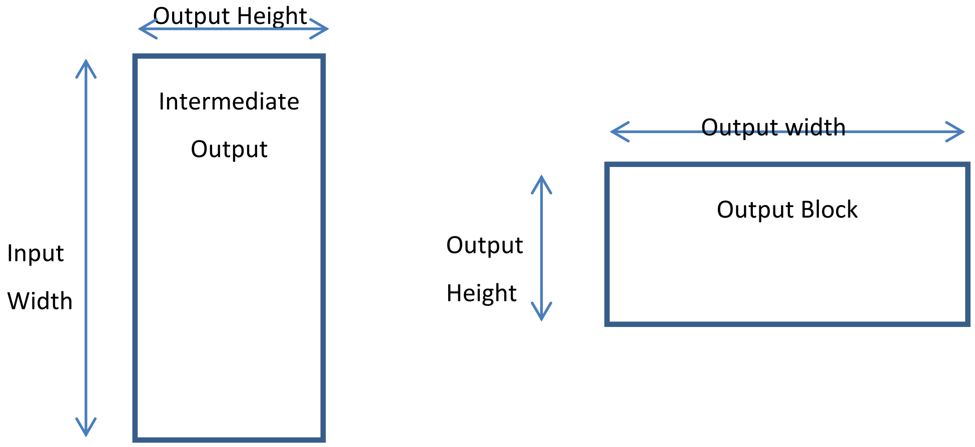
Description

The below sequence of three kernels can be used to re-size a 2D block in in vertically and store the 8 bit image (Y) in transpose manner. The resize can either be down scale or upscale. The maximum scale ratio supported is 2x.

```
yuv_scalar_pixels_look_up_kernel
yuv_scalar_interpolation_kernel
yuv_scalar_luma_copy_kerne
```



These three kernels can again be used for horizontal resize that is resizing the transposed block. At the end two such pass, the 2D block can be re-sized in vertical and horizontal direction. The intermediate (output of vertical resize pass) is truncated to 8 bit.



Similarly the below sequence of below kernels can be used for resize of inter leaved UV data (8 bit each)

```
yuv_scalar_pixels_look_up_kernel
yuv_scalar_interpolation_kernel
yuv_scalar_chroma_copy_kernel
```

Usage

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

```
void yuv_scalar_pixels_look_up_kernel
        (
        __vptr_uint32 src,
        __vptr_uint16 index,
        __vptr_uint32 outBuf,
        short int numTaps,
        short int src_w,
        short int dst_h
        )
```

- src: Pointer to input 2D block
- index: Pointer to Index buffer
- outBuf: Pointer to 2D output block
- numTaps: Number of taps for the filtering
- src_w : Input block Pitch
- dst_h : Number of output lines

```
void yuv_scalar_interpolation_kernel(
                                __vptr_uint8 inPtr,
                                __vptr_uint8 fracPtr,
                                __vptr_uint8 temp1Ptr,
                                __vptr_uint8 temp2Ptr,
                                __vptr_uint16 offsetPtr,
                                short int tempBufPitch,
                                short int fracBits,
                                short int numTaps,
                                short int src_w,
                                short int src_pitch,
                                short int dst_h)
```

- inPtr: Pointer to output of previous kernel
- fracPtr: Pointer to filter co-efficient buffer
- temp1Ptr: Pointer to 2D output block
- temp2Ptr: Pointer to 2D output block
- offsetPtr: Offset data for transpose store
- tempBufPitch: Pitch of transpose buffer
- fracBits : Q format used for filter co-efficient
- numTaps: Number of taps for the filtering
- src_w : Input block Width
- src_pitch: Input block Pitch
- dst_h : Number of output lines
-

```
void yuv_scalar_luma_copy_kernel(
                                        __vptr_uint32 outPtr,
                                        __vptr_uint32 temp1Ptr,
                                        __vptr_uint32 temp2Ptr,
                                       short int tempBufPitch,
                                       short int src_w,
                                       short int dst_h
```

- outPtr: Pointer to 2D output block
- temp1Ptr: Pointer to output of previous kernel
- temp2Ptr: Pointer to output of previous kernel
- tempBufPitch: Pitch of transpose buffer
- src_w : Input block Width
- dst_h : Number of output lines

```
    void yuv_scalar_chroma_copy_kernel(
                                        __vptr_uint8 outPtr,
                                        __vptr_uint8 temp1Ptr,
                                        __vptr_uint8 temp2Ptr,
                                       short int tempBufPitch,
                                       short int src_w,
                                       short int dst_h)
```

- outPtr: Pointer to 2D output block
- temp1Ptr: Pointer to output of previous kernel
- temp2Ptr: Pointer to output of previous kernel
- tempBufPitch: Pitch of transpose buffer
- src_w : Input block Width

- dst_h : Number of output lines

Constraints

- Maximum re-size ration supported is 2x
- The output block size (both width and height) shall be multiple of 32

Performance Considerations

- Please refer the kernel test bench code for buffer placement

49. Binary Masking

Description

The routine accepts a binary bit packed image and a binary bytemap (containing 0 and 1) of size computeWidth by computeHeight and returns a bit packed binary image which contains the output of "AND" operation of these two image. Following kernels are used for this feature :

Usage

```
void vcop_binary_masking
(
    __vptr_uint8  inByteData,
    __vptr_uint8  inBitPackedData,
    __vptr_uint8  outBitPackedData,
    unsigned short computeWidth,
    unsigned short computeHeight,
    unsigned short inputByteDataPitch,
    unsigned short inputBitDataPitch,
    unsigned short outputPitch
)
```

@inputs This kernel takes following Inputs

inByteData : Input byte data containing only 0 and 1. Size of this buffer should be blockWidth * blockHeight * sizeof(uint8_t)

inBitPackedData : Input bit data containing bit packed data. Size of this buffer should be blockWidth /8 * blockHeight * sizeof(uint8_t)

computeWidth : Width of the output of this kernel. This is basically blockWidth

computeHeight : Width of the output of this kernel. This is basically blockHeight

inputByteDataPitch : Pitch in bytes for the byte data

inputBitDataPitch : Pitch in bytes for the bit packed data

outputPitch : Pitch of the output data

@outputs This kernel produce following outputs

outBitPackedData: Pointer to the output buffer containing the output of this kernel in bit packed data. Size of this buffer should be ((computeWidth) / 8 * computeHeight * size(int8_t)

EVE features used

- Pack operation

50. Select List Elements

Description

The routine accepts a list in in uint32_t format and a byte mask indicating which locations of the list needs to be retained by marking them 1 and rest all the list elements are suppressed. Following kernels are used for this feature :

Usage

```
void vcop_select_list_elements
(
  __vptr_uint32  inputList,
  __vptr_uint32  outputList,
  __vptr_uint8   selectionMask,
  __vptr_uint16  ouputListSize,
  unsigned short listSize,
  unsigned short  selectionMaskSize
)
```

@inputs   This kernel takes following Inputs

inputList: Input list data with each entry of size a word. Size of this buffer should be listSize * sizeof(uint32_t)

selectionMask: Mask which will be used to select elements from the list, Mask will have a value of 1 at the location where we want to pick the value from the list and 0 at other places. The size of this mask should be multiple of 8. Size of this buffer should be selectionMaskSize * sizeof(uint8_t)

listSize: Total number of elements present in the list

selectionMaskSize: Size of the mask to be used in terms of bytes. This should be multiple of 8.

@outputs   This kernel produce following outputs

outputList: Pointer to the output buffer which will contain the elements which are selected from the input list based on the mask provided. User should give a worst case buffer size which is same as inputList size.

OuputListSize : Pointer to the buffer which will store the number of elements detected. Size of this buffer should be sizeof(uint8_T) * 8.

EVE features used

- Collate Store