Vision Library (VLIB) Application Programming Interface

Reference Guide

Literature Number: SPRUG00C
November 2009

TEXAS INSTRUMENTS

Reference Guide
SPRUG00C – November 2009

Vision Library (VLIB) Application Programming Interface

1 About VLIB V2.1 Release

The Vision Library (VLIB) is a collection of computer vision algorithms that have been optimized for Texas Instruments' digital media processors. The VLIB Version 2.1 software library was developed for devices with a C64x or C64x+ processing core. This Application Programming interface (API) supports rapid integration of VLIB for embedded vision applications.

These incarnations of release version 2.1 are supplied:

- vlib.l64p
- vlib_errchk.l64p
- vlib.l64
- vlib_errchk.l64
- vlib.lib
- VLIB_lib.mdl

The first two libraries are for C64x+ and the next two libraries are for C64x. vlib.l64p and vlib.l64 are compiled with full file-level optimization enabled and with no debug information. vlib_errchk.l64p and vlib_errchk.l64 versions contain more error checking of input arguments for some of the library functions. These builds are designed to produce richer error reporting for debug purposes but the added overhead can slow performance (marginally in most cases).

Self-verifying examples are provided with the library to demonstrate how to use the API. The main test application works with the latest version of TI's Code Composer Studio, version 3.3. The vlib.lib library is a bit-exact version of the library for testing in PC (Windows) environments. It was compiled using Microsoft Visual C++ 6.0. The VLIB_lib.mdl file contains Simulink blocks for development and code generation in the matlab environment.

2 Exponentially-Weighted Running Mean of a Video (16-Bit)

2.1 Introduction and Use Cases

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*.

2.2 Specification

2.2.1 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.

2.2.2 Inputs

`short`	`*runningMean`	EW running mean buffer to be updated	(SQ8.7)
`char`	`*newLuma`	Most recent luma buffer	(UQ8.0)
`unsigned int`	`*mask32packed`	Foreground mask buffer	(32-bit packed)
`short`	`weight`	Weight of the newest luma	(SQ0.15)
`unsigned int`	`pixelCount`	Number of pixels to process	(UQ32.0)

2.2.3 Output

`int`	Returns VLIB Error Status

2.2.4 Method

In the implementation shown in Equation 1, the exponential running mean is updated for those pixels where the foreground mask is zero:

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

2.2.5 APIs

```
int VLIB_updateEWRMeanS16(
            short * restrict runningMean,
            const char * restrict newLuma,
            const unsigned int * restrict mask32packed,
            const short weight,
            const unsigned int pixelCount);
```

The following function can be used to initialize a running mean buffer with luma values. In this process, all UQ8.0 luma values are converted into SQ8.7 representation.

```
int VLIB_initMeanWithLumaS16(
            short * restrict runningMean,
            const char * restrict lumaFrame,
            const unsigned int pixelCount);
```

2.2.6 Requirements

- I/O buffers are assumed to be double-word aligned in memory.
- pixelCount must be a multiple of 8.

2.3 Comments

2.3.1 Adaptation Through Running Statistics

Over the course of a day, the illumination of an outdoor scene changes drastically. A background model needs to adapt to such effects and only report changes inherent to the scene, as opposed to its appearance. One practical approach is to compute the running (moving) statistics of the scene over a period of observation.

2.3.2 Foreground Objects

Based on inference or a priori knowledge, one could classify certain pixels of a video frame as foreground object (or outlier) and exclude them from the averaging operation. This mechanism would keep foreground object pixels from influencing the running mean of the background.

2.4 Performance Benchmarks

On-chip memory performance of the kernels has been measured as.

VLIB_updateEWRMeanS16	1.0 cycles/pixel
VLIB_initMeanWithLumaS16	0.4 cycles/pixel

2.5 References

1. Chapter 15: Moving Average Filters in *Digital Signal Processing: A Practical Guide for Engineers and Scientists*, Steven W. Smith, 2002, ISBN 0-7506-7444.
2. "Moving object recognition using and adaptive background memory" in *Time-Varying Image Processing and Moving Object Recognition*, K.P. Karmann and A. von Brandt, Elsevier Science Publishers B.V., 1990.

3 Exponentially-Weighted Running Mean of a Video (32-Bit)

3.1 Introduction and Use Cases

A background subtraction algorithm commonly consists 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*.

3.2 Specification

3.2.1 Function

Updates the exponential running mean of the luma component of a video. If the foreground mask bit is set for a pixel, indicating there is obstruction by a foreground object, the running mean will not be updated for that pixel.

3.2.2 Inputs

`int`	`*runningMean`	EW running mean buffer to be updated	(SQ8.23)
`char`	`*newLuma`	Most recent luma buffer	(UQ8.0)
`unsigned int`	`*mask32packed`	Foreground mask buffer	(32-bit packed)
`int`	`weight`	Weight of the newest luma	(SQ0.31)
`unsigned int`	`pixelCount`	Number of pixels to process	(UQ32.0)

3.2.3 Output

`int`	Returns VLIB Error Status

3.2.4 Method

In the implementation shown in Equation 2, the exponential running mean is updated for those pixels where the foreground mask is zero:

$$\text{updatedMean} = (1\text{-weight}) \times \text{previousMean} + \text{weight} \times \text{newestData} \quad (2)$$

3.2.5 APIs

```
int VLIB_updateEWRMeanS32(
            int * restrict runningMean,
            const char  * restrict newLuma,
            const unsigned int * restrict mask32packed,
            const int  weight,
            const unsigned int pixelCount);
```

The following function can be used to initialize a running mean buffer with luma values. In this process, all UQ8.0 luma values are converted into SQ8.23 representation.

```
int VLIB_initMeanWithLumaS32(
            int * restrict runningMean,
            const char * restrict lumaFrame,
            const unsigned int pixelCount);
```

3.2.6 Requirements

- I/O buffers are assumed to be double-word aligned in memory.
- pixelCount must be a multiple of 4.

3.3 Comments

3.3.1 Adaptation Through Running Statistics

Over the course of a day, the illumination of an outdoor scene changes drastically. A background model needs to adapt to such effects and only report changes inherent to the scene, as opposed to its appearance. One practical approach is to compute the running (moving) statistics of the scene over a period of observation.

3.3.2 Foreground Objects

Based on inference or a priori knowledge, one could classify certain pixels of a video frame as foreground object (or outlier) and exclude them from the averaging operation. This mechanism would keep foreground object pixels from influencing the running mean of the background.

3.4 Performance Benchmarks

On-chip memory performance of the kernels has been measured as.

VLIB_updateEWRMeanS32	2.0 cycles/pixel
VLIB_initMeanWithLumaS32	0.8 cycles/pixel

3.5 References

1. Chapter 15: Moving Average Filters in *Digital Signal Processing: A Practical Guide for Engineers and Scientists*, Steven W. Smith, 2002, ISBN 0-7506-7444.
2. "Moving object recognition using and adaptive background memory" in *Time-Varying Image Processing and Moving Object Recognition*, K.P. Karmann and A. von Brandt, Elsevier Science Publishers B.V., 1990.

4 Exponentially-Weighted Running Variance of a Video (16-Bit)

4.1 Introduction and Use Cases

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*.

   The *exponentially-weighted (EW) running variance* of a pixel can be used in deciding whether an observed deviation is statistically significant.

4.2 Specification

4.2.1 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.

4.2.2 Inputs

`short`	`*runningVar`	EW running variance to be updated	(SQ12.3)
`short`	`*runningMean`	EW running mean buffer	(SQ8.7)
`char`	`*newLuma`	Most recent luma buffer	(UQ8.0)
`unsigned int`	`*mask32packed`	Foreground mask buffer	(32-bit packed)
`short`	`weight`	Weight of the newest luma	(SQ0.15)
`unsigned int`	`pixelCount`	Number of pixels to process	(UQ32.0)

4.2.3 Output

`int`	Returns VLIB Error Status

4.2.4 Method

In the implementation shown in Equation 3, the exponential running variance is updated for those pixels where the foreground mask is zero:

$$\text{updatedVar} = (1 - \text{weight}) \times \text{previousVar} + \text{weight} \times (\text{newestData} - \text{previousMean})^2 \quad (3)$$

4.2.5 APIs

```
int VLIB_updateEWRVarianceS16(
            short * restrict runningVar,
            const short * restrict runningMean,
            const char * restrict newLuma,
            const unsigned int * restrict mask32packed,
            const short weight,
            const unsigned int pixelCount);
```

The following function can be used to initialize a running variance buffer with a constant variance value. The latter is expected to be in SQ12.3 format already.

```
int VLIB_initVarWithConstS16(
            short * restrict runningVar,
            const short constVar,
            const unsigned int pixelCount);
```

4.2.6 Requirements

- I/O buffers are assumed to be double-word aligned in memory.
- pixelCount must be a multiple of 8.

4.3 Performance Benchmarks

On-chip memory performance of the kernels has been measured as.

VLIB_updateEWRVarianceS16	1.3 cycles/pixel
VLIB_initVarWithConstS16	0.1 cycles/pixel

5 Exponentially-Weighted Running Variance of a Video (32-Bit)

5.1 Introduction and Use Cases

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*. The *exponentially-weighted (EW) running variance* of a pixel can be used in deciding whether an observed deviation is statistically significant.

5.2 Specification

5.2.1 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.

5.2.2 Inputs

`int`	`*runningVar`	EW running variance to be updated	(SQ16.15)
`int`	`*runningMean`	EW running mean buffer	(SQ8.23)
`char`	`*newLuma`	Most recent luma buffer	(UQ8.0)
`unsigned int`	`*mask32packed`	Foreground mask buffer	(32-bit packed)
`int`	`weight`	Weight of the newest luma	(SQ0.31)
`unsigned int`	`pixelCount`	Number of pixels to process	(UQ32.0)

5.2.3 Output

`int`	Returns VLIB Error Status

5.2.4 Method

In the implementation shown in Equation 4, the exponential running variance is updated for those pixels where the foreground mask is zero:

$$\text{updatedVar} = (1 - \text{weight}) \times \text{previousVar} + \text{weight} \times (\text{newestData} - \text{previousMean})^2 \quad (4)$$

5.2.5 APIs

```
int VLIB_updateEWRVarianceS32(
            int * restrict runningVar,
            const int * restrict runningMean,
            const char * restrict newLuma,
            const unsigned int * restrict mask32packed,
            const int weight,
            const unsigned int pixelCount);
```

The following function can be used to initialize a running variance buffer with a constant variance value. The latter is expected to be in SQ16.15 format already.

```
int VLIB_initVarWithConstS32(
            int * restrict runningVar,
            const int constVar,
            const unsigned int pixelCount);
```

5.2.6 Requirements

- I/O buffers are assumed to be double-word aligned in memory.
- pixelCount must be a multiple of 4.

5.3 Performance Benchmarks

On-chip memory performance of the kernels has been measured as.

VLIB_updateEWRVarianceS32	2.3 cycles/pixel
VLIB_initVarWithConstS32	0.3 cycles/pixel

6 Uniformly-Weighted Running Mean of a Video (16-Bit)

6.1 Introduction and Use Cases

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 *uniformly-weighted (UW) running mean* (a.k.a., moving average).

Special requirements:

For averaging, a video buffer of N luma frames need to be stored in memory. The user is expected to maintain this buffer and pass the appropriate frame pointers to the function.

6.2 Specification

6.2.1 Function

Updates the (uniformly-weighted) running mean of the luma component of a video. If the foreground mask bit of either the newest or the oldest video frame is set, indicating there is obstruction by a foreground object, the running mean will not be updated.

6.2.2 Inputs

`short`	`*updatedMean`	Updated running mean buffer	(SQ8.7)
`short`	`*previousMean`	Previous running mean buffer	(SQ8.7)
`char`	`*newestData`	Most recent luma buffer	(UQ8.0)
`unsigned int`	`*oldestData`	Oldest luma buffer	(UQ8.0)
`unsigned int`	`*newestMask32packed`	Newest mask buffer	(32-bit packed)
`unsigned int`	`*oldestMask32packed`	Oldest mask buffer	(32-bit packed)
`unsigned int`	`pixelCount`	Number of pixels to in the luma buffer	(UQ32.0)
`unsigned char`	`frameCount`	Number of frames in video buffer	(UQ8.0)

6.2.3 Output

`int`	Returns VLIB Error Status

6.2.4 Method

In the implementation shown in Equation 5, the running mean is updated for those pixels where the foreground mask of the oldest and newest frames are zero:

$$\text{updatedMean} = \text{previousMean} + (\text{newestData} - \text{oldestData}) \div (\text{frameCount} - 1) \quad (5)$$

6.2.5 APIs

```
int VLIB_updateUWRMeanS16(
            short * restrict updatedMean,
            const short * restrict previousMean,
            const char * restrict newestData,
            const char * restrict oldestData,
            const unsigned int * restrict newestMask32packed,
            const unsigned int * restrict oldestMask32packed,
            const unsigned int pixelCount,
            const unsigned char frameCount);
```

The following function can be used to initialize a running mean buffer with luma values. In this process, all UQ8.0 luma values are converted into SQ8.7 representation.

```
int VLIB_initMeanWithLumaS16(
            short * restrict runningMean,
            const char * restrict lumaFrame,
            const unsigned int pixelCount);
```

6.2.6 Requirements

- I/O buffers are assumed to be double-word aligned in memory.
- pixelCount must be a multiple of 8.

6.3 Performance Benchmarks

On-chip memory performance has been measured as 1.0 cycles/pixel.

6.4 References

1. Chapter 15: Moving Average Filters, in *Digital Signal Processing: A Practical Guide for Engineers and Scientists*, Steven W. Smith, 2002, ISBN 0-7506-7444.

7 Uniformly-Weighted Running Variance of a Video (16-Bit)

7.1 Introduction and Use Cases

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*. The *uniformly-weighted running variance* of a pixel can be used in deciding whether an observed deviation is statistically significant.

7.2 Specification

7.2.1 Function

Updates the (uniformly-weighted) running variance of the luma component of a video. If the foreground mask bit of either the newest or the oldest video frame is set, indicating there is obstruction by a foreground object, the running variance will not be updated.

7.2.2 Inputs

`short`	`*updatedVar`	Updated running variance buffer	(SQ12.3)
`short`	`*updatedMean`	Updated running mean buffer	(SQ8.7)
`short`	`*previousMean`	Previous running mean buffer	(SQ8.7)
`short`	`*previousVar`	Previous running variance buffer	(SQ12.3)
`char`	`*newestData`	Most recent luma buffer	(SQ8.0)
`unsigned int`	`*newestMask32packed`	Newest foreground mask	(32-bit packed)
`unsigned int`	`*oldestMask32packed`	Oldest foreground mask	(32-bit packed)
`unsigned int`	`pixelCount`	Number of pixels to process	(UQ32.0)
`unsigned char`	`frameCount`	Number of frames in video buffer	(UQ8.0)

7.2.3 Output

`int`	Returns VLIB Error Status

7.2.4 Method

In the implementation shown in Equation 6, the running variance is updated for those pixels where the foreground mask of the oldest and newest frames are zero:

$$\text{updatedVar} = 1 \div (\text{frameCount}-1) \times (\text{frameCount}\times\text{previousVar} + (\text{newestData}-\text{updatedMean}) \times (\text{newestData}-\text{previousMean})) \tag{6}$$

7.2.5 APIs

```
int VLIB_updateUWRVarianceS16(
            short * restrict updatedVar,
            const short * restrict previousVar,
            const short * restrict updatedMean,
            const short * restrict previousMean,
            const char * restrict newestData,
            const unsigned int * restrict newestMask32packed,
            const unsigned int * restrict oldestMask32packed,
            const unsigned int pixelCount,
            const unsigned char frameCount);
```

The following function can be used to initialize a running variance buffer with a constant variance value. The latter is expected to be in SQ12.3 format already.

```
int VLIB_initVarWithConstS16(
            short * restrict runningVar,
            const short constVar,
            const unsigned int pixelCount);
```

7.2.6 Requirements

- I/O buffers are assumed to be double-word aligned in memory.
- pixelCount must be a multiple of 8.

7.3 Performance Benchmarks

On-chip memory performance has been measured as 2.0 cycles/pixel.

8 Statistical Background Subtraction (16-Bit)

8.1 Introduction and Use Cases

In background subtraction, thresholding can be used to decide whether a pixel's observed value deviates too far from its model (that is, the average of its past values). Assuming each pixel's variance has been modeled, one might threshold a deviation image with a (scaled) variance image.

8.2 Specification

8.2.1 Function

This function implements a statistical background segmentation algorithm

8.2.2 Inputs

`unsigned int`	`*mask32packed`	Binary mask to be computed	(32-bit packed)
`char`	`*newLuma`	Most recent luma buffer	(UQ8.0)
`short`	`*runningMean`	EW running mean buffer	(SQ8.7)
`short`	`*runningVar`	EW running variance buffer	(SQ12.3)
`short`	`thresholdGlobal`	Global threshold value	(SQ12.3)
`short`	`thresholdFactor`	Multiplicative factor for threshold	(SQ4.11)
`unsigned int`	`pixelCount`	Number of pixels to process	(UQ32.0)

8.2.3 Output

`int`	Returns VLIB Error Status

8.2.4 Method

For each pixel, the running mean and variance statistics are assumed to be known. The routine makes comparisons between three scalar values for each pixel:

1. The squared distance between the most recent luma measurement and the running mean determined by Equation 7:

$$(\text{newLuma} - \text{runningMean})^2 \quad (7)$$

2. The thresholdGlobal
3. thresholdFactor × runningVar

For a pixel to be classified as foreground, (1) needs to be greater than both (2) and (3). When these conditions are satisfied, the observation is deemed to stem from a foreground object (and not from the modeled background), and the corresponding mask pixel value is set to 1.

The comparison with (2) plays the role of assuming a minimum variance for the pixel values, as in camera noise, etc. A sequence of luma observations might be very consistent, driving the running variance to small values. In such cases, camera noise could cause a pixel to pass the foreground threshold. By setting a reasonably high camera noise value (which is a "squared" scalar), one can filter out the camera noise.

Note that the thresholdFactor is also in *squared* form: if you would like measurements which are 2 standard deviations away from the mean to be classified as foreground, the thresholdFactor should be set to 2×2=4. This variable is represented as SQ4.11 (sign bit, 4 integer bits, 11 fractional bits). In hex-format, it 4(dec) would read 0x2000.

8.2.5 APIs

```
int VLIB_subtractBackgroundS16(
            unsigned int * restrict mask32packed,
            const char * restrict newLuma,
            const short * restrict runningMean,
            const short * restrict runningVar,
            const short thresholdGlobal,
            const short thresholdFactor,
            const unsigned int PixelCount);
```

8.2.6 Requirements

- I/O buffers are assumed to be double-word aligned in memory.
- pixelCount must be a multiple of 8.

8.3 Performance Benchmarks

On-chip memory performance has been measured as 1.1 cycles/pixel.

9 Statistical Background Subtraction (32-Bit)

9.1 Introduction and Use Cases

In background subtraction, thresholding can be used to decide whether a pixel's observed value deviates too far from its model (that is, the average of its past values). Assuming each pixel's variance has been modeled, one might threshold a deviation image with a (scaled) variance image.

9.2 Specification

9.2.1 Function

This function implements a statistical background segmentation algorithm.

9.2.2 Inputs

`unsigned int`	`*mask32packed`	Binary mask to be computed	(32-bit packed)
`char`	`*newLuma`	Most recent luma buffer	(UQ8.0)
`int`	`*runningMean`	EW running mean buffer	(SQ8.23)
`int`	`*runningVar`	EW running variance buffer	(SQ16.15)
`int`	`thresholdGlobal`	Global threshold value	(SQ16.15)
`int`	`thresholdFactor`	Multiplicative factor for threshold	(SQ4.27)
`unsigned int`	`pixelCount`	Number of pixels to process	(UQ32.0)

9.2.3 Output

`int`	Returns VLIB Error Status

9.2.4 Method

For each pixel, the running mean and variance statistics are assumed to be known. The routine makes comparisons between three scalar values for each pixel:

1. The squared distance between the most recent luma measurement and the running mean as shown in Equation 8:

$$(\text{newLuma} - \text{runningMean})^2 \qquad (8)$$

2. The thresholdGlobal
3. The thresholdFactor × runningVar

For a pixel to be classified as foreground, (1) needs to be greater than both (2) and (3). When these conditions are satisfied, the observation is deemed to stem from a foreground object (and not from the modeled background), and the corresponding mask pixel value is set to 1.

The comparison with (2) plays the role of assuming a minimum variance for the pixel values, as in camera noise, etc. A sequence of luma observations might be very consistent, driving the running variance to small values. In such cases, camera noise could cause a pixel to pass the foreground threshold. By setting a reasonably high camera noise value (which is a “squared” scalar), one can filter out the camera noise.

The thresholdFactor is also in *squared* form: if you would like measurements which are two standard deviations away from the mean to be classified as foreground, the thresholdFactor should be set to 2×2=4. This variable is represented as SQ4.27 (sign bit, 4 integer bits, 27 fractional bits). In hex-format, it 4(dec) would read 0x20000000.

9.2.5 APIs

```
int VLIB_subtractBackgroundS32(
            unsigned int * restrict mask32packed,
            const char * restrict newLuma,
            const int * restrict runningMean,
            const int * restrict runningVar,
            const int thresholdGlobal,
            const int thresholdFactor,
            const unsigned int PixelCount);
```

9.2.6 Requirements

- I/O buffers are assumed to be double-word aligned in memory.
- pixelCount must be a multiple of 4.

9.3 Performance Benchmarks

On-chip memory performance has been measured as 2.3 cycles/pixel.

10 Mixture of Gaussians Background Modeling for Grayscale Video (16-Bit)

10.1 Introduction and Use Cases

In order to reliably obtain foreground blobs in complex, dynamic environments, it is often desirable to have an adaptive multi-modal background model. The Mixture of Gaussians background modeling and subtraction is a popular technique that provides such capabilities.

10.2 Specification

10.2.1 Function

Maintain a Gaussian mixture model (GMM) for each pixel in a video frame, and return a packed binary mask corresponding to the computed foreground regions for the input frame. This function assumes that the input stream contains a single channel (such as, luminance), and uses a maximum of 3 Gaussian components to model the pixel intensity variations.

10.2.2 Inputs

`char`	`*inputIm`	Input image buffer	(UQ8.0)
`unsigned short`	`*currentWts`	Buffer for current weights	(SQ0.15)
`unsigned short`	`*currentMeans`	Buffer for current means	(SQ8.7)
`unsigned short`	`*currentVars`	Buffer for current variances	(SQ12.3)
`char`	`*compIndex`	Buffer for indices indicating which mode a pixel belongs to	(UQ8.0)
`char`	`*intBuffer`	Buffer for internal use	(UQ8.0)
`unsigned int`	`*fgMask`	Computed binary foreground mask	(UQ8.0)
`int`	`imageSize`	Pixel count of input image buffer	(SQ32.0)
`unsigned short`	`updateRate1`	Update rate for weights	(SQ0.15)
`unsigned short`	`updateRate2`	Update rate for heights	(SQ0.15)
`unsigned short`	`mdThreshold`	Mahalanobis distance threshold	(SQ4.11)
`unsigned short`	`bsThreshold`	Background subtraction threshold	(SQ0.15)
`unsigned short`	`initialWt`	Initial weight for new component	(SQ0.15)
`unsigned short`	`initialVar`	Initial variance for new component	(SQ12.3)

10.2.2.1 Notes and Special Requirements

- If the input image contains N pixels, the input buffers should have the following sizes:
  - currentWts: 3.N data elements
  - currentMeans: 3.N data elements
  - CurrentVars: 3.N data elements
  - compIndex: N data elements
  - intBuffer: N data elements
  - fgMask: N/32 data elements
- All buffers should be initialized to 0 before invoking the function for the first time.
- I/O buffers are assumed to be double-word aligned in memory.

10.2.3 Output

`int`	Returns VLIB Error Status

10.2.4 APIs

```
int VLIB_mixtureOfGaussiansS16(
            const char* restrict inputIm,
            short* restrict currentWts,
            short* restrict currentMeans,
            short* restrict currentVars,
            char* restrict compIndex,
            char* restrict intBuffer,
            unsigned int* restrict fgMask,
            const int imageSize,
            const short updateRate1,
            const short updateRate2,
            const short mdThreshold,
            const short bsThreshold,
            const short initialWt,
            const short initialVar);
```

10.3 Performance Benchmarks

On-chip memory performance has been measured as 31.30 cycles/pixel.

10.4 References

1. *Adaptive background mixture models for real-time tracking*, C. Stauffer and W. Grimson, Computer Vision and Pattern Recognition, 1999.

11 Mixture of Gaussians Background Modeling for Grayscale Video (32-Bit)

11.1 Introduction and Use Cases

In order to reliably obtain foreground blobs in complex, dynamic environments, it is often desirable to have an adaptive multi-modal background model. The Mixture of Gaussians background modeling and subtraction is a popular technique that provides such capabilities.

11.2 Specification

11.2.1 Function

Maintain a Gaussian mixture model (GMM) for each pixel in a video frame, and return a packed binary mask corresponding to the computed foreground regions for the input frame. This function assumes that the input stream contains a single channel (such as, luminance), and uses a maximum of 3 Gaussian components to model the pixel intensity variations.

11.2.2 Inputs

`char`	`*inputIm`	Input image buffer	(UQ8.0)
`unsigned short`	`*currentWts`	Buffer for current weights	(SQ0.15)
`unsigned int`	`*currentMeans`	Buffer for current means	(SQ8.23)
`unsigned int`	`*currentVars`	Buffer for current variances	(SQ16.15)
`char`	`*compIndex`	Buffer for indices indicating which mode a pixel belongs to	(UQ8.0)
`char`	`*intBuffer`	Buffer for internal use	(UQ8.0)
`unsigned int`	`*fgmask`	Computed binary foreground mask	(UQ8.0)
`int`	`imageSize`	Pixel count of input image buffer	(SQ32.0)
`unsigned short`	`updateRate1`	Update rate for weights	(SQ0.15)
`unsigned int`	`updateRate2`	Update rate for heights	(SQ0.31)
`unsigned int`	`mdThreshold`	Mahalanobis distance threshold	(SQ4.27)
`unsigned short`	`bsThreshold`	Background subtraction threshold	(SQ0.15)
`unsigned short`	`initialWt`	Initial weight for new component	(SQ0.15)
`unsigned int`	`initialVar`	Initial variance for new component	(SQ16.15)

11.2.3 Notes and Special Requirements

- If the input image contains N pixels, the input buffers should have the following sizes:
  – currentWts: 3.N data elements
  – currentMeans: 3.N data elements
  – CurrentVars: 3.N data elements
  – compIndex: N data elements
  – intBuffer: N data elements
  – fdMask: N/32 data elements
- All buffers should be initialized to 0 before invoking the function for the first time.
- I/O buffers are assumed to be double-word aligned in memory.

11.2.4 Output

`int`	Returns VLIB Error Status

11.2.5 APIs

```
int VLIB_mixtureOfGaussiansS32(
            const char* restrict inputIm,
            short* restrict currentWts,
            int* restrict currentMeans,
            int* restrict currentVars,
            char* restrict compIndex,
            char* restrict intBuffer,
            unsigned int* restrict fgMask,
            const int imageSize,
            const short updateRate1,
            const int updateRate2,
            const int mdThreshold,
            const short bsThreshold,
            const short initialWt,
            const int initialVar);
```

11.3 Performance Benchmarks

On-chip memory performance has been measured as 39.13 cycles/pixel.

11.4 References

1. *Adaptive background mixture models for real-time tracking*, C. Stauffer and W. Grimson, Computer Vision and Pattern Recognition, 1999.

12 8-Bit Image Extraction From 16-Bit Background Models

12.1 Introduction and Use Cases

While a background model can contain fractional bits, you might be interested in processing or displaying only the integer portion of it. The following function is designed to help developers extract the 8 (unsigned) integer bits of a 16-bit (signed) background model. It can be applied to both running mean and variance images to extract the most significant 8 bits.

12.2 Specification

12.2.1 Inputs

`short`	`*BGmodel`	Background model	(SQa.b)
`unsigned char`	`*BGimage`	Extracted background image buffer	(UQ8.0)
`unsigned int`	`PixelCount`	Number of pixels to process	(UQ32.0)

12.2.2 Outputs

`int`	Returns VLIB Error Status

12.2.3 Method

This kernel extracts the 8-bit (unsigned) most significant integer portion of a 16-bit (signed) background model.

12.2.4 APIs

```
int VLIB_extract8bitBackgroundS16(
            const short * restrict BGmodel,
            unsigned char * restrict BGimage,
            const unsigned int pixelCount);
```

12.3 Requirements

- The buffers BGmodel and BGimage need to be double-word aligned in memory.
- The pixelCount must be a multiple of 8.

12.4 Performance Benchmarks

On-chip memory performance has been measured as 0.26 cycles/pixel.

13 32-Bit Packing and Unpacking of Binary Mask Images

13.1 Introduction and Use Cases

The background modeling and subtraction APIs of VLIB commonly operate on 32-bit packed binary mask images. The following functions are designed to help developers pack and unpack such masks efficiently.

13.2 Specification

13.2.1 Inputs

`unsigned int`	`*mask32packed`	32-bit packed binary mask buffer	(UQ32.0)
`unsigned char`	`*maskImage`	Unpacked binary mask image buffer	(UQ8.0)
`unsigned int`	`pixelCount`	Number of pixels to process	(UQ32.0)

13.2.2 Output

`int`	Returns VLIB Error Status

13.2.3 Method

These kernels convert binary images between the 32-bit packed and 8-bit unpacked formats.

13.2.4 APIs

```
int VLIB_packMask32(
            const unsigned char * restrict maskImage,
            unsigned int * restrict mask32packed,
            const unsigned int pixelCount);

int VLIB_unpackMask32(
            const unsigned int * restrict mask32packed,
            unsigned char * restrict maskImage,
            const unsigned int pixelCount);
```

13.3 Requirements

- The buffer maskImage need to be double-word aligned in memory.
- The pixelCount must be a multiple of 8.

13.4 Performance Benchmarks

On-chip memory performance for VLIB_packMask32 has been measured as 0.26 cycles/pixel.

On-chip memory performance for VLIB_unpackMask32 has been measured as 0.38 cycles/pixel.

14 Dilation

14.1 Introduction and Use Cases

Dilation, along with erosion, is an elementary morphological operation [1].

14.2 Specification

14.2.1 Function

By itself, dilation expands binary 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 and closing.

14.2.2 Inputs

`const unsigned char`	`*in_data`	Input binary image	(32-bit packed)
`unsigned char`	`*out_data`	Output binary image	(32-bit packed)
`const char`	`*mask`	3x3 filter mask(1)	
`int`	`cols`	Number of pixels to process	(in pixels)
`int`	`pitch`	Pitch of input image	(in pixels)

(1) Used in only one of the available versions of dilation.

14.2.3 Method

These functions use bit-packed binary images; that is, each pixel is represented by a bit. The results are calculated using the definition in Equation 9:

Dilation: out(u,v) = OR OR (in(u+i,v+j) AND mask(N-i,N-j)) (9)

In Equation 9, the *logical summation* OR is done over i=0,1,2 and j=0,1,2.

There are several important limitations to be aware of:

- I/O buffers are assumed to be double-word aligned and not aliased.
- The inputs `cols` and `pitch` must be multiples of 64.
- The bit-packed input and output are ordered the same way as pixels in the image. This is different from IMGLIB requirement for bit-reversed binary pixels within 32-bit words.
- If the data is a region of interest within a larger image, then `pitch` < `cols`.
- Border pixels will not contain valid data, in particular, the first and last row, as well as two rightmost columns of the output do not contain valid data.

14.2.4 APIs

```
int VLIB_dilate_bin_square(
            const unsigned char *restrict in_data,
            unsigned char *restrict out_data,
            int cols
            int pitch);

int VLIB_dilate_bin_cross(
            const unsigned char *restrict in_data,
            unsigned char *restrict out_data,
            int cols
            int pitch);

int VLIB_dilate_bin_mask(
            const unsigned char *restrict in_data,
            unsigned char *restrict out_data,
            const char *restrict mask,
            int cols
            int pitch);
```

14.3 Performance Benchmarks

The performance with all input and output data in on-chip memory is 0.27, 0.27, and 0.39 cycles per pixel, for square, cross, and mask versions of dilation, respectively.

14.4 Notes

Repeated application of dilation (resp. erosion) with a 3x3 structuring element can often be used to achieve dilations (resp. erosions) with larger structuring elements, depending on the shape and size of the large structuring element. In general, this can be achieved for odd structuring element sizes (5x5, 7x7, 9x9, …), and only if the structuring element is decomposable. In practice, repeated application of dilations (resp. erosions) with a 3x3 cross and/or a 3x3 square can be used as a substitute for dilation (resp. erosion) with commonly used large structuring elements.

If the large structuring element is decomposable or can be approximated by one that is decomposable, it is advantageous to use this approach to reduce processing time and memory consumption.

By combining these two 3x3 structuring elements a variety of larger structuring elements can be achieved:

```
     1 1 1
S3 = 1 1 1   (3x3 square)
     1 1 1
```

and

```
     0 1 0
C3 = 1 1 1   (3x3 cross)
     0 1 0
```

For example, an 11x11 structuring element that reasonably approximates a circle can be achieved by this combination (here we denote dilation by a “+” and erosion by a “–”):

```
K11 = S3 + S3 + C3 + C3 + C3 =

      0 0 0 1 1 1 1 1 0 0 0
      0 0 1 1 1 1 1 1 1 0 0
      0 1 1 1 1 1 1 1 1 1 0
      1 1 1 1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1 1 1 1
      1 1 1 1 1 1 1 1 1 1 1
      0 1 1 1 1 1 1 1 1 1 0
      0 0 1 1 1 1 1 1 1 0 0
      0 0 0 1 1 1 1 1 0 0 0
```

Similarly, if an 11x11 square is needed, this decomposition should be used:

```
S11 = S3 + S3 + S3 + S3 + S3
```

If an 11x11 diamond is desired, this decomposition is needed:

```
D11 = C3 + C3 + C3 + C3 + C3
```

Based on these decompositions and associativity and distributivity of dilation (resp. erosion), the larger dilation (resp. erosion) with K11 as an example, is implemented as follows:

```
A + K11     = A + (S3 + S3 + C3 + C3 + C3)
            = ((((A + S3) + S3) + C3) + C3) + C3
```

And

```
A - K11     =  A - (S3 + S3 + C3 + C3 + C3)
            = ((((A - S3) - S3) - C3) - C3) - C3
```

14.5 References

1. *Digital Image Processing* by R.C.Gonzales and R.E.Woods, Prentice-Hall, 2007.

15 Erosion

15.1 Introduction and Use Cases

Erosion, along with dilation, is an elementary morphological operation [1].

15.2 Specification

15.2.1 Function

By itself, erosion shrinks binary 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 and closing. VLIB_erode_bin_singlePixel erodes isolated pixels (ON pixels that do not have any ON neighbors).

15.2.2 Inputs

`const unsigned char`	`*in_data`	Input binary image	(32-bit packed)
`unsigned char`	`*out_data`	Output binary image	(32-bit packed)
`const char`	`*mask`	3x3 filter mask[(1)]	
`int`	`cols`	Number of pixels to process	(in pixels)
`int`	`pitch`	Pitch of input image	(in pixels)

(1) Used in only one of the available versions of erosion.

15.2.3 Method

These functions use bit-packed binary images; that is, each pixel is represented by a bit. The results are calculated using the definitions in Equation 10:

Erosion: out(u,v) = AND AND (in(u+i,v+j) AND mask(N-i,N-j)) (10)

In Equation 10, the *logical product* AND is done over i=0,1,2 and j=0,1,2.

There are several important limitations to be aware of:

- I/O buffers are assumed to be double-word aligned and not aliased.
- The inputs `cols` and `pitch` must be multiples of 64.
- The bit-packed input and output are ordered the same way as pixels in the image. This is different from IMGLIB requirement for bit-reversed binary pixels within 32-bit words.
- If the data is a region of interest within a larger image, then `pitch` < `cols`
- Border pixels will not contain valid data, in particular, the first and last row, as well as two rightmost columns of the output do not contain valid data.

15.2.4 APIs

```
void VLIB_erode_bin_square(
            const unsigned char *restrict in_data,
            unsigned char *restrict out_data,
            int cols
            int pitch);

void VLIB_erode_bin_cross(
            const unsigned char *restrict in_data,
            unsigned char *restrict out_data,
            int cols
            int pitch);

void VLIB_erode_bin_mask(
            const unsigned char *restrict in_data,
            unsigned char *restrict out_data,
            const char *restrict mask,
            int cols
            int pitch);

void VLIB_erode_bin_singlePixel(
            const unsigned char *restrict in_data,
            unsigned char *restrict  out_data,
            int cols,
            int pitch);
```

15.3 Performance Benchmarks

The performance with all input and output data in on-chip memory is 0.29, 0.29, 0.41, and 0.2 cycles per pixel, for square, cross, mask, and isolated pixel versions of erosion, respectively.

15.4 Notes

See Section 14.4 in the discussion on dilation regarding repeated application of a 3x3 erosion (resp. dilation) as a substitute for erosion (resp. dilation) with larger structuring elements.

15.5 References

1. *Digital Image Processing* by R.C.Gonzales and R.E.Woods, Prentice-Hall, 2007.

16 Connected Components Labeling

16.1 Introduction and Use Cases

Segmentation algorithms are often used to separate an image into salient (foreground) and non-salient (background) pixels; for example, VLIB_subtractBackgroundS16. These methods typically produce a binary image that identifies each pixel as belonging either to the foreground or background. The connected components labeling algorithm examines the binary image, groups foreground pixels that have other foreground pixels as 4- or 8-connected neighbors, and labels discrete groupings as components. Once accomplished, component properties can be measured and used to extract foreground information.

16.2 Specification

16.2.1 Function

The primary function for grouping and labeling foreground components or *blobs* in a binary image is VLIB_createConnectedComponentsList. After the handle is created and initialized by way of VLIB_initConnectedComponentsList, a 32-bit packed binary image should be supplied as input to the function such that each bit corresponds to a pixel location. For example, the most significant bit in the first 32-bit word represents the top-left corner of the binary image. By passing the handle to support functions, such as VLIB_GetCCFeatures, properties about the foreground regions in the image can be extracted.

The support function VLIB_createCCMap8Bit produces an 8-bit 2D map that labels every pixel in the image with its corresponding blob ID. Pixels associated with the background are all given ID = 0. Other support functions that extract blob information from the list are: VLIB_GetNumCCs and GetCCFeatures. The former returns the number of connected components in the list, while the latter reveals features of the component as defined by the follow structure:

```
typedef struct {
     int area;
     int xsum;
     int ysum;
     int xmin;
     int ymin;
     int xmax;
     int ymax;
     int seedx;
     int seedy;
} VLIB_CC;
```

The pixel defined by a component’s centroid is not guaranteed to be a member of the component. Thus, a guaranteed point in the connected component namely (seedx, seedy) is provided.

Additional features will be added to the structure as required. More support functions are also planned for future releases.

16.2.2 Inputs

VLIB_CCHandle	* handle	A pointer to the list handle, which is a private structure representing the labeled connected components in the binary image.	
unsigned short	inputwidth	Width of input image	(in pixels)
unsigned short	inputheight	Height of input image	(in pixels)
int	*inputImage	Input binary image mask(32-bit packed)	(SQ32.0)
void	*pBuffer	Pointer to large scratch buffer	
int	bytesBuffer	Number of bytes of scratch buffer	(SQ32.0)
int	minBlobArea	Minimum Pixel Area of each Blob	(SQ32.0)
int	connected8Flag	Set to 0 for 4 connected (no diagonal neighbors connected) or to 1 for 8 connected (all 8 pixel neighbors)	(SQ32.0)

16.2.3 Output

`int`	Returns VLIB Error Status

16.2.4 Implementation Notes

The amount of memory used by VLIB_createConnectedComponentsList depends on the binary image. To provide a buffer with sufficient size to accommodate any binary image, use the support function VLIB_calcConnectedComponentsMaxBufferSize to estimate the upper bound. The function returns the maximum required bytes to support the pathological arrangement of foreground pixels in the input image, which is generally very large.

When the binary image is preprocessed by morphological operations like erode or dilate that remove isolated pixels and small blobs, the actual upper bound needed is much smaller than the calculated maximum bytes, generally by a factor of 2 to 4, but perhaps even more. Because the amount suggested will generally require an *external* memory buffer to store the list of connected components, enabling the cache is highly recommended.

If the buffer is statically allocated only once, the initialization function VLIB_InitConnectedComponentsList only needs to be called once prior to calling VLIB_createConnectedComponentsList. However, if the allocated memory buffer address changes, that is dynamically allocated within an application, it must be called before each call to VLIB_createConnectedComponentsList. These functions are not re-entrant.

16.2.5 APIs

```
int VLIB_calcConnectedComponentsMaxBufferSize(
            unsigned short imgWidth,
            unsigned short imgHeight,
            int minBlobArea,
            int *maxBytesRequired);

int VLIB_initConnectedComponentsList(
            VLIB_CCHandle * handle,
            void * pBuffer,
            int bytesBuffer);

int VLIB_createConnectedComponentsList(
            VLIB_CCHandle * handle,
            unsigned short width,
            unsigned short rowsInImg,
            int * p32BitPackedFGMask,
            int minBlobArea,
            int connected8Flag);

int VLIB_getNumCCs(
            VLIB_CCHandle * handle,
            int * numCCs);

int VLIB_getCCFeatures(
            VLIB_CCHandle * handle,
            VLIB_CC * cc,
            short listIndex);

int VLIB_createCCMap8Bit(
            VLIB_CCHandle * restrict handle,
            unsigned char * restrict pOutMap,
            const unsigned short outCols,
            const unsigned short outRows);
```

When allocating memory for the handle to connected components, be sure to use VLIB_getSizeOfCCHandle(), which returns the size in bytes. For example,

```
Int sizeOfCCHandle =  VLIB_GetSizeOfCCHandle();
VLIB_CCHandle * handle = (VLIB_CCHandle *)
MEM_alloc(DDR2HEAP,sizeOfCCHandle,8);
```

16.3 Performance Benchmarks

VLIB_createConnectedComponentsList() and VLIB_createCCMap8Bit() are the only computation intensive APIs for connected components; the others simply make calls to internal structures. DSP performance is correlated with the relative size and number of connected components extracted from the 32-bit packed binary foreground mask. That is, larger and more numerous components will consume more DSP cycles and memory than smaller and fewer components.

Allocating buffers with memory sufficient to handle the worst case scenario given image resolution and size of components is recommended. This can be computed using VLIB_calcConnectedComponentsMaxBufferSize. VLIB_createConnectedComponentsList() performance ranges from 1.1 cycles per input pixel to 5.2 cycles/pixel; likewise, VLIB_createCCMap8Bit() ranges from 3.0 to 8.0 cycles/pixel. The algorithm is frame based and highly image dependent. The above performance estimates are average estimates for real use cases and worst case measurements may be much higher.

16.4 References

1. *Robot Vision*, Horn, MIT Press, 1986, pp. 69-71.

17 Canny Edge Detection

17.1 Introduction and Use Cases

Relative to many other edge detection methods, like Sobel and Robert's Cross, the Canny edge detector is generally regarded as the edge detector of choice because it provides robust edge detection and linking, even in noisy images.

17.2 Method

Canny edge detection produces clean, thin edges using these steps (algorithms):

- Gaussian image smoothing
- 2D gradient filtering
- Non-maximum suppression
- Hysteresis thresholding

VLIB provides these four optimized kernels so that integrators can quickly develop a Canny edge detector that is optimized for a specific platform and application[1]. A full description of the VLIB APIs for these kernels follows in Section 18 through Section 21. For a simple implementation using these component VLIB functions, please refer to the example code provided with this release (VLIB_testCannyEdgeDetector.c).

17.3 Performance Benchmarks

The overall DSP performance of Canny edge detection using VLIB kernels is largely dependent on the framework that feeds image data from one function to another. Integrators are encouraged to leverage fast L1D/L2D memory to improve the performance of VLIB kernels. Using sophisticated methods for data trafficking, including the EDMA3, multiple buffers, etc., is also necessary to achieve optimal performance. With the exception of Hysteresis thresholding, which generally requires a frame-based implementation, the other fundamental kernels in Canny can be implemented using efficient block-based frameworks.

As general guidance for framework design, the performance of a Canny edge detector using VLIB kernels is roughly 30 cycles per input pixel, depending on image content, image size, filter dimensions, and applied threshold levels; such as using 7x7 Gaussian filter, VGA resolution, thresholds that produce edge pixels in 5 – 10% of the input pixels, and at least 32kB on-chip memory.

17.4 References

1. *A Computational Approach to Edge Detection* by Canny, J., IEEE Trans. Pattern Analysis and Machine intelligence, 8:679-714, 1986.

18 Image Smoothing (for Canny Edge Detection)

18.1 Introduction and Use Cases

The first step in Canny edge detection attempts to smooth the image to remove noise and generate more reliable gradients. This 2D filter convolves a 7×7 kernel with 8-bit coefficients over 8-bit image (luma) pixels. This function can be used for Gaussian filtering when kernels approximate Gaussian coefficients. Note: This step can be implemented using convolution functions in IMGLIB2 such as IMG_conv_7x7_i8_c8s, IMG_conv_3x3_i8_c8s, etc. Refer to the IMGLIB2[1] documentation for APIs, assumptions, and benchmarks.

18.2 Specification

18.2.1 Function

Convolves input image with a smoothing kernel. Typically zero mean Gaussian.

18.2.2 Inputs

`char`	`*pInImg`	Pointer to input (luma image)
`char`	`*pOutImg`	Pointer to output (smoothed luma image)
`int`	`numPixels2Process`	Number of pixels to process
`short`	`imgWidth`	Width of image
`int8`	`p8bitMask`	Pointer to 7x7 coefficient mask
`short`	`shiftmask`	Number of bit-wise right shifts to apply to mask coefficients

18.2.3 Output

`int`	Returns VLIB Error Status

18.2.4 Method

To provide flexibility, a large 7×7 convolution filter that accepts user-specified filter coefficients is supported. Coefficients for a smaller Gaussian filter can also be used by padding the coefficients with zeros. When using this function for Canny edge detection, keep in mind that subsequent components expect a 7×7 smoothing filter to be used so applying smaller filters, such as IMG_conv_3x3_i8_c8s, will require careful adjustments to image/data pointers.

The convolution kernel accepts seven rows with imgWidth pixels for every row of imgWidth output pixels using the input mask of 7×7. This convolution operation performs a point by point multiplication of the 7×7 mask with the input image. The 49 multiplications are summed together to produce a 32-bit convolution intermediate sum. The user-defined shiftMask value is used to right-shift this convolution sum down to the byte range. The result, which is range limited between 0 to 255, is store in an output array pOutImg. The coefficients are provided as 8-bit signed values. The input image pixels are provided as 8-bit unsigned pixels and the output pixels will be in 8-bit unsigned.

18.3 References

1. http://focus.ti.com/docs/toolsw/folders/print/sprc264.html

19 2D Gradient Filtering (for Canny Edge Detection)

19.1 Introduction and Use Cases

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 [1].

19.2 Specification

19.2.1 Function

Extracts the 2D gradient vector coordinates as well as magnitude.

19.2.2 Inputs

`char`	`*pInBlk`	Pointer to input (smoothed luma image)
`short`	`*pBufGradX`	Pointer to output horizontal gradient
`short`	`*pBufGradY`	Pointer to output vertical gradient
`short`	`*pBufMag`	Pointer to output gradient magnitude
`unsigned short`	`width`	Width of image
`unsigned short`	`height`	Height of image

19.2.3 Output

`int`	Returns VLIB Error Status

19.2.4 Method

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 as shown in Equation 11 and Equation 12:

$$Gx = I(x+1,y) - I(x-1,y) \tag{11}$$

$$Gy = I(x,y+1) - I(x,y-1) \tag{12}$$

The gradient magnitude is approximated as shown in Equation 13:

$$Gmag = (|Gx| + |Gy|) \tag{13}$$

Gx, Gy and Gmag are all signed, 16-bit values.

19.2.5 APIs

```
int VLIB_xyGradientsAndMagnitude(
            unsigned char * restrict pInBlk,
            short * restrict pBufGradX,
            short * restrict pBufGradY,
            short * restrict pBufMag,
            unsigned short width,
            unsigned short height);
```

19.3 Assumptions

The 7×7 Gaussian filtering creates a 3-pixel border around the image that contains invalid data. In the interest of performance, the gradient filter processes these border pixels, but later stages will discount them appropriately. Additionally, calculating the 2D gradients vectors will require a 1-pixel border. So the gradient and magnitude outputs will have a 4-pixel border of invalid data. The gradient filter has no memory boundary alignment requirements.

19.4 Performance Benchmarks

DSP performance of this kernel running in L1/L2 memory is 0.8 cycles per input pixel.

19.5 References

1. *Mathematical Handbook for Scientists and Engineers: Definitions, Theorems, and Formulas for Reference and Review* by Korn, Theresa M. & Korn, Granino Arthur; New York: Dover Publications, pp. 157-160.

20 Non-Maximum Suppression (for Canny Edge Detection)

20.1 Introduction and Use Cases

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 [1].

20.2 Specification

20.2.1 Function

Creates an 8-bit edge map labeling each pixel location as a non-Edge (0) or possible-edge (127).

20.2.2 Inputs

`short`	`*pInMag`	Pointer to input (gradient magnitude)
`short`	`*pBufGradX`	Pointer to input horizontal gradient
`short`	`*pBufGradY`	Pointer to input horizontal gradient
`char`	`*pOutBlk`	Pointer to output gradient magnitude
`unsigned short`	`width`	Number of columns in image
`unsigned short`	`pitch`	Pitch of the input data
`unsigned short`	`height`	Number of rows in image

20.2.3 Output

`int`	Returns VLIB Error Status

20.2.4 Method

VLIB_nonMaximumSuppressionCanny creates an 8-bit edge map that labels each pixel either as a non-edge (0) or a possible-edge (127). 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.

20.2.5 APIs

```
int VLIB_nonMaximumSuppressionCanny(
            short * restrict pInMag,
            short * restrict pInGradX,
            short * restrict pInGradY,
            unsigned char * restrict pOutBlk,
            unsigned short width,
            unsigned short pitch,
            unsigned short height);
```

20.3 Assumptions

VLIB_nonMaximumSuppressionCanny uses a 3×3 kernel and operates on rows instead of pixels. The function accepts 3 rows of input (Gx, Gy and Gmag) for every single row of the edge map that is calculated. This function introduces another 1-pixel border of invalid data around the center-portion of the edge map. Before feeding the edge map into the next stage of Canny edge detection (Hysteresis Thresholding), the 5-pixel border of invalid data should be set as non-edges. However, the 5-pixel border at the top and bottom of the edge map should be handled manually. The input pointers should be the top left corner of the image where the processing starts. Take care in adjusting the pointers according to the filter used for convolution.

20.4 Performance Benchmarks

DSP performance of this kernel running in L1/L2 memory is 8.7 cycles per input pixel.

20.5 References

1. *A Computational Approach to Edge Detection* by Canny, J.; IEEE Trans. Pattern Analysis and Machine intelligence, 8:679-714, 1986.

21 Hysteresis Thresholding (for Canny Edge Detection)

21.1 Introduction and Use Cases

Hysteresis thresholding is the final stage within Canny edge detection [1]. With an edge map containing possible edges, hysteresis thresholding identifies and follows edges. Using both *High* and *Low* thresholds, it is able to maintain edge continuity by linking stronger edge segments that are connected to weaker segments. This stage is split into two functions VLIB_doublethresholding (block based) and VLIB_edgeRelaxation(Non block based).

21.2 Specification

21.2.1 Function

21.2.2 Inputs

`short`	`*pInMag`	Pointer to input (gradient magnitude)
`char`	`*edgeMap`	Pointer to edge (modified in place)
`unsigned int`	`strongEdgeListPtr`	Pointer to a buffer which holds locations of strong edges
`unsigned short`	`width`	Number of columns in image
`unsigned short`	`pitch`	Pitch of the input image
`unsigned short`	`height`	Number of rows in image
`unsigned short`	`loThresh`	Lower threshold
`unsigned short`	`hiThresh`	Higher threshold
`unsigned int`	`block_offset`	Relative offset of beginning of a block(when used in block-based mode)

21.2.3 Output

`int`	Returns VLIB Error Status

21.2.4 Method

VLIB_doublethresholding 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. VLIB_edgeRelaxation grows the edge segments by following a path of connected edges with magnitude values at or above the low threshold. Values in the edge map are modified from possible-edge (127) to edge (255) for line segments. The size of the strongEdgeListPtr is content dependent, but at its largest, should be large enough to store 32-bit representation for each edge pixel in the entire image.

21.2.5 APIs

```
int VLIB_doublethresholding(
            signed short * restrict pInMag,
            unsigned char *edgeMap,
            unsigned int * restrict strongEdgeListPtr,
            int * numStrongEdges,
            unsigned short width,
            unsigned short pitch,
            unsigned short height,
            unsigned char loThresh,
            unsigned char hiThresh,
            unsigned int block_offset);

int VLIB_edgeRelaxation(
            unsigned char *edgeMap,
            unsigned int * restrict strongEdgeListPtr,
            int * numStrongEdges,
            unsigned short width);
```

21.3 Assumptions

If an edge map is desired that only consists of non-edges (0) and edges (255), it will be necessary to remove the remaining possible-edges (127) after VLIB_edgeRelaxation completes. Edge linking is image content dependent. VLIB_edgeRelaxation is generally frame-based, so it can be difficult to partition this function into sub-image blocks, especially for large images. Use caution when locating the strongEdgeListPtr buffer in fast memory areas (L1D/L2D).

21.4 Performance Benchmarks

DSP performance of VLIB_doublethresholding kernel running in DDR2 memory is 3 cycles per input pixel.

The VLIB_edgeRelaxation kernel is frame-based and image dependent. Usually for natural images, DSP performance is less than 3 cycles per input pixel.

21.5 References

1. *A Computational Approach to Edge Detection* by Canny, J.; IEEE Trans. Pattern Analysis and Machine intelligence, 8:679-714, 1986.

22 Image Pyramid (8-Bit)

22.1 Introduction and Use Cases

Image pyramid is a data structure consisting of the original image at level 0, 2×2 sub-sampled image at Level 1, further 2×2 sub-sampled image at Level 2, and further 2×2 sub-sampled image at Level 3. It is commonly used in detection and tracking applications to reduce the amount of processing [1].

22.2 Specification

22.2.1 Function

Calculates Levels 1, 2, and 3 of an image pyramid for an 8-bit input image. The antialiasing filter used at each step is a 2×2 averaging.

22.2.2 Inputs

`char`	`*pIn`	8-bit input image	(UQ8.0)
`unsigned short`	`inCols`	Width of input image	(in pixels)
`unsigned short`	`inRows`	Height of input image	(in pixels)
`char`	`*pOut`	8-bit output data	(UQ8.0)

22.2.3 Output

`int`	Returns VLIB Error Status

22.2.4 Method

inCols must be a multiple of 8, while pIn and pOut must be 64-bit aligned.

- pIn is a pointer to an (inCols × inRows) array of unsigned char data.
- pOut is a pointer to an (inCols × inRows) × 21 ÷ 64 array of unsigned char data.

22.2.5 APIs

```
int VLIB_imagePyramid8(
            char * restrict pIn,
            unsigned short inCols,
            unsigned short inRows,
            unsigned int * restrict pOut);
```

22.3 Performance Benchmarks

The performance with all input and output data in on-chip memory is 0.97 cycles per output value.

22.4 References

1. http://web.mit.edu/persci/people/adelson/pub_pdfs/RCA84.pdf

23 Image Pyramid (16-Bit)

23.1 Introduction and Use Cases

Image pyramid is a data structure consisting of the original image at level 0, 2×2 sub-sampled image at Level 1, further 2×2 sub-sampled image at Level 2, and further 2×2 sub-sampled image at Level 3. It is commonly used in detection and tracking applications to reduce the amount of processing [1].

23.2 Specification

23.2.1 Function

Calculates Levels 1, 2, and 3 of an image pyramid for an 16-bit input image. The antialiasing filter used at each step is a 2×2 averaging.

23.2.2 Inputs

`unsigned short`	`*pIn`	16-bit input image	(UQ16.0)
`unsigned short`	`inCols`	Width of input image	(in pixels)
`unsigned short`	`inRows`	Height of input image	(in pixels)
`unsigned short`	`*pOut`	16-bit output data	(UQ16.0)

23.2.3 Output

`int`	Returns VLIB Error Status

23.2.4 Method

inCols must be a multiple of 8, while pIn and pOut must be 64-bit aligned.

- pIn is a pointer to an (inCols × inRows) array of unsigned char data.
- pOut is a pointer to an (inCols × inRows) × 21 ÷ 64 array of unsigned short data.

23.2.5 APIs

```
int VLIB_imagePyramid16(
            unsigned short * restrict pIn,
            unsigned short inCols,
            unsigned short inRows,
            unsigned short * restrict pOut);
```

23.3 Performance Benchmarks

The performance with all input and output data in on-chip memory is 2.4 cycles/output value.

23.4 References

1. http://web.mit.edu/persci/people/adelson/pub_pdfs/RCA84.pdf

24 Gaussian 5x5 Pyramid Kernel (8-Bit)

24.1 Introduction and Use Cases

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

24.2 Specification

24.2.1 Function

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 antialiasing filter used at each step is a binomial approximation to the 5x5 Gaussian filter given by:

```
 1     4     6     4     1
 4    16    24    16     4
 6    24    36    24     6    /  256
 4    16    24    16     4
 1     4     6     4     1
```

24.2.2 Inputs

`char`	`*restrict pIn`	5 x width input array	(UQ8.0)
`unsigned int`	`*restrict pB`	5 x (width-4) temporary array	(UQ16.0)
`unsigned short`	`cols`	cols = W-4; must be divisible by 8	(UQ16.0)
`unsigned short`	`pitch`	Pitch of the input data	(UQ16.0)
`unsigned short`	`rows`	rows = H; height of the input data; must be >4	(UQ16.0)
`char`	`*restrict pOut`	1 x (width-4)/2 output	(UQ8.0)

24.2.3 Output

`int`	Returns VLIB Error Status

24.2.4 Method

The value of cols = W-4 must be a multiple of 8, rows = H (height of the input data) must be > 4; while pIn, pB, and pOut must be 64-bit aligned.

24.2.5 APIs

```
int VLIB_gauss5x5PyramidKernel_8(
            unsigned char  *restrict pIn,
            unsigned short *restrict pB,
            unsigned short  cols,
            unsigned short  pitch,
            unsigned short  rows,
            unsigned char  *restrict pOut);
```

24.3 Performance Benchmarks

The compute-only performance with all buffers in L1 is 4.9 cycles per output value.

24.4 References

1. http://web.mit.edu/persci/people/adelson/pub_pdfs/RCA84.pdf

25 Gaussian 5x5 Pyramid Kernel (16-Bit)

25.1 Introduction and Use Cases

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

25.2 Specification

25.2.1 Function

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 antialiasing filter used at each step is a binomial approximation to the 5x5 Gaussian filter given by the following:

```
1    4    6    4    1
4   16   24   16    4
6   24   36   24    6    /  256
4   16   24   16    4
1    4    6    4    1
```

25.2.2 Inputs

unsigned short	*restrict	pIn	5 x width input array	(UQ16.0)
unsigned int	*restrict	pB	5 x (width-4) temporary array	(UQ32.0)
unsigned short	cols		cols = W-4; must be divisible by 8	(UQ16.0)
unsigned short	pitch		Pitch of the input data	(UQ16.0)
unsigned short	rows		rows = H; height of the input data; must be >4	(UQ16.0)
unsigned short	*restrict	pOut	1 x (width-4)/2 output	(UQ16.0)

25.2.3 Output

int	Returns VLIB Error Status

25.2.4 Method

The value of cols = W-4 must be a multiple of 8, rows = H (height of the input data) must be > 4; while pIn, pB, and pOut must be 64-bit aligned.

25.2.5 APIs

```
int VLIB_gauss5x5PyramidKernel_16(
            unsigned short *restrict pIn,
            unsigned int   *restrict pB,
            unsigned short  cols,
            unsigned short  pitch,
            unsigned short  rows,
            unsigned short *restrict pOut);
```

25.3 Performance Benchmarks

The compute-only performance with all buffers in L1 is 5.8 cycles per output value.

25.4 References

1. http://web.mit.edu/persci/people/adelson/pub_pdfs/RCA84.pdf

26 Gradient 5x5 Pyramid Kernel (8-Bit)

26.1 Introduction and Use Cases

Gradient image pyramid is a data structure 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 commonly used in detection and tracking, as well as in image fusion applications, in order to reduce the amount of processing [1].

26.2 Specification

26.2.1 Function

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:

```
         -1    -2     0     2     1
         -4    -8     0     8     4
H5 =     -6   -12     0    12     6   (horizontal)
         -4    -8     0     8     4
         -1    -2     0     2     1

         -1    -4    -6    -4    -1
         -2    -8   -12    -8    -2
V5 =      0     0     0     0     0   (vertical)
          2     8    12     8     2
          1     4     6     4     1
```

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 Equation 14 and Equation 15:

Gh = ((conv2(A,H5) + 64) >> 7) + 128; (14)

Gv = ((conv2(A,V5) + 64) >> 7) + 128; (15)

26.2.2 Inputs

`char`	`*restrict pIn`	5 x width input array	(UQ8.0)
`unsigned short`	`*restrict pB`	5 x (width-4) temporary array	(UQ16.0)
`unsigned short`	`cols`	cols = W-4; must be divisible by 8	(UQ16.0)
`unsigned short`	`pitch`	Pitch of the input data	(UQ16.0)
`unsigned short`	`rows`	rows = H; height of the input data; must be >4	(UQ16.0)
`char`	`*restrict pOut`	1 x (width-4)/2 output	(UQ8.0)

26.2.3 Output

`int`	Returns VLIB Error Status

26.2.4 Method

The value of cols = W-4 must be a multiple of 8, rows = H (height of the input data) must be > 4; while pIn, pB, and pOut must be 64-bit aligned.

26.2.5 APIs

```
int VLIB_gradientH5x5PyramidKernel_8(
            unsigned char  *restrict pIn,
            unsigned short *restrict pB,
            unsigned short  cols,
            unsigned short  pitch,
            unsigned short  rows,
            unsigned char  *restrict pOut);

int VLIB_gradientV5x5PyramidKernel_8(
            unsigned char  *restrict pIn,
            unsigned short *restrict pB,
            unsigned short  cols,
            unsigned short  pitch,
            unsigned short  rows,
            unsigned char  *restrict pOut);
```

26.3 Performance Benchmarks

The compute-only performance in L1 is:

Horizontal	7.3 cycles per output value
Vertical	9.7 cycles per output value

26.4 References

1. "Enhanced image capture through fusion" from *Proceedings of 4th International Conference on Computer Vision*by Burt, P.J. and Kolczynski, R.J., 1993.

27 Recursive IIR Filter: Horizontal, First-Order

27.1 Introduction and Use Cases

A variety of image processing algorithms can be implemented through recursive IIR filters, including smoothing and gradient/edge computations. These methods can be preferred over FIR (convolutional) filters for their computational efficiency.

27.2 Specification

27.2.1 Function

This function implements the 1st order horizontal IIR filter.

27.2.2 Inputs

char	*out	Filter output image	(UQ8.0)
char	*in	Input luma image	(UQ8.0)
int	width	Image width	(SQ31.0)
int	height	Image height	(SQ31.0)
short	weight	Filter coefficient	(SQ15.0)
char	*boundaryLeft	Array of left-boundary values	(UQ8.0)
char	*boundaryRight	Array of right-boundary values	(UQ8.0)
char	*buffer	Scratch buffer	(UQ8.0)

27.2.3 Output

int	Returns VLIB Error Status

27.2.4 Method

For each pixel, computes using Equation 16:

$$output(x,y) = 0.5 \times (output\_LR(x,y) + output\_RL(x,y)) \quad (16)$$

In Equation 16, output_LR is the causal filter component, processing pixels from left to right, and output_RL is the anti-causal component, processing pixels right to left. These are defined as in Equation 17 and Equation 18:

$$output\_LR(x,y) = weight \times input(x,y) + (1-weight) \times output\_LR(x-1, y) \quad (17)$$

$$output\_RL(x,y) = weight \times input(x,y) + (1-weight) \times output\_RL(x+1, y) \quad (18)$$

While the intermediate IIR results are computed at 16-bit precision, the output is cast to 8-bits. The left- and right-boundary values can be passed via array pointers boundaryLeft and boundaryRight. If these pointers are NULL, boundary image pixel values will be used as initial conditions.

27.2.5 APIs

```
int VLIB_recursiveFilterHoriz1stOrder(
            char *out,
            const char *in,
            const int width,
            const int height,
            const short weight,
            const char *boundaryLeft,
            const char *boundaryRight,
            char *buffer);
```

27.3 Performance Benchmarks

On-chip memory performance has been measured as 3.9 cycles/pixel.

27.4 Notes

- The scratch buffer must be at least 4×width bytes.
- The image width and height needs to be a multiple of 4.
- The input and output image buffers need to be double-word aligned.

27.5 References

1. *Fast Algorithms for Low-Level Vision* by R. Deriche, PAMI (12), 1, 1990

28 Recursive IIR Filter: Horizontal, First-Order (16 Bit)

28.1 Introduction and Use Cases

A variety of image processing algorithms can be implemented through recursive IIR filters, including smoothing and gradient/edge computations. These methods can be preferred over FIR (convolutional) filters for their computational efficiency.

28.2 Specification

28.2.1 Function

This function implements the (signed) 16-bit 1st order horizontal IIR filter.

28.2.2 Inputs

short	*out	Filter output image	(SQa.b)
short	*in	Input luma image	(SQa.b)
int	width	Image width	(SQ31.0)
int	height	Image height	(SQ31.0)
short	weight	Filter coefficient	(SQ15.0)
short	*boundaryLeft	Array of left-boundary values	(SQa.b)
short	*boundaryRight	Array of right-boundary values	(SQa.b)
short	*buffer	Scratch buffer	(SQa.b)

28.2.3 Output

int	Returns VLIB Error Status

28.2.4 Method

For each pixel, computes using Equation 19:

$$\text{output}(x,y) = 0.5\times (\text{output\_LR}(x,y) + \text{output\_RL}(x,y)) \quad (19)$$

In Equation 19, output_LR is the causal filter component, processing pixels from left to right, and output_RL is the anti-causal component, processing pixels right to left. These are defined as in Equation 20 and Equation 21:

$$\text{output\_LR}(x,y) = \text{weight} \times \text{input}(x,y) + (1-\text{weight}) \times \text{output\_LR}(x-1, y) \quad (20)$$

$$\text{output\_RL}(x,y) = \text{weight} \times \text{input}(x,y) + (1-\text{weight}) \times \text{output\_RL}(x+1, y) \quad (21)$$

Just like the input and output, the intermediate IIR results are computed at 16-bit precision. The left- and right-boundary values can be passed via array pointers boundaryLeft and boundaryRight. If these pointers are NULL, boundary image pixel values will be used as initial conditions.

28.2.5 APIs

```
int VLIB_recursiveFilterHoriz1stOrderS16(
            short *out,
            const short *in,
            const int width,
            const int height,
            const short weight,
            const short *boundaryLeft,
            const short *boundaryRight,
            short *buffer);
```

28.3 Performance Benchmarks

On-chip memory performance has been measured as 3.7 cycles/pixel.

28.4 Notes

- The scratch buffer must be at least 8×width bytes.
- The image width and height need to be a multiple of 4.
- The input and output image buffers need to be double-word aligned.

28.5 References

1. *Fast Algorithms for Low-Level Vision* by R. Deriche, PAMI (12), 1, 1990

29 Recursive IIR Filter: Vertical, First-Order

29.1 Introduction and Use Cases

A variety of image processing algorithms can be implemented through recursive IIR filters, including smoothing and gradient/edge computations. These methods can be preferred over FIR (convolutional) filters for their computational efficiency.

29.2 Specification

29.2.1 Function

This function implements the 1st order vertical IIR filter.

29.2.2 Inputs

`char`	`*out`	Filter output image	(UQ8.0)
`char`	`*in`	Input luma image	(UQ8.0)
`int`	`width`	Image width	(SQ31.0)
`int`	`height`	Image height	(SQ31.0)
`short`	`weight`	Filter coefficient	(SQ15.0)
`char`	`*boundaryTop`	Array of top-boundary values	(UQ8.0)
`char`	`*boundaryBottom`	Array of bottom-boundary values	(UQ8.0)
`char`	`*buffer`	Scratch buffer	(UQ8.0)

29.2.3 Output

`int`	Returns VLIB Error Status

29.2.4 Method

For each pixel, computes using Equation 22:

$$\text{output}(x,y) = 0.5 \times (\text{output\_TB}(x,y) + \text{output\_BT}(x,y)) \tag{22}$$

In Equation 22, output_TB is the causal filter component, processing pixels from top to bottom, and output_BT is the anti-causal component, processing pixels bottom to top. These are defined as in Equation 23 and Equation 24:

$$\text{output\_TB}(x,y) = \text{weight} \times \text{input}(x,y) + (1\text{-weight}) \times \text{output\_TB}(x, y-1) \tag{23}$$

$$\text{output\_BT}(x,y) = \text{weight} \times \text{input}(x,y) + (1\text{-weight}) \times \text{output\_BT}(x, y+1) \tag{24}$$

While the intermediate IIR results are computed at 16-bit precision, the output is cast to 8-bits. The top- and bottom-boundary values can be passed via array pointers boundaryTop and boundaryBottom. If these pointers are NULL, boundary image pixel values will be used as initial conditions.

29.2.5 APIs

```
int VLIB_recursiveFilterVert1stOrder(
            char *out,
            const char *in,
            const int width,
            const int height,
            const short weight,
            const char *boundaryTop,
            const char *boundaryBottom,
            char *buffer);
```

29.3 Performance Benchmarks

On-chip memory performance has been measured as 2.9 cycles/pixel.

29.4 Notes

- The scratch buffer must be at least 4×height bytes.
- The image width and height needs to be a multiple of 4.
- The input and output image buffers need to be double-word aligned.

29.5 References

1. *Fast Algorithms for Low-Level Vision* by R. Deriche, PAMI (12), 1, 1990

30 Recursive IIR Filter: Vertical, First-Order (16-Bit)

30.1 Introduction and Use Cases

A variety of image processing algorithms can be implemented through recursive IIR filters, including smoothing and gradient/edge computations. These methods can be preferred over FIR (convolutional) filters for their computational efficiency.

30.2 Specification

30.2.1 Function

This function implements the (signed) 16-bit 1st order vertical IIR filter.

30.2.2 Inputs

`short`	`*out`	Filter output image	(SQa.b)
`short`	`*in`	Input luma image	(SQa.b)
`int`	`width`	Image width	(SQ31.0)
`int`	`height`	Image height	(SQ31.0)
`short`	`weight`	Filter coefficient	(SQ15.0)
`short`	`*boundaryTop`	Array of top-boundary values	(SQa.b)
`short`	`*boundaryBottom`	Array of bottom-boundary values	(SQa.b)
`short`	`*buffer`	Scratch buffer	(SQa.b)

30.2.3 Output

`int`	Returns VLIB Error Status

30.2.4 Method

For each pixel, computes using Equation 25:

$$output(x,y) = 0.5 \times (output\_TB(x,y) + output\_BT(x,y)) \quad (25)$$

In Equation 25, output_TB is the causal filter component, processing pixels from top to bottom, and output_BT is the anti-causal component, processing pixels bottom to top. These are defined as in Equation 26 and Equation 27:

$$output\_TB(x,y) = weight \times input(x,y) + (1-weight) \times output\_TB(x, y-1) \quad (26)$$

$$output\_BT(x,y) = weight \times input(x,y) + (1-weight) \times output\_BT(x, y+1) \quad (27)$$

Just like the input and output, the intermediate IIR results are computed at 16-bit precision. The top- and bottom-boundary values can be passed via array pointers boundaryTop and boundaryBottom. If these pointers are NULL, boundary image pixel values will be used as initial conditions.

30.2.5 APIs

```
int VLIB_recursiveFilterVert1stOrderS16(
            short *out,
            const short *in,
            const int width,
            const int height,
            const short weight,
            const short *boundaryTop,
            const short *boundaryBottom,
            short *buffer);
```

30.3 Performance Benchmarks

On-chip memory performance has been measured as 2.6 cycles/pixel.

30.4 Notes

- The scratch buffer must be at least 8×height bytes.
- The image width and height need to be a multiple of 4.
- The input and output image buffers need to be double-word aligned.

30.5 References

1. *Fast Algorithms for Low-Level Vision* by R. Deriche, PAMI (12), 1, 1990

31 Integral Image (8-Bit)

31.1 Introduction and Use Cases

Object classification may be done by calculating image features (such as moments and/or wavelets) on a region of interest and feeding them to a classifier (such as k-NN or SVM). Integral image is an important step in calculation of a common type of image features, over-complete Haar wavelets [2]. Integral image values may be used as features themselves.

31.2 Specification

31.2.1 Function

Calculates the Integral image of an 8-bit image.

31.2.2 Inputs

`char`	`*pIn`	8-bit input image	(UQ8.0)
`unsigned short`	`inCols`	Width of input image	(in pixels)
`unsigned short`	`inRows`	Height of input image	(in pixels)
`unsigned int`	`*pLastLine`	32-bit carry-over buffer	(UQ32.0)
`unsigned int`	`*pOut`	32-bit output data	(UQ32.0)

31.2.3 Output

`int`	Returns VLIB Error Status

31.2.4 Method

The arguments pIn, pOut, and pLastLine must be 64-bit aligned. For the fixed-width version the width is assumed to be 640 pixels.

- pIn is a pointer to an (inCols × inRows) array of unsigned char data.
- pLastLine is a pointer to an (inCols × 1) array of unsigned int data.
- pOut is a pointer to an (inCols × inRows) array of unsigned int data.

31.2.5 APIs

```
int VLIB_integralImage8(
            char * restrict pIn,
            unsigned short inCols,
            unsigned short inRows,
            unsigned int * restrict pLastLine,
            unsigned int * restrict pOut);
```

31.3 Performance Benchmarks

The performance with all input and output data in on-chip memory is 2.3 cycles/pixel.

31.4 References

1. *Rapid Object Detection Using a Boosted Cascade of Simple Features* by Viola, P.; Jones, M. TR2004-043 May 2004 http://www.merl.com/reports/docs/TR2004-043.pdf
2. *Integral Image Optimizations for Embedded Vision Applications* by B.Kisacanin, Proc. IEEE Southwest Symposium on Image Analysis and interpretation, 2008; http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4512315.

32 Integral Image (16-Bit)

32.1 Introduction and Use Cases

Object classification may be done by calculating image features (such as moments and/or wavelets) on a region of interest and feeding them to a classifier (such as k-NN or SVM). Integral image is an important step in calculation of a common type of image features, over-complete Haar wavelets [2]. Integral image values may be used as features themselves.

32.2 Specification

32.2.1 Function

Calculates the Integral image of a 16-bit image.

32.2.2 Inputs

`unsigned short`	`*pIn`	16-bit input image	(UQ16.0)
`unsigned short`	`inCols`	Width of input image	(in pixels)
`unsigned short`	`inRows`	Height of input image	(in pixels)
`unsigned int`	`*pLastLine`	32-bit carry-over buffer	(UQ32.0)
`unsigned int`	`*pOut`	32-bit output data	(UQ32.0)

32.2.3 Output

`int`	Returns VLIB Error Status

32.2.4 Method

The arguments pIn, pOut, and pLastLine must be 64-bit aligned. For the fixed-width version the width is assumed to be 640 pixels.

- pIn is a pointer to an (inCols × inRows) array of unsigned short data.
- pLastLine is a pointer to an (inCols × 1) array of unsigned int data.
- pOut is a pointer to an (inCols × inRows) array of unsigned int data.

32.2.5 APIs

```
int VLIB_integralImage16(
            unsigned short * restrict pIn,
            unsigned short inCols,
            unsigned short inRows,
            unsigned int * restrict pLastLine,
            unsigned int * restrict pOut);
```

32.3 Performance Benchmarks

The performance with all input and output data in on-chip memory is 2.7 cycles/pixel.

32.4 References

1. *Rapid Object Detection Using a Boosted Cascade of Simple Features* by Viola, P.; Jones, M. TR2004-043 May 2004 http://www.merl.com/reports/docs/TR2004-043.pdf
2. *Integral Image Optimizations for Embedded Vision Applications* by B.Kisacanin, Proc. IEEE Southwest Symposium on Image Analysis and interpretation, 2008; http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4512315.

33 Hough Transform for Lines

33.1 Introduction and Use Cases

Hough transform for lines is commonly used after edge detection to determine the most dominant lines in the edge image.

33.2 Specification

33.2.1 Function

Calculates the Hough space values from the list of edge points.

33.2.2 Inputs

`unsigned short`	`* pEdgeMapList`	Points to a list of 2xlistSize values of type unsigned short which represent x and y values of edge points	(UQ16.0)
`unsigned short`	`* pOutHoughSpace`	Points to the Hough space	(UQ16.0)
`unsigned short`	`outBlkWidth`	Width of the original image	(UQ16.0)
`unsigned short`	`outBlkHeight`	Height of the original image	(UQ16.0)
`unsigned int`	`listSize`		(UQ32.0)
`unsigned short`	`thetaRange`		(UQ16.0)
`unsigned short`	`rhoMaxLength`		(UQ16.0)
`short`	`*pSIN`	Sine lookup tables	(SQ16.0)
`short`	`*pCOS`	Cosine lookup tables	(SQ16.0)
`unsigned short`	`ping`	Array of rhoMaxLength elements	(UQ16.0)
`unsigned short`	`pong`	Array of rhoMaxLength elements	(UQ16.0)
`unsigned short`	`pang`	Array of rhoMaxLength elements	(UQ16.0)
`unsigned short`	`peng`	Array of rhoMaxLength elements	(UQ16.0)

33.2.3 Output

```
unsigned short   maxHoughSpaceValue
```

33.2.4 Method

For each edge point (specified by the x and y coordinates) and for each angle theta, rho is calculated by Equation 28:

rho = x cos(theta) + y sin(theta) (28)

The corresponding value in the Hough space, located at (rho, theta), is incremented.

33.2.5 APIs

```
int VLIB_houghLineFromList(
            unsigned short * restrict pEdgeMapList,
            unsigned short * restrict pOutHoughSpace,
            unsigned short outBlkWidth,
            unsigned short outBlkHeight,
            unsigned int listSize,
            unsigned short thetaRange,
            unsigned short rhoMaxLength,
            const short *pSIN,
            const short *pCOS,
            unsigned short * restrict ping,
            unsigned short * restrict pong,
            unsigned short * restrict pang,
            unsigned short * restrict peng);
```

33.3 Performance Benchmarks

The full benefit of optimized code can be achieved if the data is not partitioned into small buffers and if at least ping, pong, pang, and peng buffers are in internal memory. The performance of 777 cycles per edge point (or 39 cycles per pixel, assuming 5% of pixels are edge points) has been achieved, with input and output data in external memory and ping, pong, pang, and peng buffers in internal memory. The number of edge points in this measurement was 3840 (5% of 320x240 image), while the size of the Hough Space in this measurement was 267x267.

33.4 Notes

- pEdgeMapList points to a list of 2xlistSize values of type unsigned short, which represent x and y values of edge points: x1,y1,x2,y2,… While it should be located in the fastest memory available, its role is cache friendly so it can be stored in the external memory.
- pOutHoughSpace points to the Hough space, which is a thetaRange×rhoMaxLength array of unsigned short. While it should be located in the fastest memory available, its role is cache friendly so it can be stored in the external memory.
- outBlkWidth and outBlkHeight represent width and height of the original image
- pSIN and pCOS are lookup tables for sine and cosine and can be generated during initialization. While it should be located in the fastest memory available, it’s role is cache friendly so it can be stored in the external memory.
- ping, pong, pang, and peng are arrays of rhoMaxLength elements of type unsigned short. These arrays should be stored in the fastest available memory.
- The function is written so that the list of edge points can be broken into sublists and the function called on them separately. This is useful if the list needs to be in the fast memory, but is too big to fit there. In that case, the Hough space should be cleared only before the call on the first sublist.

34 Harris Corner Score

34.1 Introduction and Use Cases

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.

34.2 Specification

34.2.1 Function

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.

34.2.2 Inputs

`short`	`*gradX`	Horizontal gradient of the input luma image	(SQ15.0)
`short`	`*gradY`	Vertical gradient of the input luma image	(SQ15.0)
`int`	`width`	Image width	(SQ31.0)
`int`	`height`	Image height	(SQ31.0)
`short`	`*HarrisScore`	Harris (cornerness) score	(SQ5.10)
`short`	`k`	Sensitivity parameter	(SQ0.15)
`char`	`*buffer`	Scratch buffer	(UQ8.0)

34.2.3 Output

`int`	Returns VLIB Error Status

34.2.4 Method

For each pixel, Equation 29, Equation 30 and Equation 31 together compute the 2×2 gradient covariance matrix M, where the summations are over 7×7 pixel neighborhoods:

$$M(1,1) = \mathrm{sum}(gradX)^2 \quad (29)$$

$$M(1,2) = M(2,1) = \mathrm{sum}(gradX \times gradY) \quad (30)$$

$$M(2,2) = \mathrm{sum}(gradY)^2 \quad (31)$$

The cornerness score is defined as in Equation 32, where k is a parameter, typically around 0.04. An approximation of the binary log of this value is stored in the output.

$$\det(M) - k \times \mathrm{trace}(M)^2 \quad (32)$$

34.2.5 APIs

```
int VLIB_harrisScore_7x7(
            const short * restrict gradX,
            const short * restrict gradY,
            int width,
            int height,
            short * restrict harrisscore,
            short k,
            char * buffer);
```

34.3 Performance Benchmarks

On-chip memory performance has been measured as 18.7 cycles/pixel.

34.4 Notes

- Garbage may be written in the output margins, which are 3 pixels wide on each side. If the input gradient also has a margin of 1 pixel, then there is an overall output margin of 4 pixels.
- This method uses a scratch buffer which must be at least 96**width* bytes.

34.5 References

1. http://www.csse.uwa.edu.au/~pk/Research/MatlabFns/Spatial/Docs/Harris/index.html

35 Non-Maximal Suppression

35.1 Introduction and Use Cases

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.

35.2 Specification

35.2.1 Function

35.2.2 Inputs

`short`	`*im`	Input image	(SQ15.0)
`int`	`width`	Image width	(SQ31.0)
`int`	`height`	Image height	(SQ31.0)
`short`	`thresh`	Minimum threshold for peaks	(SQ15.0)
`char`	`*out`	Binary output indicating peaks	(UQ8.0)

35.2.3 Output

`int`	Returns VLIB Error Status

35.2.4 Method

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.

35.2.5 APIs

There are three versions this function, defined for neighborhood sizes of 3x3, 5x5, and 7x7 pixels. All operate on 16-bit signed input data.

```
int VLIB_nonMaxSuppress_3x3_S16(
            const short * restrict im,
            int width,
            int height,
            short thresh,
            char * restrict out);

int VLIB_nonMaxSuppress_5x5_S16(
            const short * restrict im,
            int width,
            int height,
            short thresh,
            char * restrict out);

int VLIB_nonMaxSuppress_7x7_S16(
            const short * restrict im,
            int width,
            int height,
            short thresh,
            char * restrict out);
```

35.3 Performance Benchmarks

On-chip memory performance of the kernels has been measured as:

VLIB_nonMaxSuppress_3x3_16s	1.1 cycles/pixel
VLIB_nonMaxSuppress_5x5_16s	1.4 cycles/pixel
VLIB_nonMaxSuppress_7x7_16s	2.2 cycles/pixel

36 Lucas-Kanade Feature Tracking (Sparse Optical Flow)

36.1 Introduction and Use Cases

Tracks a set of feature points using the Lucas-Kanade method.

36.2 Specification

36.2.1 Function

The input parameters x and y correspond to pixel locations in the input image im1. Patches of 7x7 pixels centered around these points are tracked in the next frame.

The pointers outx and outy are expected to contain initial estimates of the feature location in im2. They are overwritten with the refined values after max_iters iterations. This is so that this function can be used in a coarse-to-fine strategy with image pyramids. Otherwise, the initial estimates should typically be equal to the locations in the first image.

36.2.2 Inputs

`char`	`*im1`	Input Luma image 1	(UQ8.0)
`char`	`*im2`	Input Luma image 2	(UQ8.0)
`short`	`*gradX`	X gradient of im1	(SQ15.0)
`short`	`*gradY`	Y gradient of im1	(SQ15.0)
`int`	`width`	Image width	(SQ31.0)
`int`	`height`	Image height	(SQ31.0)
`int`	`nfeatures`	Number of features	(SQ31.0)
`short`	`*x`	X feature coordinates in im1	(SQ11.4)
`short`	`*y`	Y feature coordinates in im1	(SQ11.4)
`short`	`*outx`	X feature coordinates in im2	(SQ11.4)
`short`	`*outy`	Y feature coordinates in im2	(SQ11.4)
`int`	`iters`	Number of iterations	(SQ31.0)
`char`	`*scratch`	Scratch memory	(UQ8.0)

36.2.3 Output

`int`	Returns VLIB Error Status

36.2.4 Method

This function considers a 7x7 patch centered about the feature coordinate. Bilinear sampling is used so that the tracked feature coordinates have sub-pixel accuracy.

The number of iterations is typically between 6 and 10.

36.2.5 APIs

```
int VLIB_trackFeaturesLucasKanade_7x7(
            const char * restrict im1,
            const char * restrict im2,
            const short * restrict gradX,
            const short * restrict gradY,
            int width,
            int height,
            int nfeatures,
            short * restrict x,
            short * restrict y,
            short * outx,
            short * outy,
            int max_iters,
            const char * restrict scratch);
```

36.3 Performance Benchmarks

On-chip memory performance has been measured as:

- 423 cycles per feature for startup
- 120 cycles per iteration per feature

36.4 Notes

The input pointer scratch should be pointing at a memory buffer of 384 bytes, ideally located in on-chip memory.

36.5 References

1. "An Iterative Image Registration Technique with an Application to Stereo Vision" from *Proceedings of the 7th international Joint Conference on Artificial intelligence (IJCAI '81)* by B.D. Lucas and T. Kanade, April, 1981, pp. 674-679, http://www.ri.cmu.edu/pubs/pub_2548.html.
2. http://www.ri.cmu.edu/projects/project_515.html
3. http://www.ces.clemson.edu/~stb/klt/

37 Normal Flow (16-Bit)

37.1 Introduction and Use Cases

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.

37.2 Specification

37.2.1 Function

This function takes as input the x and y gradients, the gradient magnitude, and the pixel-wise image difference and computes the normal flow vectors in the x and y directions.

37.2.2 Inputs

short	*inDiff	Pointer to array containing image difference values	(SQ15.0)
short	*Emag	Pointer to array containing gradient magnitude values	(SQ15.0)
short	*Ex	Pointer to array containing x-direction gradient	(SQ15.0)
short	*Ey	Pointer to array containing y-direction gradient	(SQ15.0)
short	*Lut	Pointer to array (Look-Up Table) containing values for integer division.	(SQ0.15)
short	T	Threshold on gradient magnitude	(SQ15.0)
char	numPixels	Number of pixels to process	(SQ31.0)
short	*normalFlowX	Pointer to array to hold computed normal flow vectors	(SQ8.7)
short	*normalFlowY	Pointer to array to hold computed normal flow vectors	(SQ8.7)

37.2.3 Output

void

37.2.4 Notes

- The LUT (look-up table) array should hold values such that LUT[n] = X, where X is the value 1/n represented in SQ0.15 format.
- 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.
- Minimum number of pixels allowed is 20 (numPixels >= 20)
- Number of pixels to be processed should be a multiple of 4.
- All arrays are double word aligned.

37.2.5 APIs

```
void VLIB_normalFlow_16(
            short * imDiff,
            short * Emag,
            short * Ex,
            short * Ey,
            short * LUT,
            short T,
            int   numPixels,
            short * normalFlowU,
            short * normalFlowV);
```

37.3 Performance Benchmarks

The performance of the function was measured as 2.65 cycles / pixel.

38 Kalman Filter With 2-Dimension Observation and 4-Dimension State Vectors (16-Bit)

38.1 Introduction and Use Cases

The Kalman filter is an efficient recursive method to estimate the state of a process from partial observations. It is used in a wide variety of vision problems, such as object tracking, background estimation, etc.

38.2 Specification

38.2.1 Function

The Kalman filter is implemented as two separate functions, one for the time update (or prediction) and the other for the measurement update (or correction). This implementation assumes a pre-determined fixed dimension for the observation and state vectors. The observation vector should be of dimension 2×1, and the state vector should have dimension 4×1.

The state of the Kalman filter is defined using the following structure. The expected bit precision for each matrix is noted in the comments. The variable sD and mD represent the dimensionality of the state and measurement vectors and have values of 4 and 2 respectively.

```
typedef struct VLIB_kalmanFilter_2x4{
     short transition[sD*sD]; // SQ15.0, state transition matrix
     short errorCov[sD*sD]; // SQ13.2, a priori error covariance matrix
     short predictedErrorCov[sD*sD]; // SQ13.2, predicted error cov matrix
     short state[sD];    // SQ10.5, state of the process
     short predictedState[sD]; // SQ10.5, predicted state of the process
     short measurement[mD*sD]; // SQ15.0, measurement matrix
     short processNoiseCov[sD*sD]; // SQ13.2, process noise cov matrix
     short measurementNoiseCov[mD*mD]; // SQ15.0, measurement noise cov
     short kalmanGain[sD*mD]; // SQ0.15, Kalman gain
     short temp1[sD*sD];
     short temp2[sD*sD];
     short temp3[sD*sD];
} VLIB_kalmanFilter_2x4;
```

38.2.2 Inputs

The inputs to VLIB_kalmanFilter_2x4_Predict (prediction step) are:

`VLIB_kalmanFilter_2x4`	`*KF`	Pointer to struct VLIB_kalmanFilter_2x4

The inputs to VLIB_kalmanFilter_2x4_Correct (correction step) are:

`VLIB_kalmanFilter_2x4`	`*KF`	Pointer to struct VLIB_kalmanFilter_2x4	
`short`	`*Z`	Pointer to array (dimension 2x1) containing measurement	(SQ10.5)
`short`	`*Res`	Pointer to array to store the residual error	(SQ10.5)

38.2.3 Output

For VLIB_kalmanFilter_2x4_Predict:

`int`	Returns VLIB Error Status

For VLIB_kalmanFilter_2x4_Correct:

`int`	Returns VLIB Error Status

38.2.4 Notes

- All the matrices in the struct VLIB_kalmanFilter_2x4 should be initialized to 0.
- The structure should be word aligned.

38.2.5 APIs

```
void VLIB_kalmanFilter_2x4_Predict(
            VLIB_kalmanFilter_2x4 * KF);

void VLIB_kalmanFilter_2x4_Correct(
            VLIB_kalmanFilter_2x4 * KF,
            short * restrict Z,
            short * restrict Residual);
```

38.3 Performance Benchmarks

For VLIB_kalmanFilter_2x4_Predict:	Performance using on-chip memory was measured as 154 cycles.
For VLIB_kalmanFilter_2x4_Correct:	Performance using on-chip memory was measured as 327 cycles.

39 Kalman Filter With 4-Dimension Observation and 6-Dimension State Vectors (16-Bit)

39.1 Introduction and Use Cases

The Kalman filter is an efficient recursive method to estimate the state of a process from partial observations. It is used in a wide variety of vision problems, such as object tracking, background estimation, etc.

39.2 Specification

39.2.1 Function

The Kalman filter is implemented as two separate functions, one for the time update (or prediction) and the other for the measurement update (or correction). This implementation assumes a pre-determined fixed dimension for the observation and state vectors. The observation vector should be of dimension 4×1, and the state vector should have dimension 6×1.

The state of the Kalman filter is defined using the following structure (the expected bit precision for each matrix is noted in the comments). The variable sD and mD represent the dimensionality of the state and measurement vectors and have values of 6 and 4 respectively.

```
typedef struct VLIB_kalmanFilter{
     short transition[sD*sD]; // SQ13.2, state transition matrix
     short errorCov[sD*sD]; // SQ13.2, a priori error covariance matrix
     short predictedErrorCov[sD*sD]; // SQ13.2, predicted error cov matrix
     short state[sD];    // 16-bit, desired Q value, state of the process
     short predictedState[sD]; // desired Q value, predicted state
     short measurement[mD*sD]; // SQ15.0, measurement matrix
     short processNoiseCov[sD*sD]; // SQ13.2, process noise cov matrix
     short measurementNoiseCov[mD*mD]; // SQ15.0, measurement noise cov
     short kalmanGain[sD*mD]; // SQ0.15, Kalman gain
     short temp1[sD*sD];
     short temp2[sD*sD];
     short temp3[sD*sD];
     int   tempBuffers[mD*mD*2];
     int   scaleFactor;    // SQ31.0
} VLIB_kalmanFilter_4x6;
```

39.2.2 Inputs

The inputs to VLIB_kalmanFilter_4x6_Predict (prediction step) are:

`VLIB_kalmanFilter_4x6`	`*KF`	Pointer to struct VLIB_kalmanFilter_4x6	

The inputs to VLIB_kalmanFilter_4x6_Correct (correction step) are:

`VLIB_kalmanFilter_4x6`	`*KF`	Pointer to struct VLIB_kalmanFilter_4x6	
`short`	`*Z`	Pointer to array containing measurement	(User-defined)
`short`	`*Res`	Pointer to array to store the residual error	(User-defined)

39.2.3 Output

For VLIB_kalmanFilter_4x6_Predict:

`void`

For VLIB_kalmanFilter_4x6_Correct:

`void`

39.2.4 Notes

- All the matrices in the struct VLIB_kalmanFilter_4x6 should be initialized to 0.
- The structure should be word aligned.
- The element scaleFactor in the structure VLIB_kalmanFilter_4x6 scales the matrix M = (H*P1*H' + R) to ensure that its inverse does not overflow 32 bits. The scaling is done by right shifting each element of M by the quantity assigned to scaleFactor. The computed inverse is then scaled back to ensure the correct result, based on the identity inv(M) = inv(M/k)/k.

39.2.5 APIs

```
void VLIB_kalmanFilter_4x6_Predict(
            VLIB_kalmanFilter_4x6 * KF);

void VLIB_kalmanFilter_4x6_Correct(
            VLIB_kalmanFilter_4x6 * KF,
            short * restrict Z,
            short * restrict Residual);
```

39.3 Performance Benchmarks

For VLIB_kalmanFilter_2x4_Predict:	Performance using on-chip memory was measured as 374.2 cycles.
For VLIB_kalmanFilter_2x4_Correct:	Performance using on-chip memory was measured as 1627.5 cycles.

40 Nelder-Mead Simplex (16-Bit)

40.1 Introduction and Use Cases

Optimization techniques are important in several vision algorithms. The Nelder-Mead simplex method is a common optimization technique used to find the minima of a given cost function.

40.2 Specification

40.2.1 Function

This function accepts as input a pointer to the cost function to be minimized and an N-dimensional coordinate vector indicating the starting point of the search. The function returns the coordinates of the found minima and the actual minimum value.

40.2.2 Inputs

`int`	`*func`	Pointer to cost function.	
`short`	`*start`	Pointer to array containing starting coordinates	User-defined
`short`	`*init_step`	Pointer to array containing the size of the initial step to be taken in each dimension to form the initial simplex	User-defined
`int`	`N`	Dimensionality of the coordinate space	(SQ31.0)
`short`	`N_inv`	Value equal to the reciprocal of N	(SQ0.15)
`int`	`MaxIteration`	Maximum number of allowed iterations to find the minima	(SQ31.0)
`int`	`EPSILON`	Stopping criterion corresponding to a threshold on the difference between the largest and smallest values in the simplex at any iteration.	User-defined
`short`	`*v`	Pointer to array of size N+1. For internal use.	
`short`	`*f`	Pointer to array of size N+1. For internal use.	
`short`	`*vr`	Pointer to array of size N. For internal use.	
`short`	`*ve`	Pointer to array of size N. For internal use.	
`short`	`*vc`	Pointer to array of size N. For internal use.	
`short`	`*vm`	Pointer to array of size N. For internal use	
`void`	`*addtlArgs`	Pointer to structure containing additional arguments to cost function	
`short`	`*minPoint`	Pointer to array to hold the coordinates of the found minima	
`int`	`*minValue`	Pointer to variable to hold the minimum found value	

40.2.3 Output

`void`

40.2.4 Notes

- All arrays should be double word aligned.
- The stooping condition works as follows: If the difference between the largest and smallest values in the simplex at any iteration is smaller than EPSILON, the function terminates.
- It is assumed that the cost function will have a 32-bit return value, and, as input, it will take 16-bit representation of the coordinates.

40.2.5 APIs

```
void VLIB_simplex(
            int (*func)(short[], void *)
            short* restrict start,
            short* restrict init_step,
            int N,
            short N_INV,
            int MaxIteration,
            int EPSILON,
            short* restrict v,
            int* restrict f,
            short* restrict vr,
            short* restrict ve,
            short* restrict vc,
            short* restrict vm,
            void* addtlArgs,
            short* restrict minPoint,
            int* restrict minValue);
```

40.3 Performance Benchmarks

The performance of the function was measured as: 75.9 cycles to find the minima of Rosenbrock's function in 3D. The minimization involved 102 iterations and 177 evaluations of the cost function.

41 Nelder-Mead Simplex for 3D Coordinate Space (16-Bit)

41.1 Introduction and Use Cases

Optimization techniques are important in several vision algorithms. The Nelder-Mead simplex method is a common optimization technique used to find the minima of a given cost function.

41.2 Specification

41.2.1 Function

This function accepts as input a pointer to the cost function to be minimized and an 3-dimensional coordinate vector indicating the starting point of the search. The function returns the coordinates of the found minima and the actual minimum value.

41.2.2 Inputs

`int`	`*func`	Pointer to cost function.	
`short`	`*start`	Pointer to array containing starting coordinates	User-defined
`short`	`*init_step`	Pointer to array containing the size of the initial step to be taken in each dimension to form the initial simplex	User-defined
`int`	`MaxIteration`	Maximum number of allowed iterations to find the minima	(SQ31.0)
`int`	`EPSILON`	Stopping criterion corresponding to a threshold on the difference between the largest and smallest values in the simplex at any iteration.	User-defined
`short`	`*v`	Pointer to array of size N+1. For internal use.	
`short`	`*f`	Pointer to array of size N+1. For internal use.	
`short`	`*vr`	Pointer to array of size N. For internal use.	
`short`	`*ve`	Pointer to array of size N. For internal use.	
`short`	`*vc`	Pointer to array of size N. For internal use.	
`short`	`*vm`	Pointer to array of size N. For internal use	
`void`	`*addtlArgs`	Pointer to structure containing additional arguments to cost function	
`short`	`*minPoint`	Pointer to array to hold the coordinates of the found minima	
`int`	`*minValue`	Pointer to variable to hold the minimum found value	

41.2.3 Output

`void`

41.2.4 Notes

- All arrays should be double word aligned.
- The stooping condition works as follows: If the difference between the largest and smallest values in the simplex at any iteration is smaller than EPSILON, the function terminates.
- It is assumed that the cost function will have a 32-bit return value, and, as input, it will take 16-bit representation of the coordinates.

41.2.5 APIs

```
void VLIB_simplex_3D(
            int (*func)(short[], void *)
            short * restrict start,
            short * restrict init_step,
            int MaxIteration,
            int EPSILON,
            short * restrict v,
            int * restrict f,
            short * restrict vr,
            short * restrict ve,
            short * restrict vc,
            short * restrict vm,
            void * addtlArgs,
            short * restrict minPoint,
            int * restrict minValue);
```

41.3 Performance Benchmarks

The performance of the function was measured as: 40.2 cycles to find the minima of Rosenbrock's function in 3D. The minimization involved 102 iterations and 177 evaluations of the cost function.

42 Legendre Moments Computation (16-Bit)

42.1 Introduction and Use Cases

Legendre Moments are orthogonal moments often used for image analysis.

42.2 Specification

42.2.1 Function

The function returns a square matrix M of dimension (Order+1) where Order is the specified maximum order of moments required. Entries M(i,j) such that i+j < Order correspond to the required Legendre moments.

There are two functions related to Legendre Moments computation, VLIB_legendreMoments_Init and VLIB_legendreMoments. If the image size and the required moment order are fixed, VLIB_legendreMoments_Init can be called just once to initialize the necessary buffers and constants.

42.2.2 Inputs

The inputs for VLIB_legendreMomentsInit are:

`short`	`*LPoly`	Buffer to hold the computed Legendre polynomial values	(UQ0.15)
`int`	`Order`	Required order of moments	(SQ31.0)
`int`	`ImH`	Image height	(SQ31.0)
`short`	`*Constant`	Pointer to variable	SQ0.15)

The inputs for VLIB_legendreMoments are:

`short`	`*Im`	Input image patch	(UQ0.15)
`short`	`*Lmoments`	Buffer to hold the computed Legendre moments	(SQ0.15)
`short`	`*LPoly`	Buffer returned from call to VLIB_LegendreMoments_Init	(SQ0.15)
`int`	`Order`	Required order of moments	(SQ31.0)
`int`	`ImH`	Image height	(SQ31.0)
`short`	`Constant`	Constant value returned by VLIB_LegendreMoments_Init	SQ0.15)

42.2.3 Output

`int`	Returns VLIB Error Status

42.2.4 Notes

- The pixel intensities should be normalized to be in [0,1].
- The image should be square, image height = image width.
- The largest image supported is 256×256.
- The largest order of moments supported is 40.
- Lmoments should be initialized to 0
- LPoly is independent of the pixel intensities, and is dependent only on the size of the image (ImH) and the Order of the moment values required
- LPoly must be of dimension (Order+1)×(ImH)
- LMoments must be of dimension (Order+1)×(Order+1)

Example:

1. Initialize LPoly, LMoments to 0 before first call to VLIB_legendreMoments
2. For subsequent calls to VLIB_legendreMoments, reuse the values in the buffer LPoly set by the first call to VLIB_legendreMoments

42.2.5 APIs

```
int VLIB_legendreMoments_Init(
            short * LPoly,
            const char Order,
            const char ImH,
            short * Constant);

int VLIB_legendreMoments(
            const * restrict Im,
            short * restrict LMoments,
            short * restrict LPoly,
            const char Order,
            const char ImH,
            const short Constant);
```

42.3 Performance Benchmarks

For a 128x128 image patch and 20th order moments, the performance using on-chip memory has been measured as in Equation 33:

$$0.68\times(ImH^2)\times(Order^2) \quad (33)$$

43 Initialization for Histogram Computation for Integer Scalars (8-Bit)

43.1 Introduction and Use Cases

Initializes arrays for histogram computation.

43.2 Specification

43.2.1 Function

Initializes buffer for 1D histogram computation by VLIB_histogram_1D_U8 and VLIB_weightedHistogram_1D_U8.

43.2.2 Inputs

`char`	`*binEdges`	Array containing the edges of the histogram bins (must be monotonically increasing)	(UQ8.0)
`int`	`numB`	Number of bins	(SQ31.0)
`char`	`*internalBuffer`	Buffer for internal use	(UQ8.0)

43.2.3 Output

`int`	Returns VLIB Error Status

43.2.4 Notes

- The values of the bin edges must increase monotonically.
- internalBuffer should be initialized to 0.
- internalBuffer should have a size equal to the length of the range of values that the input quantity can take.

  R = (*max* – *min*) + 1, where max and min are the maximum and minimum possible values that the input quantity can have.

43.2.5 APIs

```
int VLIB_histogram_1D_Init_U8(
            char * restrict binEdges,
            const int numBins,
            char * restrict histArray);
```

43.3 Performance Benchmarks

On-chip memory performance of has been measured as in Equation 34, where R is the length of the range of the quantity to be histogrammed:

$$8.6 \times R \text{ cycles} \tag{34}$$

44 Histogram Computation for Integer Scalars (8-Bit)

44.1 Introduction and Use Cases

Histograms are used commonly as a discrete measure of the distribution of a given quantity.

44.2 Specification

44.2.1 Function

Computes histogram from array of 8-bit integers using user-specified bins.

44.2.2 Inputs

`char`	`*X`	Input array of scalar values	(UQ8.0)
`int`	`numX`	Number of elements in X	(SQ31.0)
`int`	`numB`	Number of bins	(SQ31.0)
`unsigned short`	`binWeight`	Value to accumulate in histogram bins	(UQ16.0)
`char`	`*histArray`	Array for internal use, initialized by VLIB_histogram_1D_Init_U8	(UQ8.0)
`unsigned short`	`*internalH1`	Array for internal use	(UQ16.0)
`unsigned short`	`*internalH2`	Array for internal use	(UQ16.0)
`unsigned short`	`*internalH3`	Array for internal use	(UQ16.0)
`unsigned short`	`*H`	Array to hold the computed histogram	(UQ16.0)

44.2.3 Output

`int`	Returns VLIB Error Status

44.2.4 Notes

- The values in binEdges must increase monotonically.
- H[k] will hold the number of elements that satisfy Equation 35:

binEdges[k] <= X[i] < binEdges[k+1] (35)

- The last bin H[end] will hold the number of elements that satisfy Equation 36:

X[i] == binEdges[end] (36)

- histArray should be initialized by calling VLIB_histogram_1D_Init_U8.
- H, internalH1, internalH2, and internalH3 should be of length numB, initialized to 0.
- numX should be a multiple of 4
- numB should be a multiple of 2

44.2.5 APIs

```
int VLIB_histogram_1D_U8(
            char* restrict X,
            const int numX,
            const int numBins,
            const unsigned short binWeight,
            char* restrict histArray,
            unsigned short* restrict internalH1,
            unsigned short* restrict internalH2,
            unsigned short* restrict internalH3,
            unsigned short* restrict H);
```

44.3 Performance Benchmarks

On-chip memory performance has been measured as Equation 37:

$$(2.25 \times \text{numX}) + (1 \times \text{numBins}) \text{ cycles} \tag{37}$$

45 Weighted Histogram Computation for Integer Scalars (8-Bit)

45.1 Introduction and Use Cases

Histograms are used commonly as a discrete measure of the distribution of the input data. Weighted histograms permit the user he flexibility to influence the relative importance of different values in the input data.

45.2 Specification

45.2.1 Function

Computes weighted histogram from array of 8-bit integers using user-specified bins.

45.2.2 Inputs

`char`	`*X`	Input array of scalar values	(UQ8.0)
`int`	`numX`	Number of elements in X	(SQ31.0)
`int`	`numB`	Number of bins	(SQ31.0)
`unsigned short`	`*binWeight`	Array of size numX of weight that each element contributes to the histogram	(UQ16.0)
`char`	`*histArray`	Array for internal use, initialized by VLIB_histogram_1D_Init_U16	(UQ8.0)
`unsigned short`	`*internalH1`	Array for internal use	(UQ16.0)
`unsigned short`	`*internalH2`	Array for internal use	(UQ16.0)
`unsigned short`	`*internalH3`	Array for internal use	(UQ16.0)
`unsigned short`	`*H`	Array to hold the computed histogram	(UQ16.0)

45.2.3 Output

`int`	Returns VLIB Error Status

45.2.4 Notes

- H[k] will hold the number of elements that satisfy Equation 38:

  binEdges[k] <= X[i] < binEdges[k+1] (38)

- The last bin H[end] will hold the number of elements that satisfy Equation 39:

  X[i] == binEdges[end] (39)

- histArray should be initialized by calling VLIB_histogram_1D_Init_U16.
- internalH1, internalH2, and internalH3 should be of length numB, initialized to 0.
- H should be of length numB, initialized to 0.
- numX should be a multiple of 4.
- numB should be a multiple of 2.

45.2.5 APIs

```
int VLIB_weightedHistogram_1D_U8(
            char* restrict X,
            const int numX,
            const int numBins,
            unsigned short* restrict binWeight,
            char* restrict histArray,
            unsigned short* restrict H1,
            unsigned short* restrict H2,
            unsigned short* restrict H3,
            unsigned short* restrict H);
```

45.3 Performance Benchmarks

On-chip memory performance has been measured as in Equation 40:

$$(2.5 \times \text{numX}) + (1 \times \text{numBins}) \text{ cycles} \quad (40)$$

46 Initialization for Histogram Computation for Integer Scalars (16-Bit)

46.1 Introduction and Use Cases

Initializes arrays for histogram computation.

46.2 Specification

46.2.1 Function

Initializes buffer for 1D histogram computation by VLIB_histogram_1D_U16 and VLIB_weightedHistogram_1D_U16.

46.2.2 Inputs

`unsigned short`	`*binEdges`	Array containing the edges of the histogram bins (must be monotonically increasing)	(UQ16.0)
`int`	`numB`	Number of bins	(SQ31.0)
`unsigned short`	`*internalBuffer`	Buffer for internal use	(UQ16.0)

46.2.3 Output

`int`	Returns VLIB Error Status

46.2.4 Notes

- The values of the bin edges must increase monotonically.
- internalBuffer should be initialized to 0.
- internalBuffer should have a size equal to the length of the range of values that the input quantity can take.

  R = (*max* – *min*) + 1, where max and min are the maximum and minimum possible values that the input quantity can have.

46.2.5 APIs

```
int VLIB_histogram_1D_Init_U16(
            unsigned short* restrict binEdges,
            const int numBins,
            unsigned short* restrict H);
```

46.3 Performance Benchmarkss

On-chip memory performance of has been measured as in Equation 41, where R is the length of the range of the quantity to be histogrammed:

$$8.6 \times R \text{ cycles} \tag{41}$$

47 Histogram Computation for Integer Scalars (16-Bit)

47.1 Introduction and Use Cases

Histograms are used commonly as a discrete measure of the distribution of a given quantity.

47.2 Specification

47.2.1 Function

Computes histogram from array of 16-bit integers using user-specified bins.

47.2.2 Inputs

`unsigned short`	`*X`	Input array of scalar	(UQ16.0)
`int`	`numX`	Number of elements in X	(SQ31.0)
`int`	`numB`	Number of bins	(SQ31.0)
`unsigned short`	`binWeight`	Value to accumulate in histogram bins	(UQ16.0)
`unsigned short`	`*histArray`	Array for internal use, initialized by VLIB_histogram_1D_Init_U16	(UQ16.0)
`unsigned short`	`*internalH`	Array for internal use	(UQ16.0)
`unsigned short`	`*H`	Array to hold the computed histogram	(UQ16.0)

47.2.3 Output

`int`	Returns VLIB Error Status

47.2.4 Notes

- H[k] will hold the number of elements that satisfy Equation 42:

binEdges[k] <= X[i] < binEdges[k+1] (42)

- The last bin H[end] will hold the number of elements that satisfy Equation 43:

X[i] == binEdges[end] (43)

- histArray should be initialized by calling VLIB_histogram_1D_Init_U16.
- internalH should be of length numB, initialized to 0.
- H should be of length numB, initialized to 0.

47.2.5 APIs

```
int VLIB_histogram_1D_U16(
            unsigned short* restrict X,
            const int numX,
            const int numBins,
            const unsigned short binWeight,
            unsigned short* restrict histArray,
            unsigned short* restrict internalH,
            unsigned short* restrict H);
```

47.3 Performance Benchmarks

On-chip memory performance has been measured as in Equation 44:

$$(3.6 \times \text{numX}) + (1 \times \text{numBins}) \text{ cycles} \quad (44)$$

48 Weighted Histogram Computation for Integer Scalars (16-Bit)

48.1 Introduction and Use Cases

Histograms are used commonly as a discrete measure of the distribution of the input data. Weighted histograms permit the user he flexibility to influence the relative importance of different values in the input data.

48.2 Specification

48.2.1 Function

Computes weighted histogram from array of 16-bit integers using user-specified bins.

48.2.2 Inputs

`unsigned short`	`*X`	Input array of scalar values	(UQ16.0)
`int`	`numX`	Number of elements in X	(SQ31.0)
`int`	`numB`	Number of bins	(SQ31.0)
`unsigned short`	`*binWeight`	Array of size numX of weight that each element contributes to the histogram	(UQ16.0)
`unsigned short`	`*histArray`	Array for internal use, initialized by VLIB_histogram_1D_Init_U16	(UQ16.0)
`unsigned short`	`*internalH`	Array for internal use	(UQ16.0)
`unsigned short`	`*H`	Array to hold the computed histogram	(UQ16.0)

48.2.3 Output

`int`	Returns VLIB Error Status

48.2.4 Notes

- H[k] will hold the number of elements that satisfy Equation 45:

binEdges[k] <= X[i] < binEdges[k+1] (45)

- The last bin H[end] will hold the number of elements that satisfy Equation 46:

X[i] == binEdges[end] (46)

- histArray should be initialized by calling VLIB_histogram_1D_Init_U16.
- internalH should be of length numB, initialized to 0.
- H should be of length numB, initialized to 0.

48.2.5 APIs

```
int VLIB_weightedHistogram_1D_U16(
            unsigned short* restrict X,
            const int numX,
            const int numBins,
            unsigned short* restrict binWeight,
            unsigned short* restrict histArray,
            unsigned short* restrict H1,
            unsigned short* restrict H);
```

48.3 Performance Benchmarks

On-chip memory performance has been measured as in Equation 47:

$$(3.6 \times numX) + (1 \times numBins) \text{ cycles} \quad (47)$$

49 Histogram Computation for Multi-Dimensional Vectors (16-Bit)

49.1 Introduction and Use Cases

Histograms are used commonly as a discrete measure of the distribution of a given quantity.

49.2 Specification

49.2.1 Function

Histogram computation for 16-bit vector valued variables of multiple dimensions.

49.2.2 Inputs

`unsigned short`	`*X`	Array of input values, data arranged in planar form	(UQ16.0)
`int`	`numX`	Number of individual vector elements in X	(SQ31.0)
`int`	`dimX`	Dimensionality of vectors in X	(SQ31.0)
`unsigned short`	`binWeight`	Value to accumulate in histogram bins	(UQ16.0)
`unsigned short`	`*numBins`	Array of size dimX, each element specifies the number of bins required in that dimension	(UQ16.0)
`unsigned short`	`*normVals`	Array of size dimX, each element containing the normalization factor for that dimension	(UQ0.16)
`unsigned short`	`*internal1`	Buffer of size numX, for internal use (initialized to 0)	(UQ16.0)
`unsigned short`	`*internal2`	Buffer of size equal to total number of bins, for internal use (initialized to 0)	(UQ16.0)
`unsigned short`	`*H`	Array of size equal to total number of bins to hold computed histogram (initialized to 0)	(UQ16.0)

49.2.3 Output

`int`	Returns VLIB Error Status

49.2.4 Notes

- The vectors in X should be arranged in planar form (see Example below).
- H should be initialized to 0.
- internal1 and internal2 should be initialized to 0.
- The normalization factor normVals[k] for each dimension k should be set as in Equation 48, where M is the maximum value in dimension k, and d is any non-zero value:

$$normVals[k] = 1 \div (M+d), \quad (48)$$

- All arrays are double-word aligned.

Example:

Assume a 3-dimensional quantity:

```
F = [9 2 3;
     5 3 5;
     8 1 3;
     4 2 3;
     7 1 1];
```

Where the maximum possible value in each dimension is as follows:

```
Dim 1 = 10
Dim 2 = 4
Dim 3 = 5
```

The required output is a histogram with 3×5×2 bins:

```
Dim 1 = 3 bins
Dim 2 = 5 bins
Dim 3 = 2 bins
```

The following is the form of the input:

```
X = [9 5 8 4 7 2 3 1 2 1 3 5 3 3 1];
numX = 5;
dimX = 3;
binWeight = 1 (or 1/5 for a normalized histogram)
numBins = [3 5 2];
normVals = [1/11 1/5 1/6] * 65536;
internal1 = array of 0s of size 5
internal2 = array of 0s of size 30
H = array of 0s of size 30
```

49.2.5 APIs

```
int VLIB_histogram_nD_U16(
            unsigned short* restrict X,
            const int numX,
            const int dimX,
            const unsigned short binWeight,
            unsigned short* restrict numBins,
            unsigned short* restrict normVals,
            unsigned short* restrict internal1,
            unsigned short* restrict internal2,
            unsigned short* restrict H);
```

49.3 Performance Benchmarks

On-chip memory performance was measured as in Equation 49:

$$((1.25 * numX * dimX) + (3.5 * numX) + (\text{total number of bins}))\ \text{cycles} \quad (49)$$

50 Weighted Histogram Computation for Multi-Dimensional Vectors (16-Bit)

50.1 Introduction and Use Cases

Histograms are used commonly as a discrete measure of the distribution of the input data. Weighted histograms permit the user he flexibility to influence the relative importance of different values in the input data.

50.2 Specification

50.2.1 Function

Computes a weighted multi-dimensional histogram for 16-bit vector valued variables.

50.2.2 Inputs

`unsigned short`	`*X`	Array of input values, data arranged in planar form	(UQ16.0)
`int`	`numX`	Number of individual vector elements in X	(SQ31.0)
`int`	`dimX`	Dimensionality of vectors in X	(SQ31.0)
`unsigned short`	`*binWeight`	Array of size numX of weight that each element contributes to the histogram	(UQ16.0)
`unsigned short`	`*normVals`	Array of size dimX, each element specifies the number of bins required in that dimensions	(UQ16.0)
`unsigned short`	`*internal1`	Buffer of size numX, for internal use (initialized to 0)	(UQ16.0)
`unsigned short`	`*internal2`	Buffer of size equal to total number of bins, for internal use (initialized to 0)	(UQ16.0)
`unsigned short`	`*H`	Array of size equal to total number of bins to hold computed histogram (initialized to 0)	(UQ16.0)

50.2.3 Output

`int`	Returns VLIB Error Status

50.2.4 Notes

- The vectors in X should be arranged in planar form (see Example below)
- H should be initialized to 0
- internal1 and internal2 should be initialized to 0
- The normalization factor normVals[k] for each dimension k should be set as in Equation 50, where M is the maximum value in dimension k, and d > 0:

$$normVals[k] = 1/(M+d) \tag{50}$$

- All arrays are double-word aligned

Example:

Assume a 3-dimensional quantity:

```
F = [9 2 3;
     5 3 5;
     8 1 3;
     4 2 3;
     7 1 1];
```

Where the maximum possible value in each dimension is as follows:

```
Dim 1 = 10
Dim 2 = 4
Dim 3 = 5
```

The required output is a histogram with 3×5×2 bins:

```
Dim 1 = 3 bins
Dim 2 = 5 bins
Dim 3 = 2 bins
```

The following is the form of the input:

```
X = [9 5 8 4 7 2 3 1 2 1 3 5 3 3 1];
numX = 5;
dimX = 3;
binWeight = array of 1/5 of size 5
numBins = [3 5 2];
normVals = [1/11 1/5 1/6] * 65536;
internal1 = array of 0s of size 5
internal2 = array of 0s of size 30
H = array of 0s of size 30
```

50.2.5 APIs

```
int VLIB_weightedHistogram_nD_U16(
            unsigned short* restrict X,
            const int numX,
            const int dimX,
            unsigned short* binWeight,
            unsigned short* restrict numBins,
            unsigned short* restrict normVals,
            unsigned short* restrict internal1,
            unsigned short* restrict internal2,
            unsigned short* restrict H);
```

50.3 Performance Benchmarks

On-chip memory performance has been measured as in Equation 51:

$$((1.25 \times \text{numX} \times \text{dimX}) + (3.5 \times \text{numX}) + (\text{total number of bins}))\ \text{cycles} \quad (51)$$

51 Bhattacharya Distance (32-Bit)

51.1 Introduction and Use Cases

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

51.2 Specification

51.2.1 Function

This function accepts as input two arrays, p and q, of size N containing the discrete probability distributions. It returns the Bhattacharya distance, B, between p and q as a 32-bit unsigned integer as defined in Equation 52:

$$\left(1 - \sum_{i=1}^{N} \sqrt{p(i) \times q(i)}\right)^{1/2} \quad (52)$$

51.2.2 Inputs

unsigned short	*X	Pointer to array containing first probability distribution	(UQ16.0)
unsigned short	*Y	Pointer to array containing second probability distribution	(UQ16.0)
int	N	Number of elements in the probability distributions	(SQ31.0)
unsigned int	*D	Pointer to variable to store the computed Bhattacharya Distance	(SQ32.0)

51.2.3 Output

void

51.2.4 Notes

- All arrays should be double-word aligned.
- Bhattacharya distance is defined on probability distribution functions. This implies that the elements in X and Y should sum to 1, respectively.
- There should be a minimum of four elements in X and Y.

51.2.5 APIs

```
void VLIB_bhattacharyaDistance_U32(
            unsigned short * restrict X,
            unsigned short * restrict Y,
            int N,
            unsigned int * D);
```

51.3 Performance Benchmarks

The performance of the function was measured as: 45.7 × N cycles, where N is the number of elements in the input probability distribution functions.

52 L1 Distance (City Block Distance) (16-bit)

52.1 Introduction and Use Cases

L1 Distance, also called city block distance, is a measure of the distance between two vectors.

52.2 Specification

52.2.1 Function

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 Equation 53.

$$L1D = \sum_{i=1}^{N} \min\left[\left(2^{15}-1\right)\left(\left|p_i - q_i\right|\right)\right] \quad (53)$$

52.2.2 Inputs

`short restrict`	`*X`	Pointer to array containing first vector	(SQ15.0)
`short restrict`	`*Y`	Pointer to array containing second vector	(SQ15.0)
`int`	`N`	Number of elements in each vector	(SQ32.0)
`unsigned int`	`*L1D`	Pointer to variable to store the computed L1 Distance	(Q32.0)

52.2.3 Output

`void`

52.2.4 Notes

- All arrays should be double-word aligned.
- There should be a minimum of four elements in X and Y.
- If the absolute difference between two corresponding vector elements is greater than 2^15-1, then that particular value is saturated to 2^15-1.

52.2.5 APIs

```
void VLIB_L1DistanceS16(
            short* restrict X,
            short* restrict Y,
            int N,
            unsigned int* L1D)
```

52.3 Performance Benchmarks

The performance of the function was measured as: 0.54 × N cycles, where N is the number of elements in the input vectors.

53 Luminance Extraction From YUV422

53.1 Introduction and Use Cases

When the image data is stored in the YUV422 format but the processing needs to be done on its luminance component only, it is often desirable to extract the Y component and store it in a separate buffer. This is particularly useful when data needs to be contiguous.

53.2 Specification

53.2.1 Function

Extracts the luminance data from the YUV422 image.

53.2.2 Inputs

`char`	`*inputImage`	Input YUV422 image	(UQ8.0)
`unsigned short`	`inputWidth`	Width of input image	(in pixels)
`unsigned short`	`inputPitch`	Pitch of input image	(in pixels)
`unsigned short`	`inputHeight`	Height of input image	(in pixels)
`char`	`*outputImage`	Luma-only output image	(UQ8.0)

53.2.3 Output

`int`	Returns VLIB Error Status

53.2.4 Method

If the input data is in YUV422, then in order to obtain a luminance only buffer, every other byte is extracted and copied to the buffer pointed to by outputImage.

53.2.5 APIs

```
int VLIB_extractLumaFromUYUV(
            char* restrict inputImage,
            unsigned short inputWidth,
            unsigned short inputPitch,
            unsigned short inputHeight,
            char* restrict outputImage);
```

53.3 Performance Benchmarks

The performance with all input and output data in on-chip memory is 0.29 cycles/outputs.

54 Conversion From 8-Bit YUV422 Interleaved to YUV422 Planar

54.1 Introduction and Use Cases

YUV422 is a common imaging data format [1]. If the YUV color channels are interleaved, as is often the case, this function is usually beneficial for improving the performance of vision applications, as it separates the three color channels into separate buffers, color planes. This is helpful because data transfers between external and internal memory are faster for contiguous data.

54.2 Specification

54.2.1 Function

Deinterleaves color channels of an interleaved YUV422 data block.

54.2.2 Inputs

`const unsigned char`	`*yc`	Interleaved luma/chroma	(UQ8.0)
`int`	`width`	Width of input image (number of luma pixels)	(in pixels)
`int`	`pitch`	Pitch of input image (number of luma pixels)	(in pixels)
`int`	`height`	Height of input image(number of luma pixels)	(in pixels)
`unsigned char`	`*restrict y`	Luma plane (8-bit)	(in pixels)
`unsigned char`	`*restrict cr`	Cr chroma plane (8-bit)	(UQ8.0)
`unsigned char`	`*restrict cb`	Cb chroma plane (8-bit)	(UQ8.0)

54.2.3 Output

`int`	Returns VLIB Error Status

54.2.4 Method

Given pixels in the interleaved format, this function separates the three channels into separate buffers. The width must be a multiple of 8, while input and output buffers must be 64-bit aligned.

54.2.5 APIs

```
int VLIB_convertUYVYint_to_YUVpl(
            const unsigned char *yc,
            int width,
            int pitch,
            int height,
            unsigned char *restrict y,
            unsigned char *restrict cr,
            unsigned char *restrict cb);
```

54.3 Performance Benchmarks

The compute-only performance is 0.4 cycles/pixel.

54.4 References

1. *Digital Image Processing* by R.C.Gonzales and R.E.Woods, Prentice-Hall, 2007

55 Conversion From 8-bit YUV422 Interleaved to YUV420 Planar

55.1 Introduction and Use Cases

YUV420 is a common imaging data format [1]. It offers more compressed chroma data and thus a reduced bandwidth over YUV422. If the YUV422 color channels are interleaved, as is often the case, this function is usually beneficial for improving the performance of vision applications, as it separates the three color channels into separate buffers, color planes. This is helpful because data transfers between external and internal memory are faster for contiguous data.

55.2 Specification

55.2.1 Function

Deinterleaves color channels of an interleaved YUV422 data block and creates YUV420 planar format.

55.2.2 Inputs

`const unsigned char`	`*yc`	Interleaved luma/chroma	(UQ8.0)
`int`	`width`	Width of input image (number of luma pixels)	(in pixels)
`int`	`pitch`	Pitch of input image (number of luma pixels)	(in pixels)
`int`	`height`	Height of input image(number of luma pixels)	(in pixels)
`unsigned char`	`*restrict y`	Luma plane (8-bit)	(in pixels)
`unsigned char`	`*restrict cr`	Cr chroma plane (8-bit)	(UQ8.0)
`unsigned char`	`*restrict cb`	Cb chroma plane (8-bit)	(UQ8.0)

55.2.3 Output

`int`	Returns VLIB Error Status

55.2.4 Method

Given pixels in the YUV422 interleaved format, this function separates the three channels into separate buffers, and vertically subsamples the chroma information by a factor of 2. To prevent aliasing in the chroma data, the values extracted from YUV422 are averaged. The width must be a multiple of 8, the height must be a multiple of 2, while input and output buffers must be 64-bit aligned.

55.2.5 APIs

```
int VLIB_convertUYVYint_to_YUV420pl(
            const unsigned char *yc,
            int width,
            int pitch,
            int height,
            unsigned char *restrict y,
            unsigned char *restrict cr,
            unsigned char *restrict cb);
```

55.3 Performance Benchmarks

The compute-only performance is 0.41 cycles/pixel.

55.4 References

1. *Digital Image Processing* by R.C.Gonzales and R.E.Woods, Prentice-Hall, 2007

56 Conversion From 8-bit YUV422 Interleaved to HSL Planar

56.1 Introduction and Use Cases

HSL (Hue Saturation Lightness), also known as HSI (Hue Saturation Intensity), or HSB (Hue Saturation Brightness) is a popular color space for image representation, especially in computer graphics and other applications where color perception needs to be modeled better than with RGB [1]. If the input data is in the interleaved YUV422 color format, this function transforms the data into the HSL format and separates the three color channels into separate buffers, color planes.

56.2 Specification

56.2.1 Function

Calculates HSL representation of pixels represented in interleaved YUV422 format.

56.2.2 Inputs

`const unsigned char`	`*yc`	Interleaved luma/chroma	(UQ8.0)
`int`	`width`	Width of input image (number of luma pixels)	(in pixels)
`int`	`pitch`	Pitch of input image (number of luma pixels)	(in pixels)
`int`	`height`	Height of input image(number of luma pixels)	(in pixels)
`const short`	`coeff[5]`	Matrix coefficients	(SQ16.0)
`const unsigned short`	`div_table[510]`	Division table	(UQ16.0)
`unsigned char`	`*restrict H`	Pointer to H plane (8-bit)	(UQ8.0)
`unsigned char`	`*restrict S`	Pointer to S plane (8-bit)	(UQ8.0)
`unsigned char`	`*restrict L`	Pointer to L plane (8-bit)	(UQ8.0)

The matrix coefficients specified by the array coeff are typically as shown in Equation 54 for the case of RGB levels that correspond the 219-level range of Y. Expected ranges are [16..235] for Y and [16..240] for Cb and Cr.

coeff[] = { 0x2000, 0x2BDD, -0x0AC5, -0x1658, 0x3770 }; (54)

Alternatively, as shown in Equation 55for the case of RGB conversion with the 219-level range of Y expanded to fill the full RGB dynamic range. Expected ranges are [16..235] for Y and [16..240] for Cb and Cr.

coeff[] = { 0x2543, 0x3313, -0x0C8A, -0x1A04, 0x408D }; (55)

The division table is used to provide an LUT to replace integer divisions by multiplications with corresponding inverses, shifted left by 15.

56.2.3 Output

`int`	Returns VLIB Error Status

56.2.4 Method

Given pixels in the interleaved YUV422 format, this function transforms the data into the HSL format and separates the three channels into separate buffers. The width must be a multiple of 8, while input and output buffers must be 64-bit aligned.

56.2.5 APIs

```
int VLIB_convertUYVYint_to_HSLpl(
            const unsigned char *yc,
            int width,
            int pitch,
            int height,
            const short coeff[5],
            const unsigned short div_table[510],
            unsigned short *restrict H,
            unsigned char  *restrict S,
            unsigned char  *restrict L);
```

56.3 Performance Benchmarks

The compute-only performance is 113 cycles/pixel.

56.4 References

1. *Digital Image Processing* by R.C.Gonzales and R.E.Woods, Prentice-Hall, 2007

57 Conversion From 8-bit YUV422 Interleaved to LAB Planar

57.1 Introduction and Use Cases

Lab color space is an important color-opponent color space. It is derived from the CIE XYZ color space through a non-linear compression, which assures perceptual uniformity [1]. If the input data is in the interleaved YUV422 color format, this function transforms the data into the LAB format and separates the three color channels into separate buffers, color planes.

57.2 Specification

57.2.1 Function

Calculates LAB representation of pixels represented in interleaved YUV422 format.

57.2.2 Inputs

`const unsigned char`	`*yc`	Interleaved luma/chroma	(UQ8.0)
`int`	`width`	Width of input image (number of luma pixels)	(in pixels)
`int`	`pitch`	Pitch of input image (number of luma pixels)	(in pixels)
`int`	`height`	Height of input image(number of luma pixels)	(in pixels)
`const short`	`coeff[5]`	YUV to sRGB matrix coefficients	(SQ16.0)
`float`	`whitePoint[3]`	D65 = {0.950456, 1.0, 1.088754};	(float)
`float`	`*restrict L`	Pointer to L plane (8-bit)	(float)
`float`	`*restrict a`	Pointer to A plane (8-bit)	(float)
`float`	`*restrict b`	Pointer to B plane (8-bit)	(float)

The matrix coefficients specified by the array coeff are typically as shown in Equation 56 for the case of RGB levels that correspond the 219-level range of Y. Expected ranges are [16..235] for Y and [16..240] for Cb and Cr.

coeff[] = { 0x2000, 0x2BDD, -0x0AC5, -0x1658, 0x3770 }; (56)

Alternatively, as shown in Equation 57, for the case of RGB conversion with the 219-level range of Y expanded to fill the full RGB dynamic range. Expected ranges are [16..235] for Y and [16..240] for Cb and Cr.

coeff[] = { 0x2543, 0x3313, -0x0C8A, -0x1A04, 0x408D }; (57)

The white point specification is used in the normalization step of the intermediate XYZ color space. A common value is a D65 value given by Equation 58:

float whitePoint[3] = {0.950456, 1.0, 1.088754}; (58)

57.2.3 Output

`int`	Returns VLIB Error Status

57.2.4 Method

Given pixels in the interleaved YUV422 format, this function transforms the data into the LAB format and separates the three channels into separate buffers. The width must be a multiple of 8, while input and output buffers must be 64-bit aligned.

57.2.5 APIs

```
int VLIB_convertUYVYint_to_LABpl(
            const unsigned char *yc,
            int width,
            int pitch,
            int height,
            const short coeff[5],
            float whitePoint[3],
            float *restrict L,
            float *restrict a,
            float *restrict b);
```

57.3 Performance Benchmarks

The compute-only performance is 75000 cycles/pixel.

57.4 References

1. *Digital Image Processing* by R.C.Gonzales and R.E.Woods, Prentice-Hall, 2007

58 Conversion From 8-bit YUV422 Interleaved to RGB Planar

58.1 Introduction and Use Cases

Some vision applications require the data to be in the RGB format [1]. If the input data is in the interleaved YUV422 color format, this function transforms the data into the sRGB format and separates the three color channels into separate buffers, color planes. Planarization is helpful because data transfers between external and internal memory are faster for contiguous data.

58.2 Specification

58.2.1 Function

Calculates sRGB representation of pixels given in interleaved YUV422 format.

58.2.2 Inputs

`const unsigned char`	`*yc`	Interleaved luma/chroma	(UQ8.0)
`int`	`width`	Width of input image (number of luma pixels)	(in pixels)
`int`	`pitch`	Pitch of input image (number of luma pixels)	(in pixels)
`int`	`height`	Height of input image(number of luma pixels)	(in pixels)
`const short`	`coeff[5]`	Matrix coefficients	(SQ16.0)
`unsigned char`	`*restrict r`	Pointer to R plane (8-bit)	(UQ8.0)
`unsigned char`	`*restrict g`	Pointer to G plane (8-bit)	(UQ8.0)
`unsigned char`	`*restrict b`	Pointer to B plane (8-bit)	(UQ8.0)

The matrix coefficients specified by the array coeff are typically as shown in Equation 59 for the case of RGB levels that correspond the 219-level range of Y. Expected ranges are [16..235] for Y and [16..240] for Cb and Cr.

coeff[] = { 0x2000, 0x2BDD, -0x0AC5, -0x1658, 0x3770 }; (59)

Alternatively, as shown in Equation 60, for the case of RGB conversion with the 219-level range of Y expanded to fill the full RGB dynamic range. Expected ranges are [16..235] for Y and [16..240] for Cb and Cr.

coeff[] = { 0x2543, 0x3313, -0x0C8A, -0x1A04, 0x408D }; (60)

58.2.3 Output

`int`	Returns VLIB Error Status

58.2.4 Method

Given pixels in the interleaved YUV422 format, this function transforms the data into the sRGB format and separates the three channels into separate buffers. The width must be a multiple of 8, while input and output buffers must be 64-bit aligned.

58.2.5 APIs

```
int VLIB_convertUYVYint_to_RGBpl(
            const unsigned char *yc,
            int width,
            int pitch,
            int height,
            const short coeff[5],
            unsigned char *restrict r,
            unsigned char *restrict g,
            unsigned char *restrict b);
```

58.3 Performance Benchmarks

The compute-only performance is 2 cycles/pixel.

58.4 References

1. *Digital Image Processing* by R.C.Gonzales and R.E.Woods, Prentice-Hall, 2007

59 LUT-Based Conversion From 8-Bit YUV422 Interleaved to LAB Planar

59.1 Introduction and Use Cases

This function is a fast approximation to the function VLIB_convertUYVYint_to_LABpl. Lab color space is an important color-opponent color space. It is derived from the CIE XYZ color space through a non-linear compression, which assures perceptual uniformity [ ]. If the input data is in the interleaved YUV422 color format, this function transforms the data into the LAB format and separates the three color channels into separate buffers, color planes.

59.2 Specification

59.2.1 Function

Calculates LAB representation of pixels represented in interleaved YUV422 format.

59.2.2 Inputs

`unsigned char`	`*restrict yc`	Interleaved luma/chroma	(UQ8.0)
`int`	`width`	Width of input image (number of luma pixels)	(in pixels)
`int`	`pitch`	Pitch of input image (number of luma pixels)	(in pixels)
`int`	`height`	Height of input image(number of luma pixels)	(in pixels)
`int`	`d`	Defines the LUT sparsity: 1/2^(3d)	
`unsigned short`	`*restrict LabLUT`	Pointer to the Lab LUT	(UQ16.0)
`unsigned short`	`*restrict l`	Pointer to L plane	(UQ16.0)
`unsigned short`	`*restrict a`	Pointer to L plane	(UQ16.0)
`unsigned short`	`*restrict b`	Pointer to L plane	(UQ16.0)

The calculated values are stored as 16-bit values. The approximate relationship to the floating-point values calculated by the VLIB function VLIB_convertUYVYint_to_LABpl is as follows:

```
L = (unsigned short)(439.832×L_f + 3518.66 + 0.5);
a = (unsigned short)(232.394×a_f + 29513.99 + 0.5);
b = (unsigned short)(221.402×b_f + 29225.07 + 0.5);
```

The conversion back to the floating pt. representation is given by:

```
L_f = (L - 3518.66)/439.832;
a_f = (a - 29513.99)/232.394;
b_f = (b - 29225.07)/221.402;
```

Parameter d defines the sparsity of the LUT – each of three dimensions of the LUT is subsampled by a factor of 2d. The associated memory / accuracy trade-off is given in Table 1.

Table 1. LUT Associated Memory/Accuracy Tradeoff

			0 < L < 65536		0 < a < 65536		0 < b < 65536	
Param d	**Decimation Factor**	**Memory for LUT**	**Mean abs err**	**Max abs err**	**Mean abs err**	**Max abs err**	**Mean abs err**	**Max abs err**
0	1	97 MB	0	0	0	0	0	0
1	2×2×2	12 MB	0.43	15	0.98	55	0.7	28
2	4×4×	1.6 MB	1.72	46	4.23	169	2.97	99
3	8×8×8	120 KB	7.51	165	16.6	521	12.1	283
4	16×16×16	29 KB	30.7	597	61.3	1705	46.9	699

The pointer LabLUT points to the LUT, which can be generated using the initialization function VLIB_initUYVYint_to_LABpl_LUT. The initialization function takes the following arguments:

`const int`	`d`	Decimation factor	(in pixels)
`const short`	`coeff[5]`	YUV to sRGB Matrix coefficient	(SQ16.0)
`const float`	`whitePoint[3]`	D65 = {0.950456, 1.0, 1.088754};	(float)
`unsigned short`	`*lab`	Interleaved Lab values	(UQ16.0)

Parameter d, as before, determines the level of sparsity of the LUT.

The matrix coefficients specified by the array coeff are typically as shown in Equation 61 for the case of RGB levels that correspond the 219-level range of Y. Expected ranges are [16..235] for Y and [16..240] for Cb and Cr.

coeff[] = { 0x2000, 0x2BDD, -0x0AC5, -0x1658, 0x3770 } (61)

Alternatively, as shown in Equation 62, for the case of RGB conversion with the 219-level range of Y expanded to fill the full RGB dynamic range. Expected ranges are [16..235] for Y and [16..240] for Cb and Cr.

coeff[] = { 0x2543, 0x3313, -0x0C8A, -0x1A04, 0x408D }; (62)

The white point specification is used in the normalization step of the intermediate XYZ color space. A common value is a D65 value given by Equation 63.

float whitePoint[3] = {0.950456, 1.0, 1.088754}; (63)

59.2.3 Output

`int`	Returns VLIB Error Status

59.2.4 Method

Given pixels in the interleaved YUV422 format, this function transforms the data into the LAB format and separates the three channels into separate buffers. The width must be a multiple of 8, while input and output buffers must be 64-bit aligned.

59.2.5 APIs

```
int VLIB_convertUYVYint_to_LABpl_LUT(
            unsigned char * restrict yc,      /* Interleaved luma/chroma         */
            int width,                        /* width (number of luma pixels)   */
            int pitch,
            int height,
            int d,                            /* Decimation factor               */
            unsigned short * restrict LabExt, /* pointer to the Lab LUT          */
            unsigned short * restrict L,
            unsigned short * restrict a,
            unsigned short * restrict b);

int VLIB_initUYVYint_to_LABpl_LUT(
            const int d,                      /* Decimation factor               */
            const short coeff[5],             /* YUV to sRGB Matrix coefficients */
            const float whitePoint[3],        /* D65 = {0.950456, 1.0, 1.088754}; */
            unsigned short *Lab);             /* Interleaved Lab values
```

59.3 Performance Benchmarks

The compute-only performance depends on which memory is used for the LUT:

Memory	Performance
DDR2	82 cycles/pixel
L2D	39 cycles/pixel
L1D	33 cycles/pixel

The reported performance is for d = 4 (16x16x16 decimation factor) and it may be different for other values.

59.4 References

1. *Digital Image Processing* by R.C.Gonzales and R.E.Woods, Prentice-Hall, 2007

60 Conversion From 8-Bit YUV422 Semiplanar to YUV422 Planar

60.1 Introduction and Use Cases

YUV422 is a common imaging data format [1]. If the YUV is in the semiplanar format (luma is planar but chroma channels are interleaved), as is sometimes the case, this function may be useful to interleave the chroma channels.

60.2 Specification

60.2.1 Function

Deinterleaves chroma channels of a semiplanar YUV422 data block.

60.2.2 Inputs

`const unsigned char`	`*crb`	Interleaved chroma	(UQ8.0)
`int`	`width`	Width of input image (number of luma pixels)	(in pixels)
`int`	`pitch`	Pitch of input image (number of luma pixels)	(in pixels)
`int`	`height`	Height of input image(number of luma pixels)	(in pixels)
`unsigned char`	`*restrict cr`	Cr chroma plane (8-bit)	(UQ8.0)
`unsigned char`	`*restrict cb`	Cb chroma plane (8-bit)	(UQ8.0)

60.2.3 Output

`int`	Returns VLIB Error Status

60.2.4 Method

Given pixels in the semiplanar format, this function separates the chroma channels into separate buffers. The width must be a multiple of 8, while input and output buffers must be 64-bit aligned.

60.2.5 APIs

```
int VLIB_ConvertUYVYsemipl_to_YUVpl(
            const unsigned char * crcb,  /* Interleaved chroma           */
            int width,                   /* width (number of luma pixels) */
            int pitch,                   /* pitch (number of luma pixels) */
            int height,                  /* height (number of luma pixels)*/
            unsigned char *restrict cr,  /* Cr chroma plane (8-bit)       */
            unsigned char *restrict cb); /* Cb chroma plane (8-bit)       */
```

60.3 Performance Benchmarks

The compute-only performance is 0.26 cycles/pixel.

60.4 References

1. *Digital Image Processing* by R.C.Gonzales and R.E.Woods, Prentice-Hall, 2007

61 Conversion From 8-Bit YUV422 Planar to YUV422 Interleaved

61.1 Introduction and Use Cases

YUV422 is a common imaging data format [1]. If the information stored in the planar YUV format needs to be displayed it often needs to be interleaved first. This functions efficiently interleaves the YUV color channels.

61.2 Specification

61.2.1 Function

Interleaves YUV channels of a planar YUV422 data block.

61.2.2 Inputs

`const unsigned char`	`*restrict y`	The luma plane	(UQ8.0)
`const unsigned char`	`*restrict cr`	The Cr plane	(UQ8.0)
`const unsigned char`	`*restrict cb`	The Cb plane	(UQ8.0)
`int`	`width`	Width of input image (number of luma pixels)	(in pixels)
`int`	`pitch`	Pitch of input image (number of luma pixels)	(in pixels)
`int`	`height`	Height of input image(number of luma pixels)	(in pixels)
`unsigned char`	`*restrict vc`	Interleaved data	(UQ8.0)

61.2.3 Output

`int`	Returns VLIB Error Status

61.2.4 Method

Given data in the planar format, this function interleaves the data to the YUV422 format. The width must be a multiple of 8, while input and output buffers must be 64-bit aligned.

61.2.5 APIs

```
int VLIB_ConvertUYVYpl_to_YUVint(
            const unsigned char *restrict y,      /* Luma plane (8-bit)      */
            const unsigned char *restrict cr,     /* Cr chroma plane (8-bit) */
            const unsigned char *restrict cb,     /* Cb chroma plane (8-bit) */
            int width,
            int pitch,
            int height,
            unsigned char *restrict yc);          /* Interleaved luma/chroma  */
```

61.3 Performance Benchmarks

The compute-only performance is 0.7 cycles/pixel.

61.4 References

1. *Digital Image Processing* by R.C.Gonzales and R.E.Woods, Prentice-Hall, 2007

62 SAD Based Disparity Computation (8-Bit)

62.1 Introduction and Use Cases

Disparity gives a measure of depth information from stereo images. Here we provide an algorithm to compute disparity from rectified stereo images using Sum of absolute difference (SAD) based block matching.

62.2 Specification

62.2.1 Function

VLIB_disparity_SAD8 calculates the disparity at each position in a row of an 8-bit image. This function is optimized reusing the previous calculations. For the first row calculations cannot be reused; thus, VLIB_disparity_SAD_firstRow8 should be used to calculate the disparities in the first row.

62.2.2 Inputs

Both the APIs VLIB_disparity_SAD8 and VLIB_disparity_SAD_firstRow8 use the same set of inputs except the input pScratch which only the second API uses.

`const unsigned char *`	`pLeft`	Pointer to left image	(UQ8.0)
`const unsigned char *`	`pRight`	Pointer to right image	(UQ8.0)
`unsigned short *`	`pCost`	Cost corresponding to current displacement	(UQ16.0)
`unsigned short *`	`pMinCost`	Minimum cost across all displacements	(UQ16.0)
`unsigned char *`	`restrict pScratch`	Scratch Memory of size width	(UQ8.0)
`char *`	`pDisparity`	Displacement having the minimum cost	(SQ8.0)
`int`	`displacement`	Current displacement	(in pixels)
`int`	`width`	Width of the input images	(in pixels)
`int`	`pitch`	Pitch of the input images	(in pixels)
`int`	`windowSize`	Size of the block used for computing SAD	(in pixels)

62.2.3 Output

`int`	Returns VLIB Error Status

62.2.4 Method

Two images, the left and right images (8-bit), are used as inputs to the algorithm. These images are assumed to be rectified so that the disparity search is only along the row. The parameter pCost buffer is used to hold the SAD cost function for all pixels in a row and for all permissible values of horizontal displacements. VLIB_disparity_SAD8 computes the cost measure for a row and for a specified displacment. This function has to be looped over the range of disparity and then through all the rows. The function also updates the pMinCost buffer and stores the displacement which corresponds to the mininmum cost in pDisparity buffer. This is the simplest method for disparity calcuation. But the API gives out the cost measure at each pixel and each disparity which can be used for more complicated algorithms like dynamic programming,etc.

VLIB_disparity_SAD8 uses the pCost buffer corresponding to the previous row for calculations of the current row. Care has to be taken that pCost buffer is not cleared or reused for some other purpose. For the first row, we cannot reuse the calculations. Thus a separate API VLIB_disparity_SAD_firstRow8 is provided. It uses a scratch buffer pScratch of size width for internal calcuations. This is required only for the first row disparity computation and can be freed after that. The buffers pCost, pMinCost, pDisparity have to be padded up with eight extra locations as illustrated in VLIB_testDisparity8 using ARRAY_PAD macro.

62.2.5 APIs

```
int VLIB_disparity_SAD_firstRow8(
            const unsigned char *restrict pLeft,
            const unsigned char *restrict pRight,
            unsigned short *restrict pCost,
            unsigned short *restrict pMinCost,
            char *restrict pDisparity,
            int displacement,
            int width,
            int pitch,
            int windowSize );

int VLIB_disparity_SAD8(
            const unsigned char *restrict pLeft,
            const unsigned char *restrict pRight,
            unsigned short *restrict pCost,
            unsigned short *restrict pMinCost,
            unsigned char *restrict pScratch,
            char *restrict pDisparity,
            int displacement,
            int width,
            int pitch,
            int windowSize );
```

62.3 Performance Benchmarks

On-chip memory performance has been measured as:

VLIB_disparity_SAD_firstRow8	9.1 cycles/pixel
VLIB_disparity_SAD8	2.6 cycles/pixel

62.4 References

1. *Computer Vision*, pages 371-409, by Linda G. Shapiro and George C. Stockman, Prentice-Hall, 2001

63 SAD Based Disparity Computation (16-Bit)

63.1 Introduction and Use Cases

Disparity gives a measure of depth information from stereo images. Here we provide an algorithm to compute disparity from rectified stereo images using Sum of absolute difference (SAD) based block matching.

63.2 Specification

63.2.1 Function

VLIB_disparity_SAD16 calculates the disparity at each position in a row of an 16-bit image. This function is optimized reusing the previous calculations. For the first row calculations can't be reused, thus VLIB_disparity_SAD_firstRow16 should be used to calculate the disparities in the first row.

63.2.2 Inputs

Both the APIs VLIB_disparity_SAD16 and VLIB_disparity_SAD_firstRow16 use the same set of inputs except the input pScratch which only the second API uses.

`const unsigned short *`	`pLeft`	Pointer to left image	(UQ16.0)
`const unsigned short *`	`pRight`	Pointer to right image	(UQ16.0)
`unsigned short *`	`pCost`	Cost corresponding to current displacement	(UQ16.0)
`unsigned short *`	`pMinCost`	Minimum cost across all displacements	(UQ16.0)
`unsigned char *`	`restrict pScratch`	Scratch Memory of size width	(UQ8.0)
`char *`	`pDisparity`	Displacement having the minimum cost	(SQ8.0)
`int`	`displacement`	Current displacement	(in pixels)
`int`	`width`	Width of the input images	(in pixels)
`int`	`pitch`	Pitch of the input images	(in pixels)
`int`	`windowSize`	Size of the block used for computing SAD	(in pixels)

63.2.3 Output

`int`	Returns VLIB Error Status

63.2.4 Method

Two images, the left and right images (16-bit), are used as inputs to the algorithm. These images are assumed to be rectified so that the disparity search is only along the row. pCost buffer is used to hold the SAD cost function for all pixels in a row and for all permissible values of horizontal displacements. VLIB_disparity_SAD16 computes the cost measure for a row and for a specified displacment. This function has to be looped over the range of disparity and then through all the rows. The function also updates the pMinCost buffer and stores the displacement which corresponds to the mininmum cost in pDisparity buffer. This is the simplest method for disparity calcuation. But the API gives out the cost measure at each pixel and each disparity which can be used for more complicated algorithms like dynamic programming,etc.

VLIB_disparity_SAD16 uses the pCost buffer corresponding to the previous row for calculations of the current row. Care has to be taken that pCost buffer is not cleared or reused for some other purpose. For the first row, we cannot reuse the calculations. Thus a separate API VLIB_disparity_SAD_firstRow16 is provided. It uses a scratch buffer pScratch of size width for internal calcuations. This is required only for the first row disparity computation and can be freed after that. The buffers pCost, pMinCost, pDisparity have to be padded up with eight extra locations as illustrated in VLIB_testDisparity16 using ARRAY_PAD macro.

63.2.5 APIs

```
int VLIB_disparity_SAD_firstRow16(
            const unsigned short *restrict pLeft,
            const unsigned short *restrict pRight,
            unsigned short *restrict pCost,
            unsigned short *restrict pMinCost,
            unsigned char *restrict pScratch,
            char *restrict pDisparity,
            int displacement,
            int width,
            int pitch,
            int windowSize );

int VLIB_disparity_SAD16(
            const unsigned short *restrict pLeft,
            const unsigned short *restrict pRight,
            unsigned short *restrict pCost,
            unsigned short *restrict pMinCost,
            char *restrict pDisparity,
            int displacement,
            int width,
            int pitch,
            int windowSize );
```

63.3 Performance Benchmarks

On-chip memory performance has been measured as:

VLIB_disparity_SAD_firstRow16	13.6 cycles/pixel
VLIB_disparity_SAD16	3.7 cycles/pixel

63.4 References

1. *Computer Vision*, pages 371-409, by Linda G. Shapiro and George C. Stockman, Prentice-Hall, 2001