# OpenCV Reference Manual

v2.4

June, 2012

## 1 General Information

The OpenCV OCL module is a set of classes and functions to utilize OpenCL compatible device. In theroy, it supports any OpenCL 1.1 compatible device, but we only test it on AMD's and NVIDIA's GPU at this stage. The OpenCV OCL module includes utility functions, low-level vision primitives, and high-level algorithms. The utility functions and low-level primitives provide a powerful infrastructure for developing fast vision algorithms taking advangtage of OCL whereas the high-level functionality includes some state-of-the-art algorithms(such as stereo correspondence, face detector) ready to be used by the application developers.

The OpenCV OCL module is designed as a host-level API plus device-level kernels. The device-level kernels are collected as strings at OpenCV compile time and are compiled at runtime, so it need OpenCL runtime support. To correctly run the OpenCV OCL module, make sure you have OpenCL runtime provided by your device vendor, which is device driver normally.

The OpenCV OCL module is designed for ease of use and does not require any knowledge of OpenCL. Though, such a knowledge will certainly be useful to handle non-trivial cases or achieve the highest performance. It is helpful to understand the cost of various operations, what the OCL does, what the preferred data formats are, and so on. Since there is data transfer between OpenCL host and OpenCL device, the recommanded operation is transfer data from host -¿ ocl::func1 -¿ ocl::func2 -¿ .... -¿ ocl::funcN -¿ transfer data to host if needed, not transfer data from host -¿ ocl::func1 -¿ transfer data to host -¿ transfer data from host -¿ ocl::func2 .....

To enable OCL support, configure OpenCV using CMake with WIHT_OPENCL=ON. When the flag is set and if OpenCL SDK is installed, the full-featured OpenCV OCL module is built. Otherwis, the module may be not built.

Right now, the user should define the cv::ocl::Info class in the application and call cv::ocl::getDevice before any cv::ocl::func. This operation initialize OpenCL runtime and set the first found device as computing device. If there are more than one device and you want to use undefault device, you can call cv::ocl::setDevice then.

In the current version, all the thread share the same context and device so the multi-devices are not supported. We will add this feature soon.

# 2 utility functions

## 2.1 Info

class CV_EXPORTS Info
{
public:
struct Impl;
Impl *impl;
Info();
Info(const Info &m);
Info();
void release();
Info &operator = (const Info &m);
};
this class sould be maintained by the user and be passed to getDevice

## 2.2 int getDevice

parameters:
std::vector¡Info¿& oclinfo it must not be NULL
int devicetype = CVCL_DEVICE_TYPE_GPU The device type you want to use, the default value is GPU, other alternatives are CVCL_DEVICE_TYPE_DEFAULT, CVCL_DEVICE_TYPE_CPU, CVCL_DEVICE_TYPE_ACCELERATOR, CVCL_DEVICE_TYPE_ALL
return value: the number of devices found, must be greater than 0.
the function must be called before any other cv::ocl::functions, it initialize ocl runtime

## 2.3 void setDevice

parameters:
Info &oclinfo the selected OpenCL platform
int devnum = 0 the selected OpenCL device under this platform

## 2.4 void setBinpath

parameters:
const char *path the path of OpenCL kernel binaries
If you call this function and set a valid path, the OCL module will save the compiled kernel to the address in the first time and reload the binary since that. It can save compilation time at the runtime.

# 3 1 core. The Core Functionality

## 3.1 1.1 Basic Structures

### 3.1.1 cv::ocl::convertTo

Converts array to another datatype with optional scaling.

Supports CV_8UC1,CV_8UC4,CV_32SC1,CV_32SC4,CV_32FC1,CV_32FC4.

**m** The destination matrix. If it does not have a proper size or type before the operation, it will be reallocated

**rtype** The desired destination matrix type, or rather, the depth (since the number of channels will be the same with the source one). If rtype is negative, the destination matrix will have the same type as the source.

**alpha** The optional scale factor

**beta** The optional delta, added to the scaled values.

The method converts source pixel values to the target datatype. saturate cast<> is applied in the end to avoid possible overflows:

$$m(x,y) = saturate\_cast < rType > (\alpha(*this)(x,y) + \beta)$$

### 3.1.2 cv::ocl::copyTo

Copies the matrix to another one.

Supports CV_8UC1,CV_8UC4,CV_32SC1,CV_32SC4,CV_32FC1,CV_32FC4.

**m** The destination matrix. If it does not have a proper size or type before the operation, it will bereallocated

**mask** The operation mask. Its non-zero elements indicate, which matrix elements need to be copied

The method copies the matrix data to another matrix. Before copying the data, the method invokes

so that the destination matrix is reallocated if needed. While m.copyTo(m); will work as expected, i.e. will have no effect, the function does not handle the case of a partial overlap between the source and the destination matrices.

When the operation mask is specified, and the Mat::create call shown above reallocated the matrix, the newly allocated matrix is initialized with all 0's before copying the data.

### 3.1.3 cv::ocl::setTo

Sets all or some of the array elements to the specified value.

Supports CV_8UC1,CV_8UC4,CV_32SC1,CV_32SC4,CV_32FC1,CV_32FC4.

**s** Assigned scalar, which is converted to the actual array type

**mask** The operation mask of the same size as *this

This is the advanced variant of Mat::operator=(const Scalar& s) operator.

## 3.2 1.2 operations on Arrays

### 3.2.1 cv::ocl::absdiff

Computes per-element absolute difference between two arrays or between array and a scalar.

Supports all data types except CV_8SC1, CV_8SC2, CV8SC3 and CV_8SC4.

**src1** The first input array

**src2** The second input array, must be the same size and same type as src1

**sc** Scalar, the second input parameter

**dst** The destination array, it will have the same size and same type as src1

The functions absdiff compute:

- absolute difference between two arrays

dst(I) = saturate(|src1(I) - src2(I)|)

- or absolute difference between array and a scalar:

dst(I) = saturate(|src1(I) − sc|)

Where I is multi-dimensional index of array elements. In the case of multi-channel arrays each channel is processed independently.

### 3.2.2 cv::ocl::add

Computes the per-element sum of two array and a scalar.

Supports all data types except CV_8SC1, CV_8SC2, CV8SC3 and CV_8SC4.

**src1** The first source array

**src2** The second source array. It must have the same size and same type as src1

**sc** Scalar; the second input parameter

**dst** The destination array; it will have the same size and same type as src1

**mask** The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions add compute:

- the sum of two arrays:

dst(I) = saturate(src1(I) + src2(I)) if mask(I) 6= 0

- or the sum of array and a scalar:

dst(I) = saturate(src1(I) + sc) if mask(I) 6= 0

where I is multi-dimensional index of array elements.

The first function in the above list can be replaced with matrix expressions:

in the case of multi-channel arrays each channel is processed independently.

### 3.2.3 cv::ocl::bitwise_and

Calculates per-element bit-wise conjunction of two arrays and an array and a scalar.

Supports all data types except CV_64FC1, CV_64FC2, CV64FC3 and CV_64FC4.

**src1** The first source array

**src2** The second source array. It must have the same size and same type as src1

**sc** Scalar; the second input parameter

**dst** The destination array; it will have the same size and same type as src1

**mask** The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions bitwise and compute per-element bit-wise logical conjunction:

- of two arrays:

$$\mathrm{dst}(I) = \mathrm{src1}(I) \wedge \mathrm{src2}(I) \quad \text{if } \mathrm{mask}(I) \neq 0$$

- or array and a scalar:

$$\mathrm{dst}(I) = \mathrm{src1}(I) \wedge \mathrm{sc} \text{ if } \mathrm{mask}(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation, and in the case of multi-channel arrays each channel is processed independently.

### 3.2.4 cv::ocl::bitwise_not

Inverts every bit of array.

Supports CV_8U, CV_8S, CV_16U, CV_16S, CV_32S, CV_32F data types.

**src1** The source array

**dst** The destination array; it is reallocated to be of the same size and the same type as src

**mask** The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions bitwise not compute per-element bit-wise inversion of the source array:

$$\mathrm{dst}\,(I) = \neg \mathrm{src}\,(I)$$

In the case of floating-point source array its machine-specific bit representation (usually IEEE754-compliant) is used for the operation. in the case of multi-channel arrays each channel is processed independently.

### 3.2.5 cv::ocl::bitwise_or

Calculates per-element bit-wise disjunction of two arrays and an array and a scalar.

Supports all data types except CV_64FC1,CV_64FC2,CV64FC3 and CV_64FC4.

**src1** The first source array

**src2** The second source array. It must have the same size and same type as src1

**sc** Scalar; the second input parameter

**dst** The destination array; it is reallocated to be of the same size and the same type as src1

**mask** The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions bitwise or compute per-element bit-wise logical disjunction

- of two arrays:

$$\mathrm{dst}\,(I) = \mathrm{src1}\,(I) \vee \mathrm{src2}\,(I) \;\text{ if } \mathrm{mask}\,(I) \neq 0$$

- or array and a scalar:

$$\mathrm{dst}\,(I) = \mathrm{src1}\,(I) \vee \mathrm{sc} \text{ if } \mathrm{mask}\,(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. in the case of multi-channel arrays each channel is processed independently.

### 3.2.6 cv::ocl::bitwise_xor

Calculates per-element bit-wise “exclusive or” operation on two arrays and an array and a scalar.

Supports all data types except CV_64FC1,CV_64FC2,CV64FC3 and CV_64FC4.

**src1** The first source array

**src2** The second source array. It must have the same size and same type as src1

**sc** Scalar; the second input parameter

**dst** The destination array; it is reallocated to be of the same size and the same type as src1

**mask** The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions bitwise xor compute per-element bit-wise logical ”exclusive or” operation

- on two arrays:

$$\mathrm{dst}\,(I) = \mathrm{src1}\,(I) \oplus \mathrm{src2}\,(I) \;\text{ if } \mathrm{mask}\,(I) \neq 0$$

- or array and a scalar:

$$\mathrm{dst}\,(I) = \mathrm{src1}\,(I) \oplus \mathrm{sc} \text{ if } \mathrm{mask}\,(I) \neq 0$$

In the case of floating-point arrays their machine-specific bit representations (usually IEEE754-compliant) are used for the operation. in the case of multi-channel arrays each channel is processed independently.

### 3.2.7 cv::ocl::cartToPolar

Calculates the magnitude and angle of 2d vectors.

Supports only CV_32F, CV_64F data types.

**x** The array of x-coordinates; must be single-precision or double-precision floating-point array

**y** The array of y-coordinates; it must have the same size and same type as x

**magnitude** The destination array of magnitudes of the same size and same type as x

**angle** The destination array of angles of the same size and same type as x. The angles are measured in radians (0 to $2\pi$) or in degrees (0 to 360 degrees).

**angleInDegrees** The flag indicating whether the angles are measured in radians, which is default mode, or in degrees

The function cartToPolar calculates either the magnitude, angle, or both of every 2d vector (x(I),y(I)):

$$\text{magnitude}(I) = \sqrt{\text{x}(I)^2 + \text{y}(I)^2}$$

$$\text{angle}(I) = \text{a}\tan 2(\text{y}(I), \text{x}(I)) \cdot [180/\pi]$$

The angles are calculated with$\sim$0.3$^\circ$ accuracy. For the (0,0) point, the angle is set to 0.

### 3.2.8 cv::ocl::compare

Performs per-element comparison of two arrays or an array and scalar value.

Supports all data types except CV_8SC1,CV_8SC2,CV8SC3,CV_8SC4.

**src1** The first source array

**src2** The second source array; must have the same size and same type as src1

**dst** The destination array; will have the same size as src1 and type=CV 8UC1

**cmpop** The flag specifying the relation between the elements to be checked

**CMP EQ** src1(I) = src2(I) or src1(I) = value

**CMP GT** src1(I) > src2(I) or src1(I) > value

**CMP GE** src1(I) _ src2(I) or src1(I) _ value

**CMP LT** src1(I) < src2(I) or src1(I) < value

**CMP LE** src1(I) _ src2(I) or src1(I) _ value

**CMP NE** src1(I) 6= src2(I) or src1(I) 6= value

The functions compare compare each element of src1 with the corresponding element of src2 or with real scalar value. When the comparison result is true, the corresponding element of destination array is set to 255, otherwise it is set to 0:

- dst(I) = src1(I) cmpop src2(I) ? 255 : 0
- dst(I) = src1(I) cmpop value ? 255 : 0

The comparison operations can be replaced with the equivalent matrix expressions:

### 3.2.9 cv::ocl::countNonZero

Counts non-zero array elements.

Supports all data types.

**mtx** Single-channel array

The function cvCountNonZero returns the number of non-zero elements in mtx:

$$\sum_{I:\mathrm{mtx}(I)\neq 0} 1$$

### 3.2.10 cv::ocl::divide

Performs per-element division of two arrays or a scalar by an array.

Supports all data types except CV_8SC1,CV_8SC2,CV8SC3 and CV_8SC4.

**src1** The first source array

**src2** The second source array; should have the same size and same type as src1

**scale** Scale factor

**dst** The destination array; will have the same size and same type as src2

The functions divide compute:

- divide one array by another:

dst(I) = saturate(src1(I)*scale/src2(I))

- or a scalar by array, when there is no src1:

dst(I) = saturate(scale/src2(I))

The result will have the same type as src1. When src2(I)=0, dst(I)=0 too.

### 3.2.11 cv::ocl::exp

Calculates the exponent of every array element.

Supports only CV_32FC1 data type.

**src** The source array

**dst** The destination array; will have the same size and same type as src The function exp calculates the exponent of every element of the input array:

$$\mathrm{dst}\left[I\right] = e^{\mathrm{src}}\left(I\right)$$

The maximum relative error is about $7{\times}10^{-6}$ for single-precision and less than $10^{-10}$for doubleprecision. Currently, the function converts denormalized values to zeros on output. Special values (NaN, $\pm\infty$) are not handled.

### 3.2.12 cv::ocl:: flip

Flips a 2D array around vertical, horizontal or both axes.

Supports all data types.

**src** The source array

**dst** The destination array; will have the same size and same type as src

**flipCode** Specifies how to flip the array: 0 means flipping around the x-axis, positive (e.g., 1) means flipping around y-axis, and negative (e.g., -1) means flipping around both axes. See also the discussion below for the formulas.

The function flip flips the array in one of three different ways (row and column indices are 0-based):

$$dst_{ij} = \begin{cases} \text{src}_{\text{src.rows-}i\text{-1, }j} \text{ if flipCode} = 0 \\ \text{src}_{\text{src}i\text{,scr.cols-}j\text{-1}} \text{ if flipCode?} \\ \text{src}_{\text{src.rows-}i\text{-1, scr.cols-}j\text{-1}} \text{ if flipCode?} \end{cases}$$

The example scenarios of function use are:

- vertical flipping of the image (flipCode = 0) to switch between top-left and bottom-left image origin, which is a typical operation in video processing in Windows.
- horizontal flipping of the image with subsequent horizontal shift and absolute difference calculation to check for a vertical-axis symmetry (flipCode $> 0$)
- simultaneous horizontal and vertical flipping of the image with subsequent shift and absolute difference calculation to check for a central symmetry (flipCode $< 0$)
- reversing the order of 1d point arrays (flipCode $> 0$ or flipCode $= 0$)

### 3.2.13 cv::ocl::log

Calculates the natural logarithm of every array element.

Supports only CV_32FC1 data type.

**src** The source array

**dst** The destination array; will have the same size and same type as src

The function log calculates the natural logarithm of the absolute value of every element of the input array:

$$\mathrm{dst}(I) = \begin{cases} \log|\mathrm{src}(I)| & \text{if } \mathrm{src}(I) \neq 0 \\ C & \text{otherwise} \end{cases}$$

Where C is a large negative number (about -700 in the current implementation). The maximum relative error is about $7\times 10^{-6}$ for single-precision input and less than $10^{-10}$ for double-precision input. Special values (NaN, $\pm\infty$) are not handled.

### 3.2.14 cv::ocl::LUT

Performs a look-up table transform of an array.

Supports only CV_8UC1,CV_8UC4

**src** Source array of 8-bit elements

**lut** Look-up table of 256 elements. In the case of multi-channel source array, the table should either have a single channel (in this case the same table is used for all channels) or the same number of channels as in the source array

**dst** Destination array; will have the same size and the same number of channels as src, and the same depth as lut

The function LUT fills the destination array with values from the look-up table. Indices of the entries are taken from the source array. That is, the function processes each element of src as follows:

$$\mathrm{dst}(I) \leftarrow \mathrm{lut}(\mathrm{src}(I) + d)$$

where

$$d = \begin{cases} 0 \text{ if src has depth CV\_8U} \\ 128 \text{ if src has depth CV\_8S} \end{cases}$$

### 3.2.15 cv::ocl::magnitude

Calculates magnitude of 2D vectors.

Supports only CV_32F, CV_64F data types.

**x** The floating-point array of x-coordinates of the vectors

**y** The floating-point array of y-coordinates of the vectors; must have the same size as x

**magnitude** The destination array; will have the same size and same type as x

The function magnitude calculates magnitude of 2D vectors formed from the corresponding elements of x and y arrays:

$$\mathrm{dst}\left(I\right) = \sqrt{x\left(I\right)^2 + y\left(I\right)^2}$$

### 3.2.16 cv::ocl::meanStdDev

Calculates mean and standard deviation of array elements.

Supports all data types except CV_32F,CV_64F.

**mtx** The source array; it should have 1 to 4 channels

**mean** The output parameter: computed mean value

**stddev** The output parameter: computed standard deviation

The functions meanStdDev compute the mean and the standard deviation M of array elements, independently for each channel, and return it via the output parameters:

$$N = \sum_I 1$$
$$mean_c = \frac{\sum_I src(I)_c}{N}$$
$$stddev_c = \sqrt{\sum_I (src(I)_c - mean_c)^2}$$

### 3.2.17 cv::ocl::merge

Composes a multi-channel array from several single-channel arrays.

Supports all data types.

**mv** The source array or vector of the single-channel matrices to be merged. All the matrices in mv must have the same size and the same type

**count** The number of source matrices when mv is a plain C array; must be greater than zero

**dst** The destination array; will have the same size and the same depth as mv[0], the number of channels will match the number of source matrices

The functions merge merge several single-channel arrays (or rather interleave their elements) to make a single multi-channel array.

$$\mathrm{dst}(I)_c = \mathrm{mv}\,[c]\,(I)$$

The function cv::split does the reverse operation and if you need to merge several multichannel images or shuffle channels in some other advanced way.

### 3.2.18 cv::ocl:: minMax

Calculates per-element minimum and maximum of two arrays or array and a scalar.

Supports all data types.

Finds global minimum and maximum in a whole array or sub-array

**src** The source single-channel array

**minVal** Pointer to returned minimum value; NULL if not required

**maxVal** Pointer to returned maximum value; NULL if not required

**mask** The optional mask used to select a sub-array

The functions ninMax find minimum and maximum element values. The extremums are searched across the whole array, or, if mask is not an empty array, in the specified array region.

The functions do not work with multi-channel arrays.

In the case of a sparse matrix the minimum is found among non-zero elements only.

### 3.2.19 cv::ocl:: minMaxLoc

Finds global minimum and maximum in a whole array or sub-array

Supports all data types.

**src** The source single-channel array

**minVal** Pointer to returned minimum value; NULL if not required

**maxVal** Pointer to returned maximum value; NULL if not required

**minLoc** Pointer to returned minimum location (in 2D case); NULL if not required

**maxLoc** Pointer to returned maximum location (in 2D case); NULL if not required

**mask** The optional mask used to select a sub-array

The functions ninMaxLoc find minimum and maximum element values and their positions. The extremums are searched across the whole array, or, if mask is not an empty array, in the specified array region.

The functions do not work with multi-channel arrays.

In the case of a sparse matrix the minimum is found among non-zero elements only.

### 3.2.20 cv::ocl::multiply

Calculates the per-element scaled product of two arrays.

Supports all data types except CV_8SC1,CV_8SC2,CV8SC3 and CV_8SC4.

**src1** The first source array

**src2** The second source array of the same size and the same type as src1

**dst** The destination array; will have the same size and the same type as src1

**scale** The optional scale factor

The function multiply calculates the per-element product of two arrays:

dst($I$) = saturate(scale·src1(I)·src2(I))

There is also Matrix Expressions -friendly variant of the first function.

### 3.2.21 cv::ocl:: norm

Calculates absolute array norm, absolute difference norm, or relative difference norm.

Supports only CV_8UC1 data type.

**src1** The first source array

**src2** The second source array of the same size and the same type as src1

**normType** Type of the norm; see the discussion below

The functions norm calculate the absolute norm of src1 (when there is no src2):

$$norm = \begin{cases} \|src1\|_{L\infty} = \max_I |src1(I)| \text{ if normType = NORM\_INF} \\ \|src1\|_{L1} = \sum_I |src1(I)| \text{ if normType = NORM\_L1} \\ \|src1\|_{L2} = \sqrt{\sum_I src1(I)^2} \text{ if normType = NORM\_L2} \end{cases}$$

or an absolute or relative difference norm if src2 is there:

$$norm = \begin{cases} \|src1 - src2\|_{L\infty} = \max_I |src1(I) - src2(I)| \text{ if normType = NORM\_INF} \\ \|src1 - src2\|_{L1} = \sum_I |src1(I) - src2(I)| \text{ if normType = NORM\_L1} \\ \|src1 - src2\|_{L2} = \sqrt{\sum_I (src1(I)src2(I))^2} \text{ if normType = NORM\_L2} \end{cases}$$

or

$$norm = \begin{cases} \frac{\|src1-src2\|_{L\infty}}{\|src2\|_{L\infty}} \text{ if normType = NORM\_INF} \\ \frac{\|src1-src2\|_{L1}}{\|src2\|_{L1}} \text{ if normType = NORM\_L1} \\ \frac{\|src1-src2\|_{L2}}{\|src2\|_{L2}} \text{ if normType = NORM\_L2} \end{cases}$$

The functions norm return the calculated norm.

When there is mask parameter, and it is not empty (then it should have type CV_8U and the same size as src1), the norm is computed only over the specified by the mask region.

A multiple-channel source arrays are treated as a single-channel, that is, the results for all channels are combined.

### 3.2.22 cv::ocl::phase

Calculates the rotation angle of 2d vectors.

Supports only CV_32F CV_64F data types.

**x** The source floating-point array of x-coordinates of 2D vectors

**y** The source array of y-coordinates of 2D vectors; must have the same size and the same type as x

**angle** The destination array of vector angles; it will have the same size and same type as x

**angleInDegrees** When it is true, the function will compute angle in degrees, otherwise they will be measured in radians

The function phase computes the rotation angle of each 2D vector that is formed from the corresponding elements of x and y:

$$\mathrm{angle}\left(I\right) = \mathrm{atan2}\left(y\left(I\right), x\left(I\right)\right)$$

The angle estimation accuracy is $0.3^\circ$, when x(I)=y(I)=0, the corresponding angle(I) is set to 0.

### 3.2.23 cv::ocl::polarToCart

Computes x and y coordinates of 2D vectors from their magnitude and angle.

Supports only CV_32F CV_64F data type.

**magnitude** The source floating-point array of magnitudes of 2D vectors. It can be an empty matrix (=Mat()) - in this case the function assumes that all the magnitudes are =1. If it's not empty, it must have the same size and same type as angle

**angle** The source floating-point array of angles of the 2D vectors

**x** The destination array of x-coordinates of 2D vectors; will have the same size and the same type as angle

**y** The destination array of y-coordinates of 2D vectors; will have the same size and the same type as angle

**angleInDegrees** When it is true, the input angles are measured in degrees, otherwise they are measured in radians

The function polarToCart computes the cartesian coordinates of each 2D vector represented by the corresponding elements of magnitude and angle:

$$\begin{array}{l}
\mathrm{x}\left(I\right) = \mathrm{magnitude}\left(I\right)\cos\left(\mathrm{angle}\left(I\right)\right) \\
\mathrm{y}\left(I\right) = \mathrm{magnitude}\left(I\right)\sin\left(\mathrm{angle}\left(I\right)\right)
\end{array}$$

The relative accuracy of the estimated coordinates is $10^{-6}$.

### 3.2.24 cv::ocl::split

Divides multi-channel array into several single-channel arrays.

Supports all data types.

**mtx** The source multi-channel array

**mv** The destination array or vector of arrays; The number of arrays must match mtx.channels(). The arrays themselves will be reallocated if needed

The functions split split multi-channel array into separate single-channel arrays:

$$\mathrm{mv}\left[c\right]\left(I\right) = \mathrm{mtx}(I)_c$$

### 3.2.25 cv::ocl::subtract

Calculates per-element difference between two arrays or array and a scalar.

Supports all data types except CV_8SC1,CV_8SC2,CV8SC3 and CV_8SC4.

**src1** The first source array

**src2** The second source array. It must have the same size and same type as src1

**sc** Scalar; the first or the second input parameter

**dst** The destination array; it will have the same size and same type as src1

**mask** The optional operation mask, 8-bit single channel array; specifies elements of the destination array to be changed

The functions subtract compute:

- the difference between two arrays:

dst($I$) = saturate(src1($I$) - src2(I)) if mask($I$) $\neq 0$

- the difference between array and a scalar:

dst($I$) = saturate(src1($I$) - sc) if mask($I$) $\neq 0$

- the difference between scalar and an array:

dst($I$) = saturate(sc - src2($I$)) if mask($I$) $\neq 0$

where I is multi-dimensional index of array elements.

The first function in the above list can be replaced with matrix expressions:

### 3.2.26 cv::ocl::transpose

Transposes a matrix.

Supports CV_8UC1, 8UC4, 8SC4, 16UC2, 16SC2, 32SC1 and 32FC1 data types.

**src** The source array

**dst** The destination array of the same type as src

The function cv::transpose transposes the matrix src:

$\mathrm{dst}(i, j) = \mathrm{src}(j,\, i)$

Note that no complex conjugation is done in the case of a complex matrix, it should be done separately if needed.

# 4 2 imgproc. Image Processing

## 4.1 2.1 Histograms

### 4.1.1 cv::ocl::calcHist

Calculates histogram of one or more arrays

Supports only 8UC1 data type.

**arrays** Source arrays. They all should have the same depth, CV 8U, and the same size. Each of them can have an arbitrary number of channels.

**hist** The output histogram, a dense or sparse dims-dimensional array.

## 4.2 2.2 Image Filtering

### 4.2.1 cv::ocl::bilateralFilter

Applies bilateral filter to the image.

Supports 8UC1 8UC4 data types.

**src** The source 8-bit or floating-point, 1-channel or 3-channel image

**dst** The destination image; will have the same size and the same type as src

**d** The diameter of each pixel neighborhood, that is used during filtering. If it is non-positive, it's computed from sigmaSpace

**sigmaColor** Filter sigma in the color space. Larger value of the parameter means that farther colors within the pixel neighborhood (see sigmaSpace) will be mixed together, resulting in larger areas of semi-equal color

**sigmaSpace** Filter sigma in the coordinate space. Larger value of the parameter means that farther pixels will influence each other (as long as their colors are close enough; see sigmaColor). Then d>0, it specifies the neighborhood size regardless of sigmaSpace, otherwise d is proportional to sigmaSpace.

### 4.2.2 cv::ocl::blur

Smoothes image using normalized box filter

Supports data type: CV_8UC1, CV_8UC4, CV_32FC1 and CV_32FC4

**src** The source image

**dst** The destination image; will have the same size and the same type as src

**ksize** The smoothing kernel size

**anchor** The anchor point. The default value Point(-1,-1) means that the anchor is at the kernel center

**borderType** The border mode used to extrapolate pixels outside of the image, the details are shown as follows:

- BORDER_REPLICATE: aaaaaa|abcdefgh|hhhhhhh
- BORDER_REFLECT: fedcba|abcdefgh|hgfedcb
- BORDER_REFLECT_101: gfedcb|abcdefgh|gfedcba
- BORDER_WRAP: cdefgh|abcdefgh|abcdefg
- BORDER_CONSTANT: 0000|abcdefgh|0000

The function smoothes the image using the kernel:

$$\mathrm{K} = \frac{1}{\mathrm{ksize.width} * \mathrm{ksize.height}} \begin{bmatrix} 1 & 1 & 1 & \dots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

The call blur(src, dst, ksize, anchor, borderType) is equivalent to boxFilter(src, dst, src.type( ), anchor, true, borderType).

### 4.2.3 cv::ocl::boxFilter

Smoothes image using box filter

Supports data type: CV_8UC1, CV_8UC4, CV_32FC1 and CV_32FC4.

**src** The source image

**dst** The destination image; will have the same size and the same type as src

**ksize** The smoothing kernel size

**anchor** The anchor point. The default value Point(-1,-1) means that the anchor is at the kernel center

**normalize** Indicates, whether the kernel is normalized by its area or not

**borderType** The border mode used to extrapolate pixels outside of the image, the details are shown as follows:

- BORDER_REPLICATE: aaaaaa|abcdefgh|hhhhhhh
- BORDER_REFLECT: fedcba|abcdefgh|hgfedcb
- BORDER_REFLECT_101: gfedcb|abcdefgh|gfedcba
- BORDER_WRAP: cdefgh|abcdefgh|abcdefg
- BORDER_CONSTANT: 0000|abcdefgh|0000

The function smoothes the image using the kernel:

$$\mathrm{K} = \alpha \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix}$$

where

$$\alpha = \begin{cases} \frac{1}{\text{ksize.width}*\ \text{ksize.height}} & \text{when normalize=true} \\ 1 & \text{otherwise} \end{cases}$$

Unnormalized box filter is useful for computing various integral characteristics over each pixel neighborhood, such as covariation matrices of image derivatives (used in dense optical flow algorithms etc.). If you need to compute pixel sums over variable-size windows, use cv::integral.

### 4.2.4 cv::ocl::copyMakeBorder

Forms a border around the image.

Supports CV_8UC1, CV_8UC4, CV_32SC1 data types.

**src** The source image

**dst** The destination image; will have the same type as src and the size size(src.cols+left+right, src.rows+top+bottom)

**top, bottom, left, right** Specify how much pixels in each direction from the source image rectangle one needs to extrapolate, e.g. top=1, bottom=1, left=1, right=1mean that 1 ixel-wide border needs to be built

**borderType** The border type

**value** The border value if borderType==BORDER CONSTANT

The function copies the source image into the middle of the destination image. The areas to the left, to the right, above and below the copied source image will be filled with extrapolated pixels. This is not what cv::FilterEngine or based on it filtering functions do (they extrapolate pixels on-fly), but what other more complex functions, including your own, may do to simplify image boundary handling.

The function supports the mode when src is already in the middle of dst. In this case the function does not copy src itself, but simply constructs the border, e.g.:

### 4.2.5 cv::ocl::dilate

Dilates an image by using a specific structuring element.

Supports data types: CV_8UC1, CV_8UC4, CV_32FC1 and CV_32FC4.

**src** The source image

**dst** The destination image. It will have the same size and the same type as src

**element** The structuring element used for dilation. If element=Mat(), a 3×3 rectangular structuring element is used

**anchor** Position of the anchor within the element. The default value (-1, -1) means that the anchor is at the element center

**iterations** The number of times dilation is applied

The function dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken:

$$\mathrm{dst}(x, y) = \max_{(x', y'):\mathrm{element}(x', y') \neq 0} \mathrm{src}(x + x', y + y')$$

The function supports the in-place mode. Dilation can be applied several (iterations) times. In the case of multi-channel images each channel is processed independently.

### 4.2.6 cv::ocl::erode

Erodes an image by using a specific structuring element.

Supports data types: CV_8UC1, CV_8UC4, CV_32FC1 and CV_32FC4.

**src** The source image

**dst** The destination image. It will have the same size and the same type as src

**element** The structuring element used for dilation. If element=Mat(), a 3×3 rectangular structuring element is used

**anchor** Position of the anchor within the element. The default value (-1, -1) means that the anchor is at the element center

**iterations** The number of times erosion is applied

The function erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:

$$\mathrm{dst}(x, y) = \min_{(x',y'):\mathrm{element}(x',y')\neq 0} \mathrm{src}(x + x', y + y')$$

The function supports the in-place mode. Erosion can be applied several (iterations) times. In the case of multi-channel images each channel is processed independently.

### 4.2.7 cv::ocl::filter2D

Convolves an image with the kernel

Supports CV_8UC1,CV_8UC4,CV_32SC1,CV_32SC4,CV_32FC1,CV_32FC4

**src** The source image

**dst** The destination image. It will have the same size and the same number of channels as src

**ddepth** The desired depth of the destination image. If it is negative, it will be the same as src.depth()

**kernel** Convolution kernel (or rather a correlation kernel), a single-channel floating point matrix. If you want to apply different kernels to different channels, split the image into separate color planes using cv::split and process them individually

**anchor** The anchor of the kernel that indicates the relative position of a filtered point within the kernel. The anchor should lie within the kernel. The special default value (-1,-1) means that the anchor is at the kernel center

**borderType** The border mode used to extrapolate pixels outside of the image, the details are shown as follows:

- BORDER_REPLICATE: aaaaaa|abcdefgh|hhhhhhh
- BORDER_REFLECT: fedcba|abcdefgh|hgfedcb
- BORDER_REFLECT_101: gfedcb|abcdefgh|gfedcba
- BORDER_WRAP: cdefgh|abcdefgh|abcdefg
- BORDER_CONSTANT: 0000|abcdefgh|0000

The function applies an arbitrary linear filter to the image. In-place operation is supported. When the aperture is partially outside the image, the function interpolates outlier pixel values according to the specified border mode.

The function does actually computes correlation, not the convolution:

$$\mathrm{dst}(x,y) = \sum_{\substack{0 \le x' < \mathrm{kernel.cols} \\ 0 \le y' < \mathrm{kernel.rows}}} \mathrm{kernel}(x',y') * \mathrm{src}(x+x'-\mathrm{anchor.x}, y+y'-\mathrm{anchor.y})$$

That is, the kernel is not mirrored around the anchor point. If you need a real convolution, flip the kernel using cv::flip and set the new anchor to (kernel.cols - anchor.x - 1, kernel.rows - anchor.y - 1).

The function uses DFT-based algorithm in case of sufficiently large kernels ( 11×11) and the direct algorithm for small kernels.

### 4.2.8 cv::ocl::GaussianBlur

Smoothes image using a Gaussian filter

Supports CV_8UC1,CV_8UC4,CV_32SC1,CV_32SC4,CV_32FC1,CV_32FC4

**src** The source image

**dst** The destination image; will have the same size and the same type as src

**ksize** The Gaussian kernel size; ksize.width and ksize.height can differ, but they both must be positive and odd. Or, they can be zero's, then they are computed from sigma*

**sigmaX, sigmaY** The Gaussian kernel standard deviations in X and Y direction. If sigmaY is zero, it is set to be equal to sigmaX. If they are both zeros, they are computed from ksize.width and ksize.height. To fully control the result regardless of possible future modification of all this semantics, it is recommended to specify all of ksize, sigmaX and sigmaY

**borderType** The border mode used to extrapolate pixels outside of the image, the details are shown as follows:

- BORDER_REPLICATE: aaaaaa|abcdefgh|hhhhhhh
- BORDER_REFLECT: fedcba|abcdefgh|hgfedcb
- BORDER_REFLECT_101: gfedcb|abcdefgh|gfedcba
- BORDER_WRAP: cdefgh|abcdefgh|abcdefg
- BORDER_CONSTANT: 0000|abcdefgh|0000

The function convolves the source image with the specified Gaussian kernel. In-place filtering is supported.

### 4.2.9 cv::ocl::Laplacian

Calculates the Laplacian of an image

Supports only ksize = 1 and ksize = 3 8UC1 8UC4 32FC1 32FC4 data types.

**src** Source image

**dst** Destination image; will have the same size and the same number of channels as src

**ddepth** The desired depth of the destination image

**ksize** The aperture size used to compute the second-derivative filters. It must be positive and odd

**scale** The optional scale factor for the computed Laplacian values (by default, no scaling is applied

The function calculates the Laplacian of the source image by adding up the second x and y derivatives calculated using the Sobel operator:

$$\text{dst} = \Delta\text{src} = \frac{\partial^2\text{src}}{\partial x^2} + \frac{\partial^2\text{src}}{\partial y^2}$$

This is done when ksize > 1. When ksize == 1, the Laplacian is computed by filtering the image with the following 3×3 aperture:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

### 4.2.10 cv::ocl:: medianFilter

Smoothes image using median filter

Supports CV_8UC1,CV_8UC4,CV_32FC1,CV_32FC4.

**src** The source 1-, 3- or 4-channel image. When ksize is 3 or 5, the image depth should be CV 8U or CV 32F. For larger aperture sizes it can only be CV 8U

**dst** The destination array; will have the same size and the same type as src

**ksize** The aperture linear size. It must be odd and more than 1, i.e. 3, 5, 7 ...

The function smoothes image using the median filter with ksize$\times$ksize aperture. Each channel of a multi-channel image is processed independently. Inplace operation is supported.

### 4.2.11 cv::ocl::morphologyEx

Performs advanced morphological transformations.

Supports CV_8UC1,CV_8UC4,CV_32SC1,CV_32SC4,CV_32FC1,CV_32FC4

**src** Source image

**dst** Destination image. It will have the same size and the same type as src

**element** Structuring element

**op** Type of morphological operation, one of the following:

**MORPH OPEN** opening

**MORPH CLOSE** closing

**MORPH GRADIENT** morphological gradient

**MORPH TOPHAT** "top hat"

**MORPH BLACKHAT** "black hat"

**iterations** Number of times erosion and dilation are applied

The function can perform advanced morphological transformations using erosion and dilation as basic operations.

Opening:

$$\text{dst} = \text{open(src, element)} = \text{dilate(erode(src, element))}$$

Closing:

$$\text{dst} = \text{close(src, element)} = \text{erode(dilate(src, element))}$$

Morphological gradient:

$$\text{dst} = \text{morph\_grad(src, element)} = \text{dilate(src, element) - erode(src, element)}$$

"Top hat":

$$\text{dst} = \text{tophat(src, element)} = \text{src - open(src, element)}$$

"Black hat":

$$\text{dst} = \text{blackhat(src, element)} = \text{close(src, element) - src}$$

Any of the operations can be done in-place.

### 4.2.12 cv::ocl::Scharr

Calculates the first x- or y- image derivative using Scharr operator

Supports CV_8UC1,CV_8UC4,CV_32SC1,CV_32SC4,CV_32FC1,CV_32FC4.

**src** The source image

**dst** The destination image; will have the same size and the same number of channels as src

**ddepth** The destination image depth

**xorder** Order of the derivative x

**yorder** Order of the derivative y

**scale** The optional scale factor for the computed derivative values (by default, no scaling is applied)

**delta** The optional delta value, added to the results prior to storing them in dst

**borderType** The border mode used to extrapolate pixels outside of the image, the details are shown as follows:

- BORDER_REPLICATE: aaaaaa|abcdefgh|hhhhhhh
- BORDER_REFLECT: fedcba|abcdefgh|hgfedcb
- BORDER_REFLECT_101: gfedcb|abcdefgh|gfedcba
- BORDER_WRAP: cdefgh|abcdefgh|abcdefg
- BORDER_CONSTANT: 0000|abcdefgh|0000

The function computes the first x- or y- spatial image derivative using Scharr operator. The call

Scharr(src, dst, ddepth, xorder, yorder, scale, delta, borderType).

is equivalent to

Sobel(src, dst, ddepth, xorder, yorder, CV SCHARR, scale, delta, borderType).

### 4.2.13 cv::ocl::sepFilter2D

Applies separable linear filter to an image

Supports CV_8UC1,CV_8UC4,CV_32SC1,CV_32SC4,CV_32FC1,CV_32FC4

**src** The source image

**dst** The destination image; will have the same size and the same number of channels as src

**ddepth** The destination image depth

**rowKernel** The coefficients for filtering each row

**columnKernel** The coefficients for filtering each column

**anchor** The anchor position within the kernel; The default value (-1, 1) means that the anchor is at the kernel center

**delta** The value added to the filtered results before storing them

**borderType** The border mode used to extrapolate pixels outside of the image, the details are shown as follows:

- BORDER_REPLICATE: aaaaaa|abcdefgh|hhhhhhh
- BORDER_REFLECT: fedcba|abcdefgh|hgfedcb
- BORDER_REFLECT_101: gfedcb|abcdefgh|gfedcba
- BORDER_WRAP: cdefgh|abcdefgh|abcdefg
- BORDER_CONSTANT: 0000|abcdefgh|0000

The function applies a separable linear filter to the image. That is, first, every row of src is filtered with 1D kernel rowKernel. Then, every column of the result is filtered with 1D kernel columnKernel and the final result shifted by delta is stored in dst.

### 4.2.14 cv::ocl::Sobel

Calculates the first, second, third or mixed image derivatives using an extended Sobel operator

Supports CV_8UC1,CV_8UC4,CV_32SC1,CV_32SC4,CV_32FC1,CV_32FC4

**src** The source image

**dst** The destination image; will have the same size and the same number of channels as src

**ddepth** The destination image depth

**xorder** Order of the derivative x

**yorder** Order of the derivative y

**ksize** Size of the extended Sobel kernel, must be 1, 3, 5 or 7

**scale** The optional scale factor for the computed derivative values (by default, no scaling is applied)

**delta** The optional delta value, added to the results prior to storing them in dst

**borderType** The border mode used to extrapolate pixels outside of the image, the details are shown as follows:

- BORDER_REPLICATE: aaaaaa|abcdefgh|hhhhhhh
- BORDER_REFLECT: fedcba|abcdefgh|hgfedcb
- BORDER_REFLECT_101: gfedcb|abcdefgh|gfedcba
- BORDER_WRAP: cdefgh|abcdefgh|abcdefg
- BORDER_CONSTANT: 0000|abcdefgh|0000

In all cases except 1, an ksize×ksize separable kernel will be used to calculate the derivative. When ksize = 1, a3×1 or 1×3 kernel will be used (i.e. no Gaussian smoothing is done). ksize = 1 can only be used for the first or the second x- or y- derivatives.

There is also the special value ksize = CV SCHARR (-1) that corresponds to a 3×3 Scharr filter that may give more accurate results than a 3×3 Sobel. The Scharr aperture is

$$\begin{bmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{bmatrix}$$

for the x-derivative or transposed for the y-derivative.

The function calculates the image derivative by convolving the image with the appropriate kernel:

$$\mathrm{dst} = \frac{\partial^{xorder+yorder}\mathrm{src}}{\partial x^{xorder}\partial y^{yorder}}$$

The Sobel operators combine Gaussian smoothing and differentiation, so the result is more or less resistant to the noise. Most often, the function is called with (xorder = 1, yorder = 0, ksize = 3) or (xorder = 0, yorder = 1, ksize = 3) to calculate the first x- or y- image derivative. The first case corresponds to a kernel of:

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

and the second one corresponds to a kernel of:

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

## 4.3 2.3 Geometric Image Transformations

### 4.3.1 cv::ocl::resize

Resizes an image.

Supports CV_8UC1, CV_8UC4, CV_32FC1 and CV_32FC4 data types.

**src** Source image

**dst** Destination image. It will have size dsize (when it is non-zero) or the size computed from src.size() and fx and fy. The type of dst will be the same as of src.

**dsize** The destination image size. If it is zero, then it is computed as:

dsize = Size(round(fx*src.cols), round(fy*src.rows))

. Either dsize or both fx or fy must be non-zero.

**fx** The scale factor along the horizontal axis. When 0, it is computed as (double)dsize.width/src.cols

**fy** The scale factor along the vertical axis. When 0, it is computed as (double)dsize.height/src.rows

**interpolation** The interpolation method:

**INTER NEAREST** nearest-neighbor interpolation

**INTER LINEAR** bilinear interpolation (used by default)

**INTER AREA** resampling using pixel area relation. It may be the preferred method for image decimation, as it gives moire-free results. But when the image is zoomed, it is similar to the INTER NEAREST method

**INTER CUBIC** bicubic interpolation over 4x4 pixel neighborhood

**INTER LANCZOS4** Lanczos interpolation over 8x8 pixel neighborhood

The function resize resizes an image src down to or up to the specified size. Note that the initial dst type or size are not taken into account. Instead the size and type are derived from the src, dsize, fx and fy. If you want to resize src so that it fits the pre-created dst, you may call the function as:

If you want to decimate the image by factor of 2 in each direction, you can call the function this way:

### 4.3.2 cv::ocl::warpAffine

Applies an affine transformation to an image.

Supports INTER_NEAREST, INTER_LINEAR, INTER_CUBIC types.

**src** Source image

**dst** Destination image; will have size dsize and the same type as src

**M** 2×3 transformation matrix

**dsize** Size of the destination image

**flags** A combination of interpolation methods, see cv::resize, and the optional flag WARP INVERSE MAP that means that M is the inverse transformation (dst→src)

The function warpAffine transforms the source image using the specified matrix:

$$\mathrm{dst}(x, y) = \mathrm{src}(\mathrm{M}_{11}x + \mathrm{M}_{12}y + \mathrm{M}_{13}, \mathrm{M}_{21}x + \mathrm{M}_{22}y + \mathrm{M}_{23})$$

when the flag WARP INVERSE MAP is set. Otherwise, the transformation is first inverted with cv::invertAffineTransform and then put in the formula above instead of M. The function can not operate in-place.

### 4.3.3 cv::ocl::warpPerspective

Applies a perspective transformation to an image.

Supports INTER_NEAREST, INTER_LINEAR, INTER_CUBIC types.

**src** Source image

**dst** Destination image; will have size dsize and the same type as src

**M** 3×3 transformation matrix

**dsize** Size of the destination image

**flags** A combination of interpolation methods, see cv::resize, and the optional flag WARP INVERSE MAP that means that M is the inverse transformation (dst→src)

The function warpPerspective transforms the source image using the specified matrix:

$$\mathrm{dst}(x, y) = \mathrm{src}(\frac{\mathrm{M}_{11}x + \mathrm{M}_{12}y + \mathrm{M}_{13}}{\mathrm{M}_{31}x + \mathrm{M}_{32}y + \mathrm{M}_{33}}, \frac{\mathrm{M}_{21}x + \mathrm{M}_{22}y + \mathrm{M}_{23}}{\mathrm{M}_{31}x + \mathrm{M}_{32}y + \mathrm{M}_{33}})$$

when the flag WARP INVERSE MAP is set. Otherwise, the transformation is first inverted with cv::invert and then put in the formula above instead of M. The function can not operate in-place.

## 4.4 2.4 Miscellaneous Image Transformations

### 4.4.1 cv::ocl::cvtColor

Converts image from one color space to another.

Supports.CV_8UC1,CV_8UC4,CV_32SC1,CV_32SC4,CV_32FC1,CV_32FC4

**src** The source image, 8-bit unsigned, 16-bit unsigned (CV 16UC...) or single-precision floatingpoint

**dst** The destination image; will have the same size and the same depth as src

**code** The color space conversion code; see the discussion

**dstCn** The number of channels in the destination image; if the parameter is 0, the number of the channels will be derived automatically from src and the code

The function converts the input image from one color space to another. In the case of transformation to-from RGB color space the ordering of the channels should be specified explicitly (RGB or BGR).

The conventional ranges for R, G and B channel values are:

- 0 to 255 for CV_8U inages
- 0 to 65535 for CV_16U images and
- 0 to 1 for CV_32F images.

Of course, in the case of linear transformations the range does not matter, but in the non-linear cases the input RGB image should be normalized to the proper value range in order to get the correct results, e.g. for RGB→L*u*v* transformation. For example, if you have a 32-bit floatingpoint image directly converted from 8-bit image without any scaling, then it will have 0..255 value range, instead of the assumed by the function 0..1. So, before calling cvtColor, you need first to scale the image down:

The function can do the following transformations:

- Transformations within RGB space like adding/removing the alpha channel, reversing the channel order, conversion to/from 16-bit RGB color (R5:G6:B5 or R5:G5:B5), as well as

$$\text{RGB}\,[\text{A}]\ \text{to Gray: } Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

and

$$\text{Gray to RGB}\,[\text{A}] : R \leftarrow Y, G \leftarrow Y, B \leftarrow Y, A \leftarrow 0$$

The conversion from a RGB image to gray is done with:

- RGB←→CIE XYZ.Rec709 with D65 white point (CV_BGR2XYZ, CV_RGB2XYZ, CV_XYZ2BGR, CV XYZ2RGB):

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} \leftarrow \begin{bmatrix} 3.240479 & -1.53715 & -0.498535 \\ -0.969256 & 1.875991 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

X, Y and Z cover the whole value range (in the case of floating-point images Z may exceed 1).

- RGB←→YCrCb JPEG (a.k.a. YCC) (CV_BGR2YCrCb, CV_RGB2YCrCb, CV_YCrCb2BGR, CV_YCrCb2RGB)

$$\begin{gathered} Y \leftarrow 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \\ Cr \leftarrow (R - Y) \cdot 0.713 + delta \\ Cb \leftarrow (B - Y) \cdot 0.564 + delta \\ R \leftarrow Y = 1.403 \cdot (Cr - delta) \\ G \leftarrow Y - 0.344 \cdot (Cr - delta) - 0.714 \cdot (Cb - delta) \\ B \leftarrow Y + 1.773 \cdot (Cb - delta) \end{gathered}$$

where

$$delte = \begin{cases} 128 & \text{for 8-bit images} \\ 32768 & \text{for16-bit images} \\ 0.5 & \text{for floating-point images} \end{cases}$$

Y, Cr and Cb cover the whole value range.

- RGB←→HSV (CV_BGR2HSV, CV_RGB2HSV, CV_HSV2BGR, CV_HSV2RGB) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$Y \leftarrow max(R, G, B)$$

$$S \leftarrow \begin{cases} \frac{V - \min(R, G.B)}{V} & \text{if } V \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$H \leftarrow \begin{cases} 60(G - B/S) & \text{if } V = R \\ 120 + 60(B - R)/S & \text{if } V = G \\ 240 + 60(R - G)/S & \text{if } V = B \end{cases}$$

if H < 0 then H←H + 360 On output 0≤L≤1, 0≤S≤1, 0≤H≤360.

The values are then converted to the destination data type:

**8-bit images**

$V \leftarrow 255 \cdot V$, $S \leftarrow 255 \cdot S$, $H \leftarrow H/2$ (to fit to 0 to 255)

**16-bit images (currently not supported)**

$V <$ -65535$\cdot V$, $S \quad <$ -65535$\cdot S$, $H \quad < -H$

**32-bit images** H, S, V are left as is

- RGB←→CIE L*a*b* (CV_BGR2Lab, CV_RGB2Lab, CV_Lab2BGR, CV_Lab2RGB) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit the 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$X \leftarrow X/Xn$, where $Xn$=0.950456
$Z \leftarrow Z/Zn$, where $Zn$=1.088754

$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \leq 0.008856 \end{cases}$$

$$a \leftarrow 500(f(X) - f(Y)) + delta$$
$$b \leftarrow 200(f(Y) - f(Z)) + delta$$

where

$$f(t) = \begin{cases} t^{1/3} & \text{for } t > 0.008856 \\ 7.787t + 16/116 & \text{for } t \leq 0.008856 \end{cases}$$

and

$$delta = \begin{cases} 128 & \text{for 8-bit images} \\ 0 & \text{for floating-point images} \end{cases}$$

On output 0≤ $L$ ≤100, -127≤ $a$ ≤127, -127≤ $b$ ≤127
The values are then converted to the destination data type:
**8-bit images**
L←L*255/100, a←a + 128, b←b + 128
**16-bit images** currently not supported
**32-bit images** L, a, b are left as is

- RGB←→CIE L*u*v* (CV_BGR2Luv, CV_RGB2Luv, CV_Luv2BGR, CV_Luv2RGB) in the case of 8-bit and 16-bit images R, G and B are converted to floating-point format and scaled to fit 0 to 1 range

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

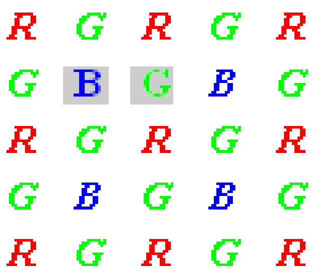
$$L \leftarrow \begin{cases} 116 * Y^{1/3} - 16 & \text{for } Y > 0.008856 \\ 903.3 * Y & \text{for } Y \leq 0.008856 \end{cases}$$

$$u' \leftarrow 4 * X/(X + 15 * Y + 3Z)$$
$$v' \leftarrow 9 * Y/(X + 15 * Y + 3Z)$$
$$u \leftarrow 13 * L * (u' - u_n) \quad \text{where} \quad u_n = 0.19793943$$
$$v \leftarrow 13 * L * (v' - v_n) \quad \text{where} \quad v_n = 0.46831096$$

On output 0≤ $L$ ≤100, -134≤ $u$ ≤220, -140≤ $v$ ≤122.

The values are then converted to the destination data type:
**8-bit images**
$L \leftarrow 255/100L$, $u \leftarrow 255/354(u + 134)$, $v \leftarrow 255/256(v + 140)$
**16-bit images** currently not supported
**32-bit images** L, u, v are left as is
The above formulas for converting RGB to/from various color spaces have been taken from multiple sources on Web, primarily from the Charles Poynton site http://www.poynton. com/ColorFAQ.html

- Bayer→RGB (CV_BayerBG2BGR, CV_BayerGB2BGR, CV_BayerRG2BGR, CV_BayerGR2BGR, CV_BayerBG2RGB, CV_BayerGB2RGB, CV_BayerRG2RGB, CV_BayerGR2RGB) The Bayer pattern is widely used in CCD and CMOS cameras. It allows one to get color pictures from a single plane where R,G and B pixels (sensors of a particular component) are interleaved like this:



The output RGB components of a pixel are interpolated from 1, 2 or 4 neighbors of the pixel having the same color. There are several modifications of the above pattern that can be achieved by shifting the pattern one pixel left and/or one pixel up. The two letters C1 and C2 in the conversion constants CV Bayer C1C2 2BGR and CV Bayer C1C2 2RGB indicate the particular pattern type - these are components from the second row, second and third columns, respectively. For example, the above pattern has very popular "BG" type.

### 4.4.2 cv::ocl::integral

Calculates the integral of an image.

Supports only CV_8UC1 source type.

**image** The source image, $W \times H$, 8-bit or floating-point (32f or 64f)

**sum** The integral image, $(W + 1)\times(H + 1)$, 32-bit integer or floating-point (32f or 64f)

**sqsum** The integral image for squared pixel values, $(W + 1)\times$ $(H + 1)$, double precision floatingpoint (64f)

The functions integral calculate one or more integral images for the source image as following:

$$\text{sum}(X,Y) = \sum_{x<X,y<Y} \text{image}(x,y)$$

$$\text{sqsum}(X,Y) = \sum_{x<X,y<Y} \text{image}(x,y)^2$$

$$\text{tilted}(X,Y) = \sum_{y<Y,abs(x-X+1)\leq Y-y-1} \text{image}(x,y)$$

Using these integral images, one may calculate sum, mean and standard deviation over a specific up-right or rotated rectangular region of the image in a constant time, for example:

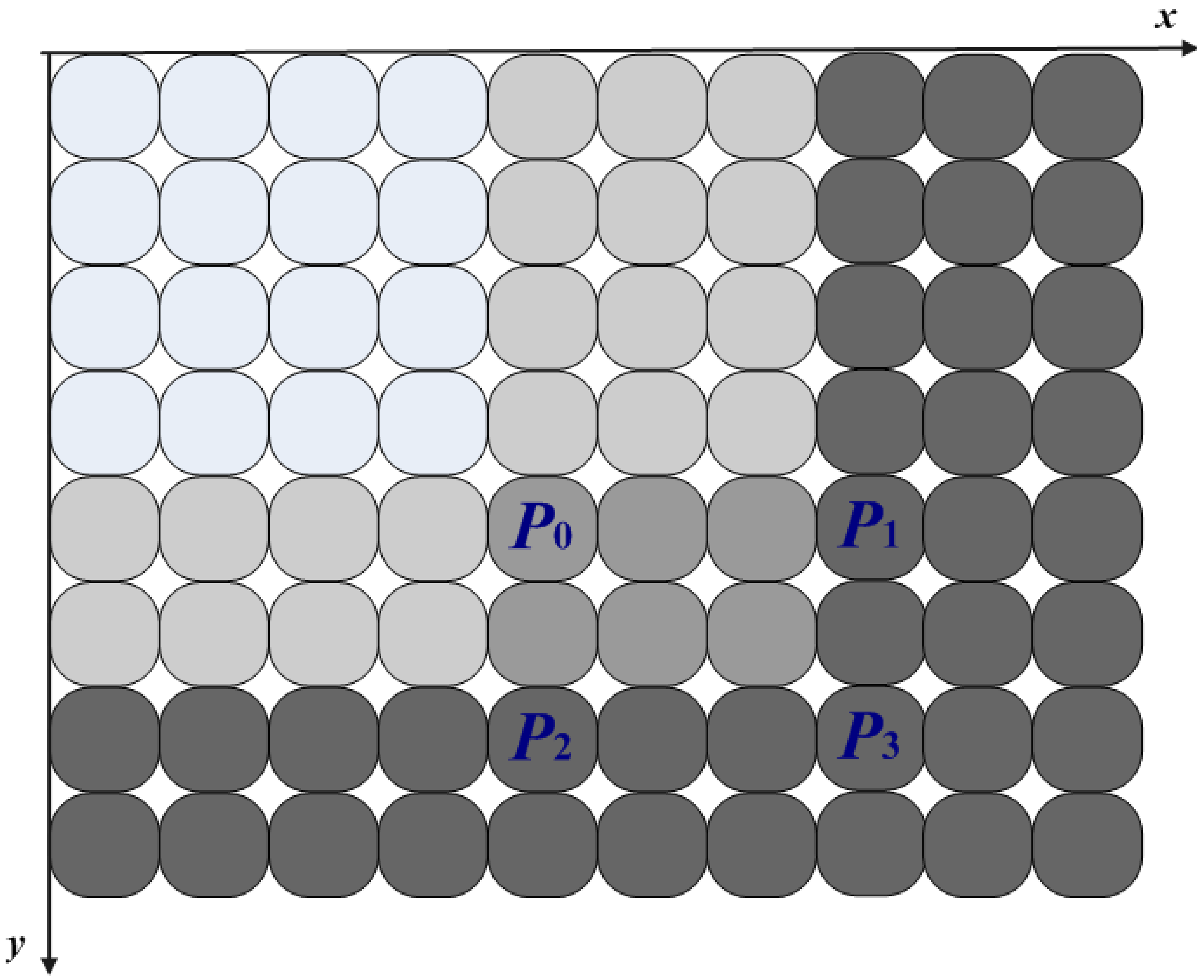
$$\sum_{x_1\leq x<x_2,y_1\leq y<y_2} \text{image}(x,y) = \text{sum}(x_2,y_2)-\text{sum}(x_1,y_2)-\text{sum}(x_2,y_1)+\text{sum}(x_1,y_1)$$

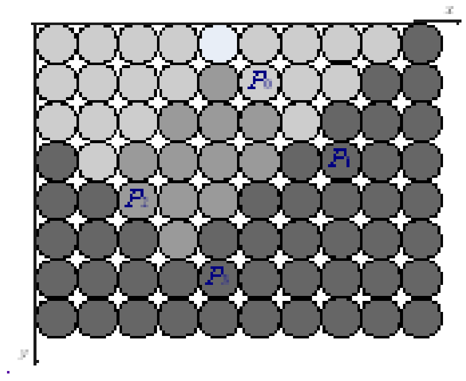
It makes possible to do a fast blurring or fast block correlation with variable window size, for example. In the case of multi-channel images, sums for each channel are accumulated independently.

As a practical example, the next figure shows the calculation of the integral of a straight rectangle Rect(3,3,3,2) and of a tilted rectangle Rect(5,1,2,3). The selected pixels in the original image are shown, as well as the relative pixels in the integral images sum and tilted.

$P_0 = \{y, x\} = \{4, 4\}$
$P_1 = \{y, x + w\} = \{4, 7\}$
$P_2 = \{y + h, x\} = \{6, 4\}$
$P_3 = \{y + h, x + w\} = \{6, 7\}$

$P_0 = \{y, x\} = \{1, 5\}$
$P_1 = \{y + h, x + w\} = \{3, 7\}$
$P_2 = \{y + h, x - h\} = \{4, 2\}$
$P_3 = \{y + w + h, x + w - h\} = \{6, 4\}$


x
y
$P_0$
$P_1$
$P_2$
$P_3$



### 4.4.3 cv::ocl::threshold

Applies a fixed-level threshold to each array element

Supports CV_8UC1 and CV_32FC1 data types.

**src** Source array (single-channel, 8-bit of 32-bit floating point)

**dst** Destination array; will have the same size and the same type as src

**thresh** Threshold value

**maxVal** Maximum value to use with THRESH BINARY and THRESH BINARY INV thresholding types

**thresholdType** Thresholding type (see the discussion)

The function applies fixed-level thresholding to a single-channel array. The function is typically used to get a bi-level (binary) image out of a grayscale image or for removing a noise, i.e. filtering out pixels with too small or too large values. There are several types of thresholding that the function supports that are determined by thresholdType:

**THRESH BINARY**

$$\mathrm{dst}(x,y) = \begin{cases} \max \mathrm{Val} & \text{if src(x,y) ¿ thresh} \\ 0 & \text{otherwise} \end{cases}$$

**THRESH BINARY INV**

$$\mathrm{dst}(x,y) = \begin{cases} 0 & \text{if src(x,y) ¿ thresh} \\ \max \mathrm{Val} & \text{otherwise} \end{cases}$$

**THRESH TRUNC**

$$\mathrm{dst}(x,y) = \begin{cases} \text{threshold} & \text{if src(x,y) ¿ thresh} \\ \text{src(x,y)} & \text{otherwise} \end{cases}$$

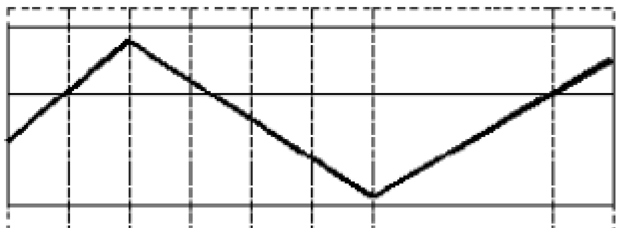
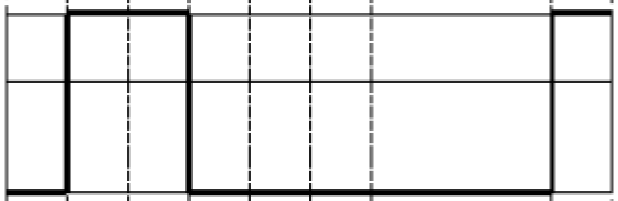
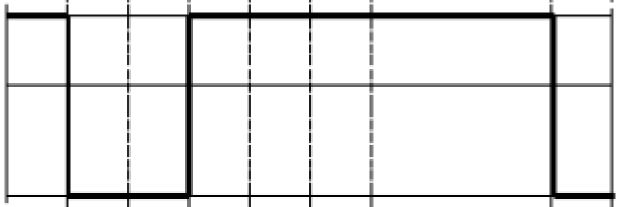
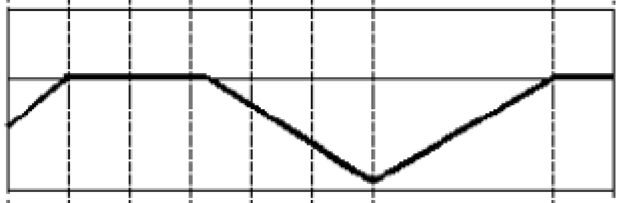
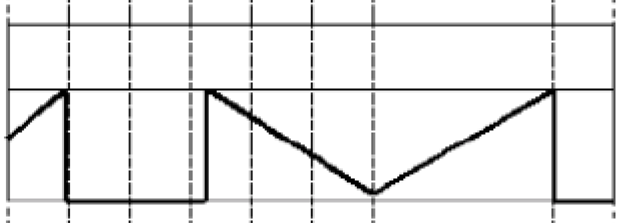
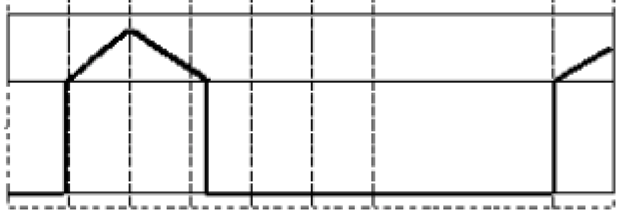
**THRESH TOZERO**

$$\mathrm{dst}(x,y) = \begin{cases} \text{src(x,y)} & \text{if src(x,y) ¿ thresh} \\ 0 & \text{otherwise} \end{cases}$$

**THRESH TOZERO INV**

$$\mathrm{dst}(x,y) = \begin{cases} 0 & \text{if src(x,y) ¿ thresh} \\ \text{src(x,y)} & \text{otherwise} \end{cases}$$

Also, the special value THRESH OTSU may be combined with one of the above values. In this case the function determines the optimal threshold value using Otsu's algorithm and uses it instead of the specified thresh. The function returns the computed threshold value. Currently, Otsu's method is implemented only for 8-bit images.

| | |
|---|---|
|  | Value and Threshold Level |
|  | Threshold Binary |
|  | Threshold Binary, Inverted |
|  | Truncate |
|  | Threshold to Zero, Inverted |
|  | Threshold to Zero |