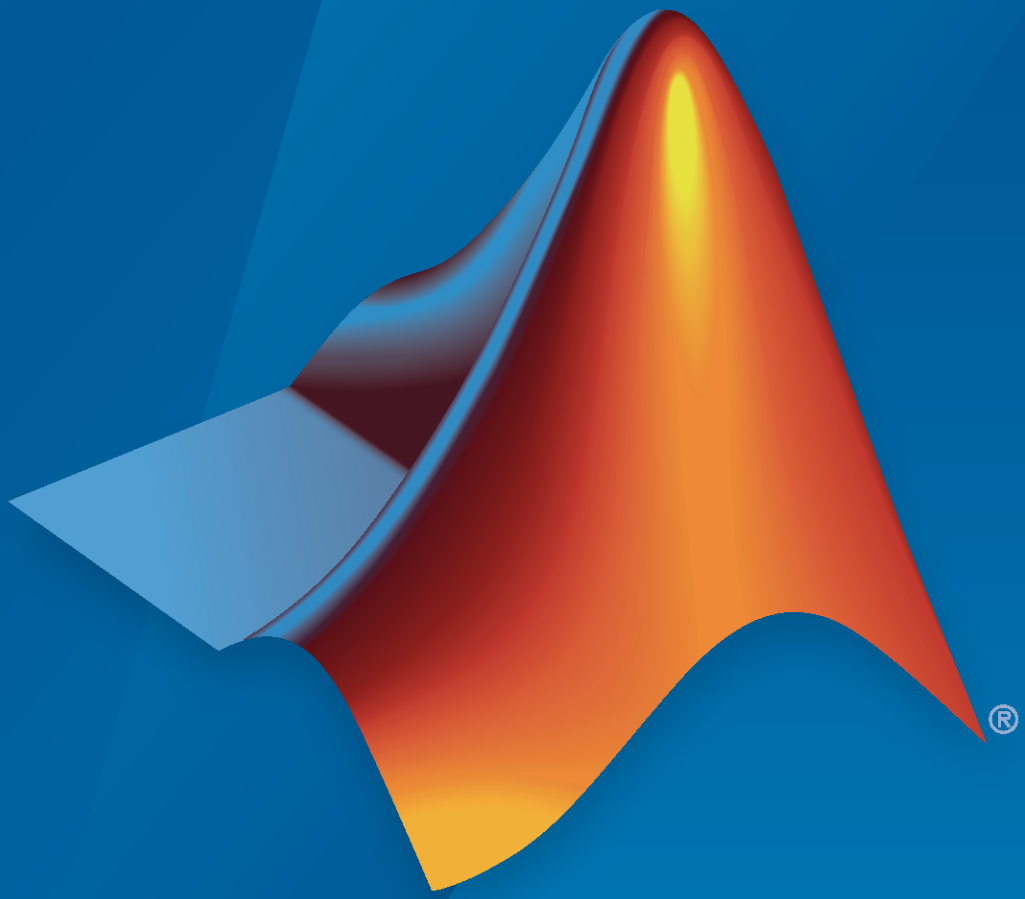
Vision HDL Toolbox™ Release Notes



MATLAB®

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

Sales and services: `www.mathworks.com/sales_and_services`

User community: `www.mathworks.com/matlabcentral`

Technical support: `www.mathworks.com/support/contact_us`

Phone: 508-647-7000

The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Vision HDL Toolbox™ Release Notes

Contents

R2023a

R2022b

R2022a

R2021b

R2021a

R2020b

R2020a

R2019b

R2019a

R2018b

R2018a

R2017b

R2017a

R2016b

R2016a

R2015b

R2023a

Version: 2.7

New Features

Bug Fixes

Downsample images

The Image Resizer block resizes images by two-dimensional downsampling of the input frames. The block offers bilinear or nearest neighbor interpolation and an antialiasing filter. The “Change Image Size” example is updated to use the new block.

This functionality is also available as a `visionhdl.ImageResizer` System object™.

High frame rate object tracking

The “Object Tracking using 2-D FFT” example uses a 2-D FFT to track the location of an object in TIR video frames. The example supports 128 fps for frames up to 1080p. This example requires the DSP HDL Toolbox™ product.

Lidar ground plane segmentation

The “Ground Plane Segmentation of Lidar Data on FPGA” example shows how to identify ground points in a lidar image. Ground plane removal is a common preprocessing step for point-cloud classification algorithms and simultaneous localization and mapping (SLAM) workflows.

Preprocess HDMI video input for YOLO v2 deep learning network, and postprocess images on ARM

The “Integrate YOLO v2 Vehicle Detector System on SoC” example extends the “Deploy and Verify YOLO v2 Vehicle Detector on FPGA” example by targeting the postprocessing logic to the ARM processor of an SoC device.

The “YOLO v2 Vehicle Detector with Live Camera Input on Zynq-Based Hardware” example adds live HDMI video input and output to the “Integrate YOLO v2 Vehicle Detector System on SoC” example, and shows how to deploy the design to the Xilinx® Zynq® UltraScale+™ MPSoC ZCU102 Evaluation Kit. This example requires Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware. The example uses a new `RGB for DL Processor` reference design. The reference design passes the HDMI input to the preprocessing logic and also writes the input frame to PS DDR. After preprocessing, the design writes the resized and normalized images to PL DDR where the DL processor can access it. After the DL processor writes the output back to DDR, the postprocessing code reads the output frames to calculate and overlay bounding boxes. You can view the modified frames from the HDMI output on the board or by using the Video Capture HDMI block in Simulink®.

Edge Detector, Median Filter, ROI Selector, and Lookup Table support two pixels per clock streaming

In R2022b, the Edge Detector, Median Filter, ROI Selector, and Lookup Table blocks supported 4- or 8-pixel vector input. For R2023a, they also support 2-pixel vector input.

R2022b

Version: 2.6

New Features

Bug Fixes

Customize CLAHE example parameters

The Contrast Limited Adaptive Histogram Equalization example was released in R2019b and implements adaptive histogram equalization for hardware. This algorithm clips the peak histogram values and redistributes them across neighboring tiles by using a bilinear interpolation filter. In R2022b, the example model provides parameters for the number of tiles (from to 2 through 16), the number of histogram bins (from 32 through 4096), and the input pixel bit width (from 8 to 16 bits). You can also change the input image size and the algorithm adjusts the tile sizes based on the new dimensions.

Debug YOLO v2 vehicle detector design on hardware (requires Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware)

The Debug YOLO v2 Vehicle Detector on FPGA example shows how to debug a design on hardware by visualizing internal signals from a vehicle detector design deployed on the Xilinx Zynq UltraScale+ MPSoC ZCU102 board. The example uses the FPGA data capture and AXI manager features of HDL Verifier™ Support Package for Xilinx FPGA Boards to set triggers and capture the signals of interest into MATLAB®. The debug example extends the Deploy and Verify YOLO v2 Vehicle Detector on FPGA example.

Stream high frame rate video from MIPI camera interface to Simulink (requires Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware)

Use the Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware to capture video from a LI-IMX274MIPI-FMC module connected to a Xilinx Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit. The MIPI reference design supports frame rates up to 120 fps and resolutions up to 1080p. Explore your video processing design in Simulink and target the design to the FPGA.

Multipixel Streaming: Implement gamma correction algorithm for high-frame-rate video

The Gamma Corrector block now supports multipixel streams. The HDL implementation replicates the algorithm for each pixel in parallel.

The block supports input and output column vectors of *NumPixels* values. *NumPixels* can be 2, 4, or 8. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Histogram and Pixel Stream Aligner support two pixels per clock streaming

In R2022a, the Histogram block supported 4- or 8-pixel vector input. For R2022b, it also supports 2-pixel vector input.

In R2022a, the Pixel Stream Aligner block supported 4- or 8-pixel vector input. For R2022b, it also supports 2-pixel vector input.

R2022a

Preprocess Input Images for YOLOv2 Deep Learning Network (requires Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware)

The Deploy and Verify YOLO v2 Vehicle Detector on FPGA example shows how to resize input images and normalize their pixel value ranges to meet the requirements of the deep learning network. The example deploys the design to FPGA by using Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware.

Image Warp Example

The Image Warp example shows how to implement geometric transform algorithms suitable for image processing on FPGAs.

Image Undistortion Example

The Image Undistortion example shows how to remove lens distortion with an implementation suitable for FPGAs.

Video Stabilization Example

The Video Stabilization example implements a feature-based algorithm suitable for FPGAs. This algorithm helps reduce shaking between frames. The design includes these vision processing steps.

- FAST feature detector
- BRIEF feature descriptors
- Feature matching
- RANSAC affine transform estimation

Multipixel Streaming: Implement histogram and bilateral filter algorithms for high-frame-rate video

The Histogram and Bilateral Filter blocks now support multipixel streams.

The HDL implementation replicates the algorithm for each pixel in parallel.

The blocks support input column vectors of *NumPixels* values. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix. The Bilateral Filter returns an output vector of the same size as the input vector. The Histogram block output interface is the same whether the input is scalar or vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Two Pixels Per Clock Streaming

The Frame To Pixels and Pixels To Frame blocks now support multipixel streams that have 2 pixels per clock cycle. These Vision HDL Toolbox blocks also accept and return 2-pixel streams.

- Line Buffer
- Pixel Stream FIFO
- Image Filter
- Demosaic Interpolator
- Bilateral Filter
- Color Space Converter

Functionality being removed or changed

Change in default frame dimensions of visionhdl.FrameToPixels and visionhdl.PixelsToFrame System objects

Behavior change

The default values of the custom formats have changed. In previous releases, the default dimensions of the `visionhdl.FrameToPixels` and `visionhdl.PixelsToFrame` System objects matched the 1080p format. Starting in R2022a, the default values match the default values of the Frame To Pixels and Pixels To Frame blocks, which define a small custom frame as shown in this code.

```
a = visionhdl.FrameToPixels('VideoFormat','Custom')

a =

  visionhdl.FrameToPixels with properties:

         NumComponents: 1
             NumPixels: 1
           VideoFormat: 'Custom'
    ActivePixelsPerLine: 32
       ActiveVideoLines: 18
     TotalPixelsPerLine: 42
        TotalVideoLines: 28
     StartingActiveLine: 6
             FrontPorch: 5
```

This change means you might need to specify additional dimensions for custom formats where previously the default value resulted in a valid format.

These objects now update frame dimension properties when you select a standard format. This change means you can refer to the object properties to determine the dimensions being used by the format.

```
a = visionhdl.FrameToPixels('VideoFormat','480p')

a =

  visionhdl.FrameToPixels with properties:

    NumComponents: 1
        NumPixels: 1
      VideoFormat: '480p'

TPPL480 = a.TotalPixelsPerLine
```

```
ans =

   800
```

Both these changes also apply to the `visionhdl.FILFrameToPixels` and `visionhdl.FILPixelsToFrame` System objects.

R2021b

Version: 2.4

New Features

Bug Fixes

Multipixel-Multicomponent Streaming: Implement color space conversion and demosaic interpolation algorithms for high-frame-rate color video

The Color Space Converter and Demosaic Interpolator blocks now support multipixel-multicomponent streams.

The HDL implementation replicates the algorithm for each pixel in parallel.

The Color Space Converter block supports input matrices of *NumPixels*-by-3 values, and output matrices of *NumPixels*-by-*NumComponents* values, where *NumComponents* is 3 or 1. The Demosaic Interpolator block accepts an input vector of *NumPixels*-by-1 values and returns an output matrix of *NumPixels*-by-3 values. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Image Normalization Example: Scale pixel values to specified output range

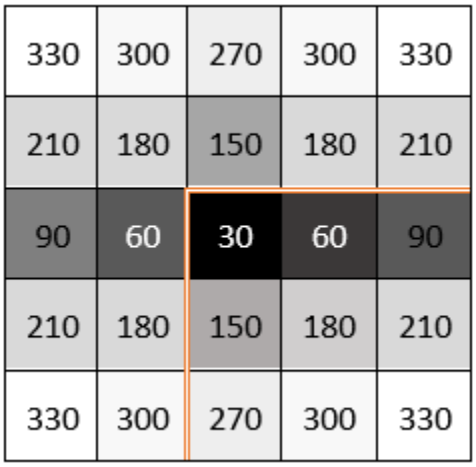
Image normalization helps prepare input images for deep learning algorithms. The Image Normalization Using External Memory example shows how to scale pixel values in a frame to a given range. The example model provides the option to dynamically calculate the minimum and maximum pixel values from each frame or to specify an expected input range. The example then scales the input pixel values to a specified output range.

Image Rotation Example: Rotate frames by small angles

Small-angle rotation helps prepare input images for vision processing algorithms such as feature extraction and matching. The Rotate Image by Small Acute Angle example computes the transform matrix for an angle between –10 and 10 degrees and uses these values as the input to the hardware algorithm. The hardware algorithm performs an affine transform and computes the output pixel intensities by using bilinear interpolation. This implementation does not require external memory.

Reflection Padding: Pad image frames by reflecting around the edge pixel

Pad the edge of a frame by reflecting around the edge-pixel value. This padding method helps reduce edge contrast effects and can improve results for machine learning while maintaining the original frame size.



To use this feature, set the **Padding method** parameter to `Reflection` on any of these blocks.

- Line Buffer
- Image Filter
- Bilateral Filter
- Median Filter
- Corner Detector

For more information on padding methods, see Edge Padding.

Generate FPGA designs with AXI-4 Stream Video interfaces (requires Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware)

When you use the Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware, you can now select an AXI-4 Stream Video interface for your generated HDL code. Using this standard interface enables you to integrate your generated IP core into a custom reference design or a larger video processing system.

MPSoC Prototyping: Target designs to Xilinx Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit (requires Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware)

Target your video processing algorithms to the Xilinx Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit with an Avnet® FMC-HDMI-CAM module. In step 1.1 of the HDL Workflow Advisor, set the **Target platform** parameter to `ZCU106 FMC-HDMI-CAM`.

R2021a

External Memory Modeling Examples: Model and deploy streaming video algorithms that require random access to memory (Requires SoC Blockset)

The Vertical Video Flipping Using External Memory example shows how to use SoC Blockset™ blocks to model random-access external memory for streaming vision applications. Then, to generate code for FPGA and processor designs, and deploy the design on a board, it uses the **SoC Builder** tool.

The Contrast Limited Adaptive Histogram Equalization with External Memory example shows how to use the SoC Blockset workflow to model frame buffer memory for a CLAHE design.

Multipixel-Multicomponent Streaming: Implement Pixel Stream Aligner, Pixel FIFO, and ROI Selector blocks for high-frame-rate color video

The Pixel Stream Aligner, Pixel Stream FIFO, and ROI Selector blocks now support streams that are both multicomponent and multipixel.

The HDL implementation replicates the algorithm for each pixel and component in parallel.

The blocks support input and output matrices of *NumPixels*-by-*NumComponents* pixels. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the matrix.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware is moved to Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware

Starting in R2021a, the Computer Vision Toolbox™ Support Package for Xilinx Zynq-Based Hardware is named Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware. To use this support package in R2021a, you must have the Vision HDL Toolbox product. For more information, see Vision HDL Toolbox Supported Hardware.

MPSoC Prototyping: Target designs to Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit (requires Vision HDL Toolbox Support Package for Xilinx Zynq-Based Hardware)

Target your video processing algorithms to the Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit, with an Avnet FMC-HDMI-CAM module. In step 1.1 of the HDL Workflow Advisor, set the **Target platform** parameter to `ZCU102 FMC-HDMI-CAM`.

R2020b

Version: 2.2

New Features

Bug Fixes

Harris Corner Detector Block and System Object: Detect features using intersecting edges algorithm

The Corner Detector block now provides a choice between the FAST algorithm and the Harris and Stephens interconnecting edges algorithm. See Harris Corner Detection.

Region of Interest (ROI) Resource Sharing: Share hardware resources and streaming control signals between vertically-aligned regions

The ROI Selector block provides an option to share hardware resources when selecting vertically aligned regions. Regions in the same column share the same `pixelcontrol` bus output.

Select the **Reuse output ports for vertically aligned regions** checkbox, and provide a set of regions that are aligned in columns and do not overlap vertically within each column. You can specify up to 1024 regions per column. To divide a frame into tiled regions that are compatible with vertical reuse, use the `visionhdlframetoregions` function.

Blob Analysis Example: Detect and label connected components in streaming video

The Blob Analysis example shows how to implement a single-pass 8-way connected component labeling algorithm, and perform blob analysis to give the centroid, bounding box, and area of each blob. The model supports up to 1080p@60fps video.

Multipixel and Multicomponent Streaming: Implement Lookup Table and Pixel Stream Aligner for high-frame-rate or color video

You can now process 4 or 8 pixels on each cycle when using the Lookup Table and Pixel Stream Aligner blocks. These blocks now also accept multicomponent streams, but you cannot use multicomponent and multipixel together.

The HDL implementation replicates the algorithm for each pixel or component in parallel.

For multipixel streaming, the blocks support input and output column vectors of 4 or 8 pixels. For multicomponent streaming, the blocks support input and output row vectors of 2, 3, or 4 components. In both cases, the **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2020a

Version: 2.1

New Features

Bug Fixes

Corner Detector Block and System Object: Detect features using FAST algorithm

The Corner Detector block detects corners using the features-from-accelerated-segment test (FAST) algorithm. You can specify a minimum contrast threshold as a parameter or port and select from three metrics that determine a corner: 5 out of 8, 7 out of 12, or 9 out of 16 pixels. These metrics specify how many pixels in a circle of pixels must meet the minimum contrast for the center pixel to be considered a corner.

Line Buffer with No Padding: Specify option to not add padding for blocks that use line buffer memory

You can now configure the Line Buffer block and blocks that use an internal line buffer to not add padding around the boundaries of the active frame. This option reduces the hardware resources used by the block and the blanking required between frames but affects the accuracy of the output pixels at the edges of the frame. To use this option, set the **Padding method** parameter to `None`. For an example, see Increase Throughput with Padding None.

This change affects these blocks:

- Line Buffer
- Bilateral Filter
- Corner Detector
- Edge Detector
- Image Filter
- Median Filter
- Binary morphology blocks: Closing, Dilation, Erosion, and Opening

Resizing Example: Downsize an image frame by a specified factor

The Image Resize example shows how to downsample an image using the bilinear, bicubic, or Lanczos-2 algorithm.

Fog Rectification Example: Enhance hazy images to improve clarity

The Fog Rectification example removes fog from color input images, and then enhances the contrast of the defogged image by stretching the range of intensity values.

Stereo Rectification Example: Align pairs of images from stereo cameras

The Stereo Image Rectification example undistorts and rectifies pairs of stereo input images.

Multicomponent Multipixel Streaming: Process high-frame-rate or high-resolution color video

The Frame To Pixel and Pixel To Frame blocks now support multicomponent multipixel streams. The MultiPixel-MultiComponent Video Streaming example shows how to use a multipixel and multicomponent pixel stream to process a high-resolution color image.

Multipixel Streaming: Perform binary morphology on high-frame-rate or high-resolution video

You can process 4 or 8 pixels on each cycle when using these binary morphology blocks: Closing, Dilation, Erosion, and Opening.

These blocks now support input and output vectors of 4 or 8 pixels. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the vector.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

R2019b

Multipixel Streaming: Process high-frame-rate or high-resolution video on FPGA

To support high-frame-rate or high-resolution video processing, such as 4k UHD, the Vision HDL Toolbox streaming video interface can now process 4 or 8 pixels on each cycle. For an example of how to use this feature, see Filter Multipixel Video Streams.

When you configure the Frame To Pixels and Pixels To Frame blocks, set the **Number of pixels** parameter to `4` or `8`. With this setting, the output of the Frame To Pixels block is a vector of 4 or 8 pixel values on each time step.

The Image Filter, Edge Detector, and Median Filter blocks now support input and output vectors of 4 or 8 pixels. The **ctrl** ports remain scalar, and the control signals in the `pixelcontrol` bus apply to all pixels in the vector. The Line Buffer block can accept an input vector of 4 or 8 pixels and returns a *KernelHeight*-by-*NumberOfPixels* matrix.

Video formats for multipixel streams must have horizontal dimensions divisible by the **Number of pixels** parameter value. These horizontal dimensions are set by the following parameters: **Active pixels per line**, **Total pixels per line**, **Front porch**, and **Back porch**. Standard video protocols 480p, 720p, 1080p, and 4k UHD support either 4 or 8 pixels at a time.

You can simulate System objects with a multipixel streaming interface, but you cannot generate HDL code for System objects that use multipixel streams. To generate HDL code for multipixel algorithms, use the equivalent Simulink blocks.

External Memory Modeling and Debugging: Simulate and deploy memory interfaces and measure performance (requires SoC Blockset)

For designs that require external memory, such as designs that buffer an entire image frame, you can use SoC Blockset to model a memory controller and multiple memory channels. This model calculates and visualizes memory bandwidth, burst counts, and transaction latencies in simulation. You can generate HDL code for the memory controller and channels by using the **SoC Builder** app. You can also deploy an AXI memory interconnect monitor on your FPGA, which can return memory transaction information for debugging and visualization in Simulink. See the Histogram Equalization Using Video Frame Buffer (SoC Blockset) and Analyze Memory Bandwidth Using Traffic Generators (SoC Blockset) SoC Blockset examples.

For an overview of memory modeling options with Computer Vision System Toolbox™ Support Package for Xilinx Zynq-Based Hardware or SoC Blockset, see Modeling External Memory.

Adaptive Histogram Equalization: Preprocess images to improve contrast

The FPGA Implementation of Contrast Limited Adaptive Histogram Equalization example shows how to implement adaptive histogram equalization for hardware. This algorithm clips the peak histogram values and redistributes them across neighboring tiles by using a bilinear interpolation filter.

Acceleration for System objects

You can speed up calls to Vision HDL Toolbox System objects by enabling a simulation mode that uses code generation. Use this command to configure a System object for code generation before calling the object.

```
myobj.simulateUsing('Code generation');
```

For an example, see Edge Detection Using Sobel Method on the `visionhdl.EdgeDetector` reference page.

Increased histogram sizes

You can now configure the Histogram block to have 2048 or 4096 histogram bins.

Removal of MATLAB Compiler support

Vision HDL Toolbox System objects and functions are no longer deployable with MATLAB Compiler™.

R2019a

Version: 1.8

New Features

Bug Fixes

Low-Light Enhancement Example: Enhance low-light color images to improve visibility

The FPGA Implementation of Low Light Enhancement example shows how to implement a hardware-targeted haze-removal technique for low-light images.

Model and deploy algorithms that use an AXI master external memory interface (requires Computer Vision System Toolbox Support Package for Xilinx Zynq-Based Hardware)

The FPGA reference design now supports an AXI master interface to external memory. This interface provides read and write access to any address. See Image Rotation with Zynq-Based Hardware (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

The Image Rotation with Zynq-Based Hardware example provides a simplified simulation model of the external memory interface. When mapping the physical ports of the reference design, map the memory interface signals to the target AXI master read and write interfaces. For details, see Model External Memory Interfaces (Computer Vision Toolbox Support Package for Xilinx Zynq-Based Hardware).

Horizontal and Vertical Counter Block: Count active lines and pixels of a pixel stream

The HV Counter block returns active-line and active-pixel counts that indicate the current position in a video frame or region-of-interest.

Increased kernel size limits for Image Filter block

The Image Filter block now allows for a coefficient kernel with up to 64-by-64 elements. Previously, the block restricted the coefficient kernel size to 16-by-16 elements.

R2018b

Programmable 2-D FIR Coefficients: Use an input port to load filter coefficients at the start of each frame

The Image Filter block now accepts coefficients from an input port. Each dimension of the matrix must have at least 2 and no more than 16 elements. The block samples the values from the **coeff** port at the start of a frame only and ignores any changes within a frame.

Compatibility Considerations

In previous releases, you could specify a row vector of coefficients, that is, a matrix of 1-by-N elements. Now, the coefficient matrix must have at least 2 elements in each dimension.

Image Pyramid Example: Generate resized pixel streams from an input pixel stream

The Image Pyramid for FPGA example produces a set of resized pixel streams from an input pixel stream. The model generates smaller streams by successively down-sampling the input stream using a Gaussian filter. Image pyramid algorithms are used in many feature detection and classification algorithms, including convolutional neural networks (CNN).

FAST Corner Detection Example: Detect corners using the features-from-accelerated-segment test (FAST) algorithm

The FAST Corner Detection example shows how to find corners in grayscale images using a metric based on 12 out of 16 pixels in a circle. The algorithm also implements nonmaximal suppression to find the best corners. Corner detection is the basis of many image-point-matching algorithms, such as creating panoramas, motion tracking and stabilization, and stereo vision.

Stereo Disparity Example: Compute disparity between left and right stereo camera images

The FPGA Implementation of Stereo Disparity using Semi-Global Block Matching example shows how to measure disparity between pairs of stereo camera images by using the semi-global block matching (SGBM) algorithm. This algorithm is similar to the `disparity` function in Computer Vision System Toolbox.

External Memory Modeling Examples: Model and deploy algorithms that use an external frame buffer (requires Computer Vision System Toolbox Support Package for Xilinx Zynq-Based Hardware)

The support package reference design now supports adding an external memory interface to a frame buffer. The frame buffer stores a single frame and returns that frame when requested. The frame buffer maintains the streaming video control signals for the output frame. The reference design implements the frame buffer interface using 2-Channel AXI Video DMA.

The Histogram Equalization with Zynq-Based Hardware (Computer Vision System Toolbox Support Package for Xilinx Zynq-Based Hardware) and Lane Detection with Zynq-Based Hardware (Computer Vision System Toolbox Support Package for Xilinx Zynq-Based Hardware) examples include a

simplified simulation model of an external memory interface. When you map the physical ports of the reference design, select the frame buffer target interface for the signals that connect to the memory interface model. For details, see Model External Memory Interfaces (Computer Vision System Toolbox Support Package for Xilinx Zynq-Based Hardware).

Improved Line Buffer

The line buffer now handles bursty data, that is, noncontiguous valid signals within a pixel line. This implementation uses fewer hardware resources due to improved padding logic and native support for kernel sizes with an even number of lines. This change affects the Line Buffer block and these blocks that use an internal line buffer:

- Bilateral Filter
- Demosaic Interpolator
- Edge Detector
- Image Filter
- Median Filter
- Binary morphology blocks: Closing, Dilation, Erosion, and Opening
- Grayscale morphology blocks: Grayscale Closing, Grayscale Dilation, Grayscale Erosion, and Grayscale Opening also use the new line buffer architecture. However, when you use a 2-D kernel of all 1s or a row-vector kernel, noncontiguous valid signals within a pixel line are not supported. As a workaround, use the Pixel Stream FIFO block to buffer an input stream and return image lines that have contiguous valid pixels.

This resource and performance data is the synthesis results from the generated HDL targeted to a Xilinx Zynq-7000 ZC706 FPGA. The synthesis results were generated using Xilinx Vivado® 2017.4. The Line Buffer block is configured with symmetric padding and a line buffer size of 2048. The table shows both odd and even neighborhood sizes.

	5-by-5 Kernel, R2018b	**5-by-5 Kernel, R2018a**	**6-by-6 Kernel, R2018b**	**6-by-6 Kernel, R2018a**
Clock frequency	300 MHz, 0.5 slack	300 MHz, 0.55 slack	300 MHz, 0.31 slack	250 MHz,0.43 slack
LUT	647	673	790	901
Slice registers	1452	1068	1844	1368
BRAM	4	4	5	5.5

Compatibility Considerations

The latency of the line buffer is now reduced by a few cycles for some configurations. You might need to rebalance parallel path delays in your models that contain a Line Buffer block or blocks that have an internal line buffer. A best practice is to synchronize parallel paths in your models using the pixel stream control signals rather than inserting a specific number of delays.

R2018a

Version: 1.6

New Features

Bug Fixes

Pothole Detection Example: Overlay a centroid marker and text label to identify potholes

This example extends the previous cartooning example to include calculating a centroid and overlaying a centroid marker and text label on detected potholes. See Pothole Detection.

Pixel Stream FIFO Block: Convert bursty video sources to contiguous lines

The Pixel Stream FIFO block rebuffers a video stream to create image lines that have contiguous valid pixels. Use this block to buffer bursty video sources, such as DMA data, or a Camera Link® source that has valid pixels every *N* clock cycles.

For an example that shows how to use the Pixel Stream FIFO block on such sources, see Buffer Bursty Data Using Pixel Stream FIFO Block.

Separable Filter Example: Use the Line Buffer block to implement a hardware-efficient custom filter

This example shows how to design a separable filter using the Line Buffer block. Separable filters use fewer hardware resources than equivalent 2-D filters. The example explains how to determine if a filter is separable, and how to choose fixed-point data types. See Using the Line Buffer to Create Efficient Separable Filters.

R2017b

Version: 1.5

New Features

Bug Fixes

Bilateral Filter Block and System Object: Apply a Gaussian filter with edge preservation

The Bilateral Filter block performs two-dimensional bilateral filtering of the input video. The block calculates filter coefficients based on the spatial and intensity standard deviations that you specify.

This release also includes an equivalent System object, `visionhdl.BilateralFilter`.

Birds-Eye View Block and System Object: Transform a front-facing camera view to an overhead view

The Birds-Eye View block warps the front-facing camera images to a top-down perspective, according to physical camera parameters that you specify. The Lane Detection example is updated to use the new block. See Lane Detection.

This release also includes an equivalent System object, `visionhdl.BirdsEyeView`.

Line Buffer Block and System Object: Store a sliding window of pixels for developing custom filter algorithms

The Line Buffer block provides a sliding *N*-by-1 column vector of pixels from a video stream. The line memory handles video control signals and edge padding, and is pipelined for high-speed video designs. To compose a neighborhood for further processing, use the **shiftEnable** output signal to store the output columns, including padding, in a shift register.

This release also includes an equivalent System object, `visionhdl.LineBuffer`.

Cartoon Image Abstraction Example: Extract features using the Bilateral Filter block

This example show how to emphasize edges in an image by using bilateral filtering and gradient generation. The original RGB image is quantized to a reduced number of colors, then the cartoon lines are overlaid on the quantized version of the input image. See Generate Cartoon Images Using Bilateral Filtering.

R2017a

Version: 1.4

New Features

Bug Fixes

Pixel Stream Aligner: Synchronize two video streams for comparison or overlay

The Pixel Stream Aligner block synchronizes two pixel streams by delaying one stream to match the timing of a reference stream. You can use this block to align streams for overlaying, comparing, or combining two streams, such as in a Gaussian blur operation. Connect the delayed stream as the reference, and the earlier stream to the `pixel` and `ctrl` ports.

This release also includes an equivalent System object, `visionhdl.PixelStreamAligner`.

Corner Detection Example: Overlay detected corners using the Pixel Stream Aligner

The Corner Detection example is updated to use the Pixel Stream Aligner block to implement the overlay of the detected corners onto the original image.

Lane Detection Example: Process 480p video and compute ego lanes in FPGA

The Lane Detection example now accepts 480p input video, without padding. To accommodate the larger birds-eye-view frame, the design does not accept new input while processing the current frame. Input frames that arrive before the previous frame is finished are dropped. The example now determines which detected lanes are the ego lanes, and removes outliers, in hardware.

R2016b

Lane Detection Example: Reference design demonstrating FPGA acceleration of a lane detection algorithm

This example shows FPGA acceleration of lane-marking detection. The design includes an FPGA-based candidate generator and a software-based polynomial fitting engine. See Lane Detection.

Measure Timing Block and System Object: Measure video signal timing from the pixel control bus

Use the Measure Timing block to investigate the blanking intervals between active frames in streaming video data. This block observes the control signals in the pixel control bus in your model, and returns the timing characteristics of the frames.

This release also includes an equivalent System object, `visionhdl.MeasureTiming`.

AXI4-Stream Video Interface: Generate an HDL IP core with an AXI4-Stream Video interface for your video algorithm (requires HDL Coder)

When your synthesis tool is Xilinx Vivado, HDL Coder™ can generate an IP core with an AXI4-Stream Video interface for your video algorithm. To generate an IP core, model your video algorithm using the streaming pixel interface. Then, in the **Target platform interface table**, map the pixel data and pixel control bus ports to the `AXI4-Stream Video Master` or `AXI4-Stream Video Slave` interfaces.

You can integrate the generated IP core into the `Default video system` reference design or your own custom video reference design.

See Model Design for AXI4-Stream Video Interface Generation.

Computer Vision on Xilinx Zynq-Based Hardware: Generate and verify vision algorithms on a prototype board connected to a live HDMI video stream

The Computer Vision System Toolbox Support Package for Xilinx Zynq-Based Hardware (introduced April 2016) supports verification and prototyping of vision algorithms on Zynq-based hardware.

HDL Coder is required for customizing the algorithms running on the FPGA fabric of the Zynq device. Embedded Coder® is required for customizing the algorithms running on the ARM® processor of the Zynq device. Using this support package, you can:

- Target your video processing algorithms to Zynq hardware from Simulink. This includes support for Vision HDL Toolbox blocks.
- Stream HDMI signals into Simulink to explore designs with real data.
- Generate HDL vision IP cores, using HDL Coder. This includes support for algorithms that use Vision HDL Toolbox blocks.
- Deploy algorithms and visualize them using HDMI output on a screen.

For additional information, see Computer Vision System Toolbox Support Package for Xilinx Zynq-Based Hardware.

Optimized grayscale morphology using Van Herk algorithm

The grayscale morphology blocks and objects now implement the Van Herk algorithm for line, square, or rectangle structuring elements with more than 8 columns. This algorithm uses fewer hardware resources, and has higher latency, than the previous comparator tree implementation.

This change affects these blocks and objects:

- Grayscale Closing
- Grayscale Dilation
- Grayscale Erosion
- Grayscale Opening
- `visionhdl.GrayscaleClosing`
- `visionhdl.GrayscaleDilation`
- `visionhdl.GrayscaleErosion`
- `visionhdl.GrayscaleOpening`

Compatibility Considerations

Due to the latency change, you might need to rebalance parallel path delays in your models that contain morphology blocks. A best practice is to use the pixel stream control signals to synchronize parallel paths in your models, rather than inserting a specific number of delays.

The latency of a Van Herk kernel for a neighborhood of $m \times n$ pixels is $2m + \log_2(n)$. The block implements this kernel for line, square, or rectangle structuring elements more than 8 pixels wide, with no pixels set to zero.

The latency of a comparison tree kernel for a neighborhood of $m \times n$ pixels is $\log_2(m)+\log_2(n)$. The block implements this kernel for structuring elements smaller than 8 pixels wide, or those with one or more pixels set to zero.

Simpler way to call System objects

Instead of using the `step` method to perform the operation defined by a System object, you can call the object with arguments, as if it were a function. The `step` method continues to work. This feature improves the readability of scripts and functions that use many different System objects.

For example, if you create a `visionhdl.LookupTable` System object named `invertgray`, then you call the System object as a function with that name.

```
invertgray = visionhdl.LookupTable(uint8(linspace(255,0,256));
for p = 1:numPixelsPerFrame
    [pixOut(p),ctrlOut(p)] = invertgray(pixIn(p),ctrlIn(p));
end
```

The equivalent operation using the `step` method is:

```
invertgray = visionhdl.LookupTable(uint8(linspace(255,0,256));
for p = 1:numPixelsPerFrame
    [pixOut(p),ctrlOut(p)] = step(invertgray,pixIn(p),ctrlIn(p));
end
```

When the `step` method has the System object as its only argument, the function equivalent has no arguments. You must call this function with empty parentheses. For example, `step(sysobj)` and `sysobj()` perform equivalent operations.

R2016a

Version: 1.2

New Features

Bug Fixes

ROI Selector: Select a region of interest from a streaming video source

The new block, ROI Selector, selects a region of interest (ROI) from a video stream. You can specify one or more regions using input ports or mask parameters. The block returns each new region as streaming pixel data and corresponding `pixelcontrol` bus.

This release also includes an equivalent System object, `visionhdl.ROISelector`.

Grayscale Morphology: Perform dilation, erosion, opening, and closing operations on grayscale inputs

Perform grayscale morphology using these new blocks and System objects:

- Grayscale Closing
- Grayscale Dilation
- Grayscale Erosion
- Grayscale Opening
- `visionhdl.GrayscaleClosing`
- `visionhdl.GrayscaleDilation`
- `visionhdl.GrayscaleErosion`
- `visionhdl.GrayscaleOpening`

Larger frame size for statistics computations

The Image Statistics block and `visionhdl.ImageStatistics` System object now support input regions up to 64^4 (16,777,216) pixels in size.

R2015b

Version: 1.1

New Features

Bug Fixes

Corner Detection Example: Detect intersecting edges with the Harris algorithm

This example uses the Image Filter block to implement the Harris & Stephens corner detection algorithm. See “Corner Detection” in Vision HDL Toolbox Examples.

MATLAB Compiler Integration: Generate standalone executables for System objects

All System objects in Vision HDL Toolbox support generating executables with MATLAB Compiler.

HDL code generation for structure arguments in MATLAB

HDL Coder now supports code generation for structure arguments of functions. For Vision HDL Toolbox, this simplifies the arguments of functions targeted for HDL code generation. Previously, you had to flatten the structure into the component control signals.

```
function [pixOut,hStartOut,hEndOut,vStartOut,vEndOut,validOut] = ...
      HDLTargetedDesign(pixIn,hStartIn,hEndIn,vStartIn,vEndIn,validIn)
```

With HDL code generation support for structures, the arguments can now include the control signal structure.

```
function [pixOut,ctrlOut] = HDLTargetedDesign(pixIn,ctrlIn)
```

The structure becomes individual control signals in the generated Verilog® or VHDL® code.

Improved line buffer performance

This release improves the HDL performance of blocks and objects that have internal line memory. The synthesized HDL code for the line buffer now supports HD video at 60fps on the Xilinx Zynq-7000 ZC702 board, and 4k video at 30fps on the Xilinx Zynq-7000 ZC706 board. The following blocks and System objects use the improved line buffer code:

- Demosaic Interpolator
- Edge Detector
- Image Filter
- Median Filter
- Closing
- Dilation
- Erosion
- Opening

For example, the table shows the R2015b performance of the Demosaic Interpolator, using **Gradient-corrected linear** interpolation, and synthesized with Xilinx Vivado for these target boards.

Xilinx Zynq-7000 ZC702	Xilinx Zynq-7000 ZC706
HD input video	4k input video

Xilinx Zynq-7000 ZC702	Xilinx Zynq-7000 ZC706
200 MHz	375 MHz
Consumes: • no DSP48s • 2.5% of the LUTS • 1.5% of the slice registers • 8 BRAMS (4%)	Consumes: • no DSP48s • 0.6% of the LUTS • 0.4% of the slice registers • 8 BRAMS (1%)

In the previous release, the performance is shown below.

Xilinx Zynq-7000 ZC702	Xilinx Zynq-7000 ZC706
HD input video	4k input video
135 MHz (need 150 MHz for 60 fps)	230 MHz (need 300 MHz for 30 fps)
Consumes: • no DSP48s • 2.6% of the LUTS • 1.5% of the slice registers • 8 BRAMS (4%)	Consumes: • no DSP48s • 0.5% of the LUTS • 0.3% of the slice registers • 8 BRAMS (1%)

R2015a

Version: 1.0

New Features

Video synchronization signal controls for handling nonideal timing and resolution variations

Vision HDL Toolbox blocks and System objects accept and return video data as a serial stream of pixel data and control signals. The protocol mimics the timing of a video system, including inactive intervals between frames. Each block or object operates without full knowledge of the image format, and can tolerate imperfect timing of lines and frames. See Streaming Pixel Interface.

Configurable frame rates and sizes, including 60FPS for high-definition (1080p) video

To support HD video applications, Vision HDL Toolbox blocks and System objects generate HDL code capable of running at 150 MHz.

For supported video formats, see the Frame To Pixels block.

Frame-to-pixel and pixel-to-frame conversions to integrate with frame-based processing capabilities in MATLAB and Simulink

In MATLAB, use the `visionhdl.FrameToPixels` object to convert framed video data to a stream of pixels and control signals.

In Simulink, use the Frame To Pixels block to convert framed video data to a stream of pixels and control signals.

Image processing, video, and computer vision algorithms with a pixel-streaming architecture, including image enhancement, filtering, morphology, and statistics

Vision HDL Toolbox blocks and System objects implement hardware-friendly architectures. For the list of blocks and System objects provided in this product, see HDL-Optimized Algorithm Design.

Implicit on-chip data handling using line memory

Some Vision HDL Toolbox blocks and System objects include internal memory for a small number of lines as required for the calculation at each image pixel.

The line memory stores *kernel size - 1*-by-*active pixels per line* pixels. Set **Line buffer size** to a power of two that accommodates *active pixels per line*.

Support for HDL code generation and real-time verification

Vision HDL Toolbox provides libraries of blocks and System objects that support HDL code generation. To generate HDL code from these designs, you must have an HDL Coder license. HDL Coder also enables you to generate scripts and test benches for use with 3rd party HDL simulators.

If you have an HDL Verifier license, you can use the FPGA-in-the-loop feature to prototype your HDL design on an FPGA board. HDL Verifier also enables you to cosimulate a Simulink model with an HDL design running in a 3rd party simulator.

See HDL Code Generation and Verification