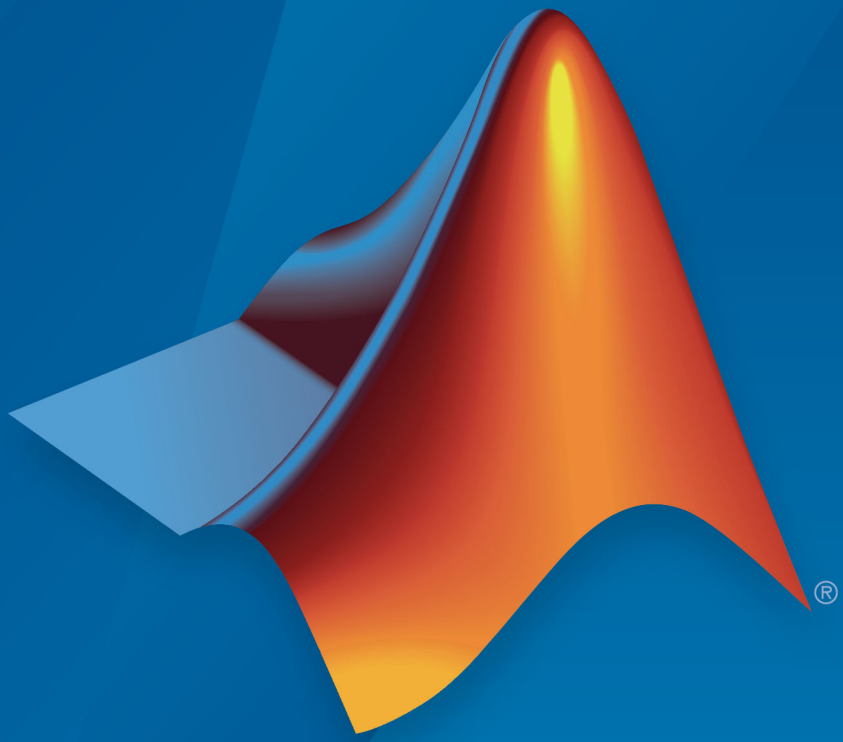
Vision HDL Toolbox™ Release Notes



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.
3 Apple Hill Drive
Natick, MA 01760-2098

Vision HDL Toolbox™ Release Notes

Contents

R2018b

R2018a

R2017b

R2017a

R2016b

R2016a

R2015b

R2015a

R2018b

Version: 1.7

New Features

Bug Fixes

Compatibility Considerations

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

Version: 1.3

New Features

Bug Fixes

Compatibility Considerations

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
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