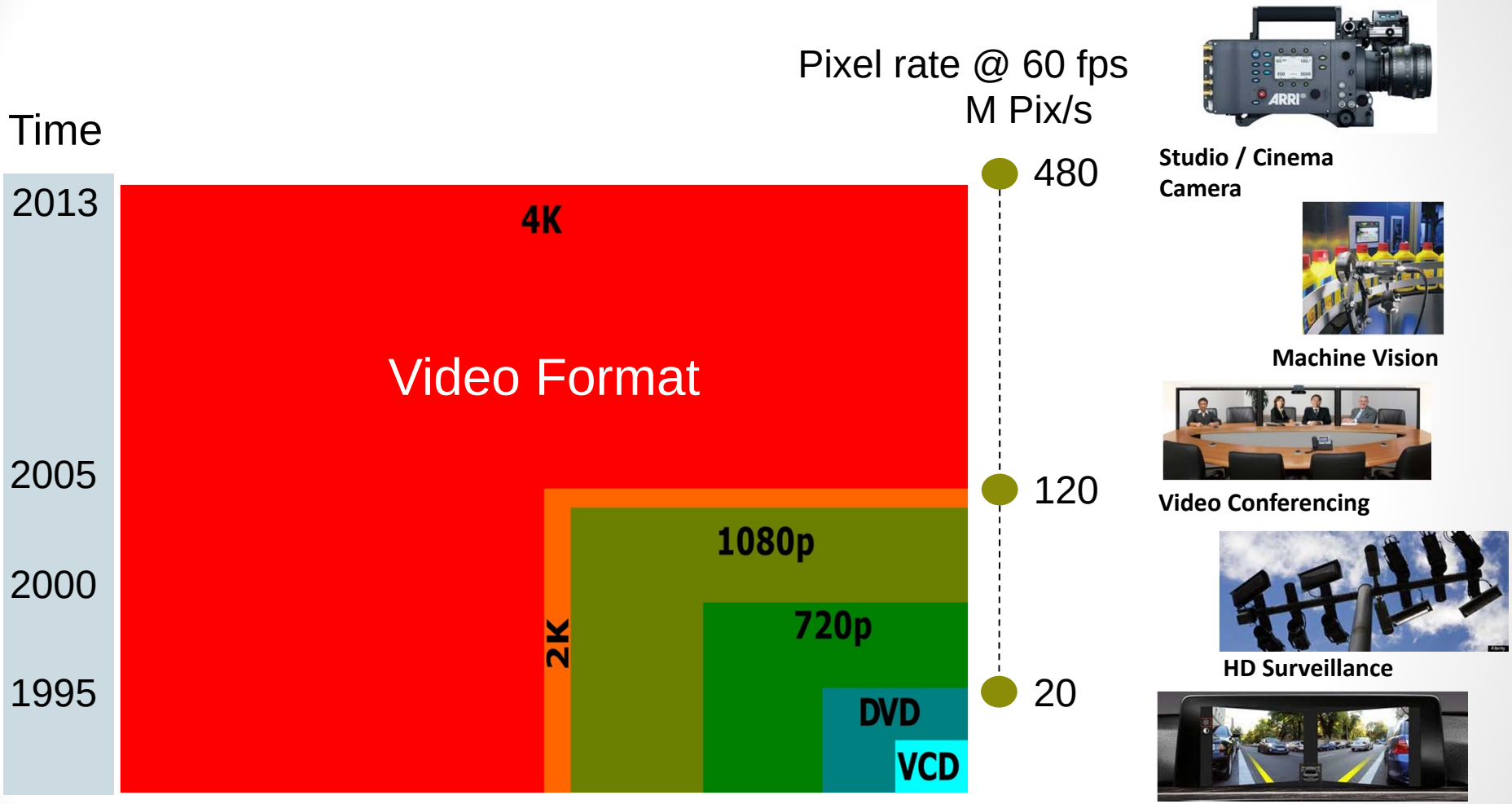
Accelerating OpenCV in Embedded Vision

Stephen Neuendorffer
Principal Engineer, High Level Synthesis
Xilinx

#TheVisionShow

- Trends in vision processing
  - ?
- Key bottlenecks in embedded video processing
  - ?
- Using OpenCV in embedded video processing
  - ?

Increasing Video Resolutions



High Resolution Video moving to Camera/Vision applications

Multiple Cameras

Machine Vision

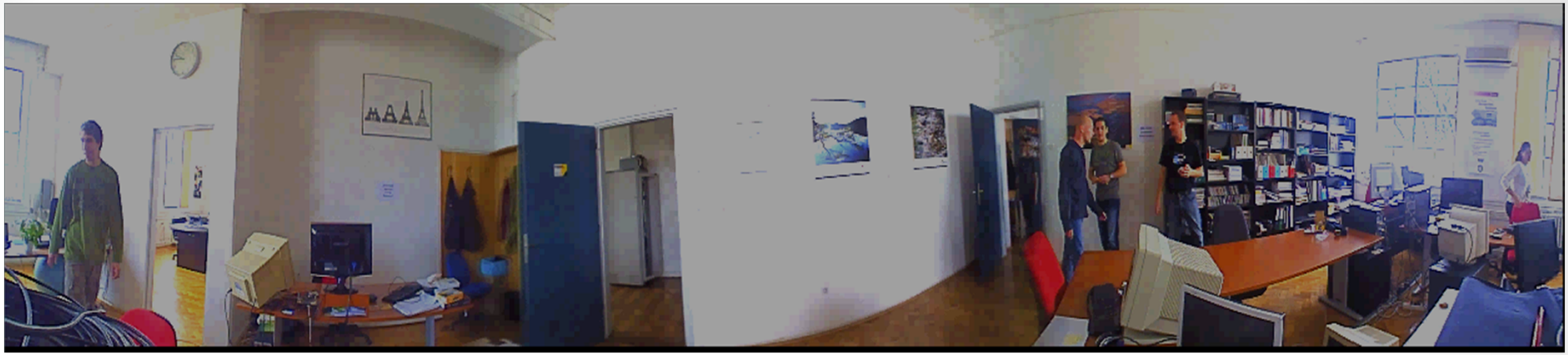
Video Conferencing

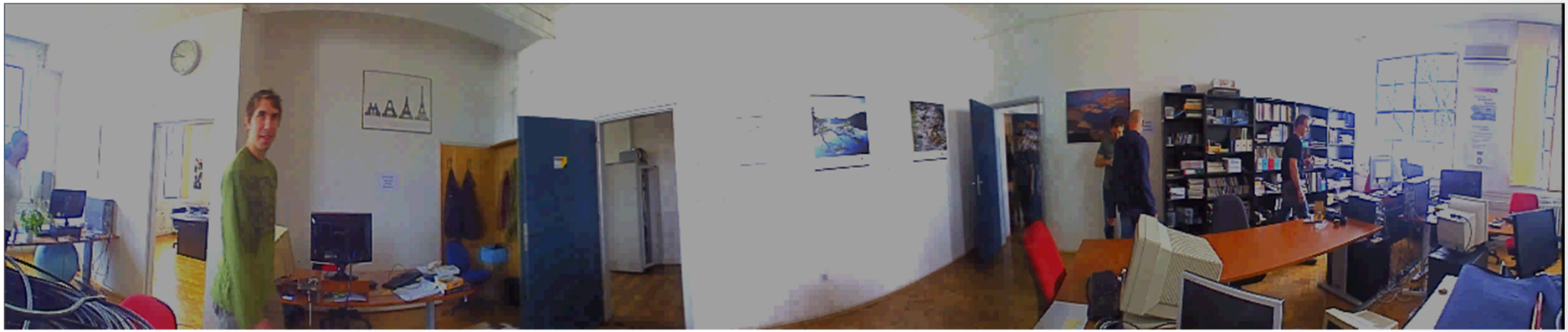
HD Surveillance

Driver Assist

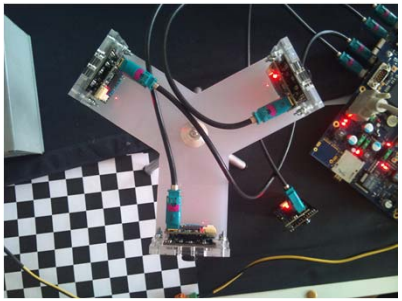
- Stereo (+ IR) cameras for depth map
- Robots operating in unrestricted environments
- Simultaneous location and mapping
- Immersive environments
- Massive camera scaling
- Multiple camera stitching and tracking
- Semantic understanding and decision making

An Example





3*1 MP Camera Stitching from Xylon
(http://www.xylon.com)

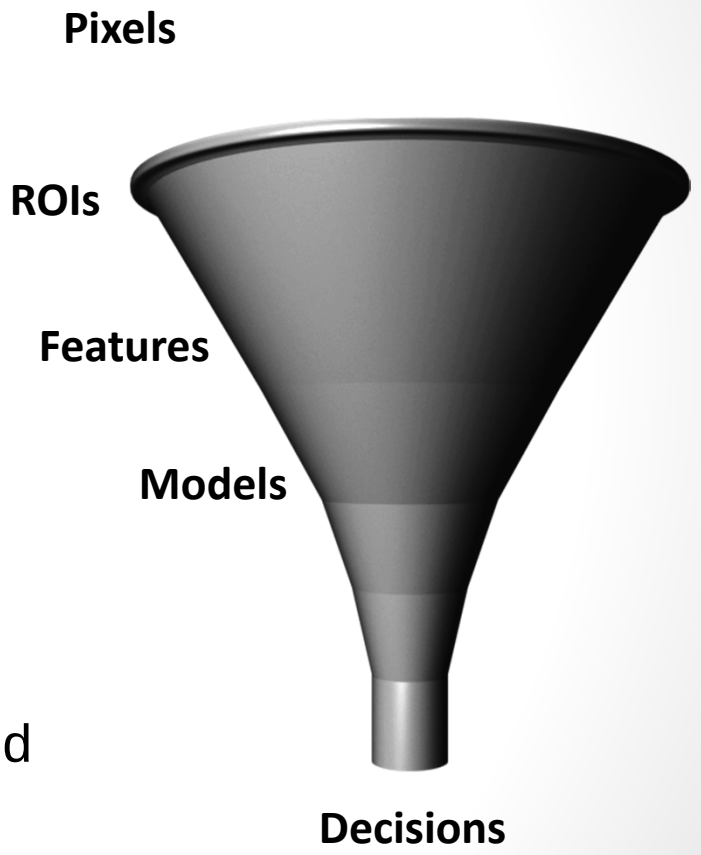


- Collect the right information
  - Fuse 3 cameras into one image
- Detect interesting things
  - There are 4 people
- Identify the things
  - Person in the green shirt is Fred
- Understand the environment
  - Fred is leaving
- Make a decision
  - Turn on the lights in the next room and warm up Fred's car

Similar challenges in many vision applications

The Funnel

- Collect the right information
  - Fuse 3 cameras into one image
- Detect interesting things
  - There are 4 people
- Identify the things
  - Person in the green shirt is Fred
- Understand the environment
  - Fred is leaving
- Make a decision
  - Turn on the lights in the next room and warm up Fred's car



Similar challenges in many vision applications

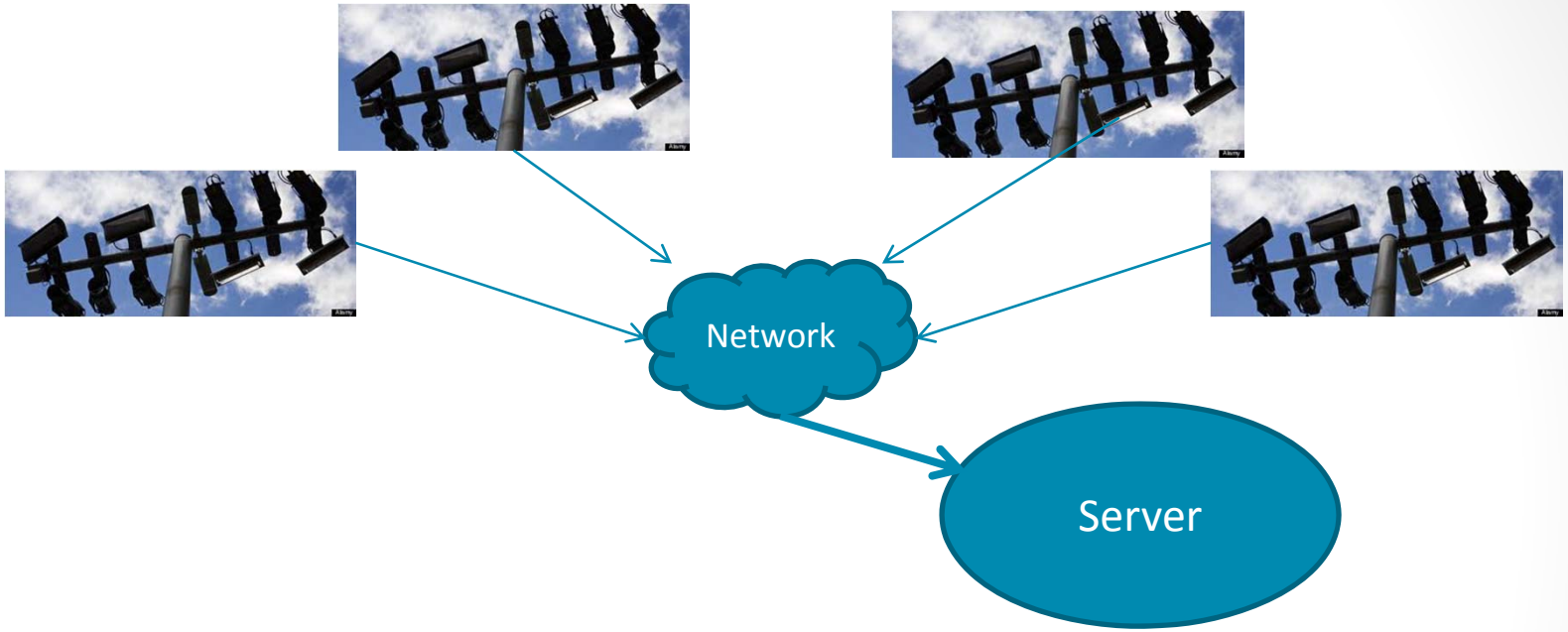
Video Processing Rates

Design approach	RISC Proc.	Proc. w/ accels.	Folded datapath	Pipelined datapath	Replicated datapath
clock:sample	1000:1	100:1	10:1	1:1	1:10
Sample Rate (200MHz clock)	200Ks/s	2Ms/s	20Ms/s	200Ms/s	2 Gs/s
Computation (500 ops/sample)	100M op/s	1G op/s	10G op/s	100G op/s	1 T op/s
	Feature Processing			FullHD Pixel rate Processing	

Arm A9 processors
1-2 Gops each

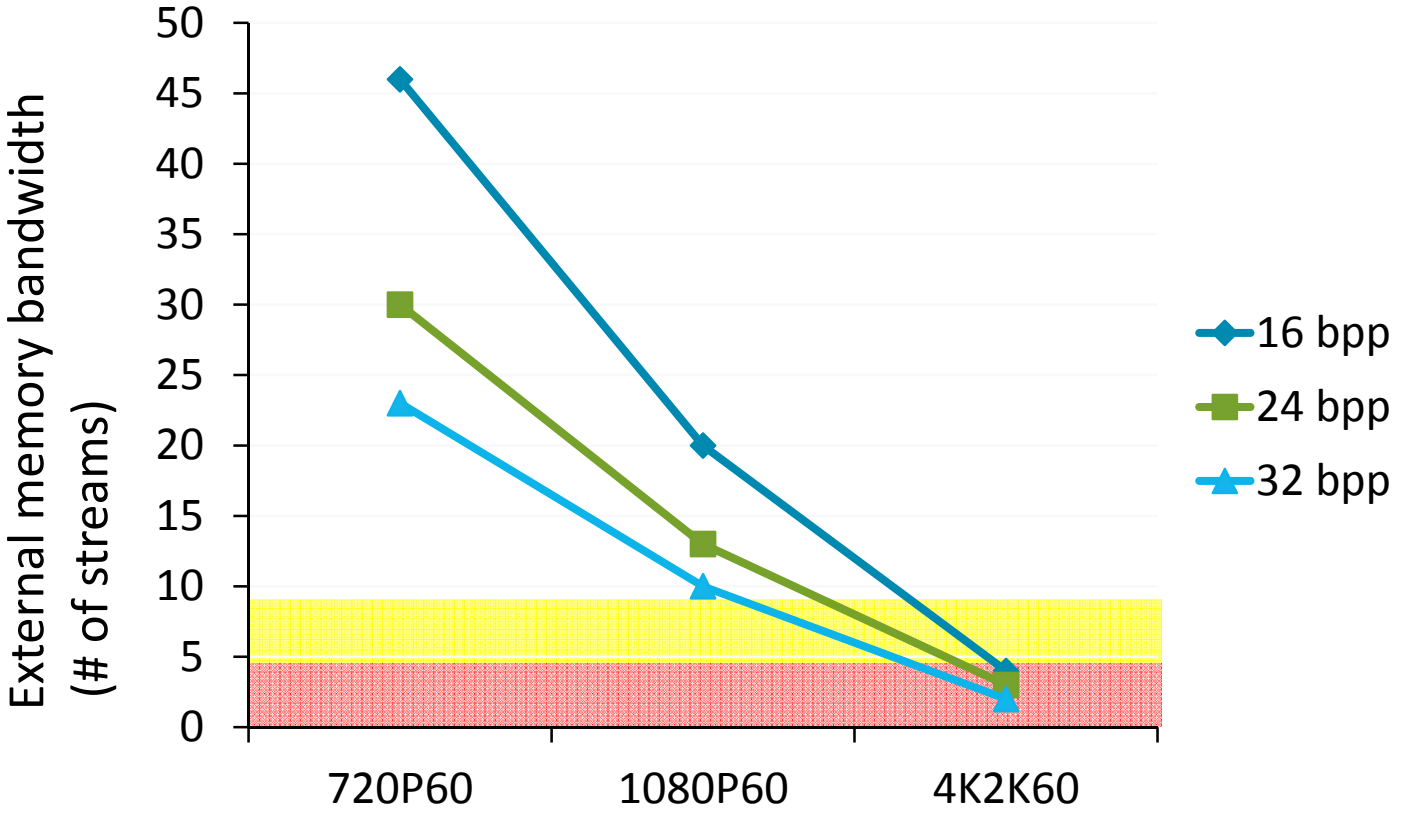
specialized hardware
10 – 500 Gops

In the network?



- 1080P60 ~= 125 Mpixels/sec * 8 bpp ~= 1Gbit/sec
  - Completely saturates a Gigabit Ethernet link
- Some solutions
  - ‘Exotic’ networks (10GigE)
  - Point-to-point (SDI/HDMI)
  - Compression (H264/H265)

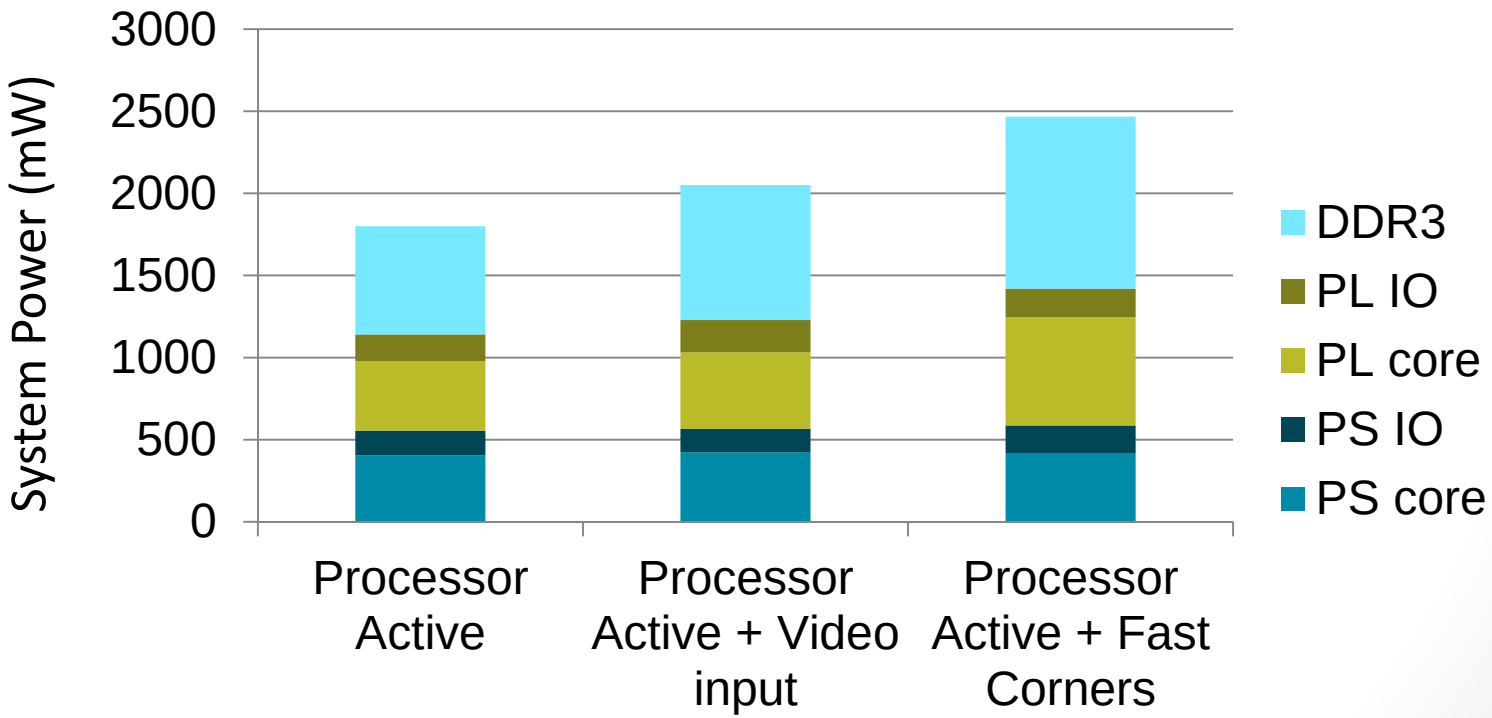
Video Streams to Memory



For embedded memory: 32 bit DDR3-800, 80% efficiency

Process and decide without storing whenever possible!

- External Memory Power can also be significant
- Real-time HD video processing can be very efficient in FPGAs (200-300 mW incremental power)



Data collected from Xilinx Application Note 1157

Analytics Challenges

Network Challenge
- More data than can be stored or sent
- Compression vs. Storage
- Intelligent Recording

Dumb Cameras -> Smarter Cameras

Power Challenge
- Power over Ethernet
- Thermal Issue
- Power Management

Efficient Embedded Processing

Cost Challenge
- Existing infrastructure
- Increase in camera types
- New applications without new hardware

Increasing Integration with programmability

OpenCV Overview

- Open Source Computer Vision (OpenCV) is widely used to develop computer vision applications
  - Library of 2500+ optimized video functions
  - Optimized for desktop processors and GPUs
  - Tens of thousands of users

- Big Questions
  - How to get these applications into an embedded system?
  - What is right mix of processing for different parts of an application?

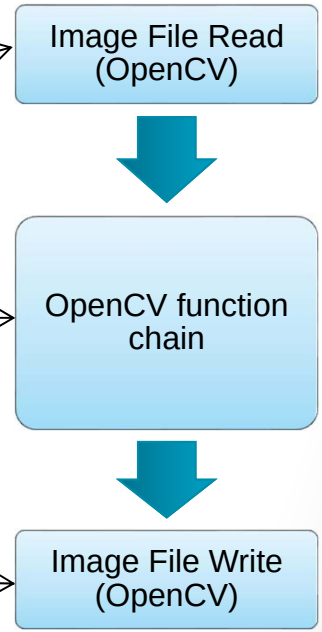
OpenCV modules

- core - key data structures
- imgproc - image filtering
- features2d - corner, blob extraction and matching
- stitching - lens correction, image stitching, stereo rectification
- objdetect - classifiers
- video - optical flow
- videostab - video stabilization
- flann - flann feature matching
- ml - machine learning, classifier training
- calib3d - lens calibration and stereo alignment
- highgui - platform independent GUI toolkit

OpenCV Code

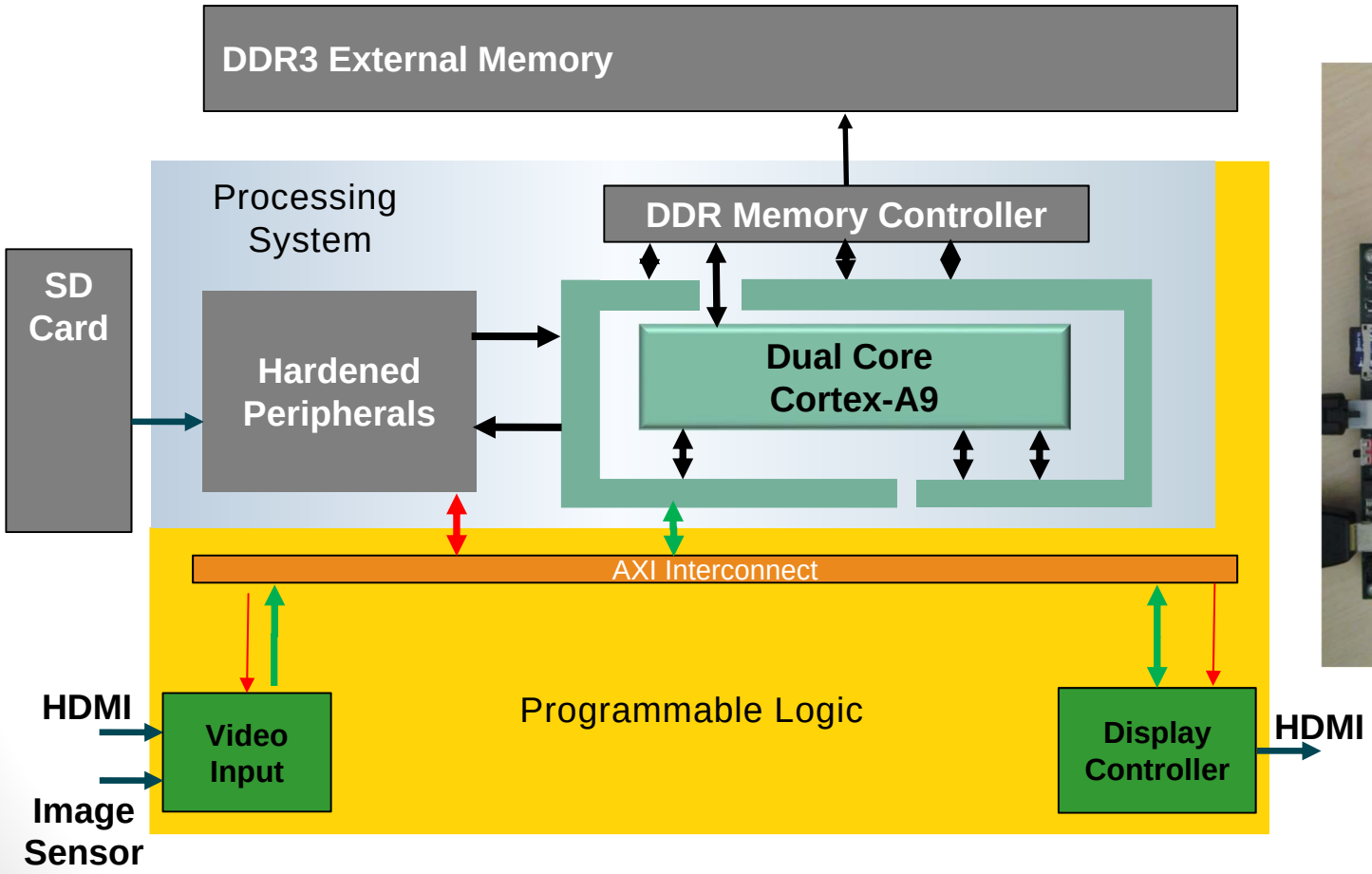
- One image input, one image output
  - Processed by chain of functions sequentially

```
…
IplImage* src=cvLoadImage("test_1080p.bmp");
IplImage* dst=cvCreateImage(cvGetSize(src),
                    src->depth, src->nChannels);

cvSobel(src, dst, 1, 0);
cvSubS(dst, cvScalar(100,100,100), src);
cvScale(src, dst, 2, 0);
cvErode(dst, src);
cvDilate(src, dst);

cvSaveImage("result_1080p.bmp", dst);
cvReleaseImage(&src);
cvReleaseImage(&dst);
…
```



Example Architecture

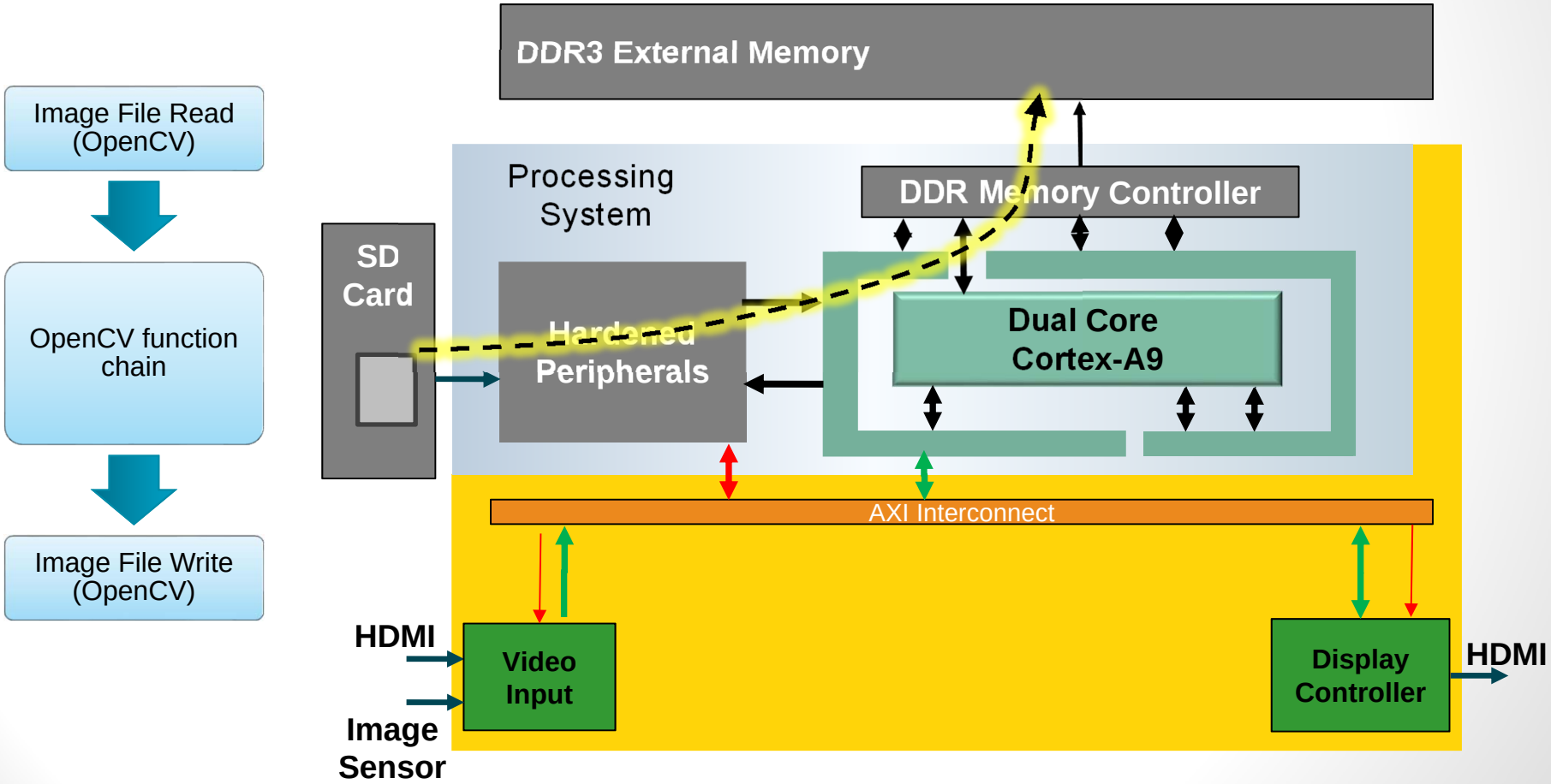
- Xilinx 'Zynq' Video and Imaging Kit
  - Supports HDMI I/O and VITA2000 Image Sensor

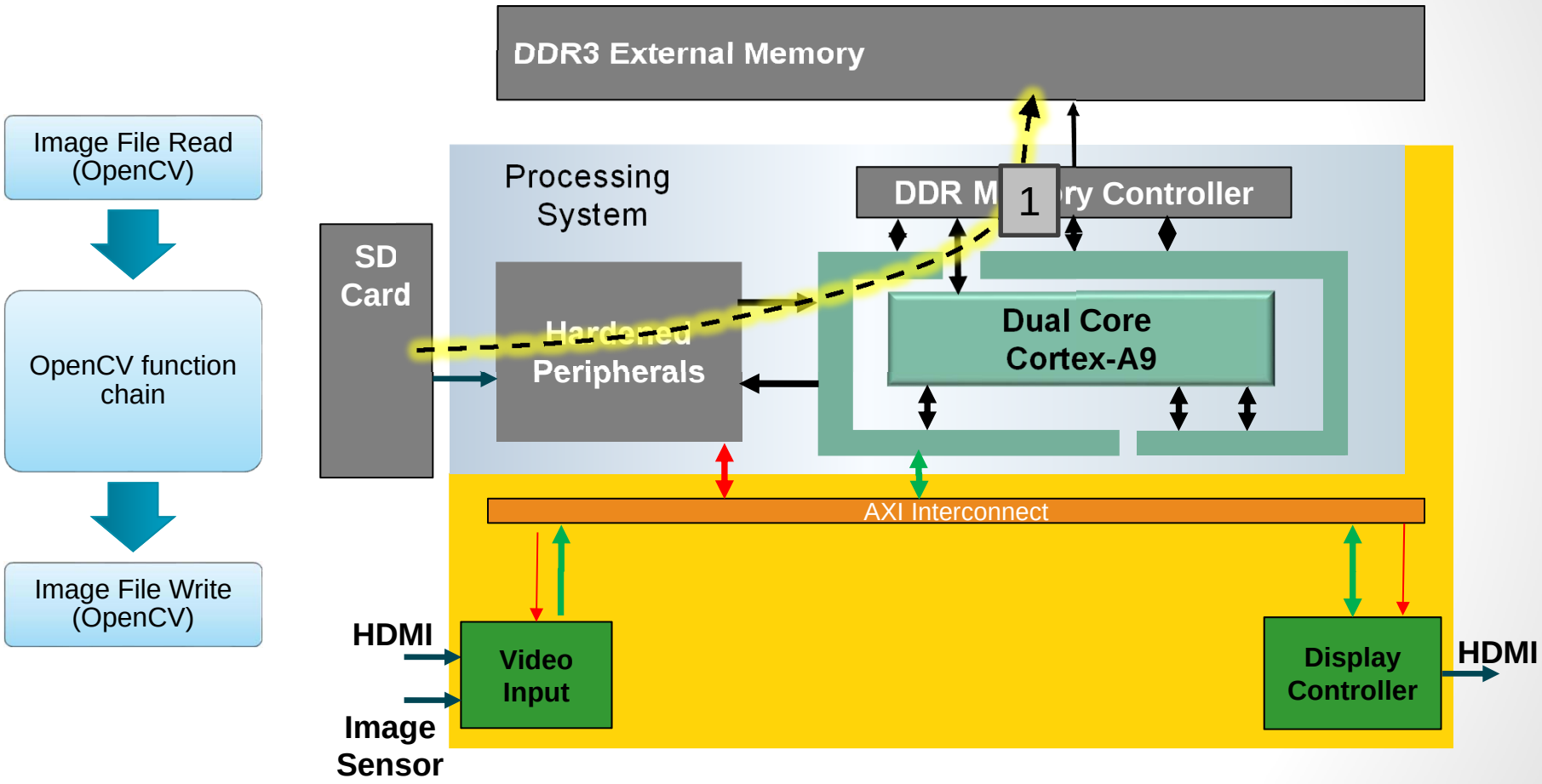


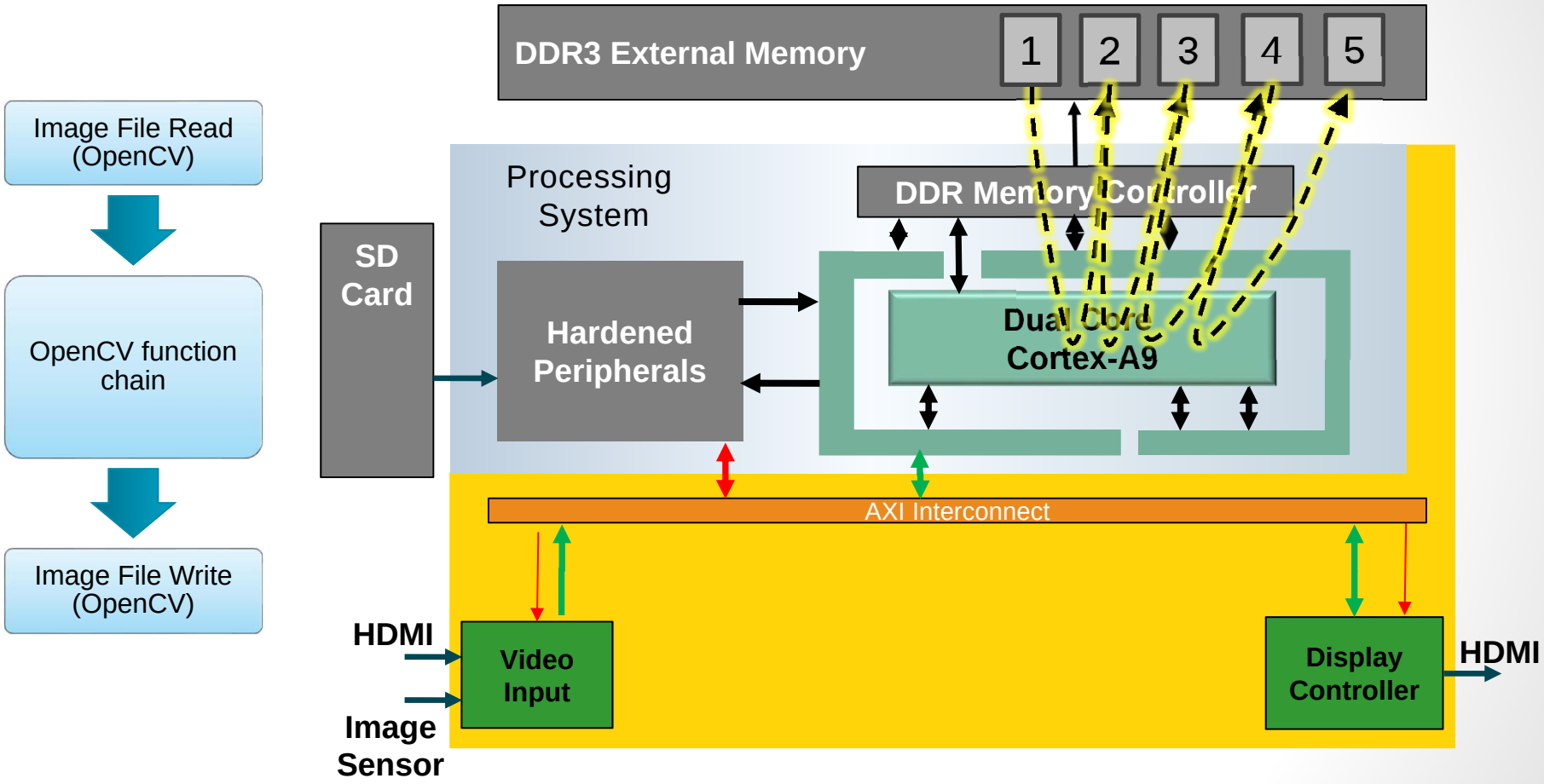
http://www.xilinx.com/zynq

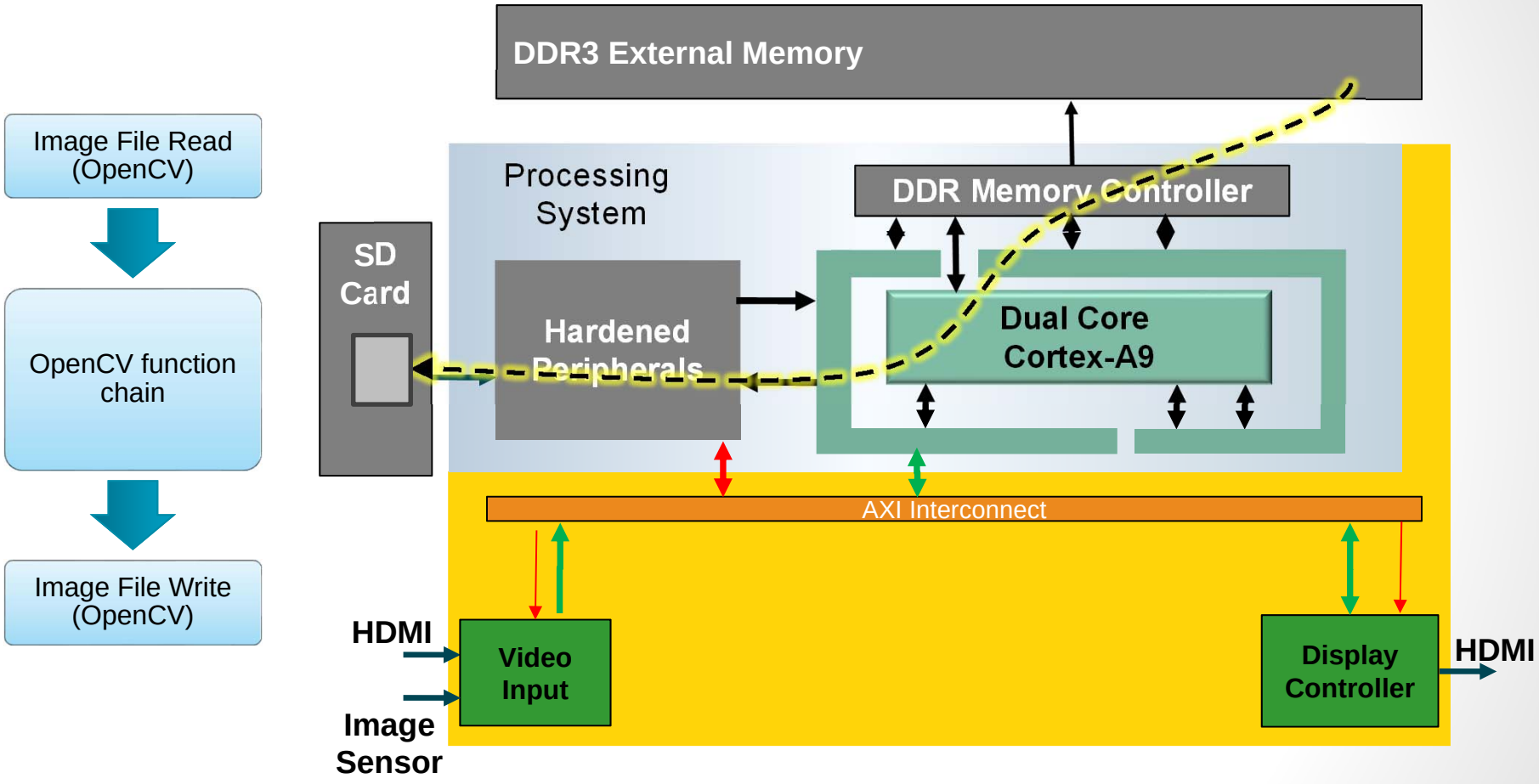
OpenCV on ARM

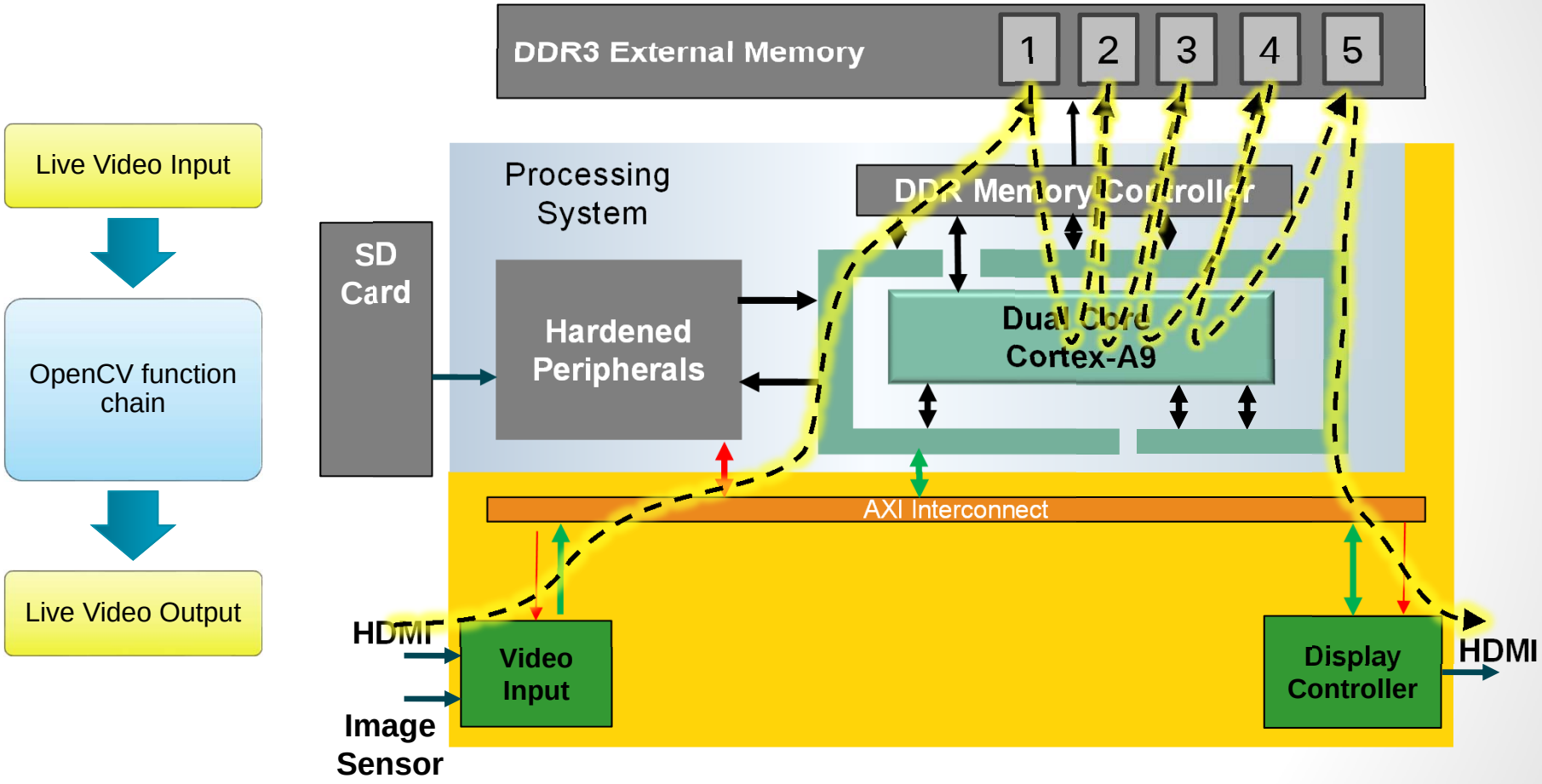
- OpenCV-based image processing built around sequential processing of frame buffers in external memory











ARM Optimized CV

- Explicit prefetching and use of ARM NEON instructions can give significant speedup over generic code
- Throughput drops with composition of functions

Performance of UncannyCV library (http://www.uncannyvision.com)

	Throughput * (Megapixels per sec)	Acceleration * (vs. OpenCV on ARM)
FAST9 Corner Detection	24	2x
Canny Edge Detection	25	3x
Harris Corner Detection	15.7	6.5x
Erosion/Dilation	153	6.5x
5x5 convolution	96	22x
LBP Face Detection		3x
HOG Pedestrian Detection		9x

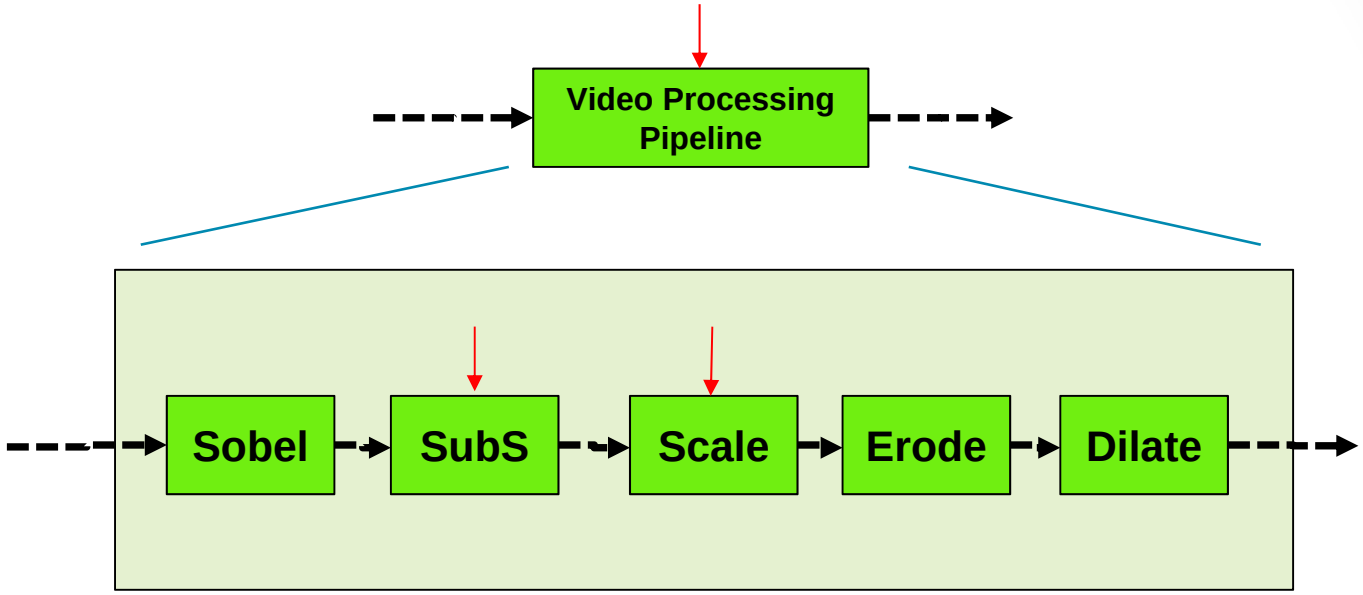
* based on single core 1GHz Cortex-A9 (2 wide SIMD)

ARM Limitations

- Poor access locality -> small caches perform poorly
  - 720P ~= 1 M pixels/frame ~= 4 MByte/frame @ 32 bpp
  - Need multiple frames in cache for best performance
  - Need good cache prefetching performance

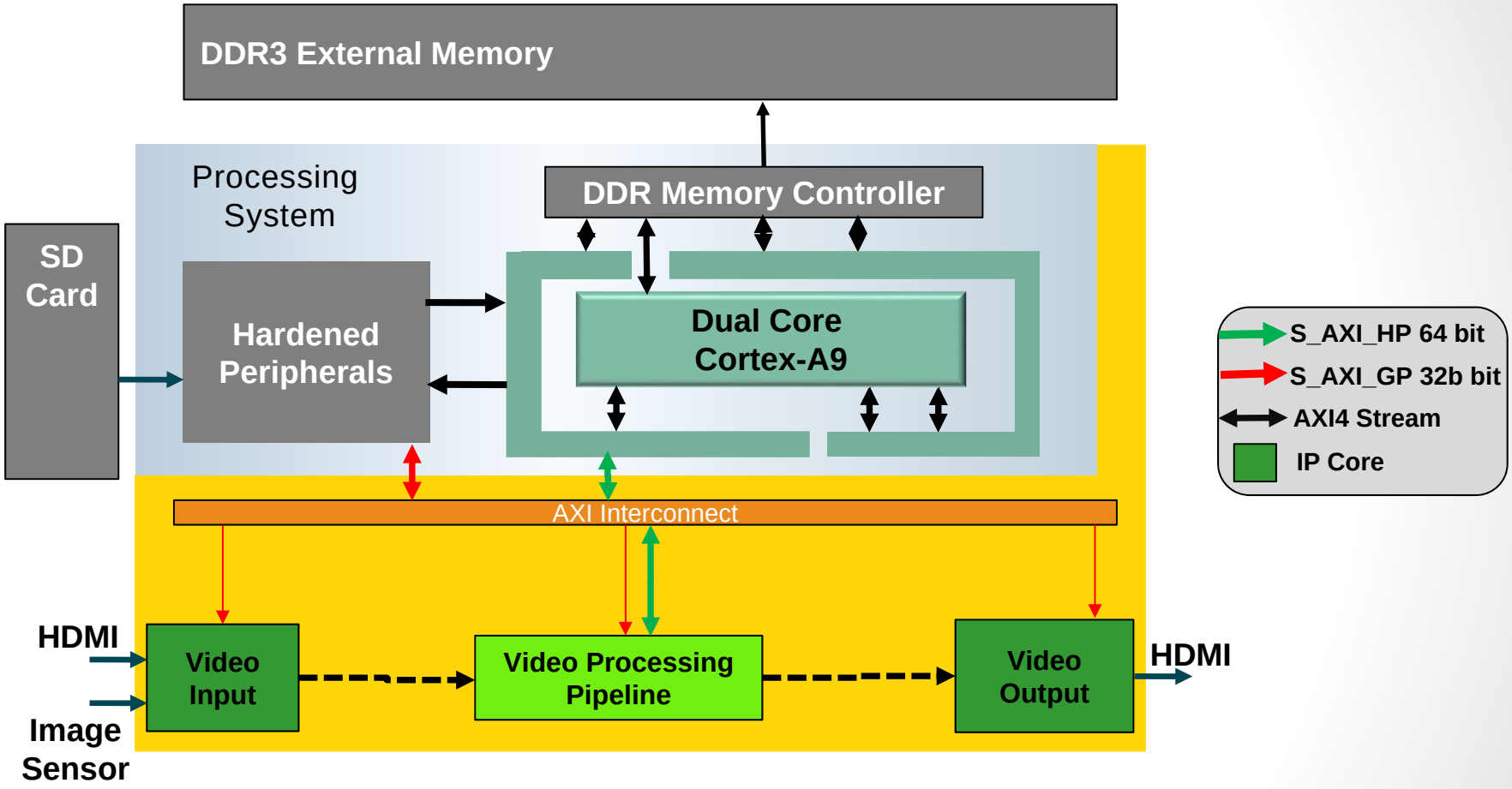
- Generic processors limited in parallel operations/number of cores.
  - 720P60 ~= 60 Mpixels/sec ~= 16 cycles per pixel @ 1 Ghz
  - Complex algorithms need many operations per cycle.

Real-time HD processing requires solving both problems



- Avoids need for intermediate buffers
  - Pixels (or tiles of pixels) can be passed directly from one block to another
- Scales well to complex algorithms
  - Additional processing need not share computing resources

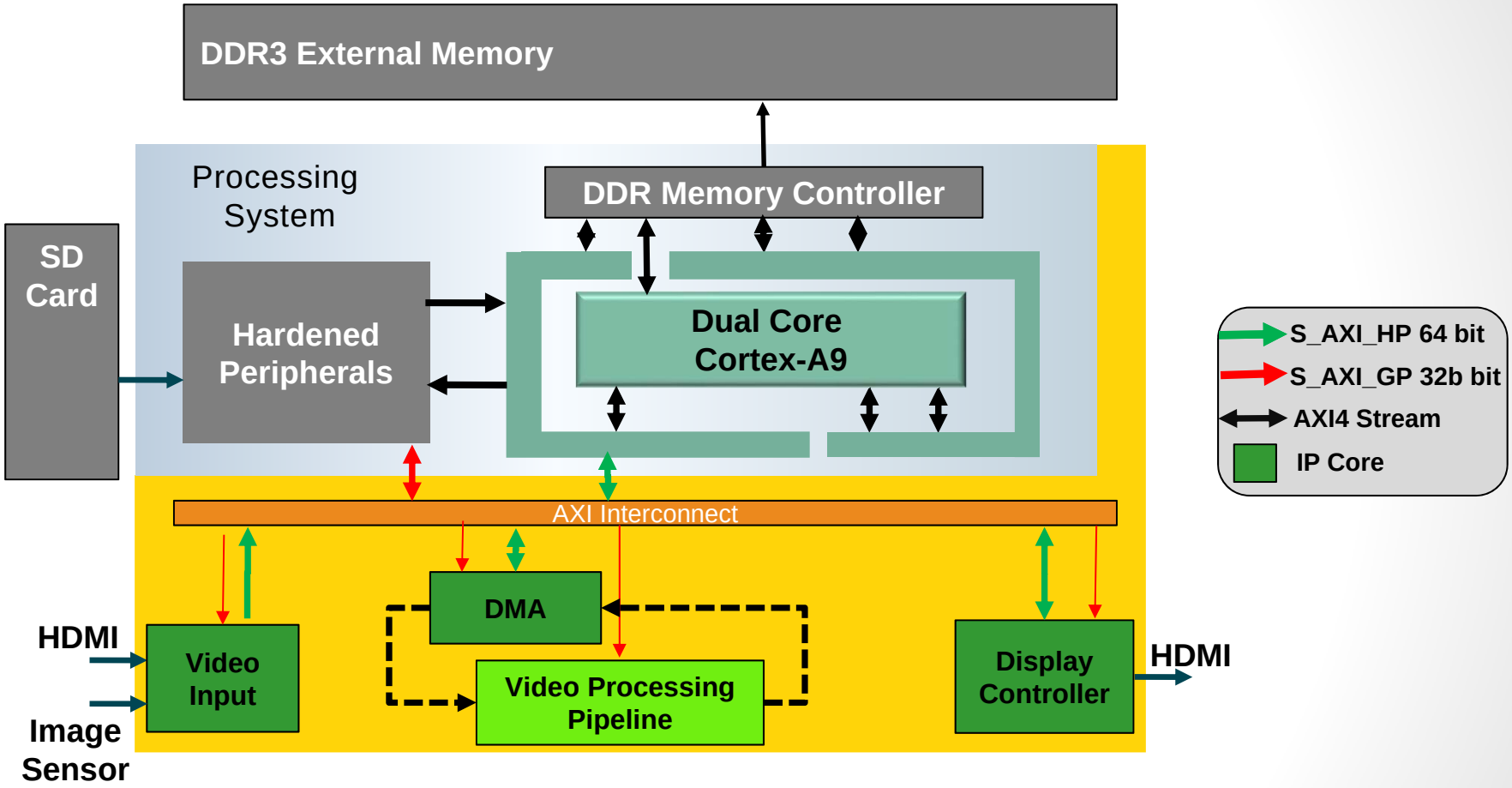
Embedded Vision HW



Pro: Reduced memory bandwidth

Con: Less flexibility and ability to interact with the processor

Embedded Vision HW

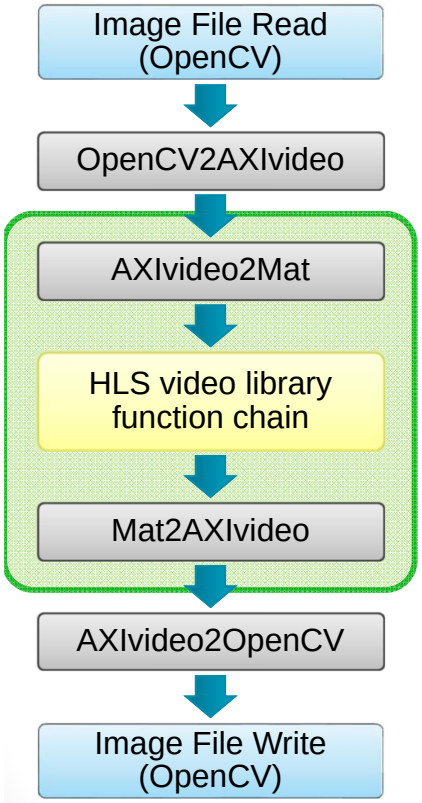


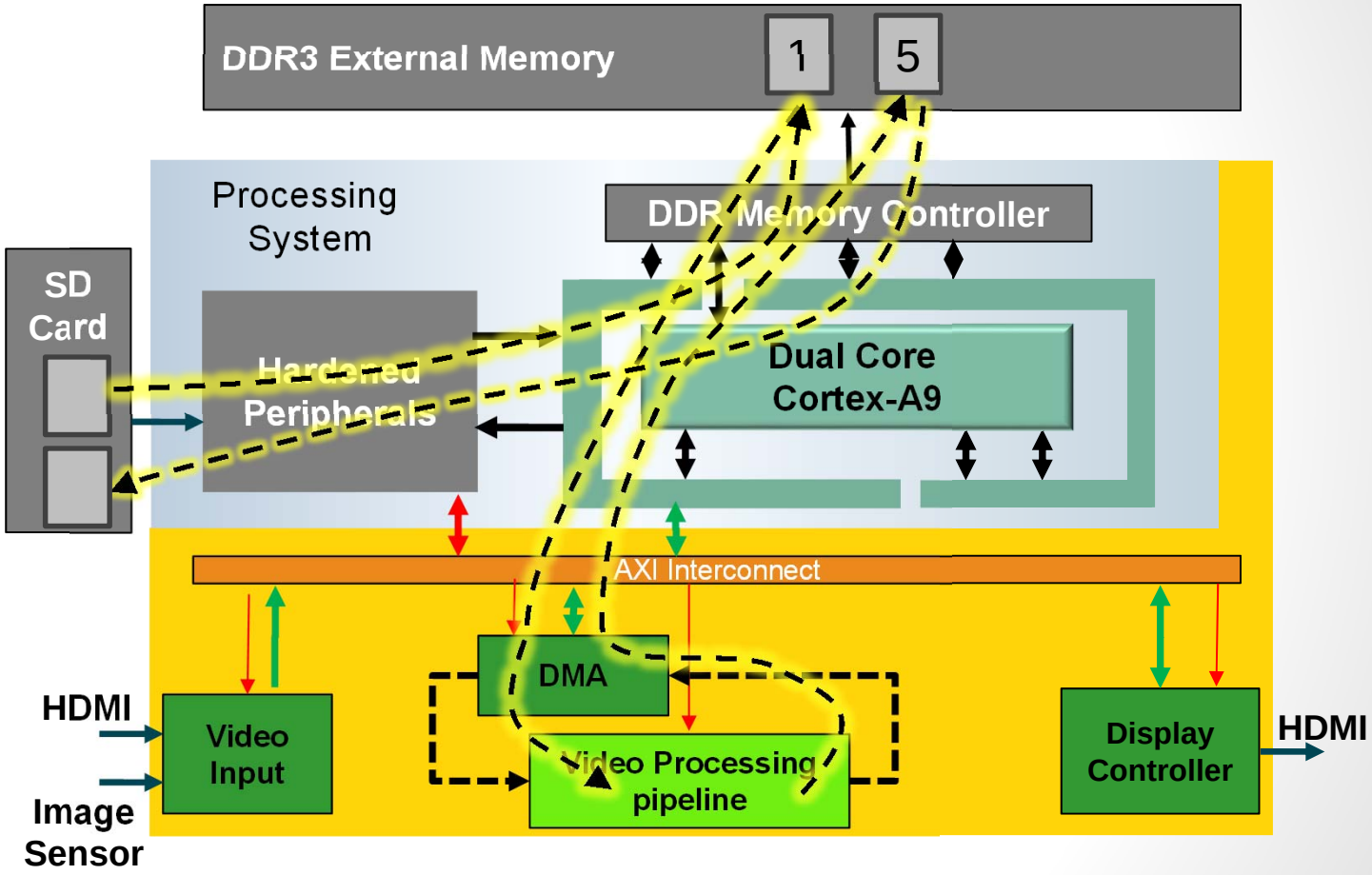
Pro: Processor manages buffers

Con: Requires 4 video streams to external memory

e.g., Zynq Video TRD: http://www.wiki.xilinx.com/Zynq+Base+TRD+2013.4

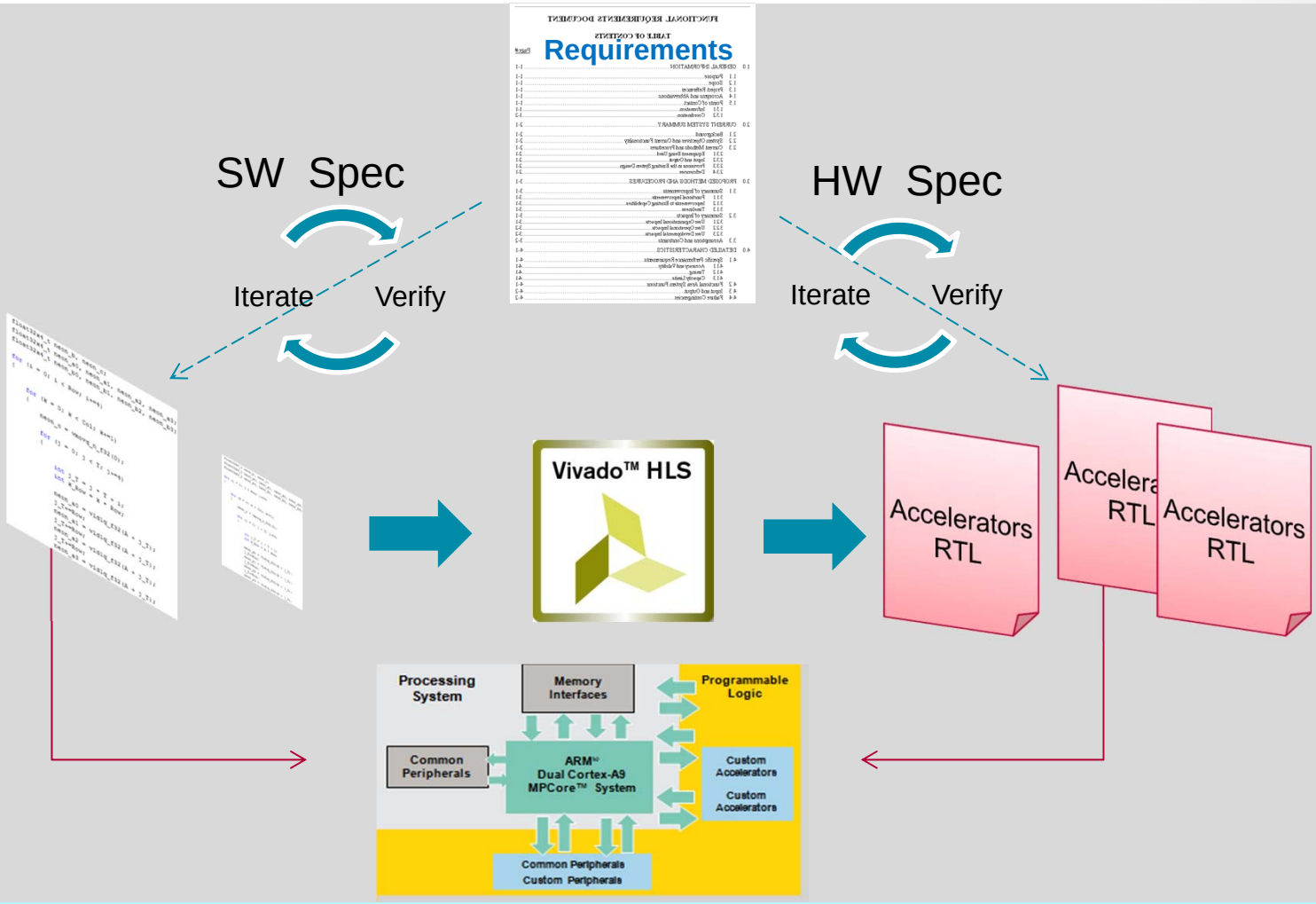
Accelerating Video Processing



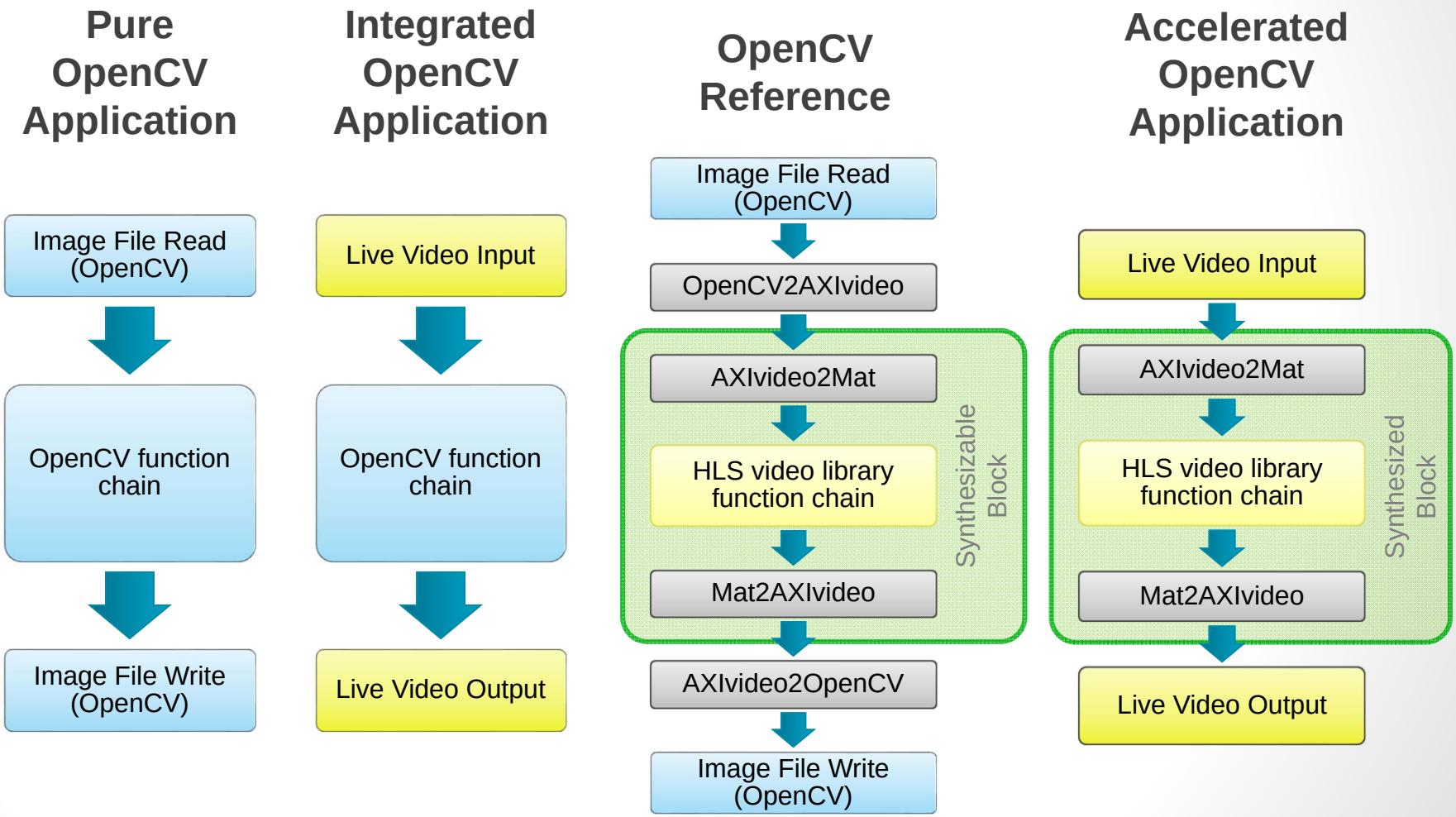


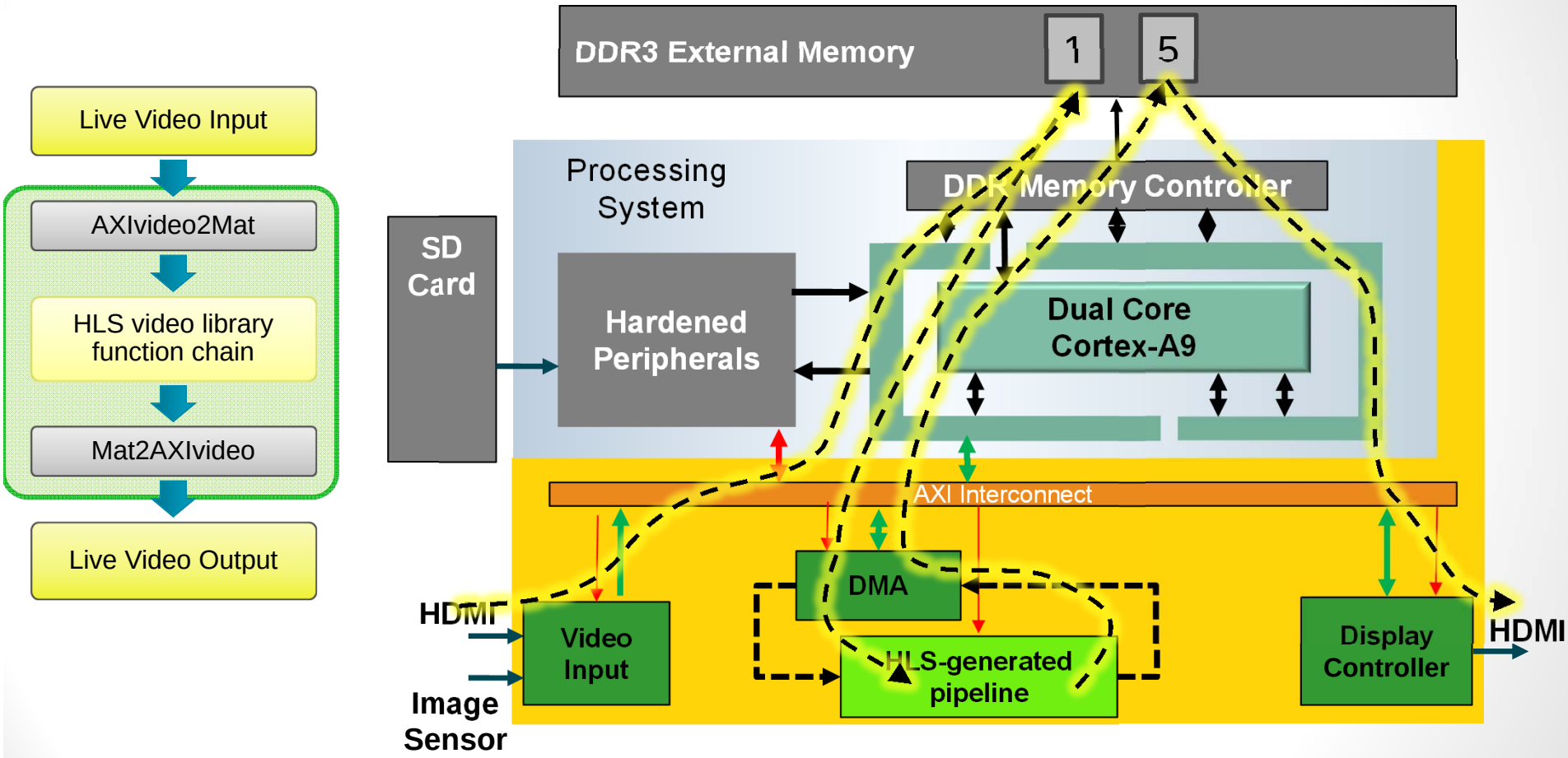
- Most pixel processing algorithms can be chained together with minimal buffering
  - e.g., most filters
  - Each buffer written and read by exactly one function
- Some algorithms require some buffering, but much less than a frame
  - Lens distortion correction
  - Stereo rectification
  - Processing with reconvergent paths
- Some algorithms really require frame buffers
  - Algorithms that process multiple frames (e.g., optical flow)
  - Algorithms with state (e.g., background subtraction)

Using High-Level Synthesis



Accelerates Algorithmic C to Co-Processing Accelerator Integration





HLS video libraries

- OpenCV functions are not directly synthesizable with HLS
  - Dynamic buffer allocation
- HLS video libraries are similar to OpenCV image processing functions
  - Require ‘streaming’ architectures

Video Library functions

- C++ code contained in `hls` namespace
- Similar interface, equivalent behavior with OpenCV, e.g.
  - OpenCV library:
  - HLS video library:

```
cvScale(src, dst, scale, shift);
```

```
hls::Scale<...>(src, dst, scale, shift);
```

- Some constructor arguments have corresponding or replacement template parameters: e.g.,
  - OpenCV library:
  - HLS video library:

```
cv::Mat mat(rows, cols, CV_8UC3);
```

```
hls::Mat<ROWS, COLS, HLS_8UC3> mat(rows, cols);
```

- ROWS and COLS specify the maximum size of an image processed

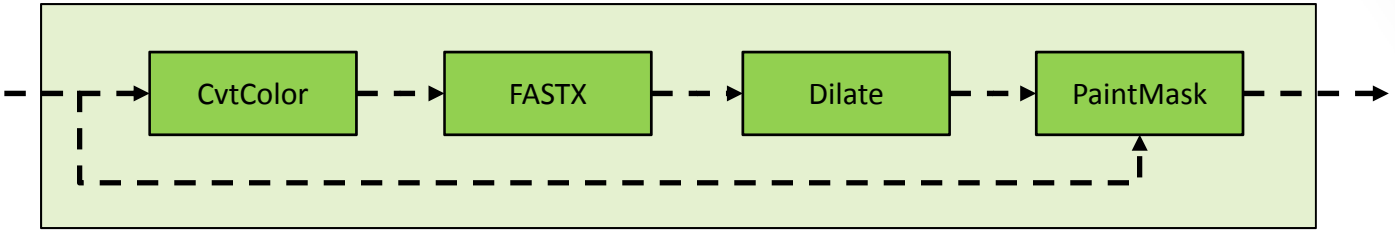
A More Complex Example

- This code is not ‘streaming’ and must be rewritten
  - Random access and in-place operation on ‘dst’

```
void opencv_image_filter(IplImage* img, IplImage* dst ) {
  IplImage* gray = cvCreateImage(cvSize(img->width,img->height), 8, 1 );
  cvCvtColor( img, gray, CV_BGR2GRAY );
  std::vector<cv::KeyPoint> keypoints;
  cv::Mat gray_mat(gray,0);
  cv::FAST(gray_mat, keypoints, 20,true );
  int rect=2;
  cvCopy(img,dst);
  for (int i=0; i<keypoints.size(); i++) {
    cvRectangle(dst,
      cvPoint(keypoints[i].pt.x,keypoints[i].pt.y),
      cvPoint(keypoints[i].pt.x+rect,keypoints[i].pt.y+rect),
    cvScalar(255,0,0),1);
  }
  cvReleaseImage( &gray );
}
```

opencv_top.cpp

Streaming Code




```
void opencv_image_filter(IplImage* src, IplImage* dst)
{
    IplImage* gray = cvCreateImage( cvGetSize(src), 8, 1 );
    IplImage* mask = cvCreateImage( cvGetSize(src), 8, 1 );
    IplImage* dmask = cvCreateImage( cvGetSize(src), 8, 1 );
    std::vector<cv::KeyPoint> keypoints;
    cv::Mat gray_mat(gray,0);

    cvCvtColor(src, gray, CV_BGR2GRAY );
    cv::FAST(gray_mat, keypoints, 20, true);
    GenMask(mask, keypoints);
    cvDilate(mask,dmask);
    cvCopy(src,dst);
    PaintMask(dst,dmask,cvScalar(255,0,0));

    cvReleaseImage( &mask );
    cvReleaseImage( &dmask );
    cvReleaseImage( &gray );
}
```

HLS Implementation

- Synthesizable code
  - Note '#pragma HLS stream"

```
    hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      _src(rows,cols);
    hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      _dst(rows,cols);
    hls::AXIvideo2Mat(input, _src);
    hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      src0(rows,cols);
    hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC3>      src1(rows,cols);
#pragma HLS stream depth=20000 variable=src1.data_stream
    hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC1>      mask(rows,cols);
    hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC1>      dmask(rows,cols);
    hls::Scalar<3,unsigned char> color(255,0,0);
    hls::Duplicate(_src,src0,src1);
    hls::Mat<MAX_HEIGHT,MAX_WIDTH,HLS_8UC1>      gray(rows,cols);
    hls::CvtColor<HLS_BGR2GRAY>(src0,gray);
    hls::FASTX(gray,mask,20,true);
    hls::Dilate(mask,dmask);
    hls::PaintMask(src1,dmask,_dst,color);
    hls::Mat2AXIvideo(_dst, output);
```

1080P60 Corner Detection

FPGA Pipelines

- Efficiently implement real-time video processing
- Throughput stays constant with more complex programs

Performance of Xilinx HLS video library

	Throughput * (Megapixels per sec)	Acceleration * (vs. OpenCV on ARM)
FAST9 Corner Detection	124	10x
Canny Edge Detection	124	14x
Harris Corner Detection	124	50x
Erosion/Dilation	124	5x
5x5 convolution	124	27x

* FPGA target is 1080P60
compared to single core 1GHz Cortex-A9

Conclusion

- Trends in vision processing
  - Higher resolution and more cameras -> smarter cameras
- Key bottlenecks in embedded video processing
  - Memory + computation -> dedicated processing pipelines
- Using OpenCV in embedded video processing
  - Run on ARM for non-bottleneck portions
  - Migrate to NEON acceleration for 2-20x speedup
  - Use as golden reference for HLS design in FPGAs for 10-100x speedup and real-time processing

Stephen Neuendorffer

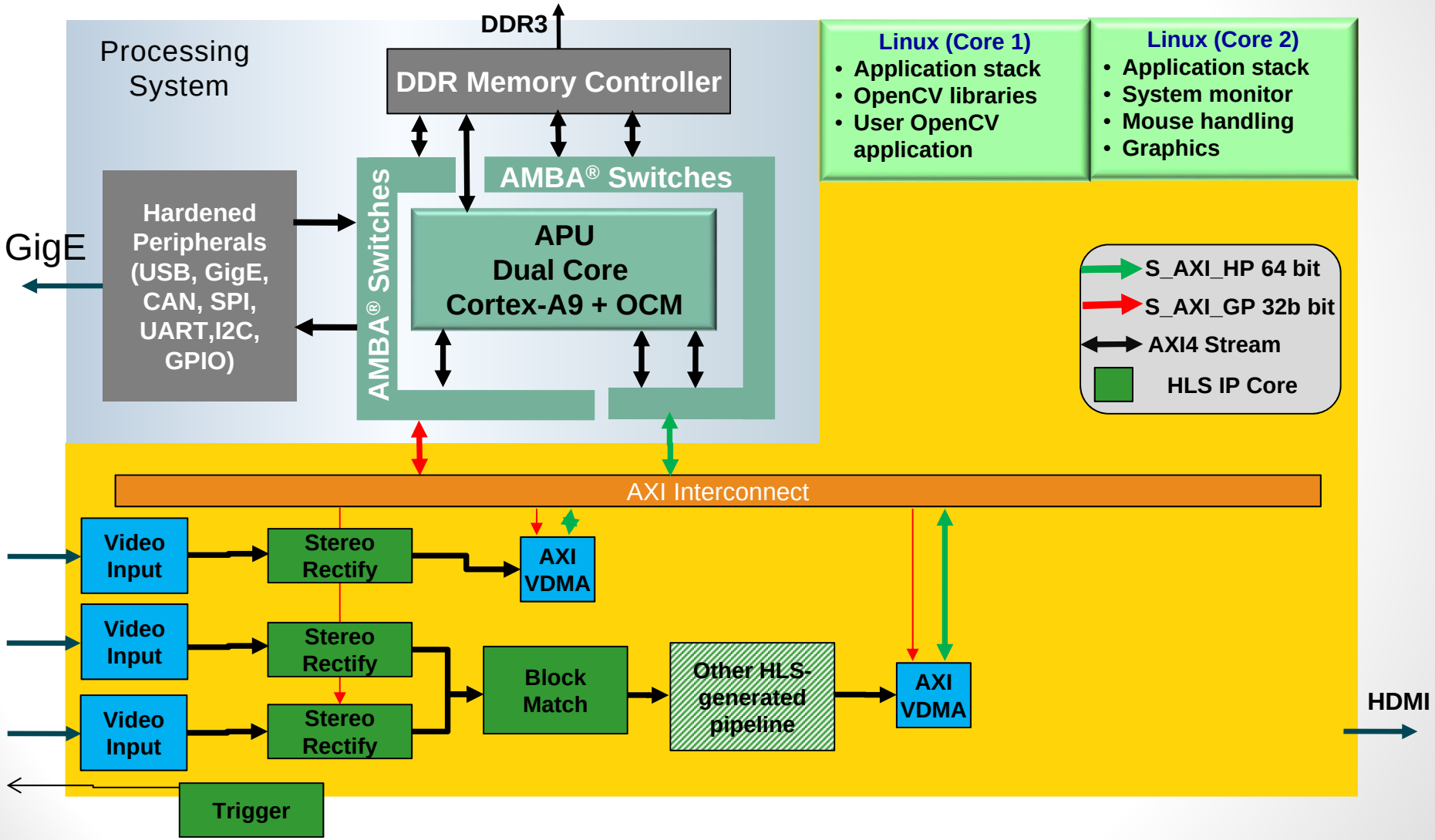
Principal Engineer

Xilinx
2100 Logic drive
San Jose, California
USA

Phone: +1 (408) 626-6359
Email: stephenn@xilinx.com

www.xilinx.com

Stereo Camera Architecture



#TheVisionShow