Tegra X1 and VisionWorks/OpenVX

Computer Vision on a Chip

Thierry Lepley

December 2015

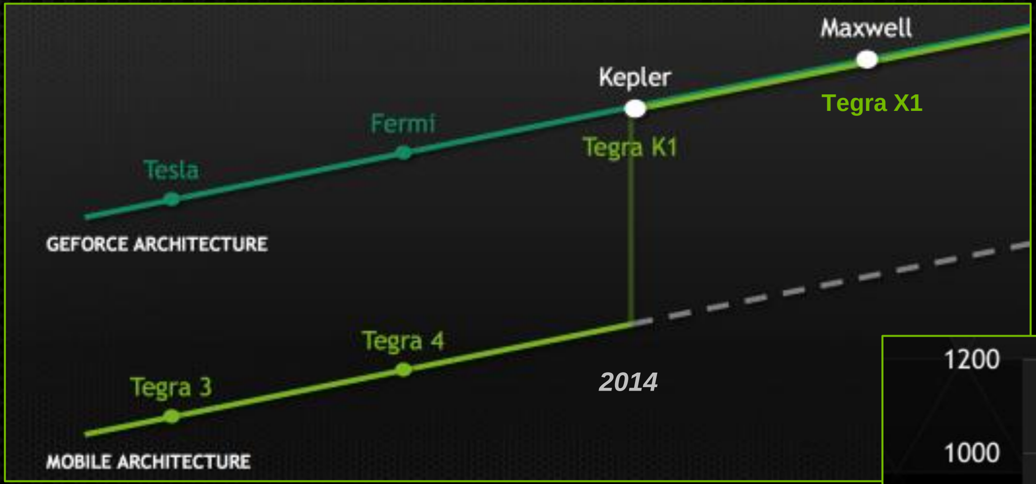
Agenda

1. Tegra X1 at a glance
2. Tegra X1 and Image processing
3. VisionWorks toolkit / OpenVX
4. Simple code example

Agenda

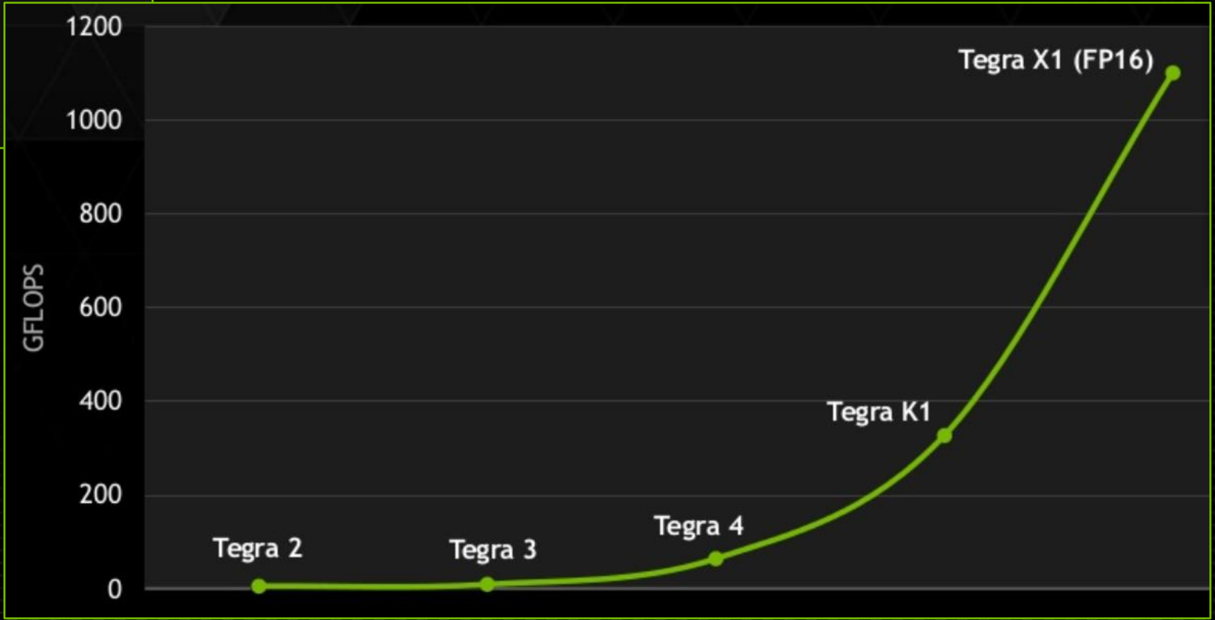
1. Tegra X1 at a glance
2. Tegra X1 and Image processing
3. VisionWorks toolkit / OpenVX
4. Simple code example

Embedded GPU : Evolution



Programming continuum
from embedded to super computing

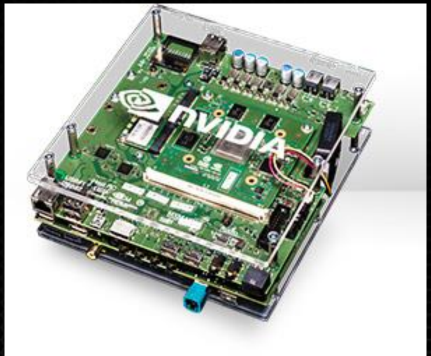
More compute performance
More performance per watt



Tegra K1 : first CUDA capable SoC

- Tegra TK1 :
  - 4+1 cores A15 (32 bits) or 2 cores (64 bits) ARM
  - Kepler architecture with 192 CUDA cores
    1 GK110 SM minus dynamic parallelism (compute capability 3.2)
  - 28 nm, up to 850 Mhz

Automotive



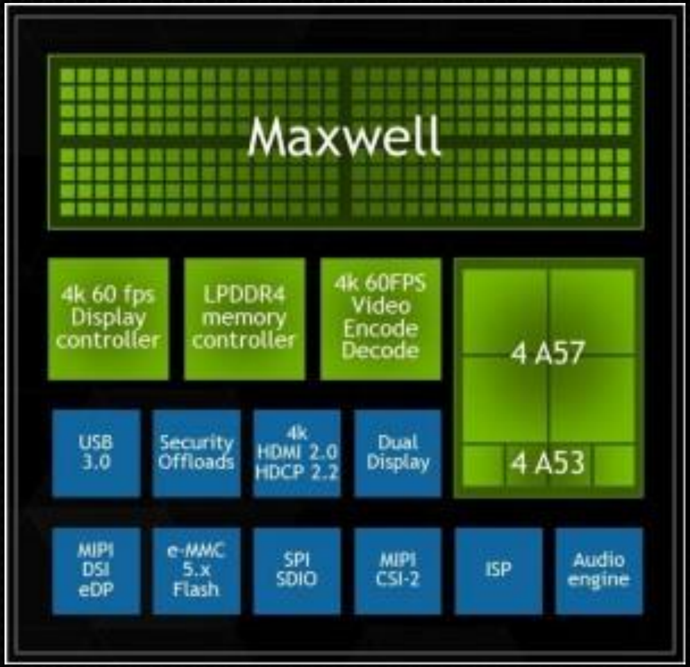
JETSON TK1 Pro

Embedded

192 $

JETSON TK1

TX1 : Main Features



- **4+4 cores (A57+A53) ARM 64-bit**
- Maxwell GPU (compute capability 5.3)
  - 2 SM, 256 CUDA cores
  - Dynamic parallelism
  - Supports CUDA 7
  - 2 x FP16 vector
- 20 nm, up to 1 Ghz, < 10 Watts

AlexNet image classifier (source: http://devblogs.nvidia.com/parallelforall)

platform	img / s	Power (AP+DRAM)	Perf/watt	Efficiency
Intel i7-6700K	242	62.5W	3.88	1x
Jetson TX1	258	5.7W	45	**11.5x**

TX1 : CUDA Capability (1/2)

TK1

TX1

<table>
<tr><th>Feature Support</th><th colspan="5">Compute Capability</th></tr>
<tr><th>(Unlisted features are supported for all compute capabilities)</th><th>2.x</th><th>3.0</th><th>3.2</th><th>3.5, 3.7, 5.0, 5.2</th><th>5.3</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td><td colspan="5" rowspan="13">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td></tr>
<tr><td>Warp vote functions (Warp Vote Functions)</td></tr>
<tr><td>Double-precision floating-point numbers</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())</td></tr>
<tr><td>__ballot() (Warp Vote Functions)</td></tr>
<tr><td>__threadfence_system() (Memory Fence Functions)</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or() (Synchronization Functions)</td></tr>
<tr><td>Surface functions (Surface Functions)</td></tr>
<tr><td>3D grid of thread blocks</td><td colspan="5">Yes</td></tr>
<tr><td>Unified Memory Programming</td><td>No</td><td colspan="4">Yes</td></tr>
<tr><td>Funnel shift (see reference manual)</td><td colspan="2">No</td><td colspan="3">Yes</td></tr>
<tr><td>Dynamic Parallelism</td><td colspan="3">No</td><td colspan="2">Yes</td></tr>
<tr><td>Half-precision floating-point operations: addition, subtraction, multiplication, comparison, warp shuffle functions, conversion</td><td colspan="4">No</td><td>Yes</td></tr>
</table>

TX1 : CUDA Capability (2/2)

TK1

TX1

	Compute Capability							
Technical Specifications	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3
Maximum number of resident grids per device (Concurrent Kernel Execution)	16		4	32				16
Maximum dimensionality of grid of thread blocks	3							
Maximum x-dimension of a grid of thread blocks	65535	2^{31}-1						
Maximum y- or z-dimension of a grid of thread blocks	65535							
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	1024							
Maximum z-dimension of a block	64							
Maximum number of threads per block	1024							
Warp size	32							
Maximum number of resident blocks per multiprocessor	8	16				32		
Maximum number of resident warps per multiprocessor	48	64						
Maximum number of resident threads per multiprocessor	1536	2048						
Number of 32-bit registers per multiprocessor	32 K	64 K			128 K	64 K		
Maximum number of 32-bit registers per thread block	32 K	64 K						32 K
Maximum number of 32-bit registers per thread	63			255				
Maximum amount of shared memory per multiprocessor	48 KB				112 KB	64 KB	96 KB	64 KB
Maximum amount of shared memory per thread block	48 KB							
Number of shared memory banks	32							
Amount of local memory per thread	512 KB							
Constant memory size	64 KB							
Cache working set per multiprocessor for constant memory	8 KB					10 KB		
Cache working set per multiprocessor for texture memory	12 KB	Between 12 KB and 48 KB						

TX1 Platforms / Boards

Automotive

Drive PX

Embedded

JETSON TX1

599 $

Education
299$

CSI interface : up to 6 cameras

http://www.nvidia.com/object/embedded-systems.html

https://developer.nvidia.com/embedded-computing

Development on TX1

- Same development environment as on desktop
  - L4T (Linux for Tegra) based on Ubuntu14.04
  - SSH for remote connection; can also plug keyboard/move/screen
  - Development :
    - Usual GCC, GDB tools
    - *No mandatory cross compilation*
  - *Tools*
    - **NVIDIA Visual Profiler (Visual Studio / Linux-Eclipse)**
    - **CUDA MEMCHECK**
- *Other*
  - **Tegra System Profile** (CPU/GPU combined profiling)

Agenda

1. Tegra X1 at a glance
2. Tegra X1 and Image processing
3. VisionWorks toolkit / OpenVX
4. Simple code example

Image Processing on Tegra X1

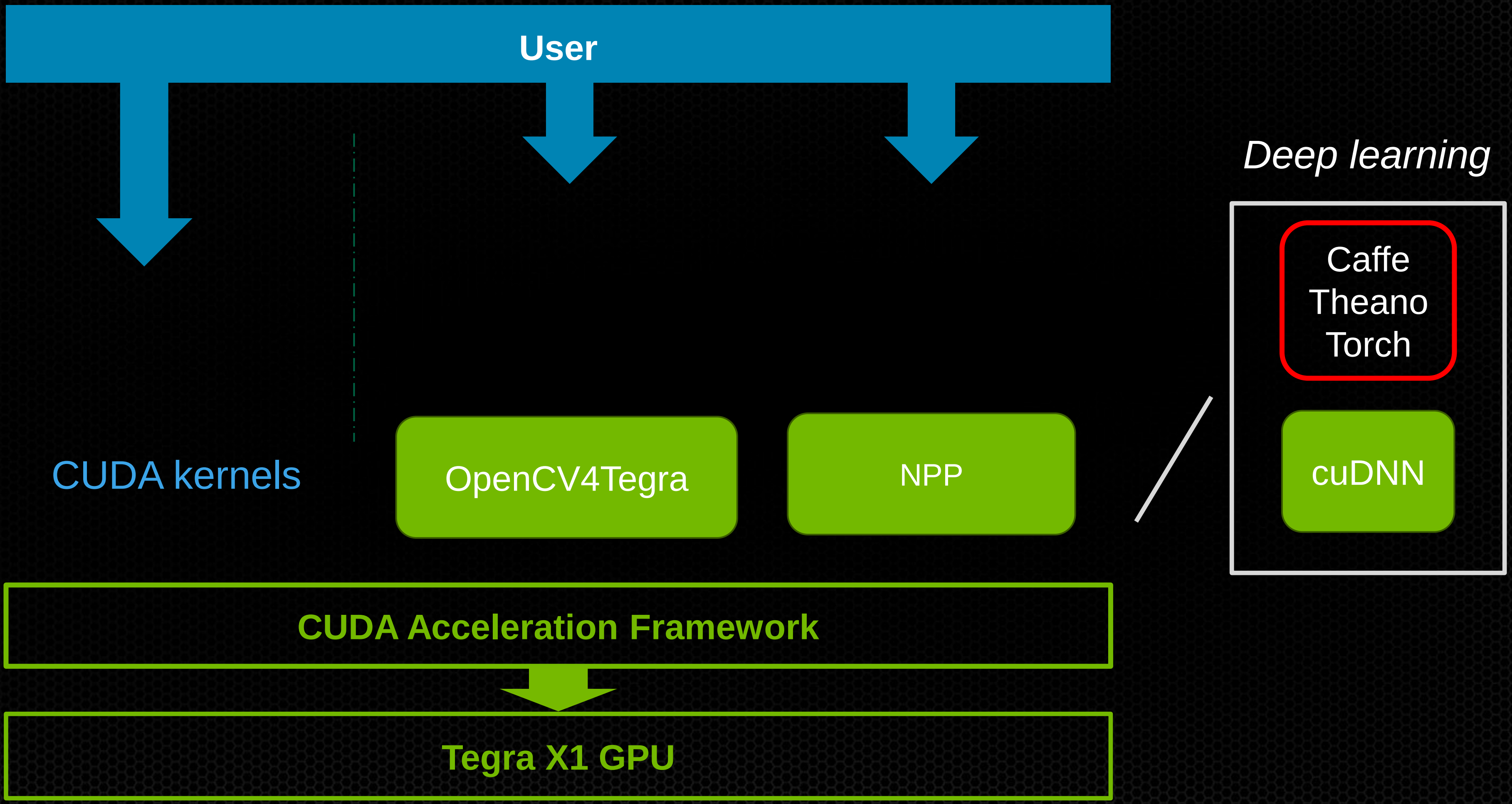
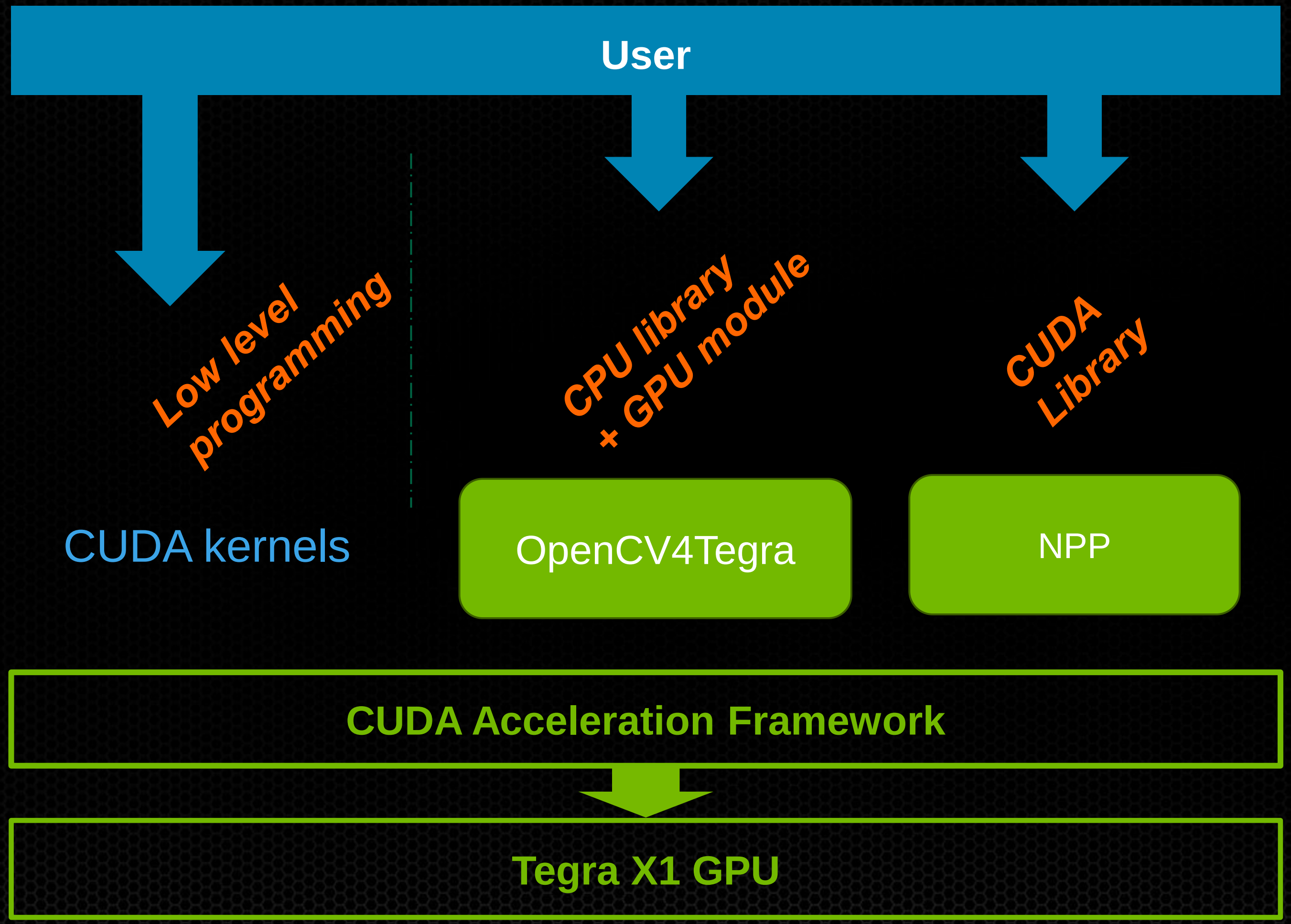


Image Processing on Tegra X1



Developer Problematic

- Low value added re-optimizing classical image processing primitives for each new architecture
  - ➢ Use a library of image processing primitive
- Different flavors of memory hierarchy across platforms
  Ex: Tegra - shared memory, Desktop - discrete memory
  - ➢ Abstract the memory
- Systems Heterogeneous (GPU + CPU + HW accelerators)
  - ➢ Abstract CUDA to fully cover Tegra
- All can't be in a library
  - ➢ Need user extensibility

OpenVX Approach

1. Library of optimized Vision primitives
    1. ~40 primitives (from simple arithmetic to more complex keypoint detection/tracking). Will increase as consensus reached within industry
    2. Border modes supports, notion of valid region
    3. Extensible : users can add their own primitives
2. Set of opaque CV data containers
    - Image, pyramid, array, matrix, scalar, LUT, Convolution matrix, Remap matrix, Distribution, Threshold
    - Advanced : images created from handle / ROI, delays, …
3. Framework for assembling and executing primitives
    1. Immediate mode : fast development, one time processing
    2. Graph mode : repeating processing (ex: video processing)

OpenVX at a Glance

- Standard defined by Khronos (OpenGL, OpenCL, OpenMax etc..)
- Majors OpenVX goals
    1. Define a subset of relevant primitives and image/data format
    2. Enable acceleration on complex and heterogeneous architectures
    3. Enable more optimization opportunities than OpenCV
- Timeline
    - Started early 2012, OpenVX 1.0 released in Oct 2014, 1.0.1 in May 2015
    - NVIDIA releases the first public OpenVX 1.0.1 production implementation (Nov 2015)
- Some of the contributors : TI, Itseez, NVIDIA, Qualcomm, Samsung, Intel, AMD, CEVA, Axis, Cognivue, ST

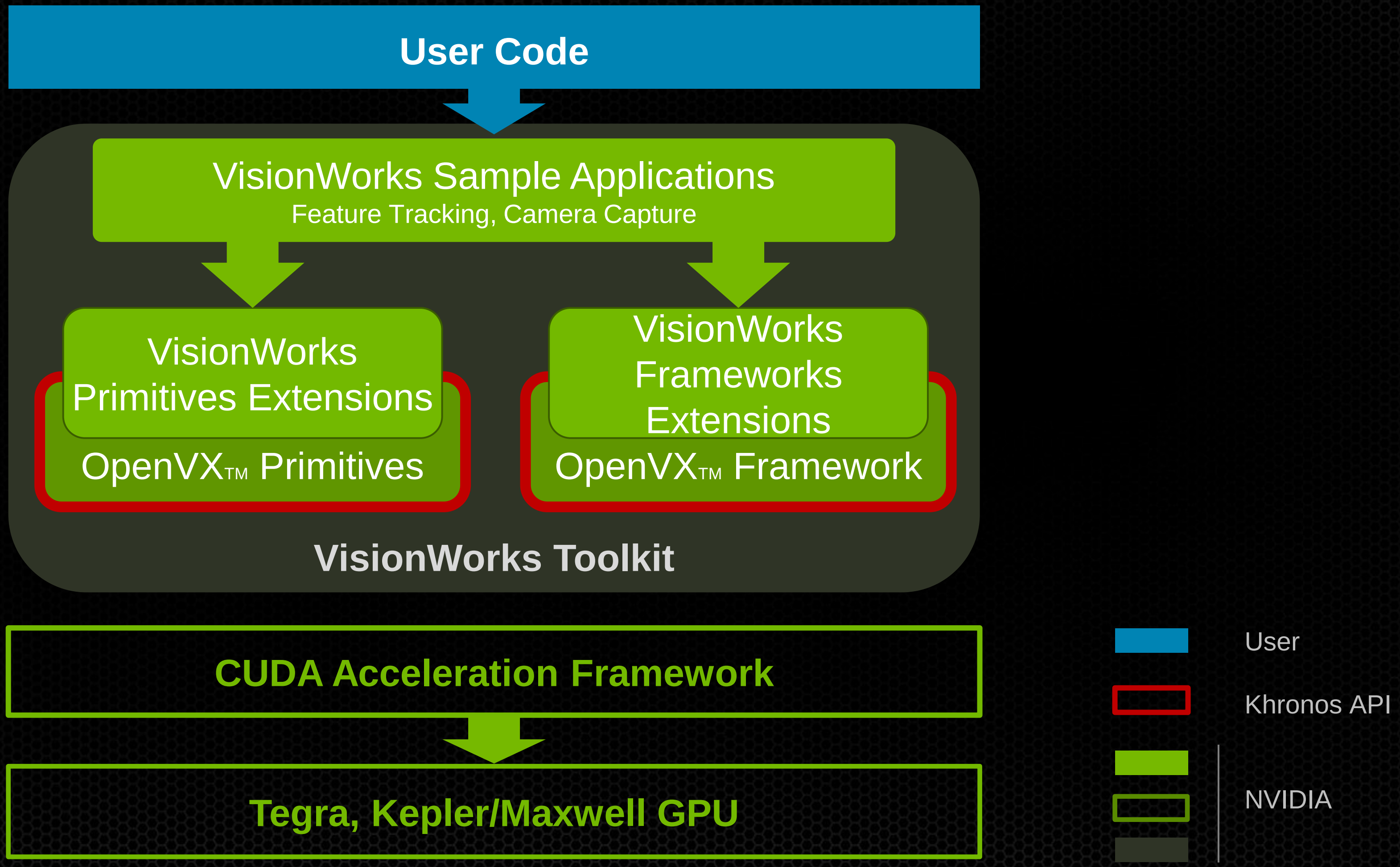
Agenda

1. Tegra X1 at a glance
2. Tegra X1 and Image processing
3. VisionWorks toolkit / OpenVX
4. Simple code example

What is VisionWorks

- Production implementation of OpenVX + extensions + CUDA interop
- High performance & robust Computer Vision primitives for the TX1
- Performance portability across different NVIDIA systems

VisionWorks Software Stack



VisionWorks Primitives

IMAGE ARITHMETIC
Absolute Difference
Accumulate Image
Accumulate Squared
Accumulate Weighted
Add / Subtract / Multiply
Channel Combine
Channel Extract
Color Convert +
CopyImage
Convert Depth
Magnitude
Not / Or / And / Xor
Phase
Table Lookup
Threshold

FLOW & DEPTH
Median Flow
Optical Flow (LK)
Semi-Global Matching
Stereo Block Matching

GEOMETRIC TRANSFORMS
Affine Warp +
Perspective Warp +
Flip Image
Remap
Scale Image +

FILTERS
BoxFilter
Convolution
Dilation Filter
Erosion Filter
Gaussian Filter
Gaussian Pyramid
Laplacian3x3
Median Filter
Sobel 3x3
Scharr3x3

FEATURES
Canny Edge Detector
Fast Corners +
Fast Track
Harris Corners +
Harris Track
Hough Circles
Hough Lines

ANALYSIS
Histogram
Histogram Equalization
Integral Image
Mean Std Deviation
Min Max Locations

Primitive	*Standard OpenVX*
Primitive +	*Standard with NVIDIA Extensions*
Primitive	*NVIDIA Proprietary*

VisionWorks Objects : Data

Images

- Image: vx_image +
  (RGB, different flavors of YUV, gray scale)
- Image Pyramid : vx_pyramid +

Arrays

- Array : vx_array +
- Distribution : vx_distribution +
- Look-up-table : vx_lut +

Matrices

- Matrix: vx_matrix +
- Convolution : vx_convolution +
- Remap : vx_remap +

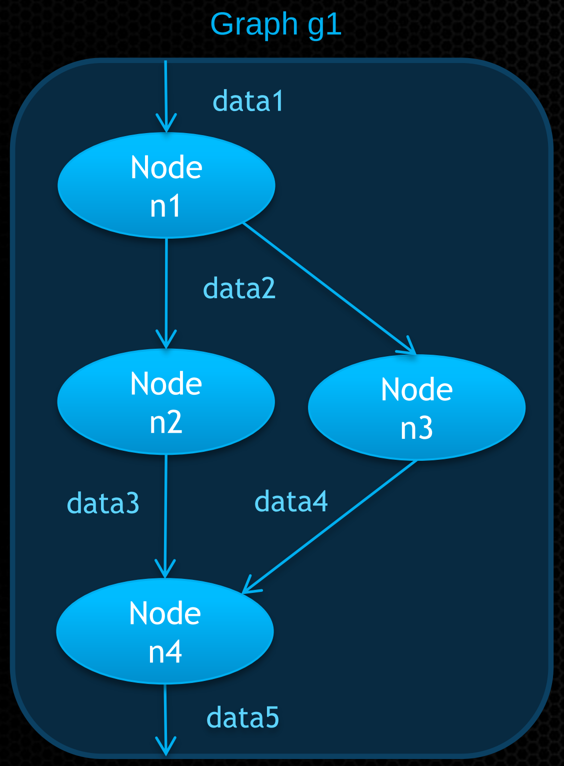
Scalars

- Scalar : vx_scalar +
- Threshold : vx_threshold +

Object + Standard OpenVX with NVIDIA Extensions (ex: access from CUDA)

VisionWorks Objects : Framework

- The 'world' : vx_context +
- CV 'pipeline'
  - Graph : vx_graph +
  - Graph node (instance of kernel): vx_node
- CV function/primitive: vx_kernel +
- Parameter (node & kernel): vx_parameter
- Circular buffer of data objects : vx_delay



Object	*Standard OpenVX*
Object +	*Standard with NVIDIA Extensions*

Objects Philosophy

- The application only gets <u>references</u> to objects
  - The application *can't destroy* an object, only *release* it
  - The object stays alive until not referenced anymore
- Object life cycle
  1. <u>Create</u> an object and get a reference to it

```
vx_image img = vxCreateImage(context, 640, 480, VX_DF_IMAGE_RGB);
```

  2. <u>Use/manipulate</u> the object
  3. <u>Release</u> the object reference when the no need to use this object anymore

```
vxReleaseImage(&img);
```

✓ VisionWorks manages object resource allocation and destruction

Data Objects are Opaque

- Data Object Content Access
  - Controlled : access to *explicit* and *temporary* (no permanent pointer)
  - Usage given by the application (RO, RW, WO)
  - Two access modes : ‘copy’ and ‘map’
  - Two access memory spaces : CPU or CUDA

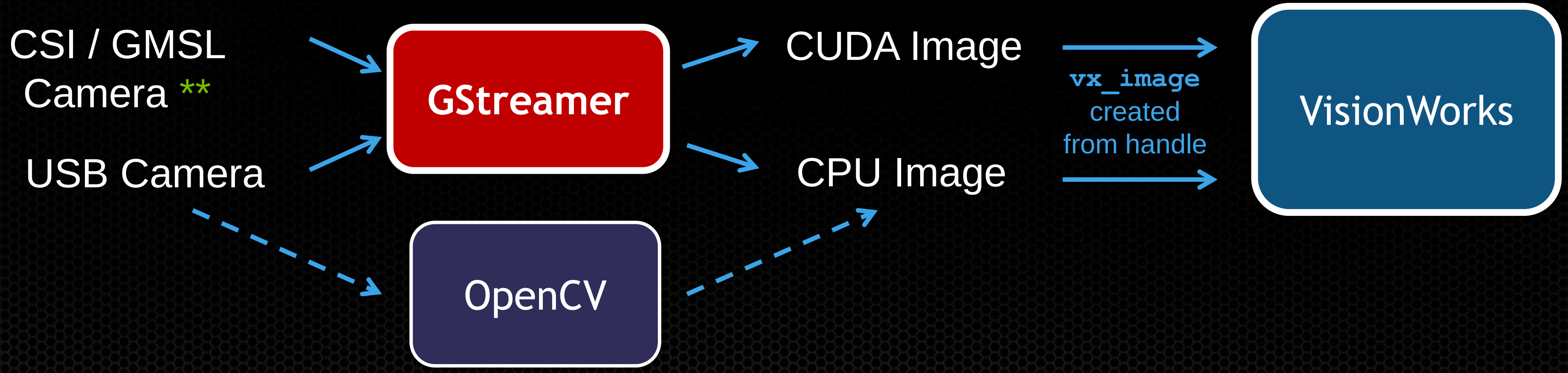
```
void *devptr = 0; vx_rectangle_t rect = {0, width, 0, height};
vxAccessImagePatch (image, &rect, &addr, 0, &devptr, NVX_WRITE_ONLY_CUDA);
// …
vxCommitImagePatch (image, &rect, 0, &addr, devptr);
```

✓ VisionWorks manages
  - Data movements across the memory hierarchy
  - Memory layout of data objects (*with exceptions*)
  - ➢ *Best for acceleration and performance portability*

Efficient I/O : Zero Copy

- Image created 'from handle' (application memory)
  - Import an external image without making a copy
    - Can be CPU or CUDA memory
    - Memory layout under application control in this case
  - For both input and output images

```
vx_image img = vxCreateImageFromHandle(context, VX_DF_IMAGE_RGB, addr, ptrs,
                                       NVX_IMPORT_TYPE_CUDA);
```



** NVIDIA Embedded platforms support CSI Cameras; NVIDIA Automotive platforms support GMSL Cameras

Primitives Execution

- *Immediate* execution
  - Blocking calls similar to OpenCV usage model
  - ✓ Useful for fast prototyping, one-shot processing

```
vxuBox3x3(context, in0, tmp);
vxuAbsDiff(context, tmp, in1, out);
```

- *Graph*
  - Primitive sequence given ahead-of-time
  - ✓ More optimization opportunities
  - ✓ Useful for video stream processing

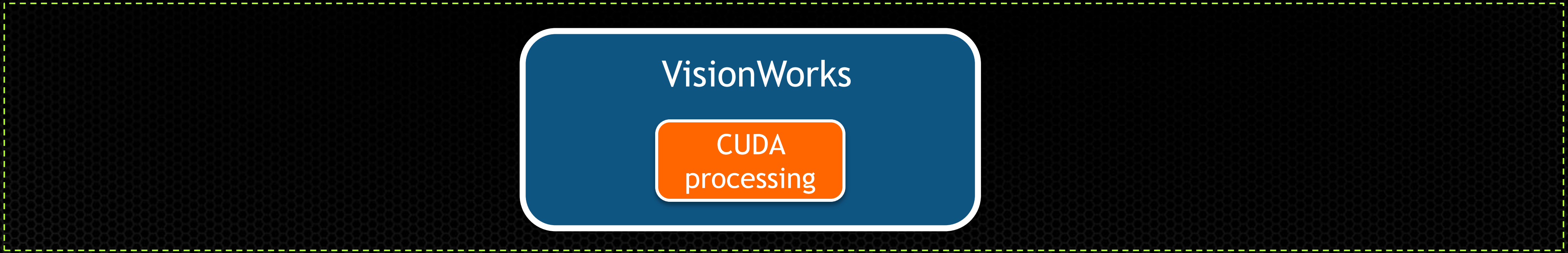
```
vx_graph graph = vxCreateGraph(context);

vxBox3x3Node(graph, in0, tmp);
vxAbsDiffNode(graph, tmp, in1, out);

vxVerifyGraph(graph);
vxProcessGraph(graph);
```

Different flavours of CUDA interop

Image created from CUDA handle

Access Objects from CUDA

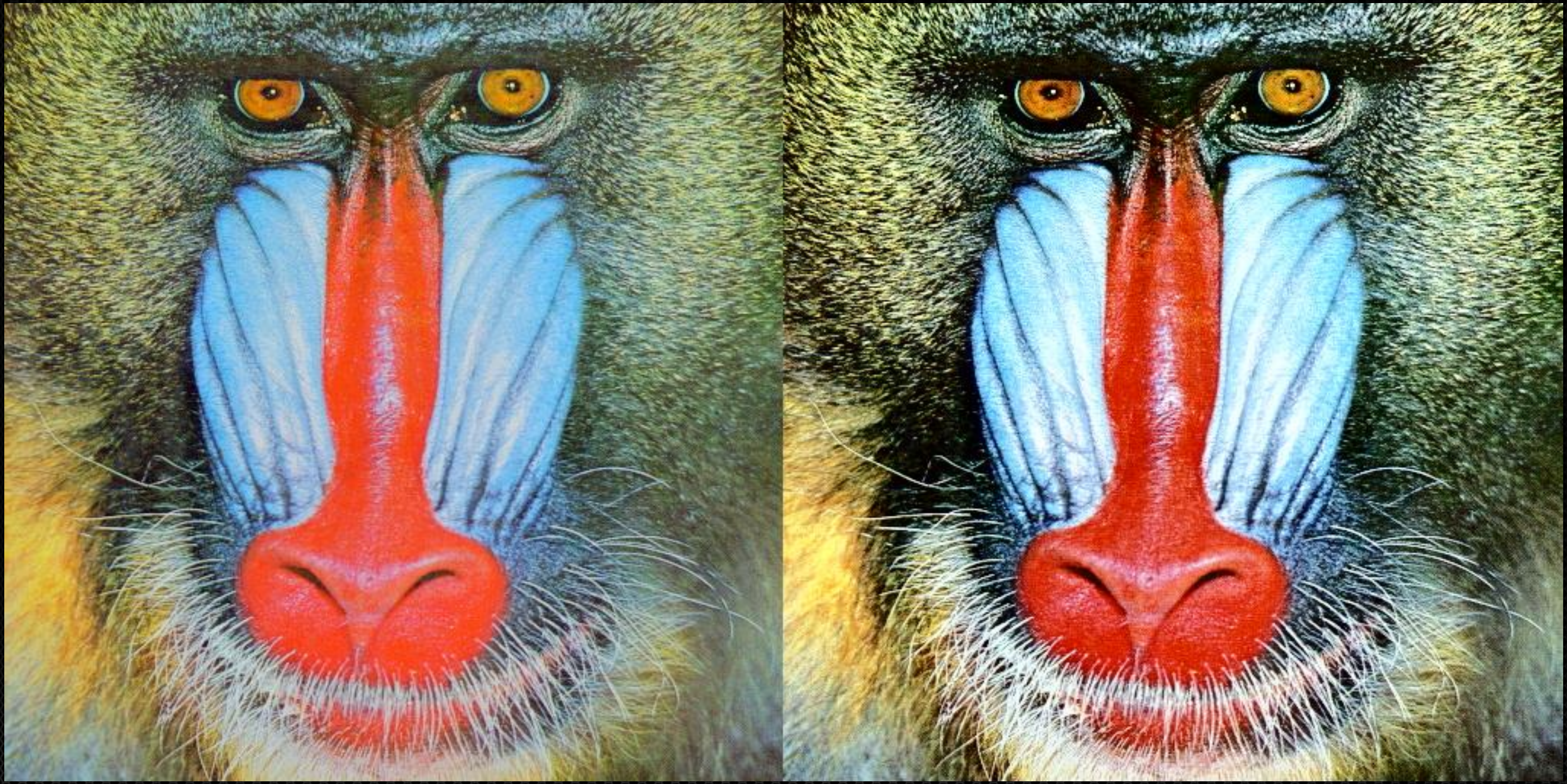
User Kernel CUDA



Agenda

1. Tegra X1 at a glance
2. Tegra X1 and Image processing
3. VisionWorks toolkit / OpenVX
4. Simple code example

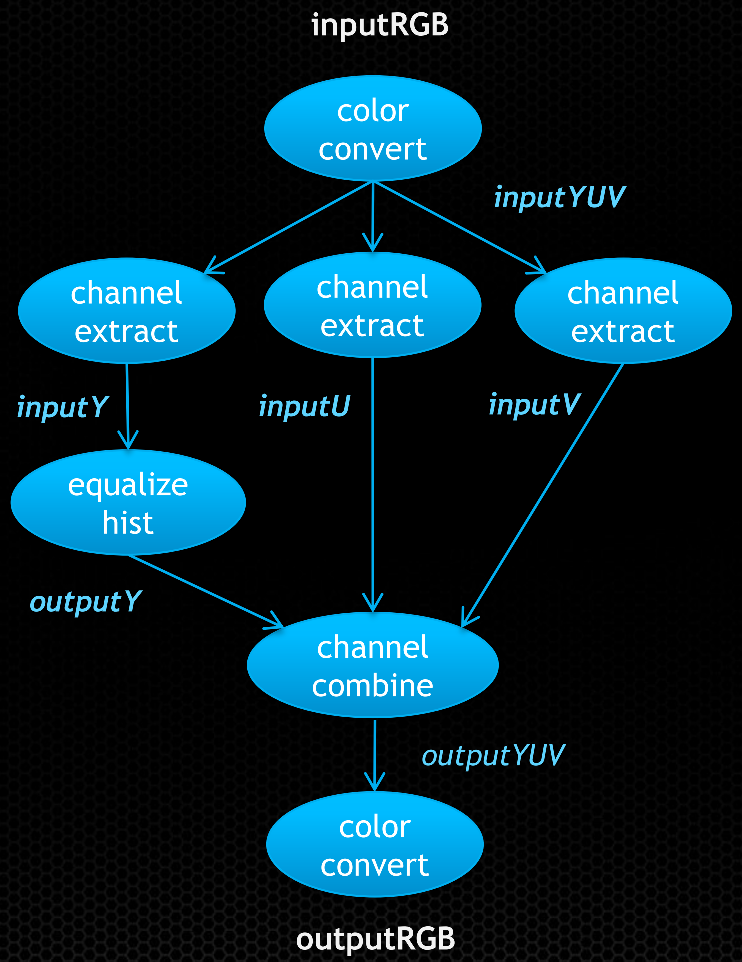
Histogram Equalization



Histogram Equalization: What is in the Toolkit ?

- Image Equalization
  - `EqualizeHist`: Histogram equalization of a gray scale image
- Image conversion
  - `ColorConvert`: Convert between color formats
  - `ChannelExtract`: Extraction of a particular channel of a color image
  - `ChannelCombine`: Create a color image from separate channels
- Framework
  - *Virtual images*: Intermediate graph images, can be optimized out

Histogram Equalization Processing

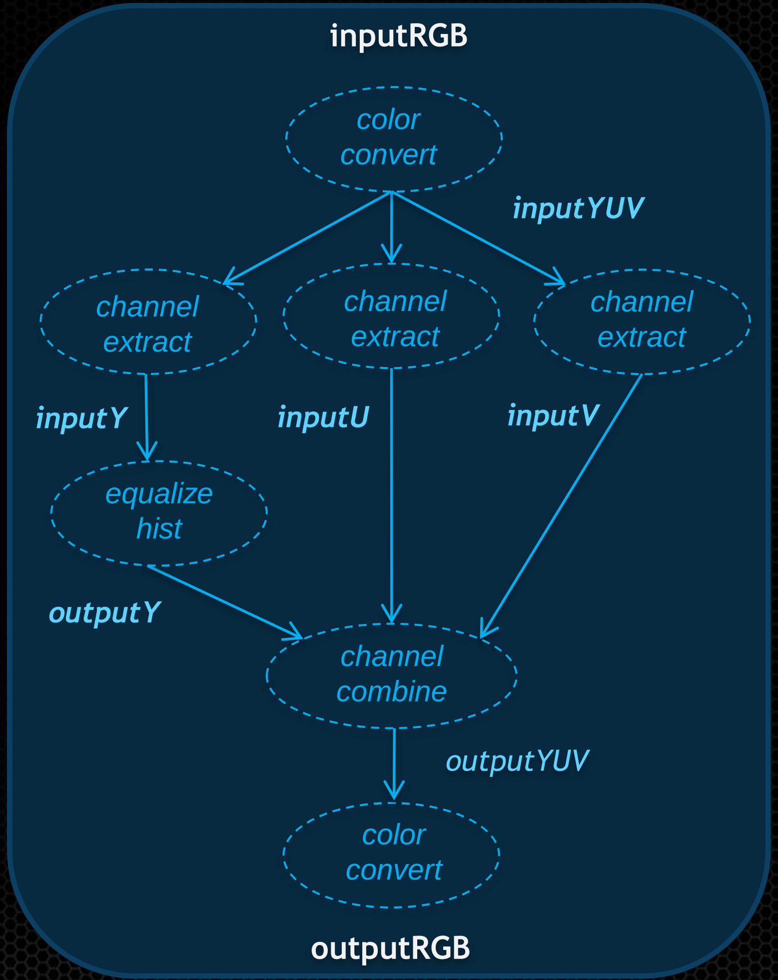


What Needs To Be Done

1. Preparation
   a) Create an OpenVX context
   b) Create the histogram equalization graph
      - Create the graph object
      - Create data objects
      - Create Nodes
   c) Verify the graph
2. Processing
   a) Execute the graph on a video sequence

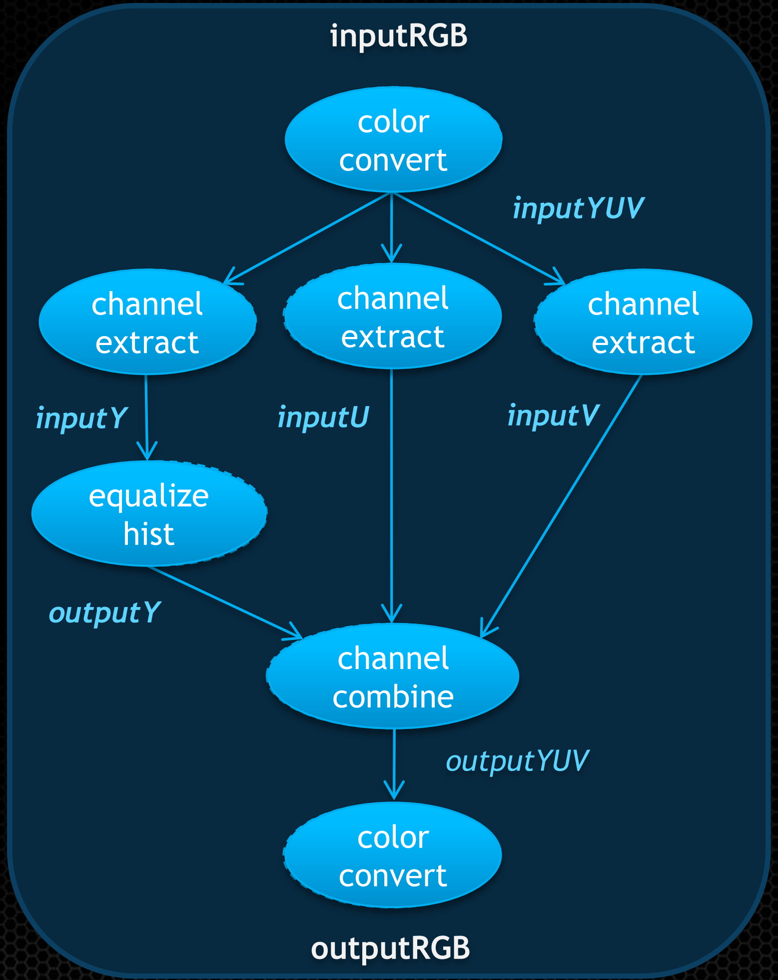
Context & Data Objects

```
// Create the context
vx_context context = vxCreateContext ();

// Import the input data into OpenVX
vx_image inputRGB  = vxCreateImageFromHandle (context, VX_DF_IMAGE_RGB,
                                               addr, ptrs, NVX_IMPORT_TYPE_CUDA);
// Create the output image
vx_image outputRGB = vxCreateImage (context, width, height, VX_DF_IMAGE_RGB);

// Create the graph (necessary for creating virtual objects)
vx_graph graph = vxCreateGraph(context);

// Create intermediate graph data objects
vx_image inputYUV = vxCreateVirtualImage (context, 0, 0, VX_DF_IMAGE_IYUV);
vx_image outputYUV = vxCreateVirtualImage (context, 0, 0, VX_DF_IMAGE_IYUV);
vx_image inputY = vxCreateVirtualImage (context, 0, 0, VX_DF_IMAGE_VIRT);
vx_image inputU = vxCreateVirtualImage (context, 0, 0, VX_DF_IMAGE_VIRT);
vx_image inputV = vxCreateVirtualImage (context, 0, 0, VX_DF_IMAGE_VIRT);
vx_image outputY = vxCreateVirtualImage (context, 0, 0, VX_DF_IMAGE_VIRT);
```



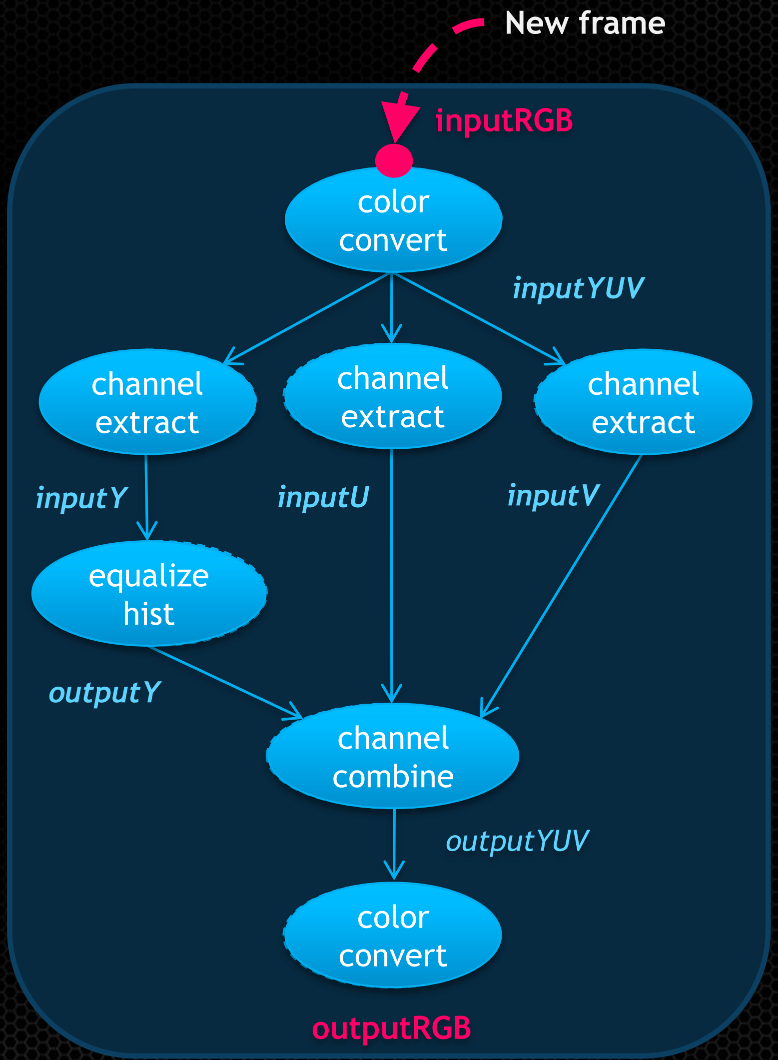
Nodes Creation

```
// RGB to Y conversion nodes
vxColorConvertNode (graph, inputRGB, inputYUV);

// Extraction of channels
vxChannelExtractNode (graph, inputYUV, VX_CHANNEL_Y,  inputY);
vxChannelExtractNode (graph, inputYUV, VX_CHANNEL_U,  inputU);
vxChannelExtractNode (graph, inputYUV, VX_CHANNEL_V,  inputV);

// Histogram equalization (gray scale)
vxEqualizeHistNode (graph, inputY, outputY);

// Build the output image
vxChannelCombineNode (graph, outputY, inputU, inputV, NULL, outputYUV);

// Y to RGB conversion nodes
vxColorConvertNode (graph, outputYUV, outputRGB);

// Graph verification
status = vxVerifyGraph (graph);
```



Execution

```
// Context & data objects creation
// <…>
// Graph creation & verification
// <…>

// Main processing loop
for (;;) {
    void *devptr = 0;
    vx_rectangle_t rect = {0, width, 0, height};
    vxAccessImagePatch (inputRGB, &rect, &addr, 0, &devptr, NVX_WRITE_ONLY_CUDA);
    // Get next frame data into 'devptr' here
    // <…>
    vxCommitImagePatch (inputRGB, &rect, 0, &addr, devptr);

    // Process graph
    vxProcessGraph (graph);

    // Use outputRGB
    // <…>
}
```



NVIDIA VISIONWORKS™

Now available on Tegra X1

and Desktop Ubuntu 14.04

https://developer.nvidia.com/embedded/visionworks