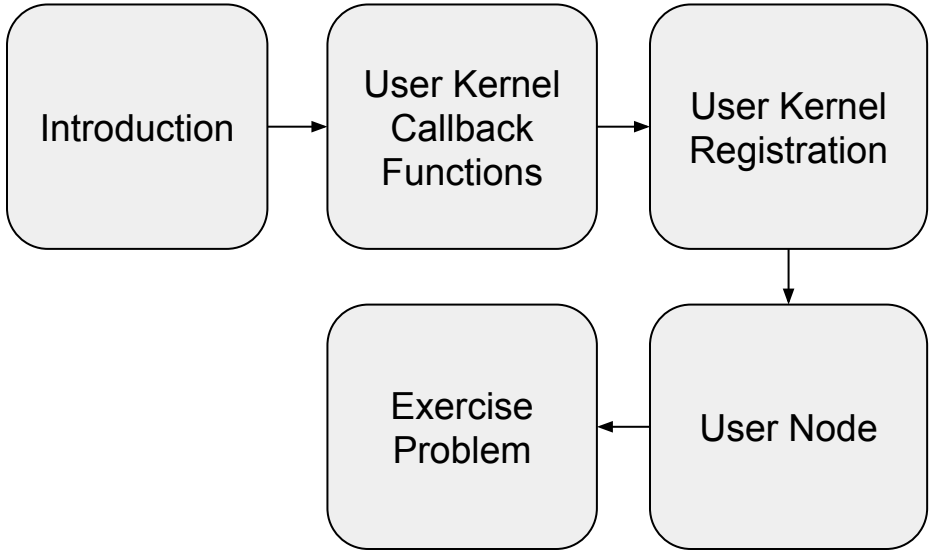
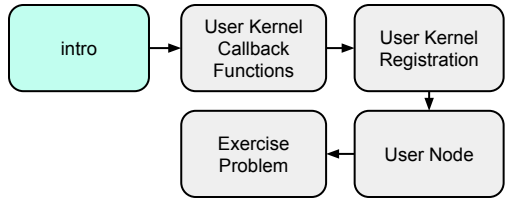
Tutorial Practice Session

Step 3: User Kernels and Nodes

Agenda



Kernel and Node Terminology



- **KERNEL**

Implementation code of a CV primitive, generic with regard to data objects

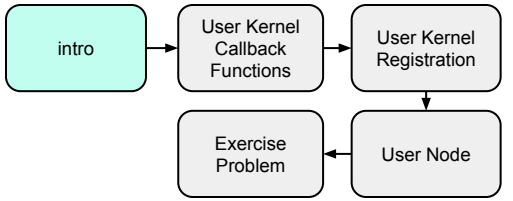
- C++ world: Class
- C world : Function

- **NODE**

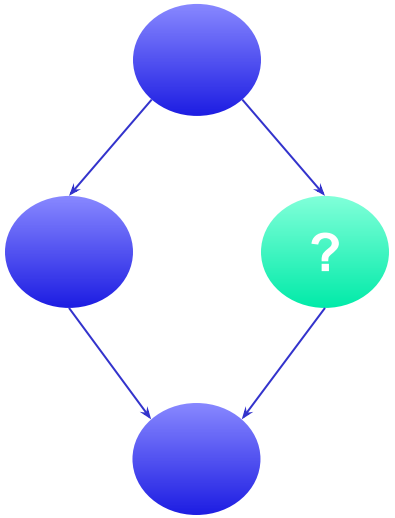
Instance of a kernel in a graph with well defined data objects parameters

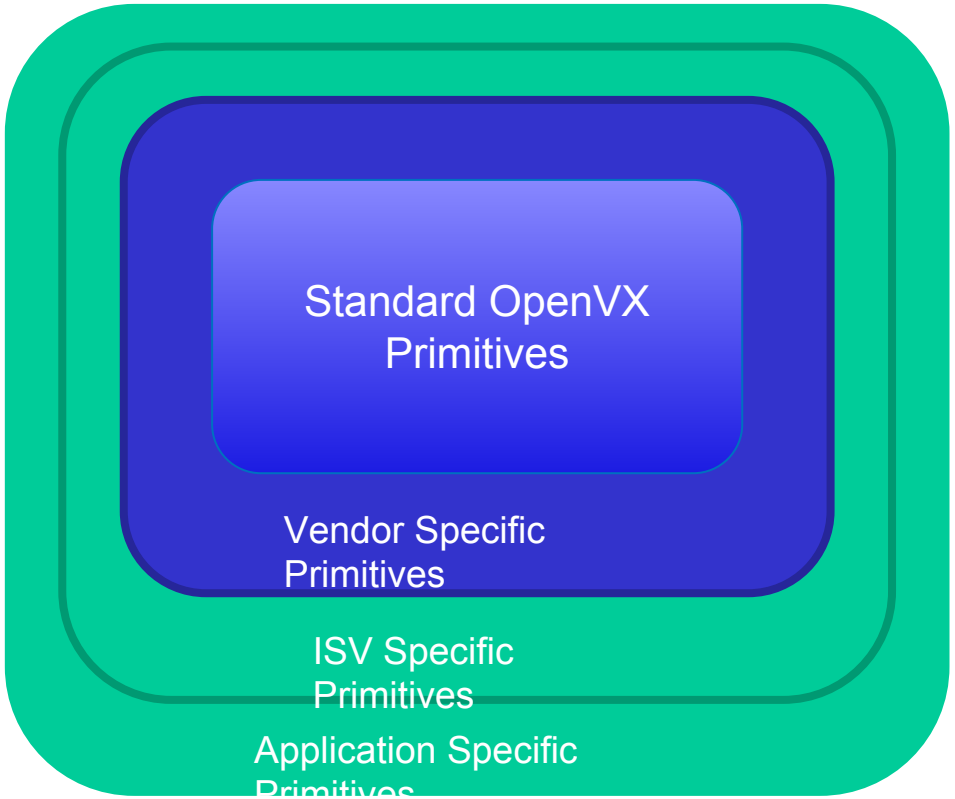
- C++ world: Object
- C world : Function Call

User Kernel, What for ?

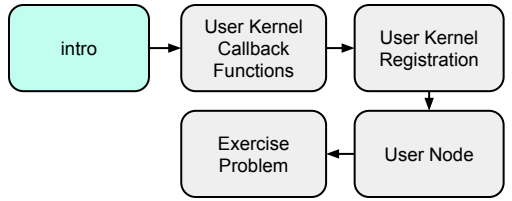


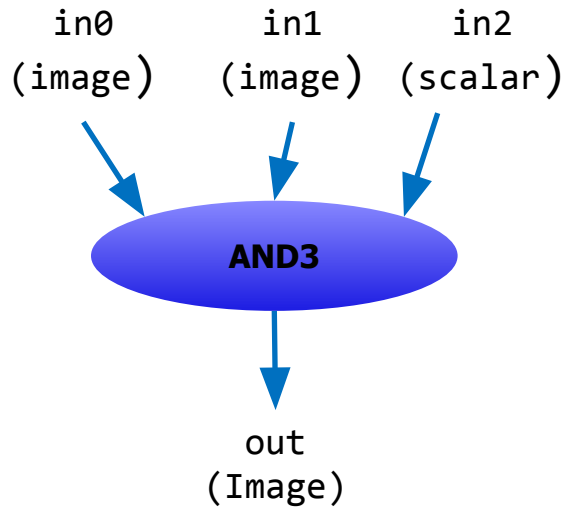
- Extend the set of CV primitives
- Avoid breaking the execution flow





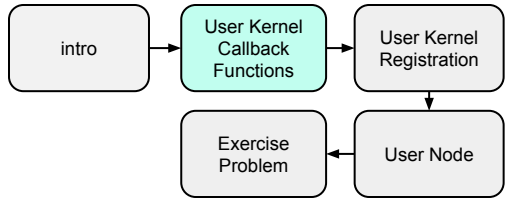
Illustrative Example





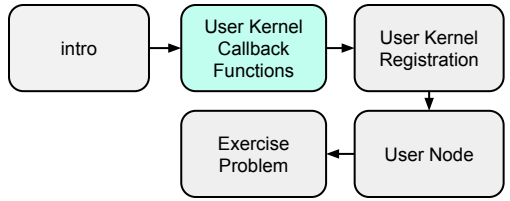

```
// Supports only U8
vxAnd3Node(vx_graph graph,
    vx_image in0, vx_image in1, vx_scalar in2,
    vx_image out)
```

User Kernels Callbacks



1. **Input validation function**
   Called at graph (re)verification
2. **Output validation function**
   Called at graph (re)verification
3. **Initialization function (optional)**
   Called at graph (re)verification
4. **Deinitializer function (optional)**
   Called at graph re-verification and destruction
5. **Process function (Host)**
   Called at node execution

Kernel Input Validation



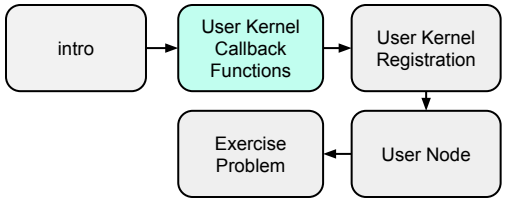
- Prototype

```
vx_status (vx_kernel_input_validate_f)(vx_node node, vx_uint32 index);
```

- AND3 Example

```
vx_status And3InputValidator(vx_node node, vx_uint32 index) {
   vx_parameter param = vxGetParameterByIndex(node, index);
   vx_reference input = NULL;
   vxQueryParameter(param, VX_PARAMETER_ATTRIBUTE_REF, &input, sizeof(input));
   if( index == 0 || index == 1 ) {
     vx_df_image format = 0;
     vxQueryImage((vx_image)input, VX_IMAGE_ATTRIBUTE_FORMAT, &format, sizeof(format));
     if (format != VX_DF_IMAGE_U8) return VX_ERROR_INVALID_FORMAT;
   }
   if( index == 1 ) {
      // Check if width & height of first two image parameters are same
   }
   if( index == 2 ) {
     vx_enum scalar_type = 0;
     vxQueryScalar((vx_scalar)input, VX_SCALAR_ATTRIBUTE_TYPE, &type, sizeof(type));
     if(type != VX_TYPE_UINT8) return VX_ERROR_INVALID_TYPE;
   }
   // release all references using vxReleaseParameter/vxReleaseImage/vxReleaseScalar ...
   return VX_SUCCESS;
}
```

Kernel Output Validation



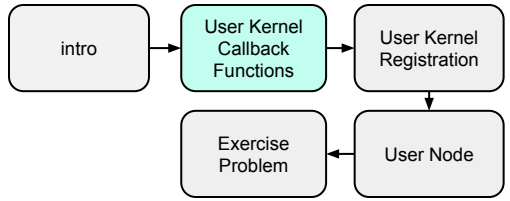
- Prototype

```
typedef vx_status (vx_kernel_output_validate_f)(vx_node node,
                   vx_uint32 index, vx_meta_format meta);
```

- AND3 Example

```
vx_status And3OutputValidator(vx_node node, vx_uint32 index, vx_meta_format meta) {
    vx_parameter param = vxGetParameterByIndex(node, 0);
    vx_image input = NULL;
    vxQueryParameter(param, VX_PARAMETER_ATTRIBUTE_REF, &input, sizeof(input));
    vxReleaseParameter(&param)

    vx_uint32 width = 0, height = 0;
    vxQueryImage(input, VX_IMAGE_ATTRIBUTE_WIDTH, &width, sizeof(width));
    vxQueryImage(input, VX_IMAGE_ATTRIBUTE_HEIGHT, &height, sizeof(height));

    vxSetMetaFormatAttribute(meta, VX_IMAGE_ATTRIBUTE_WIDTH, &width, sizeof(width));
    vxSetMetaFormatAttribute(meta, VX_IMAGE_ATTRIBUTE_HEIGHT, &height, sizeof(height));
    vx_df_image format = VX_DF_IMAGE_U8;
    vxSetMetaFormatAttribute(meta, VX_IMAGE_ATTRIBUTE_FORMAT, &format, sizeof(format));

  return VX_SUCCESS;
}
```

Kernel Process Function



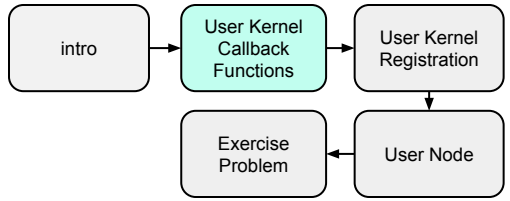
- Prototype

```
typedef vx_status (vx_kernel_f)(vx_node node,
                    const vx_reference * params, vx_uint32 n);
```

- AND3 Example

```
vx_status And3Process(vx_node node, const vx_reference * obj, vx_uint32 n) {
   vx_image in0 = (vx_image)obj[0];
   vx_image in1 = (vx_image)obj[1]
   out  = (vx_image)obj[3];

   vx_uint8 * pin0 = 0; vxAccessImagePatch(in0, …, &pin0, VX_READ_ONLY);
   vx_uint8 * pin1 = 0; vxAccessImagePatch(in1, …, &pin1, VX_READ_ONLY);
   vx_uint8 * pout = 0; vxAccessImagePatch(out, …, &pout, VX_WRITE_ONLY);
   vx_uint8 scalar = 0; vxReadScalar((vx_scalar)obj[2], &inp2);

   // perform pixel-wise bitwise-AND operation into pout …

   vxCommitImagePatch(in0, …, pin0);
   vxCommitImagePatch(in1, …, pin1);
   vxCommitImagePatch(out, …, pout);

   return VX_SUCCESS;
}
```

Kernel Init/Deinit Functions



1. Initializer (Optional)

```
typedef vx_status (vx_kernel_initialize_f)(vx_node node,
                                           const vx_reference * params, vx_uint32 num);
```

- Can request scratch memory to be used by the process callback (e. g. temporary buffer)

```
vx_size size = ...;
vxSetNodeAttribute(node, VX_NODE_ATTRIBUTE_LOCAL_DATA_SIZE, &size, sizeof(size));
```

- Can allocate and initialize some memory to be used by the process callback (e.g. lookup table)

```
MyData * ptr = new MyData;
vxSetNodeAttribute(node, VX_NODE_ATTRIBUTE_LOCAL_DATA_PTR, &ptr, sizeof(ptr));
```

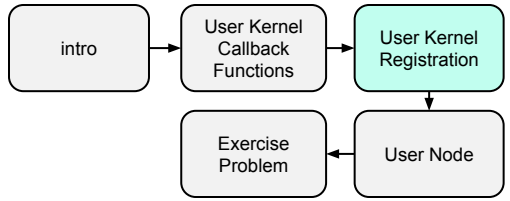
2. Deinitializer (Optional)

```
typedef vx_status (vx_kernel_deinitialize_f)(vx_node node,
                                             const vx_reference * params, vx_uint32 num);
```

- De-allocate memory allocated at initialization

```
vxQueryNode(node, VX_NODE_ATTRIBUTE_LOCAL_DATA_PTR, &ptr, sizeof(ptr));
delete ptr;
```

User Kernel Registration

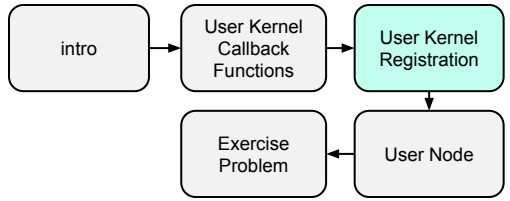


- How to register a user kernel with a context?

```
// Create a kernel object
vx_kernel kernel = vxAddKernel(context,
      "user.kernel.and3", USER_KERNEL_AND3,
                          And3Process,
      4, // NumParams
                       And3InputValidator,
      And3OutputValidator
      NULL, NULL // Init/Deinit
      );
// Add parameters
vxAddParameterToKernel(kernel, 0, VX_INPUT, VX_TYPE_IMAGE, VX_PARAMETER_STATE_REQUIRED);
...
vxAddParameterToKernel(kernel, 3, VX_OUTPUT, VX_TYPE_IMAGE, VX_PARAMETER_STATE_REQUIRED);

// Add Kernel attributes (Optional)

// Finalize shall be called after specifying all parameters
vxFinalizeKernel(kernel);
```

- Now the kernel can be retrieved with vxGetKernelByEnum(), vxGetKernelByName()

OpenVX Kernel Identification



- **User kernels should have unique kernel enums and names**

```
#define USER_LIBRARY_ARITH 0

USER_KERNEL_VX_KERNEL_BASE(VX_ID_DEFAULT, USER_LIBRARY_ARITH) + 0x4
```

Consistent with the kernel name "user.kernel.and3"

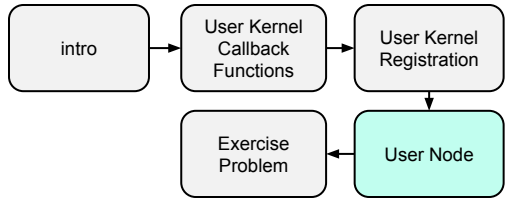
Unique Vendor ID For custom kernels use VX_ID_DEFAULT until you obtain a company ID

Library ID Controlled by the User

Kernel Index in the library Controlled by the User

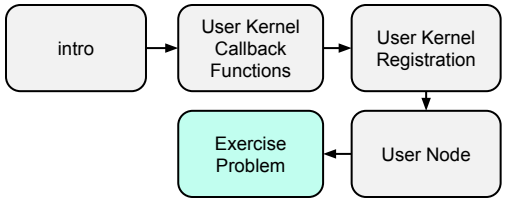
- **Up to 256 kernel libraries per vendor ID**
  - Up to 4096 kernels per kernel library

Create a User Node




```
vx_node vxAnd3Node(vx_image in0, vx_image in1, vx_scalar in2, vx_image out) {

  vx_kernel kernel = vxGetKernelByEnum(context, USER_KERNEL_AND3);
  vx_node node = vxCreateGenericNode(graph, kernel);
  vxReleaseKernel(&kernel);

  // Set data object as parameters
  vxSetParameterByIndex(node, 0, (vx_reference)in0);
  vxSetParameterByIndex(node, 1, (vx_reference)in1);
  vxSetParameterByIndex(node, 2, (vx_reference)scalar);
  vxSetParameterByIndex(node, 3, (vx_reference)out);

  return node;
}
```

Exercise Problem



- **OpenVX 1.0.1 built-in functions supports only 3x3 median filter:**

```
vx_node vxMedian3x3Node( vx_graph graph,
                         vx_image input,
                         vx_image output );
```

- **Create a user kernel for NxN median filter using OpenCV medianBlur function:**

```
vx_node userMedianBlurNode( vx_graph graph,
                            vx_image input,
                            vx_image output,
                            vx_int32 ksize );
```

  - **OpenCV medianBlur usage:**

```
cv::Mat mat_input(  height, width, CV_8U, ptr_input,  addr_input .stride_y );
cv::Mat mat_output( height, width, CV_8U, ptr_output, addr_output.stride_y );
cv::medianBlur( mat_input, mat_output, ksize );
```