Application Note

Developing Custom OpenVX Nodes for A72 Cores on TDA4x

Johnny Kim

ABSTRACT

Texas Instruments Processor SDK RTOS provides the TIOVX and vision_apps frameworks for developing heterogeneous image processing and AI pipelines on TDA4x devices. While the SDK includes a comprehensive set of standard OpenVX kernels, many applications require custom processing functions that are not supported by the built-in kernel library. This application note describes the complete workflow for developing and integrating a custom OpenVX node that executes on an Arm Cortex-A72 core, including kernel interface definition, Host and Target Kernel implementation, kernel registration, build integration, and graph execution within the vision_apps framework. A grayscale threshold operation is used as a reference implementation to illustrate parameter validation, shared-memory buffer access, and runtime processing. Although a simple threshold algorithm is used for demonstration purposes, the methodology presented in this document can be readily applied to application-specific image processing, sensor processing, and AI pre-processing or post-processing functions.

Table of Contents

Trademarks

1 Introduction

The TDA4x family of SoCs integrates heterogeneous processing resources, including Arm Cortex-A72 cores, Cortex-R5F cores, C7x DSP cores, and hardware accelerators, enabling efficient execution of image processing, computer vision, and AI workloads. To simplify software development across these heterogeneous cores, Texas Instruments provides the TIOVX framework as part of Processor SDK RTOS. TIOVX implements the OpenVX standard and allows applications to be constructed as graph-based processing pipelines composed of interconnected processing nodes.

The SDK includes a comprehensive set of built-in OpenVX kernels for common image processing and vision algorithms. However, many real-world applications require proprietary algorithms or application-specific processing functions that are not provided by the standard kernel library. Typical examples include custom image processing, sensor data processing, AI pre-processing and post-processing, and application-specific decision logic. Integrating these functions as custom OpenVX nodes enables them to participate in the standard graph execution model while remaining compatible with the existing TIOVX infrastructure. This application note describes the complete workflow for developing and integrating a custom OpenVX node that executes on an Arm Cortex-A72 core within the vision_apps framework. The document explains how to define the kernel interface, implement the Host and Target Kernels, register the kernel package, integrate it into the build system, and instantiate the custom node in an OpenVX graph. A simple grayscale threshold operation is used as a reference example to illustrate the implementation process, although the same methodology can be readily extended to more sophisticated application-specific algorithms.

2 Understanding OpenVX Kernel Architecture

This chapter introduces the basic architecture of a custom OpenVX kernel in TIOVX. Before implementing a custom node, it is helpful to understand how the OpenVX framework separates graph construction from runtime execution and how processing is distributed across heterogeneous cores.

2.1 OpenVX Node Execution Flow

An OpenVX application is organized as a graph consisting of interconnected processing nodes. Each node represents a kernel that performs a specific operation on one or more input objects and produces one or more output objects.

When a graph is created, the Host Kernel validates node parameters and configures the graph. During graph execution, the framework dispatches each node to its assigned processing target, where the corresponding Target Kernel performs the actual computation.

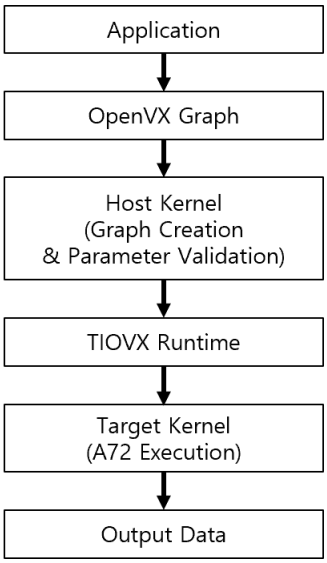

Figure 2-1. Custom OpenVX Node Execution Flow

2.2 Host and Target Kernels

A custom OpenVX kernel consists of two software components.

The **Host Kernel** executes during graph creation and verification. Its primary responsibilities include defining the kernel interface, validating node parameters, configuring output metadata, and registering the kernel with the OpenVX framework. Since the Host Kernel is involved only during graph construction, it does not contain the processing algorithm itself. The **Target Kernel** executes when the graph is scheduled. It implements the processing algorithm, accesses input and output data objects, and performs the required computation on the assigned core. Depending on the application, Target Kernels can execute on various processing cores supported by TIOVX.

2.3 A72 Target Execution Model

In this application note, the custom node is implemented for execution on an Arm Cortex-A72 core. During graph execution, the TIOVX framework dispatches the node to the MPU target, where the Target Kernel executes the application-specific algorithm.

Using an A72 target simplifies development because the processing algorithm can be implemented using standard C/C++ code without requiring DSP-specific programming techniques or hardware accelerator APIs. Once the overall development flow is understood, the same methodology can be adapted to other execution targets, such as Cortex-R5F or C7x DSPs, by implementing the corresponding Target Kernels.

3 Creating a Custom Kernel Package

This chapter describes the structure of the custom threshold kernel package used in this application note. The package is organized as separate public interface, Host Kernel, A72 Target Kernel, and demonstration application components. This organization follows the modular structure used by vision_apps and allows the Host and Target implementations to be built and registered independently.

The reference package implements a custom OpenVX threshold node for U8 grayscale images. The node accepts an input image and a threshold scalar, executes the threshold operation on the Arm Cortex-A72 core through TIVX_TARGET_MPU_0, and produces a U8 output image.

3.1 Package Structure

The custom kernel package is located under the vision_apps/kernels directory. A separate demonstration application is placed under vision_apps/apps/basic_demos.

```
vision_apps
 |
 +---apps
 |   \---basic_demos
 |       \---app_custom_threshold
 |               app_custom_threshold.c
 |               concerto.mak
 |
 \---kernels
     \---custom_threshold
         +---a72
         |       concerto.mak
         |       vx_custom_threshold_target.c
         |       vx_kernels_custom_threshold_target.c
         |
         +---host
         |       concerto.mak
         |       tivx_custom_threshold_kernels_priv.h
         |       tivx_custom_threshold_node_api.c
         |       vx_custom_threshold_host.c
         |       vx_kernels_custom_threshold_host.c
         |
         \---include
             \---TI
                     tivx_custom_threshold.h
                     tivx_custom_threshold_kernels.h
                     tivx_custom_threshold_nodes.h
```

Table 3-1. Custom Threshold Package Components

Component	Purpose
include/TI	Defines the public kernel interface, module and kernel names, parameter indices, kernel load and unload APIs, and node creation API
host	Implements Host Kernel validation and initialization, Host-side module registration, and the public node creation function
a72	Implements the runtime processing callbacks and Target Kernel registration for TIVX_TARGET_MPU_0
app_custom_threshold	Creates and executes a reference OpenVX graph and verifies the threshold output
host/concerto.mak	Builds the Host Kernel library
a72/concerto.mak	Builds the A72 Target Kernel library
app_custom_threshold/concerto.mak	Builds the demonstration executable and links the Host and Target libraries

The reference implementation uses separate concerto.mak files for the Host Kernel, A72 Target Kernel, and demonstration application. The vision_apps build system normally discovers these build files from their

respective directories. Therefore, an additional package-level concerto.mak file is not required under kernels/custom_threshold.

This separation is important because the Host Kernel and Target Kernel perform different roles. The Host Kernel defines and validates the OpenVX interface, while the A72 Target Kernel implements the runtime processing executed on the MPU target. The application links both components and uses their public APIs to load the kernel package and create the custom node.

3.2 Defining the Kernel Interface

The public interface of the custom threshold package is divided into three header files under include/TI.

1) Aggregate Package Header

The tivx_custom_threshold.h header provides a single include point for applications using the package.

```
#include <TI/tivx.h>
#include <TI/tivx_custom_threshold_kernels.h>
#include <TI/tivx_custom_threshold_nodes.h>
```

Applications can include this header to access both the kernel package management APIs and the public node creation API.

2) Kernel Definitions and Package APIs

The tivx_custom_threshold_kernels.h header defines the module name and the unique kernel name used by both the Host and Target Kernel implementations.

```
#define TIVX_MODULE_NAME_CUSTOM_THRESHOLD "custom_threshold"
#define TIVX_KERNEL_CUSTOM_THRESHOLD_NAME "com.ti.custom.threshold"
```

The module name identifies the kernel package during module registration and loading. The kernel name identifies the custom threshold kernel when the Host Kernel, Target Kernel, and application create or locate the corresponding OpenVX node.

The numerical user kernel ID is not fixed in the public header. Instead, it is allocated at runtime by the Host Kernel using **vxAllocateUserKernelId()**. This avoids assigning a hard-coded kernel enumeration value and allows the OpenVX context to provide an available user kernel ID.

The same header also defines the parameter indices shared by the Host Kernel, Target Kernel, and node creation API.

```
enum{
    TIVX_KERNEL_CUSTOM_THRESHOLD_INPUT_IDX = 0,
    TIVX_KERNEL_CUSTOM_THRESHOLD_THRESHOLD_IDX,
    TIVX_KERNEL_CUSTOM_THRESHOLD_OUTPUT_IDX,
    TIVX_KERNEL_CUSTOM_THRESHOLD_MAX_PARAMS
};
```

Table 3-2. Custom Threshold Node Parameters

Index	Parameter	Direction	OpenVX type	Description
TIVX_KERNEL_CUSTOM_THRESHOLD_INPUT_IDX	Input image	Input	VX_TYPE_IMAGE	U8 grayscale source image
TIVX_KERNEL_CUSTOM_THRESHOLD_THRESHOLD_IDX	Threshold value	Input	VX_TYPE_SCALAR	U8 scalar used for pixel comparison
TIVX_KERNEL_CUSTOM_THRESHOLD_OUTPUT_IDX	Output image	Output	VX_TYPE_IMAGE	U8 binary output image

Table 3-2. Custom Threshold Node Parameters (continued)

TIVX_KERNEL_CUSTOM_THRESHOLD_MAX_PARAMS	Parameter count	-	-	Total number of node parameters

The kernel package header also declares the APIs used to load and unload the custom kernels.

```
void tivxCustomThresholdLoadKernels(vx_context context);
void tivxCustomThresholdUnLoadKernels(vx_context context);
```

These functions coordinate the Host Kernel module and A72 Target Kernel registration. The application calls the load function after creating the OpenVX context and calls the unload function before releasing the context.

The A72 Target Kernel registration functions are also declared in the public interface.

```
void tivxRegisterCustomThresholdTargetA72Kernels(void);
void tivxUnRegisterCustomThresholdTargetA72Kernels(void);
```

These functions register and unregister the Target Kernel implementation associated with TIVX_TARGET_MPU_0.

3) Public Node Creation API

The tivx_custom_threshold_nodes.h header declares the helper function used by an application to instantiate the custom threshold node.

```
VX_API_ENTRY vx_node VX_API_CALL
tivxCustomThresholdNode(
    vx_graph graph,
    vx_image input,
    vx_scalar threshold,
    vx_image output);
```

This function accepts the OpenVX graph and the three node parameters and returns a vx_node object. Internally, the implementation creates the node using the kernel name TIVX_KERNEL_CUSTOM_THRESHOLD_NAME and associates the input image, threshold scalar, and output image with their corresponding parameter indices.

4 Developing a Custom OpenVX Node for the A72 Core: A Threshold Example

This chapter describes how to implement a custom OpenVX node that executes on the Arm Cortex-A72 core using the TIOVX framework. Although a simple threshold operation is used as the example algorithm, the development process presented in this chapter is applicable to a wide range of CPU-based image-processing functions.

The implementation consists of five major steps. First, a Host Kernel is created to define the kernel interface and parameters visible to the OpenVX framework. Next, a Target Kernel is implemented to execute the algorithm on the A72 core. The Target Kernel accesses image buffers through the TIOVX shared-memory interface before performing the application-specific image processing. Finally, the Host and Target kernels are packaged and registered so that the custom node can be instantiated from an OpenVX application.

4.1 Implementing the Host Kernel

The Host Kernel defines the OpenVX user kernel that is exposed to the application. Unlike the Target Kernel, which performs the actual image processing, the Host Kernel is responsible for describing the kernel interface to the OpenVX framework. This includes defining the kernel name, parameter types, validation callback, initialization callback, and supported execution targets.

The Host Kernel implementation is located in the following source file.

vision_apps/kernels/custom_threshold/host/**vx_custom_threshold_host.c**

The primary function in this file is **tivxAddKernelCustomThreshold()**, which performs the complete registration of the custom kernel. The registration process consists of four main steps:

- Create the OpenVX user kernel.
- Define the required input and output parameters.
- Register the supported execution target.
- Finalize the kernel registration.

```
/* Create the OpenVX user kernel and associate the callback functions */
kernel = vxAddUserKernel(context, TIVX_KERNEL_CUSTOM_THRESHOLD_NAME, kernel_id,
            tivxAddKernelCustomThresholdValidate, tivxAddKernelCustomThresholdInitialize, NULL);

/* Define the input image parameter */
vxAddParameterToKernel(kernel,  0, VX_INPUT, VX_TYPE_IMAGE, VX_PARAMETER_STATE_REQUIRED);

/* Define the input threshold value */
vxAddParameterToKernel(kernel, 1, VX_INPUT, VX_TYPE_SCALAR, VX_PARAMETER_STATE_REQUIRED);

/* Define the output image parameter */
vxAddParameterToKernel(kernel, 2, VX_OUTPUT, VX_TYPE_IMAGE, VX_PARAMETER_STATE_REQUIRED);

/* Register the kernel for execution on the A72 target */
tivxAddKernelTarget(kernel, TIVX_TARGET_MPU_0);

/* Complete the kernel registration */
vxFinalizeKernel(kernel);
```

vxAddUserKernel() creates a new OpenVX user-defined kernel and associates it with the callback functions that will be invoked by the framework. In this example, the validation callback verifies that the supplied parameters satisfy the kernel requirements, while the initialization callback configures the valid image region before graph execution.

The three calls to **vxAddParameterToKernel()** define the kernel interface exposed to the application. The threshold example requires one input image, one scalar threshold value, and one output image. Each parameter is assigned an index that must match the parameter order used when the node is created.

The call to **tivxAddKernelTarget()** specifies that this kernel executes on the A72 core (TIVX_TARGET_MPU_0). Additional targets can be registered if multiple implementations are available.

Finally, **vxFinalizeKernel()** completes the registration process and makes the kernel available for graph verification and execution.

During graph verification, the OpenVX framework invokes the validation callback to verify the parameter configuration before the graph is executed. The validation callback checks that the input image format and scalar type match the kernel requirements. It also configures the metadata of the output image so that the framework can allocate the correct image object before graph execution. The simplified implementation is below.

```
/* Query the input image format */
vxQueryImage(input, VX_IMAGE_FORMAT, &format, sizeof(format));

/* Query the scalar type */
vxQueryScalar(threshold, VX_SCALAR_TYPE, &type, sizeof(type));

/* Configure the output image metadata */
vxSetMetaFormatAttribute(meta, VX_IMAGE_WIDTH, &width, sizeof(width));
vxSetMetaFormatAttribute(meta, VX_IMAGE_HEIGHT, &height, sizeof(height));
vxSetMetaFormatAttribute(meta, VX_IMAGE_FORMAT, &format, sizeof(format));
```

The initialization callback is executed after successful validation and before graph execution begins. For this threshold example, the callback configures the valid rectangle so that the output image has the same valid region as the input image. Since the operation performs a one-to-one pixel transformation without changing the image dimensions, no additional initialization is required.

4.2 Implementing the A72 Target Kernel

While the Host Kernel defines the kernel interface, the Target Kernel implements the actual execution of the custom operation on the target core. For this example, the Target Kernel runs on the Arm Cortex-A72 core and is responsible for receiving the input and output object descriptors, accessing the image buffers, executing the threshold operation, and returning the processed result to the OpenVX framework.

The Target Kernel implementation is located in the following source file.

vision_apps/kernels/custom_threshold/a72/**vx_custom_threshold_target.c**

The Target Kernel consists of four primary callback functions.

- Register the Target Kernel with the TIOVX framework.
- Create any resources required before graph execution.
- Process each node execution.
- Delete resources when the graph is released.

The Target Kernel is registered by calling **tivxAddTargetKernel()**, which associates the custom kernel with its callback functions.

```
/* Register the Target Kernel on the A72 core */
vxAddTargetKernel( kernel_id, TIVX_TARGET_MPU_0,
        /* Create callback */
        tivxCustomThresholdCreate,
        /* Process callback */
        tivxCustomThresholdProcess,
        /* Delete callback */
        tivxCustomThresholdDelete,
        NULL, NULL);
```

The registration associates the custom kernel with the A72 execution target (TIVX_TARGET_MPU_0) and specifies the callback functions that will be invoked during the node lifecycle.

The **Create callback** is executed once after the graph has been successfully verified. It is typically used to allocate resources, initialize internal data structures, or prepare hardware resources required by the kernel. Since the threshold example performs only simple CPU-based image processing, no additional resources are required, and the callback performs minimal initialization.

The **Process callback** contains the application-specific image processing algorithm. During each graph execution, the OpenVX framework invokes this callback with the object descriptors corresponding to the node parameters. The callback converts the shared-memory addresses into CPU-accessible pointers, maps the image buffers into the A72 address space, performs the image processing, and finally unmaps the buffers before returning control to the framework.

The **Delete callback** is called when the graph is released. It is responsible for releasing any resources allocated by the Create callback. As no persistent resources are allocated in this example, the callback simply returns successfully.

The Process callback is the core of the Target Kernel implementation. The following sections describe how image buffers are accessed on the A72 core and how the threshold operation is implemented.

4.3 Accessing Image Buffers on the A72 Core

Unlike a standard C application, a Target Kernel does not receive image pointers directly. Instead, the OpenVX framework passes object descriptors that describe the input and output objects. These descriptors contain shared-memory addresses, which must be converted into CPU-accessible pointers before the image data can be processed.

To safely access image buffers on the A72 core, the Target Kernel performs four steps:

- Convert the shared-memory address to a target pointer.
- Map the buffer into the CPU address space.
- Process the image data.
- Unmap the buffer after processing.

Below codes show the essential steps required to access an image buffer from the A72 core.

```
/* Convert the shared-memory address to a target pointer */
src_target_ptr = tivxMemShared2TargetPtr(&src_desc->mem_ptr[0]);
dst_target_ptr = tivxMemShared2TargetPtr(&dst_desc->mem_ptr[0]);

/* Map the image buffers into the CPU address space */
tivxMemBufferMap(src_target_ptr, src_desc->mem_size[0], VX_MEMORY_TYPE_HOST, VX_READ_ONLY);
tivxMemBufferMap(dst_target_ptr, dst_desc->mem_size[0], VX_MEMORY_TYPE_HOST, VX_WRITE_ONLY);

/* Process the image data */
:

/* Release the mapped buffers */
tivxMemBufferUnmap(src_target_ptr, src_desc->mem_size[0], VX_MEMORY_TYPE_HOST, VX_READ_ONLY);
tivxMemBufferUnmap(dst_target_ptr, dst_desc->mem_size[0], VX_MEMORY_TYPE_HOST, VX_WRITE_ONLY);
```

The object descriptor contains the location of the image buffer in the shared memory managed by the TIOVX framework. Since this address is not directly accessible by the A72 core, the Target Kernel first calls **tivxMemShared2TargetPtr()** to obtain a CPU-accessible pointer.

Before accessing the image data, the buffer must be mapped into the core address space using **tivxMemBufferMap()**. The mapping operation ensures that the CPU accesses the most recent contents of the shared buffer and maintains cache coherency between heterogeneous cores.

Once the buffer has been mapped, the Target Kernel can access the image pixels using standard C pointers. After processing is complete, **tivxMemBufferUnmap()** is called to release the mapped buffer and synchronize any modified data back to shared memory.

Because TIOVX manages image buffers through shared memory, every Target Kernel that directly accesses image data should follow this mapping and unmapping procedure.

After the buffers have been successfully mapped, the image pixels can be accessed row by row using the image stride information stored in the object descriptor. Below shows the basic pointer calculation used by the threshold example.

```
/* Compute the address of each image row */
src_row = src_target_ptr + (y * src_desc->imagepatch_addr[0].stride_y);
dst_row = dst_target_ptr + (y * dst_desc->imagepatch_addr[0].stride_y);

/* Access pixels using standard array indexing */
pixel = src_row[x];
dst_row[x] = pixel;
```

The stride_y field specifies the number of bytes between consecutive image rows. Instead of assuming that image rows are stored contiguously, the Target Kernel uses this value to calculate the starting address of each row. This approach allows the same implementation to support different image widths, memory alignments, and buffer layouts managed by the OpenVX framework.

With the image buffers successfully mapped and the pixel addresses calculated, the Target Kernel is ready to execute the application-specific image processing algorithm, which is described in the next section.

4.4 Implementing the Threshold Algorithm

After the image buffers have been mapped into the A72 address space, the Target Kernel can perform application-specific image processing using standard C code. In this example, a simple threshold operation is used to demonstrate where the algorithm is implemented within the Process callback.

The threshold operation compares each input pixel against a user-defined threshold value. Pixels greater than the threshold are assigned the maximum intensity value (255), while all other pixels are assigned 0. The algorithm is intentionally simple so that the focus remains on the overall Custom OpenVX Node development process rather than the image-processing algorithm itself.

```
/* Read each input pixel */
pixel = src_row[x];
/* Compare the pixel against the threshold */
if (pixel > threshold) {
    /* Write the foreground value */
    dst_row[x] = 255;
}
else {
    /* Write the background value */
    dst_row[x] = 0;
}
```

The input and output row pointers are calculated using the image addressing information described in the previous section. Once the row pointers have been obtained, individual pixels can be accessed using standard array indexing, allowing the image processing algorithm to be implemented in the same manner as a conventional C application.

Although this example implements a threshold operation, the overall development framework remains unchanged for other CPU-based image-processing algorithms. The Host Kernel implementation, Target Kernel registration, and image buffer access described in the previous sections can be reused without modification. In most cases, only the algorithm-specific pixel processing code inside the Process callback needs to be replaced.

For example, the same framework can be used to implement image filtering, color space conversion, edge detection, feature extraction, AI preprocessing, or other custom image-processing functions. By separating the framework-specific implementation from the algorithm itself, developers can efficiently create new A72-based OpenVX kernels while reusing the same Host and Target Kernel infrastructure.

4.5 Registering the Kernel Package

After implementing both the Host Kernel and the Target Kernel, the final step is to package and register the custom kernel so that it can be used by an OpenVX application. The TIOVX framework separates Host and

Target registration into independent kernel packages, allowing applications to load only the required modules during initialization.

The package registration code is implemented in the following source files.

vision_apps/kernels/custom_threshold/host/**vx_kernels_custom_threshold_host.c**

vision_apps/kernels/custom_threshold/a72/**vx_kernels_custom_threshold_target.c**

The Host package publishes the custom kernel interface to the OpenVX framework, while the Target package registers the execution callbacks for the A72 core. During application initialization, both packages are loaded before the custom node can be instantiated.

The Host Kernel package is responsible for publishing the custom kernel to the OpenVX framework. During package initialization, the registration function invokes the Host Kernel registration routine described in Section 4.1.

```
/* Publish the custom kernel package */
tivxRegisterModule(TIVX_MODULE_NAME_CUSTOM_THRESHOLD,
        tivxRegisterCustomThresholdKernels,
        tivxUnRegisterCustomThresholdKernels);

/* Register the Host Kernel */
tivxAddKernelCustomThreshold(context);
```

The Target Kernel package registers the execution callbacks described in Section 4.2. During initialization, the package associates the custom kernel with the A72 core so that the OpenVX framework can dispatch node execution to the correct target.

```
/* Register the Target Kernel package */
tivxRegisterTargetKernels();

/* Register the Target Kernel on the A72 core */
tivxAddTargetKernelCustomThreshold();
```

Once both packages have been registered, the application loads the custom kernel package during initialization. After the package has been loaded, the custom node can be created and inserted into an OpenVX graph in the same manner as any standard OpenVX node.

```
/* Load the custom kernel package */
vxLoadKernels(context, TIVX_MODULE_NAME_CUSTOM_THRESHOLD);

/* Create the custom node */
node = tivxCustomThresholdNode(graph, input, threshold, output);

/* Verify and execute the graph */
vxVerifyGraph(graph);
vxProcessGraph(graph);
```

From the application developer's perspective, the custom node is used in exactly the same way as a built-in OpenVX node. The OpenVX framework automatically invokes the Host Kernel during graph verification and dispatches execution to the registered Target Kernel during graph processing. This abstraction allows developers to focus on implementing application-specific algorithms while relying on the TIOVX framework to manage kernel registration, memory handling, and execution on heterogeneous cores.

5 Integrating the Custom Node into vision_apps

The previous chapter described how to develop a custom OpenVX node, including the implementation of the Host Kernel, the A72 Target Kernel, image buffer access, and kernel package registration. Once the custom kernel has been implemented, it can be integrated into a vision_apps application and used in the same manner as a standard OpenVX node.

This chapter demonstrates the complete integration process using the app_custom_threshold reference application. Rather than explaining every implementation detail, the focus is on how the application initializes the OpenVX framework, loads the custom kernel package, creates and executes the OpenVX graph, and validates the processing result. The key code is highlighted to illustrate the overall application flow while avoiding unnecessary implementation details.

The reference application is located in the following directory.

vision_apps/apps/basic_demos/**app_custom_threshold**/

The application follows the standard OpenVX execution flow.

- Initialize the application and the OpenVX framework.
- Load the custom kernel package.
- Create the OpenVX graph and custom node.
- Verify and execute the graph.
- Validate the output image and release all resources.

The following sections describe each step using the app_custom_threshold example.

5.1 Implementing the app_custom_threshold Example

The app_custom_threshold application demonstrates how to integrate the custom threshold node into a vision_apps application. Once the custom kernel package has been registered, the custom node can be created and executed using the same programming model as any standard OpenVX node.

The application follows the typical OpenVX execution flow. It initializes the framework, loads the custom kernel package, creates the processing graph, executes the graph, validates the output image, and finally releases all allocated resources. The following code highlight the main application flow while omitting non-essential code for clarity.

```
/* Initialize the application and create the OpenVX context */
status = appInit();

context = vxCreateContext();
tivxCustomThresholdLoadKernels(context);

/* Create the graph and OpenVX data objects */
graph = vxCreateGraph(context);

input  = vxCreateImage(context, APP_CT_WIDTH, APP_CT_HEIGHT, VX_DF_IMAGE_U8);
output = vxCreateImage(context, APP_CT_WIDTH, APP_CT_HEIGHT, VX_DF_IMAGE_U8);

threshold_scalar = vxCreateScalar(context, VX_TYPE_UINT8, &threshold_value);

/* Copy the test data into the input image */
status = vxCopyImagePatch(
    input,
    &rect,
    0u,
    &addr,
    input_data,
    VX_WRITE_ONLY,
    VX_MEMORY_TYPE_HOST);

/* Create and execute the custom threshold node */
node = tivxCustomThresholdNode(
    graph,
```

```
    input,
    threshold_scalar,
    output);

status = vxVerifyGraph(graph);
status = vxProcessGraph(graph);

/* Read the output image and compare it with the expected result */
status = vxCopyImagePatch(
    output,
    &rect,
    0u,
    &addr,
    output_data,
    VX_READ_ONLY,
    VX_MEMORY_TYPE_HOST);

for (y = 0u; y < APP_CT_HEIGHT; y++) {
    for (x = 0u; x < APP_CT_WIDTH; x++) {
        if (output_data[y][x] != expected_data[y][x]) {
            fail_count++;
        }
    }
}

/* Release the OpenVX resources */
vxReleaseNode(&node);
vxReleaseGraph(&graph);
vxReleaseScalar(&threshold_scalar);
vxReleaseImage(&input);
vxReleaseImage(&output);

tivxCustomThresholdUnLoadKernels(context);
vxReleaseContext(&context);

appDeInit();
```

The application begins by initializing the vision_apps framework and creating an OpenVX context. The custom kernel package is then loaded, making the custom threshold node available to the application. After creating the required OpenVX objects, including the input image, output image, and threshold scalar, the application constructs the processing graph using the public node API introduced in Chapter 4.

Once the graph has been created, the node target is assigned to the Arm Cortex-A72 core using **vxSetNodeTarget()**. The graph is then verified by calling **vxVerifyGraph()**, which checks the validity of the graph configuration before execution. After successful verification, **vxProcessGraph()** executes the graph and invokes the Host and Target Kernel callbacks registered by the custom kernel package.

Finally, the application compares the generated output image with the reference image to verify the processing result. After validation, all OpenVX objects and the custom kernel package are released, completing the application execution.

5.2 Example Execution Log

After building the application, the vx_app_custom_threshold.out executable is launched from the target Linux console. During execution, the application initializes the OpenVX framework, executes the custom threshold node, prints the resulting output image, and reports the execution status. The following console output shows a successful execution of the application.

```
root@j721s2-evm:/opt/vision_apps# ./vx_app_custom_threshold.out

APP_CUSTOM_THRESHOLD: start
APP: Init ... !!!
    ...
    39.779936 s: VX_ZONE_INFO: [tivxInitLocal:211] Initialization Done !!!
    39.779948 s: VX_ZONE_INFO: Globally Disabled VX_ZONE_INFO

APP_CUSTOM_THRESHOLD: output image
  0   0 255 255
  0   0 255 255
```

```
APP_CUSTOM_THRESHOLD: PASS

APP: Deinit ... !!!
APP: Deinit ... Done !!!
APP_CUSTOM_THRESHOLD: end

root@j721s2-evm:/opt/vision_apps#
```

The console output confirms that the application successfully initializes the OpenVX framework and executes the custom threshold node. The printed pixel values correspond to the generated output image and provide a simple verification of the thresholding result. Finally, the **APP_CUSTOM_THRESHOLD: PASS** message indicates that the output matches the expected reference and that the custom node has been successfully integrated into the vision_apps framework and executed on the Arm Cortex-A72 core.

6 Summary

This application note presented a practical workflow for developing a custom OpenVX node that executes on the Arm Cortex-A72 core using the TIOVX framework provided by Texas Instruments Processor SDK RTOS. A threshold node was used as a reference implementation to illustrate the complete development process, including kernel interface definition, Host Kernel implementation, A72 Target Kernel development, image buffer access, kernel package registration, and integration into the vision_apps framework.

The example demonstrated how application-specific processing can be incorporated into an OpenVX graph while preserving the standard TIOVX execution model. Although the threshold operation was intentionally chosen for its simplicity, the overall development methodology is applicable to a wide range of custom image processing, sensor processing, and AI pre-processing or post-processing algorithms. By modifying only the processing logic within the Process() callback, developers can reuse the same Host Kernel, Target Kernel, memory access, and graph integration framework to accelerate the development of custom A72-based OpenVX nodes.

As vision and edge AI applications continue to evolve, integrating application-specific processing into heterogeneous computing platforms becomes increasingly important. The workflow presented in this application note provides a practical foundation for extending the TIOVX framework with custom functionality while maintaining compatibility with the existing Processor SDK RTOS software architecture. By leveraging the flexibility of the Arm Cortex-A72 core together with the graph-based execution model of OpenVX, developers can rapidly prototype, validate, and deploy custom processing nodes for a broad range of embedded vision and AI applications.

7 References

- Texas Instruments, TDA4VE, product page.
- Texas Instruments, Texas Instruments, J721S2, TDA4AL, TDA4VL, TDA4VE, AM68A Technical Reference Manual, technical reference manual.
- Texas Instruments, Texas Instruments, TDA4VE TDA4AL TDA4VL Jacinto™ Processors, Silicon Revision 1.0 datasheet, datasheet.
- Texas Instruments, PROCESSOR-SDK-J721S2, product page.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025