Application Note

在 TDA4x 上为 A72 内核开发自定义 OpenVX 节点

Johnny Kim

摘要

德州仪器 (TI) Processor SDK RTOS 提供了 TIOVX 和 vision_apps 框架，用于在 TDA4x 器件上开发异构图像处理和 AI 流水线。虽然 SDK 包含一整套标准 OpenVX 内核，但许多应用程序需要内置内核库不支持的自定义处理函数。本应用手册介绍了开发和集成在 Arm Cortex-A72 内核上执行的自定义 OpenVX 节点的完整工作流程，包括内核接口定义、主机和目标内核实现、内核注册、构建集成和 vision_apps 框架内的图形执行。灰度阈值操作用作参考实现，以说明参数验证、共享存储器缓冲区访问和运行时处理。尽管简单的阈值算法用于演示目的，但本文中介绍的方法可轻松应用于特定于应用程序的图像处理、传感器处理以及 AI 预处理或后处理功能。

内容

1 简介.....2
2 了解 OpenVX 内核架构.....3
2.1 OpenVX 节点执行流程.....3
2.2 主机和目标内核.....3
2.3 A72 目标执行模型.....3
3 创建自定义内核包.....4
3.1 包结构.....4
3.2 定义内核接口.....5
4 为 A72 内核开发定制 OpenVX 节点：阈值示例.....7
4.1 实现主机内核.....7
4.2 实现 A72 目标内核.....8
4.3 访问 A72 内核上的图像缓冲区.....8
4.4 实现阈值算法.....9
4.5 注册内核包.....10
5 将自定义节点集成到 vision_apps 中.....12
5.1 实现 app_custom_threshold 示例.....12
5.2 执行日志示例.....13
6 总结.....15
7 参考资料.....16

商标

1 简介

TDA4x 系列 SoC 集成了异构处理资源，包括 Arm Cortex-A72 内核、Cortex-R5F 内核、C7x DSP 内核和硬件加速器，可高效执行图像处理、计算机视觉和 AI 工作负载。为了简化这些异构内核之间的软件开发，德州仪器 (TI) 在 Processor SDK RTOS 中提供了 TIOVX 框架。TIOVX 实现 OpenVX 标准，允许将应用构建为由互连处理节点组成的基于图形的处理流水线。

该 SDK 包含一整套内置 OpenVX 内核，适用于常见的图像处理和视觉算法。但是，许多实际应用程序都需要专有算法或应用特定的处理函数，而标准内核库不提供这些函数。典型示例包括自定义图像处理、传感器数据处理、AI 预处理和后处理以及应用特定的决策逻辑。通过将这些功能集成为自定义 OpenVX 节点，它们能够参与标准图形执行模型，同时与现有的 TIOVX 基础结构保持兼容性。本应用手册介绍了开发和集成在 vision_apps 框架内的 Arm Cortex-A72 内核上执行的自定义 OpenVX 节点的完整工作流程。本文档介绍了如何定义内核接口、实现主机和目标内核、注册内核包、将其集成到构建系统中，以及如何在 OpenVX 图形中实例化自定义节点。以简单的灰度阈值操作作为说明实施过程的参考示例，尽管相同的方法可轻松扩展到更复杂的特定于应用程序的算法。

2 了解 OpenVX 内核架构

本章介绍了 TIOVX 中自定义 OpenVX 内核的基本架构。在实现自定义节点之前，了解 OpenVX 框架如何将图形构建与运行时执行分开以及如何在异构内核之间分布处理会很有帮助。

2.1 OpenVX 节点执行流程

OpenVX 应用程序被组织为由相互连接的处理节点组成的图形。每个节点表示对一个或多个输入对象执行特定操作并生成一个或多个输出对象的内核。

创建图形时，主机内核会验证节点参数并配置图形。在图形执行期间，框架会将每个节点分派到其分配的处理目标，相应的目标内核会在其中执行实际计算。

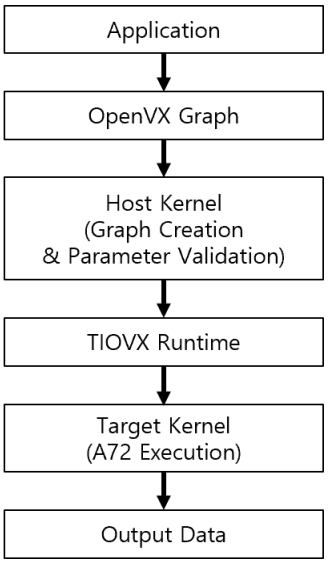

图 2-1. 自定义 OpenVX 节点执行流程

2.2 主机和目标内核

自定义的 OpenVX 内核由两个软件元件组成。

主机内核在图形创建和验证期间执行。其主要职责包括定义内核接口、验证节点参数、配置输出元数据，以及向 OpenVX 框架注册内核。由于主机内核仅在图形构造过程中涉及，因此它不包含处理算法本身。**目标内核**在调度图形时执行。它实现处理算法、访问输入和输出数据对象，并在分配的内核上执行所需的计算。根据应用程序的不同，目标内核可以在 TIOVX 支持的各种处理内核上执行。

2.3 A72 目标执行模型

在本应用手册中，实现了自定义节点以在 Arm Cortex-A72 内核上执行。在图形执行期间，TIOVX 框架会将节点分派至 MPU 目标，由目标内核在该处执行特定于应用程序的算法。

使用 A72 目标可以简化开发，因为可以使用标准 C/C++ 代码来实现处理算法，而无需特定于 DSP 的编程技术或硬件加速器 API。了解整个开发流程后，通过实现相应的目标内核，可以将相同的方法调整用于其他执行目标，例如 Cortex-R5F 或 C7x DSP。

3 创建自定义内核包

本章介绍了本应用手册中使用的自定义阈值内核包的结构。该包组织为单独的公共接口、主机内核、A72 目标内核和演示应用程序元件。这种组织方式遵循 vision_apps 使用的模块化结构，并允许独立构建和注册主机和目标实现。

参考包实现了 U8 灰度图像的自定义 OpenVX 阈值节点。节点接受输入图像和阈值标量，通过 TIVX_TARGET_MPU_0 在 Arm Cortex-A72 内核上执行阈值操作，并生成 U8 输出图像。

3.1 包结构

自定义内核包位于 vision_apps/kernels 目录下。vision_apps/apps/basic_demos 下放置了一个单独的演示应用程序。

```
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
```

表 3-1. 自定义阈值包元件

元件	用途
include/TI	定义公共内核接口、模块和内核名称、参数索引、内核加载和卸载 API，以及节点创建 API
主机	实现主机内核验证和初始化、主机端模块注册和公共节点创建函数
a72	为 TIVX_TARGET_MPU_0 实现运行时处理回调和目标内核注册
app_custom_threshold	创建和执行参考 OpenVX 图形并验证阈值输出
host/concerto.mak	构建主机内核库
a72/concerto.mak	构建 A72 目标内核库
app_custom_threshold/concerto.mak	构建演示可执行文件并链接主机和目标库

此参考实现对主机内核、A72 目标内核和演示应用程序使用单独的 concerto.mak 文件。vision_apps 构建系统通常会从各自的目录中发现这些构建文件。因此，kernels/custom_threshold 下不需要额外的包级 concerto.mak 文件。

这种分离非常重要，因为主机内核和目标内核执行的角色不同。主机内核定义和验证 OpenVX 接口，而 A72 目标内核实现在 MPU 目标上执行的运行时处理。应用程序链接这两个元件，使用它们的公共 API 加载内核包并创建自定义节点。

3.2 定义内核接口

自定义阈值包的公共接口分为 include/TI 下的三个头文件。

1) 聚合包头文件

tivx_custom_threshold.h 头文件为使用该包的应用程序提供了一个包含点。

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

应用程序可以包含此头文件来访问内核包管理 API 和公共节点创建 API。

2) 内核定义和包 API

tivx_custom_threshold_kernels.h 头文件定义了主机和目标内核实现使用的模块名称和唯一内核名称。

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

模块名称在模块注册和加载过程中标识内核包。内核名称在主机内核、目标内核和应用程序创建或找到相应的 OpenVX 节点时定义自定义阈值内核。

公共头文件中的数字用户内核 ID 不是固定的，而是由主机内核在运行时使用 **vxAllocateUserKernelId()** 来分配。这避免了分配硬编码的内核枚举值，并允许 OpenVX 上下文提供可用的用户内核 ID。

同一头文件还定义了主机内核、目标内核和节点创建 API 共享的参数索引。

```
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
};
```

表 3-2. 自定义阈值节点参数

索引	参数	方向	OpenVX 类型	说明
TIVX_KERNEL_CUSTOM_THRESHOLD_INPUT_IDX	输入图像	输入	VX_TYPE_IMAGE	U8 灰度源图像
TIVX_KERNEL_CUSTOM_THRESHOLD_THRESHOLD_IDX	阈值	输入	VX_TYPE_SCALAR	用于像素比较的 U8 标量
TIVX_KERNEL_CUSTOM_THRESHOLD_OUTPUT_IDX	输出图像	输出	VX_TYPE_IMAGE	U8 二进制输出图像
TIVX_KERNEL_CUSTOM_THRESHOLD_MAX_PARAMS	参数计数	-	-	节点参数总数

内核包头文件还声明用于加载和卸载自定义内核的 API。

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

这些函数负责协调主机内核模块和 A72 目标内核注册。该应用程序在创建 OpenVX 上下文后调用加载函数，并在释放上下文之前调用卸载函数。

A72 目标内核注册函数也在公共接口中声明。

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

这些函数注册和注销与 TIVX_TARGET_MPU_0 关联的目标内核实现。

3) 公共节点创建 API

Tivx_custom_threshold_nodes.h 头文件声明应用程序用于实例化自定义阈值节点的辅助函数。

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

此函数接受 OpenVX 图形和三个节点参数，并返回 vx_node 对象。在内部，实现使用内核名称 TIVX_KERNEL_CUSTOM_THRESHOLD_NAME 创建节点，并将输入图像、阈值标量和输出图像与其相应的参数索引关联起来。

4 为 A72 内核开发定制 OpenVX 节点：阈值示例

本章介绍了如何实现一个自定义 OpenVX 节点，该节点使用 TIOVX 框架在 Arm Cortex-A72 内核上执行。尽管使用简单的阈值操作作为示例算法，但本章介绍的开发过程适用于各种基于 CPU 的图像处理功能。

实现工作包括五个主要步骤。首先，创建主机内核来定义 OpenVX 框架可见的内核接口和参数。接下来会实现目标内核以在 A72 内核上执行算法。在执行应用特定图像处理之前，目标内核通过 TIOVX 共享存储器接口访问图像缓冲区。最后，对主机和目标内核进行打包和注册，以便可以从 OpenVX 应用中实例化自定义节点。

4.1 实现主机内核

主机内核定义了暴露给应用的 OpenVX 用户内核。与执行实际图像处理的目标内核不同，主机内核负责描述 OpenVX 框架的内核接口。这包括定义内核名称、参数类型、验证回调、初始化回调和支持的执行目标。

主机内核实现位于以下源文件中。

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

该文件中的主要函数是 **tivxAddKernelCustomThreshold()**，它执行自定义内核的完整注册。注册过程包括四个主要步骤：

- 创建 OpenVX 用户内核。
- 定义所需的输入和输出参数。
- 注册支持的执行目标。
- 完成内核注册。

```
/* 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() 创建一个新的 OpenVX 用户定义内核，并将其与将由框架调用的回调函数相关联。在本例中，验证回调会验证提供的参数是否满足内核要求，而初始化回调在执行图形之前配置有效的映像区域。

对 **vxAddParameterToKernel()** 的三个调用定义了暴露给应用的内核接口。该阈值示例需要一个输入图像、一个标量阈值和一个输出图像。为每个参数分配一个索引，该索引必须与创建节点时使用的参数顺序相匹配。

调用 **tivxAddKernelTarget()** 指定该内核在 A72 内核 (TIVX_TARGET_MPU_0) 上执行。如果有多个实现可用，则可以注册其他目标。

最后，**vxFinalizeKernel()** 完成注册过程，并使内核可用于图形验证和执行。

在图形验证期间，OpenVX 框架会调用验证回调，以在执行图形之前验证参数配置。验证回调会检查输入图像格式和标量类型是否符合内核要求。它还配置输出映像的元数据，以便框架可以在执行图形之前分配正确的图像对象。简化实现如下所示。

```
/* 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));
```

初始化回调在成功验证后和开始执行图形之前执行。对于此阈值示例，回调配置有效矩形，以便输出映像与输入映像具有相同的有效区域。该操作会在不更改图像尺寸的情况下执行一对一像素变换，因此无需额外的初始化。

4.2 实现 A72 目标内核

当主机内核定义内核接口时，目标内核会在目标内核上实现自定义操作的实际执行。在本示例中，目标内核在 Arm Cortex-A72 内核上运行，负责接收输入和输出对象描述符、访问图像缓冲区、执行阈值操作，并将处理后的结果返回到 OpenVX 框架。

目标内核实现位于以下源文件中。

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

目标内核包含四个主要回调函数。

- 通过 TIOVX 框架注册目标内核。
- 在执行图形之前创建所需的任何资源。
- 处理每个节点的执行。
- 释放图形时删除资源。

目标内核是通过调用 **tivxAddTargetKernel()** 来注册，该调用将自定义内核与其回调函数相关联。

```
/* 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);
```

注册会将自定义内核与 A72 执行目标 (TIVX_TARGET_MPU_0) 相关联，并指定将在节点生命周期期间调用的回调函数。

成功验证图形后，将执行一次 **Create 回调**。它通常用于分配资源，初始化内部数据结构或准备内核所需的硬件资源。由于阈值示例仅执行基于 CPU 的简单映像处理，因此不需要额外资源，回调执行极少的初始化。

Process 回调包含应用特定的图像处理算法。在每次图形执行过程中，OpenVX 框架使用与节点参数相对应的对象描述符来调用此回调。回调将共享存储器地址转换为 CPU 可访问的指针，将图像缓冲区映射到 A72 地址空间，执行图像处理，最后取消映射缓冲区，然后再将控制返回到框架。

释放图形时调用 **Delete 回调**。它负责释放由 Create 回调分配的任何资源。由于本例中不分配持久资源，因此回调只是成功返回。

Process 回调是目标内核实现的核心。以下各节介绍了如何在 A72 内核上访问图像缓冲区，以及如何实现阈值操作。

4.3 访问 A72 内核上的图像缓冲区

与标准 C 应用不同，目标内核不会直接接收图像指针。相反，OpenVX 框架会传递用于描述输入和输出对象的对象描述符。这些描述符包含共享存储器地址，必须先将其转换为 CPU 可访问的指针，然后才能处理图像数据。

要安全访问 A72 内核上的图像缓冲区，目标内核会执行四个步骤：

- 将共享存储器地址转换为目标指针。

- 将缓冲区映射到 CPU 地址空间。
- 处理图像数据。
- 处理后取消映射缓冲区。

以下代码展示了从 A72 内核访问图像缓冲区所需的基本步骤。

```
/* 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);
```

对象描述符包含 TIOVX 框架管理的共享存储器中图像缓冲区的位置。由于该地址不能由 A72 内核直接访问，因此目标内核首先调用 **tivxMemShared2TargetPtr()** 以获取 CPU 可访问的指针。

在访问图像数据之前，必须使用 **tivxMemBufferMap()** 将缓冲区映射到核心地址空间。映射操作确保 CPU 访问共享缓冲区的最新内容，并保持异构内核之间的缓存一致性。

映射缓冲区后，目标内核可以使用标准 C 指针访问图像像素。处理完成后，调用 **tivxMemBufferUnmap()** 以释放映射的缓冲区并将任何修改后的数据同步回共享内存。

由于 TIOVX 通过共享内存管理图像缓冲区，因此每个直接访问图像数据的目标内核都应遵循此映射与取消映射过程。

成功映射缓冲区后，可以使用存储在对象描述符中的图像跨度信息逐行访问图像像素。下面显示了阈值示例使用的基本指针计算。

```
/* 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;
```

stride_y 字段用于指定连续图像行之间的字节数。目标内核使用此值来计算每行的起始地址，而不是假设图像行连续存储。该方法允许相同的实现方案支持由 OpenVX 框架管理的不同图像宽度、存储器对齐方式和缓冲区布局。

成功映射图像缓冲区并计算像素地址后，目标内核就可以执行应用特定的图像处理算法，下一节将对此进行介绍。

4.4 实现阈值算法

将图像缓冲区映射到 A72 地址空间后，目标内核可以使用标准 C 代码执行应用特定的图像处理。在本例中，使用一个简单的阈值操作来演示在进程回调中实现算法的位置。

阈值操作将每个输入像素与用户定义的阈值进行比较。大于阈值的像素被分配最大强度值 (255)，而所有其他像素被分配为 0。该算法特意设计得简单，以便重点关注定制 OpenVX 节点的总体开发流程，而不是图像处理算法本身。

```
/* 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;
}
```

使用上一节中所述的图像寻址信息计算输入和输出行指针。获得行指针后，可以使用标准数组索引来访问各个像素，从而能够以与传统 C 应用相同的方式实现图像处理算法。

尽管此示例实现了阈值操作，但对于其他基于 CPU 的图像处理算法，整体开发框架保持不变。前面几节中描述的主机内核实现、目标内核注册和图像缓冲区访问可以重复使用，无需修改。在大多数情况下，只需替换进程回调内算法特定的像素处理代码。

例如，同一个框架可用于实现图像滤波、颜色空间转换、边缘检测、特征提取、AI 预处理，或其他自定义图像处理功能。通过将框架特定实现与算法本身分离，开发人员可以高效地创建新的基于 A72 的 OpenVX 内核，同时可重复使用相同的主机和目标内核基础设施。

4.5 注册内核包

实现主机内核和目标内核后，最后一步是打包并注册自定义内核，以便 OpenVX 应用可以使用自定义内核。TIOVX 框架将主机和目标注册分成独立的内核包，允许应用在初始化期间仅加载所需的模块。

包注册码在以下源文件中实现。

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

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

主机包将自定义内核接口发布到 OpenVX 框架，而目标包注册 A72 内核的执行回调。在应用初始化期间，两个包都在自定义节点实例化之前加载。

主机内核包负责将自定义内核发布到 OpenVX 框架。在包初始化期间，注册函数调用第 4.1 节中所述的主机内核注册例程。

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

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

目标内核包会注册第 4.2 节中所述的执行回调。在初始化期间，包将自定义内核与 A72 内核相关联，以便 OpenVX 框架可以将节点执行调度到正确的目标。

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

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

注册两个包后，应用将在初始化期间加载自定义内核包。加载包后，可以创建自定义节点并将其插入 OpenVX 图形中，具体方式与任何标准 OpenVX 节点相同。

```
/* 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);
```

从应用开发人员的角度来看，自定义节点的使用方式与内置 OpenVX 节点的使用方式完全相同。OpenVX 框架在图形验证期间自动调用主机内核，并在图形处理期间将执行补丁分配至注册的目标内核。利用该提取，开发人员可以专注于实现应用特定的算法，同时利用 TIOVX 框架来管理异构内核注册、存储器处理和执行。

5 将自定义节点集成到 vision_apps 中

上一章介绍了如何开发自定义 OpenVX 节点，包括主机内核、A72 目标内核、图像缓冲区访问和内核包注册的实现。实现自定义内核后，可将其集成到 vision_apps 应用中，并以与标准 OpenVX 节点相同的方式使用。

本章演示了使用 app_custom_threshold 参考应用的完整集成过程。章节内容并未解释每个实现细节，重点在于应用如何初始化 OpenVX 框架、加载自定义内核包、创建和执行 OpenVX 图形，以及验证处理结果。突出显示了关键代码以说明整个应用流程，同时避免不必要的实现细节。

参考应用位于以下目录中。

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

该应用遵循标准的 OpenVX 执行流程。

- 初始化应用和 OpenVX 框架。
- 加载自定义内核包。
- 创建 OpenVX 图形和自定义节点。
- 验证并执行图形。
- 验证输出图像并释放所有资源。

以下各节使用 app_custom_threshold 示例介绍了每个步骤。

5.1 实现 app_custom_threshold 示例

app_custom_threshold 应用演示了如何将自定义阈值节点集成到 vision_apps 应用中。注册自定义内核包后，可以使用与任何标准 OpenVX 节点相同的编程模型创建和执行自定义节点。

该应用遵循典型的 OpenVX 执行流程。它初始化框架、加载自定义内核包、创建处理图、执行图形、验证输出映像，最终释放所有分配的资源。为了清晰起见，以下代码突出显示了主应用流程，同时省略了非必要代码。

```
/* 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();
```

该应用首先初始化 vision_apps 框架并创建 OpenVX 上下文。然后加载自定义内核包，使自定义阈值节点可用于应用。创建所需的 OpenVX 对象（包括输入图像，输出图像和阈值标量）后，该应用将使用第 4 章中介绍的公共节点 API 构建处理图。

创建图形后，将使用 **vxSetNodeTarget()** 将节点目标分配给 Arm Cortex-A72 内核。然后通过调用 **vxVerifyGraph()** 来验证图形，后者在执行之前检查图形配置的有效性。验证成功后，**vxProcessGraph()** 执行图形并调用自定义内核包注册的主机和目标内核回调。

最后，应用将生成的输出图像与参考图像进行比较，以验证处理结果。验证后，会释放所有 OpenVX 对象和自定义内核包，完成应用执行。

5.2 执行日志示例

构建应用后，从目标 Linux 控制台启动 vx_app_custom_threshold.out 可执行文件。在执行期间，应用会初始化 OpenVX 框架、执行自定义阈值节点、打印生成的输出映像，并报告执行状态。以下控制台输出显示应用成功执行。

```
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#
```

控制台输出确认应用成功初始化 OpenVX 框架并执行自定义阈值节点。打印的像素值对应于生成的输出图像，并提供对阈值设置结果的简单验证。最后，设置 **APP_CUSTOM_THRESHOLD: PASS** 消息指示输出与预期基准匹配，以及自定义节点已成功集成到 vision_apps 框架并在 Arm Cortex-A72 内核上执行。

6 总结

本应用手册介绍了使用德州仪器 (TI) Processor SDK RTOS 提供的 TIOVX 框架开发在 Arm Cortex-A72 内核上执行的自定义 OpenVX 节点的实用工作流程。阈值节点用作参考实现来说明完整的开发过程，包括内核接口定义、主机内核实现、A72 目标内核开发、图像缓冲区访问、内核包注册，以及与 vision_apps 框架的集成。

该示例演示了如何在保留标准的 TIOVX 执行模型的同时，将应用程序的特定处理功能合并到 OpenVX 图形中。尽管为了简单起见特意选择阈值运行，但整体开发方法适用于各种自定义图像处理、传感器处理和 AI 预处理或后处理算法。通过仅修改 Process() 回调中的处理逻辑，开发人员可以重复使用相同的主机内核、目标内核、内存访问和图形集成框架，以加快基于 A72 的自定义 OpenVX 节点的开发。

随着视觉和边缘 AI 应用程序的不断发展，将应用特定处理集成到异构计算平台中变得越来越重要。本应用手册介绍的工作流程为利用自定义功能扩展 TIOVX 框架，同时保持与现有 Processor SDK RTOS 软件架构的兼容性提供了实用基础。通过利用 Arm Cortex-A72 内核的灵活性与 OpenVX 基于图形的执行模型，开发人员可以快速为各种嵌入式视觉和 AI 应用程序进行原型设计、验证和部署自定义处理节点。

7 参考资料

- 德州仪器 (TI)，TDA4VE，产品页面。
- 德州仪器 (TI)，J721S2、TDA4AL、TDA4VL、TDA4VE、AM68A 技术参考手册，技术参考手册。
- 德州仪器 (TI)，TDA4VE TDA4AL TDA4VL Jacinto™ 处理器，器件版本 1.0，数据表。
- 德州仪器 (TI)，PROCESSOR-SDK-J721S2，产品页面。

重要通知和免责声明

TI“按原样”提供技术和可靠性数据（包括数据表）、设计资源（包括参考设计）、应用或其他设计建议、网络工具、安全信息和其他资源，不保证没有瑕疵且不做出任何明示或暗示的担保，包括但不限于对适销性、与某特定用途的适用性或不侵犯任何第三方知识产权的暗示担保。

这些资源可供使用 TI 产品进行设计的熟练开发人员使用。您将自行承担以下全部责任：(1) 针对您的应用选择合适的 TI 产品，(2) 设计、验证并测试您的应用，(3) 确保您的应用满足相应标准以及任何其他安全、安保法规或其他要求。

这些资源如有变更，恕不另行通知。TI 授权您仅可将这些资源用于研发本资源所述的 TI 产品的相关应用。严禁以其他方式对这些资源进行复制或展示。您无权使用任何其他 TI 知识产权或任何第三方知识产权。对于因您对这些资源的使用而对 TI 及其代表造成的任何索赔、损害、成本、损失和债务，您将全额赔偿，TI 对此概不负责。

TI 提供的产品受 TI 销售条款)、TI 通用质量指南 或 ti.com 上其他适用条款或 TI 产品随附的其他适用条款的约束。TI 提供这些资源并不会扩展或以其他方式更改 TI 针对 TI 产品发布的适用的担保或担保免责声明。 除非德州仪器 (TI) 明确将某产品指定为定制产品或客户特定产品，否则其产品均为按确定价格收入目录的标准通用器件。

TI 反对并拒绝您可能提出的任何其他或不同的条款。

最后更新日期：2025 年 10 月