参考資料

Application Note

TDA4x の A72 コア向けカスタム OpenVX ノードの開発

Johnny Kim

概要

テキサス インスツルメンツのプロセッサ SDK RTOS は、TDA4x デバイス上で異種プロセッサを活用した画像処理と AI パイプラインを開発するための TIOVX および vision_apps フレームワークを提供しています。この SDK には、標準的な OpenVX カーネルで構成された包括的なセットが付属していますが、多くのアプリケーションでは、内蔵カーネル ライブラリでサポートされていないカスタム処理機能が必要です。このアプリケーション ノートでは、Arm Cortex-A72 コアで実行されるカスタム OpenVX ノードを開発および統合するための包括的なワークフローについて説明します。これには、カーネル インターフェイス定義、ホストおよびターゲット カーネルの実装、カーネル登録、ビルド統合、vision_apps フレームワーク内でのグラフ実行が含まれます。パラメータの検証、共有メモリ バッファへのアクセス、およびランタイム処理を具体的に示すためのリファレンス実装として、グレイスケール スレッショルド処理を使用しています。デモンストレーション用として単純なスレッショルド アルゴリズムを使用していますが、このドキュメントで説明する手法は、特定用途向けの画像処理、センサ処理、AI 前処理または後処理機能に簡単に適用できます。

目次

商標

1 概要

TDA4x SoC ファミリには、Arm Cortex-A72 コア、Cortex-R5F コア、C7x DSP コア、ハードウェア アクセラレータなどの異種の処理リソースが統合されており、画像処理、コンピュータ ビジョン、AI ワークロードを効率的に実行できます。これらの異種コアにわたるソフトウェア開発を簡素化するために、テキサス インスツルメンツはプロセッサ SDK RTOS の一部として TIOVX フレームワークを提供しています。TIOVX は OpenVX 規格を実装しており、相互接続された処理ノードで構成される、グラフ ベースの処理パイプラインとしてアプリケーションを構築できます。

この SDK には、一般的な画像処理やビジョン アルゴリズムに対応した包括的な組み込み OpenVX カーネル群が提供されています。ただし、実際のアプリケーションの多くは、標準カーネル ライブラリでは提供されていない独自のアルゴリズムまたは特定用途向けの処理機能を必要とします。代表的な例として、カスタム画像処理、センサ データ処理、AI 前処理と後処理、特定用途向けの意思決定ロジックがあります。これらの機能をカスタムの OpenVX ノードとして統合すると、既存の TIOVX インフラストラクチャとの互換性を維持しながら、標準的なグラフ実行モデルに組み込むことが可能になります。このアプリケーション ノートでは、vision_apps フレームワーク内の Arm Cortex-A72 コアで実行されるカスタム OpenVX ノードを開発および統合するための一連のワークフローについて説明しています。このドキュメントでは、カーネル インターフェイスの定義方法、ホスト カーネルとターゲット カーネルの実装方法、カーネル パッケージの登録方法、ビルド システムへの統合方法、OpenVX グラフでのカスタム ノードのインスタンス化方法を説明しています。実装プロセスを説明するための参考例として、単純なグレイスケール スレッショルド処理を使用していますが、同様の手法は、より高度で特定用途向けのアルゴリズムにも容易に拡張可能です。

2 OpenVX カーネル アーキテクチャについて理解する

この章では、TIOVX のカスタム OpenVX カーネルの基本的なアーキテクチャについて説明します。カスタム ノードを実装する前に、OpenVX フレームワークがグラフ構造とランタイム実行をどのように分離し、処理が異種コアにどのように分散されるかを理解しておくと役立ちます。

2.1 OpenVX ノード実行フロー

OpenVX アプリケーションは、相互接続された処理ノードで構成されるグラフとして構築されます。各ノードは、1 つまたは複数の入力オブジェクトに対して特定の処理を実行し、1 つまたは複数の出力オブジェクトを生成するカーネルを表します。

グラフが作成されると、ホスト カーネルはノード パラメータを検証し、グラフを構成します。グラフの実行中に、フレームワークは各ノードを割り当てられた処理ターゲットにディスパッチし、対応するターゲット カーネルが実際の計算を実行します。

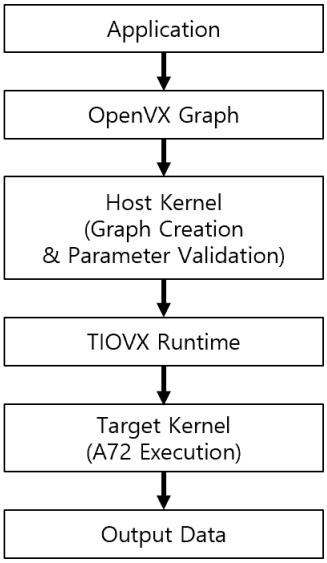

図 2-1. カスタム OpenVX ノード実行フロー

2.2 ホスト カーネルおよびターゲット カーネル

カスタムの OpenVX カーネルは、2 つのソフトウェア コンポーネントで構成されています。

ホスト カーネルはグラフの作成と検証中に実行されます。主な役割は、カーネル インターフェイスの定義、ノード パラメータの検証、出力メタデータの設定、OpenVX フレームワークへのカーネルの登録などです。ホスト カーネルはグラフ作成時にのみ関与しているため、処理アルゴリズム自体は含まれていません。**ターゲット カーネル**は、グラフがスケジュールされたときに実行されます。処理アルゴリズムを実装し、入力および出力データ オブジェクトにアクセスし、割り当てられたコアで必要な計算を実行します。アプリケーションに応じて、ターゲット カーネルは TIOVX がサポートしているさまざまな処理コア上で実行することができます。

2.3 A72 ターゲット実行モデル

このアプリケーション ノートでは、Arm Cortex-A72 コアで実行するためにカスタム ノードが実装されています。グラフの実行中に、TIOVX フレームワークはノードを MPU ターゲットにディスパッチし、そこでターゲット カーネルが特定用途向けのアルゴリズムを実行します。

A72 ターゲットを使用すると、DSP 固有のプログラミング手法やハードウェア アクセラレータ API を必要とせずに、標準の C/C++ コードを使用して処理アルゴリズムを実装できるため、開発が簡素化されます。全体的な開発フローを理解すれば、対応するターゲット カーネルを実装することで、Cortex-R5F や C7x DSP などの他の実行ターゲットに対しても同じ手法を適用できます。

3 カスタム カーネル パッケージの作成

この章では、このアプリケーション ノートで使用するカスタム スレッショルド カーネル パッケージの構造について説明します。このパッケージは、パブリック インターフェイス、ホスト カーネル、A72 ターゲット カーネル、およびデモンストレーション用アプリケーションという個別のコンポーネントで構成されています。この組織は vision_apps で使用されるモジュール構造に従い、ホストとターゲットの実装を個別に構築および登録できます。

このリファレンス パッケージは、U8 グレイスケール画像向けのカスタム OpenVX スレッショルド ノードを実装しています。ノードは入力画像とスレッショルド スカラーを受け入れ、TIVX_TARGET_MPU_0 経由で Arm Cortex-A72 コアのスレッショルド処理を実行し、U8 出力画像を生成します。

3.1 パッケージ構造

カスタム カーネル パッケージは、vision_apps/kernels ディレクトリにあります。別途、デモンストレーション用アプリケーションが、vision_apps/apps/basic_demos 下に配置されています。

```
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. カスタム スレッショルド パッケージの構成要素

構成要素	目的
include/TI	パブリック カーネル インタフェイス、モジュール名とカーネル名、パラメータ インデックス、カーネルのロードおよびアンロード用 API、ノード作成用 API を定義します
host	ホスト カーネルの検証と初期化、ホスト側モジュール登録、およびパブリック ノード作成機能を実装します
a72	TIVX_TARGET_MPU_0 のランタイム処理コールバックおよびターゲット カーネル登録を実装します
app_custom_threshold	リファレンス OpenVX グラフを作成および実行し、スレッショルド出力を検証します
host/concerto.mak	ホスト カーネル ライブラリをビルドします
a72/concerto.mak	A72 ターゲット カーネル ライブラリをビルドします
app_custom_threshold/concerto.mak	デモンストレーション用の実行可能ファイルをビルドし、ホスト ライブラリとターゲット ライブラリをリンクします

リファレンス実装では、ホスト カーネル、A72 ターゲット カーネル、およびデモンストレーション アプリケーションに対してそれぞれ個別の concerto.mak ファイルを使用しています。vision_apps ビルド システムは通常、これらのビルド ファイルをそれぞれのディレクトリから検出します。そのため、kernels/custom_threshold 下に追加のパッケージ レベルの concerto.mak ファイルは必要ありません。

ホスト カーネルとターゲット カーネルが異なる役割を果たすため、この分離は重要です。ホスト カーネルは OpenVX インターフェイスを定義および検証し、A72 ターゲット カーネルは MPU ターゲットで実行されるランタイム処理を実装しています。アプリケーションは両方のコンポーネントをリンクし、それらのパブリック API を使用してカーネル パッケージをロードし、カスタム ノードを作成します。

3.2 カーネル インターフェイスの定義

カスタム スレッショルド パッケージのパブリック インターフェイスは、include/TI 下の 3 つのヘッダー ファイルに分割されています。

1) パッケージ ヘッダーを集約する

tivx_custom_threshold.h ヘッダーは、パッケージを使用するアプリケーションに対して、単一のインクルード ポイントを提供します。

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

アプリケーションには、カーネル パッケージ管理 API とパブリック ノード作成 API の両方にアクセスするために、このヘッダーを含めることができます。

2) カーネル定義とパッケージ API

tivx_custom_threshold_kernels.h ヘッダーは、ホストおよびターゲットのカーネル実装の双方で使用されるモジュール名と一意のカーネル名を定義します。

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

モジュール名は、モジュールの登録およびロード中にカーネル パッケージを識別します。カーネル名は、ホスト カーネル、ターゲット カーネル、およびアプリケーションが対応する OpenVX ノードを作成または検索するときに、カスタム スレッショルド カーネルを識別します。

数値形式のユーザー カーネル ID は、パブリック ヘッダーで固定されていません。代わりに、**vxAllocateUserKernelId()** を使用してホスト カーネルによって実行時に割り当てられます。これにより、ハード コードされたカーネル列挙値の割り当てが回避され、OpenVX コンテキストが利用可能なユーザー カーネル ID を提供できるようになります。

同じヘッダーでは、ホスト カーネル、ターゲット カーネル、ノード作成 API で共有されるパラメータ インデックスも定義されています。

```
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. カスタム スレッショルド ノードのパラメータ

インデックス	パラメータ	方向	OpenVX の種類	説明
TIVX_KERNEL_CUSTOM_THRESHOLD_INPUT_IDX	入力画像	入力	VX_TYPE_IMAGE	U8 グレイスケール ソース画像

表 3-2. カスタム スレッショルド ノードのパラメータ (続き)

TIVX_KERNEL_CUSTOM_THRESHOLD_THRESHOLD_IDX	スレッショルドの値	入力	VX_TYPE_SCALAR	ピクセル比較に使用される U8 スカラー
TIVX_KERNEL_CUSTOM_THRESHOLD_OUTPUT_IDX	出力画像	出力	VX_TYPE_IMAGE	U8 バイナリ出力画像
TIVX_KERNEL_CUSTOM_THRESHOLD_MAX_PARAMS	パラメータ数	-	-	ノード パラメータの合計数

カーネル パッケージ ヘッダーは、カスタム カーネルのロードとアンロードに使用される API も宣言しています。

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

これらの関数は、ホスト カーネル モジュールと A72 ターゲット カーネルの登録を調整します。アプリケーションは、OpenVX コンテキストを作成した後に LOAD 関数を呼び出し、コンテキストを解放する前に UNLOAD 関数を呼び出します。

A72 ターゲット カーネル登録関数もパブリック インターフェイスで宣言されています。

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

これらの関数は、TIVX_TARGET_MPU_0 に関連するターゲット カーネル実装を登録および登録解除します。

3) パブリック ノード作成 API

tivx_custom_threshold_nodes.h ヘッダーは、カスタム スレッショルド ノードをインスタンス化するためにアプリケーションによって使用されるヘルパー関数を宣言します。

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

この関数は OpenVX グラフと 3 つのノード パラメータを受け取り、vx_node オブジェクトを返します。内部的には、この実装はカーネル名 TIVX_KERNEL_CUSTOM_THRESHOLD_NAME を使用してノードを生成し、入力画像、スレッショルド スカラー、および出力画像をそれぞれのパラメータ インデックスに関連付けます。

4 A72 コア向けカスタム OpenVX ノードの開発：スレッショルドの例

この章では、TIOVX フレームワークを使用して Arm Cortex-A72 コアで実行されるカスタム OpenVX ノードを実装する方法について説明します。サンプル アルゴリズムとして単純なスレッショルド処理を使用しますが、この章で説明する開発プロセスは、さまざまな CPU ベースの画像処理機能に適用できます。

実装は 5 つの主要なステップで構成されています。まず、OpenVX フレームワークに表示されるカーネル インターフェイスとパラメータを定義するために、ホスト カーネルを作成します。次に、A72 コアでアルゴリズムを実行するためのターゲット カーネルを実装します。ターゲット カーネルは、特定用途向けの画像処理を実行する前に、TIOVX 共有メモリ インターフェイスを介して画像バッファにアクセスします。最後に、OpenVX アプリケーションからカスタム ノードをインスタンス化できるように、ホスト カーネルとターゲット カーネルをパッケージおよび登録します。

4.1 ホスト カーネルの実装

ホスト カーネルは、アプリケーションに公開される OpenVX ユーザー カーネルを定義します。実際の画像処理を行うターゲット カーネルとは異なり、ホスト カーネルは OpenVX フレームワークへのカーネル インターフェイスの記述を担当しています。これには、カーネル名、パラメータ タイプ、検証コールバック、初期化コールバック、およびサポートされている実行ターゲットの定義が含まれます。

ホスト カーネルの実装は、次のソース ファイルにあります。

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

このファイルにおける主要な関数は **tivxAddKernelCustomThreshold()** であり、これはカスタム カーネルの完全な登録を行います。登録プロセスは、主に 4 つのステップで構成されています。

- OpenVX ユーザー カーネルを作成します。
- 必要な入力および出力パラメータを定義します。
- サポートされている実行ターゲットを登録します。
- カーネル登録を完了します。

```
/* 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() は新しい OpenVX ユーザ定義カーネルを作成し、フレームワークによって呼び出されるコールバック関数に関連付けます。この例では、検証コールバックが、指定されたパラメータがカーネルの要件を満たしていることを確認しながら、初期化コールバックが、グラフの実行に先立って有効な画像領域を構成します。

vxAddParameterToKernel() の 3 つの呼び出しは、アプリケーションに公開されるカーネル インターフェイスを定義します。スレッショルドの例では、1 つの入力画像、1 つのスカラー スレッショルド値、1 つの出力画像が必要です。各パラメータには、ノードの作成時に使用されるパラメータの順序と一致するインデックスが割り当てられます。

tivxAddKernelTarget() の呼び出しは、このカーネルが A72 コア (TIVX_TARGET_MPU_0) で実行されることを指定します。複数の実装が利用可能な場合は、追加のターゲットを登録できます。

最後に、**vxFinalizeKernel()** は登録プロセスを完了し、カーネルをグラフの検証と実行に使用できるようにします。

グラフの検証中、OpenVX フレームワークは検証コールバックを呼び出して、グラフが実行される前にパラメータ設定を検証します。検証コールバックは、入力画像形式とスカラー型がカーネル要件と一致しているかどうかを確認します。また、フレームワークがグラフの実行前に正しい画像オブジェクトを割り当てることができるように、出力画像のメタデータも構成します。簡略化した実装は以下の通りです。

```
/* 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));
```

初期化コールバックは、検証が成功した後、グラフの実行が開始される前に実行されます。このスレッショルド処理の例では、出力画像の有効領域が入力画像の有効領域と同一になるように、コールバックが有効矩形を設定します。この処理は画像の寸法を変更せずにピクセルごとの 1 対 1 の変換を行うため、追加の初期化は不要です。

4.2 A72 ターゲット カーネルの実装

ホスト カーネルがカーネル インタフェイスを定義するのに対し、ターゲット カーネルはターゲット コア上でカスタム処理の実際の実行を実装します。この例では、ターゲット カーネルは Arm Cortex-A72 コア上で動作し、入出力オブジェクト記述子の受信、画像バッファへのアクセス、スレッショルド処理の実行、処理した結果の OpenVX フレームワークへの返却を行います。

ターゲット カーネルの実装は、次のソース ファイルにあります。

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

ターゲット カーネルは 4 つの主要なコールバック関数で構成されています。

- TIOVX フレームワークでターゲット カーネルを登録します。
- グラフの実行前に必要なリソースを作成します。
- 各ノードの実行を処理します。
- グラフが解放されたら、リソースを削除します。

ターゲット カーネルは、カスタム カーネルをコールバック関数に関連付ける **tivxAddTargetKernel()** を呼び出すことによって登録されます。

```
/* 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);
```

この登録では、カスタム カーネルを A72 実行ターゲット (TIVX_TARGET_MPU_0) に関連付け、ノードのライフサイクル中に呼び出されるコールバック関数を指定します。

グラフが正常に検証された後、**Create コールバック**が 1 回実行されます。これは通常、リソースの割り当て、内部データ構造の初期化、カーネルが必要とするハードウェア リソースの準備に使用されます。スレッショルドの例は単純な CPU ベースの画像処理のみを実行するため、追加のリソースは必要ありません。コールバックは最小限の初期化を実行します。

Process コールバックには、特定用途向けの画像処理アルゴリズムが含まれています。各グラフの実行中、OpenVX フレームワークはノード パラメータに対応するオブジェクト記述子を渡して、このコールバックを呼び出します。コールバックは共有メモリ アドレスを CPU からアクセス可能なポインタに変換し、画像バッファを A72 アドレス空間にマッピングし、画像処理を実行して、最後にフレームワークに制御を返す前にバッファのマッピングを解除します。

Delete コールバックは、グラフが解放されると呼び出されます。これは、Create コールバックによって割り当てられたリソースを解放する役割を担います。この例では永続的なリソースが割り当てられないため、コールバックは単に正常終了します。

Process コールバックは、ターゲット カーネル実装の中核です。以下のセクションでは、A72 コア上で画像バッファにアクセスする方法と、スレッショルド処理の実装方法について説明します。

4.3 A72 コアの画像バッファへのアクセス

標準的な C アプリケーションとは異なり、ターゲット カーネルは画像ポインタを直接受信しません。代わりに、OpenVX フレームワークは入出力オブジェクトを記述するオブジェクト記述子を渡します。これらの記述子には共有メモリ アドレスが含まれており、画像データを処理する前に、それらを CPU からアクセス可能なポインタに変換する必要があります。

A72 コアの画像バッファに安全にアクセスするために、ターゲット カーネルは次の 4 つのステップを実行します。

- 共有メモリ アドレスをターゲット ポインタに変換します。
- バッファを CPU アドレス空間にマップします。
- 画像データを処理します。
- 処理後にバッファのマッピングを解除します。

以下のコードに、A72 コアから画像バッファにアクセスするために必要な重要な手順を示します。

```
/* 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);
```

オブジェクト記述子には、TIOVX フレームワークによって管理される共有メモリ内の画像バッファの場所が含まれています。このアドレスは A72 コアからは直接アクセスできないため、ターゲット カーネルは最初に **tivxMemShared2TargetPtr()** を呼び出して CPU からアクセス可能なポインタを取得します。

画像データにアクセスする前に、バッファを **tivxMemBufferMap()** を使用してコア アドレス空間にマップする必要があります。マッピング処理により、CPU は共有バッファの最新のコンテンツにアクセスできるようになり、異種コア間のキャッシュ コヒーレンシーが維持されます。

バッファがマップされると、ターゲット カーネルは標準的な C ポインタを使用して画像ピクセルにアクセスすることができます。処理が完了すると、**tivxMemBufferUnmap()** が呼び出され、マップされたバッファが解放されるとともに、変更されたデータが共有メモリに同期されます。

TIOVX は共有メモリを介して画像バッファを管理するため、画像データに直接アクセスするすべてのターゲット カーネルは、このマッピングとマッピング解除の手順に従う必要があります。

バッファのマッピングが正常に完了すると、オブジェクト記述子に格納されている画像のストライド情報を使用して、画像のピクセルに行単位でアクセスできます。以下は、スレッショルドの例で使用されている基本的なポインタ計算を示しています。

```
/* 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 フィールドは、連続する画像行間のバイト数を指定します。ターゲット カーネルは、画像の行が連続的に格納されていると仮定する代わりに、この値を使用して各行の開始アドレスを計算します。このアプローチにより、OpenVX フレームワークで管理されるさまざまな画像幅、メモリ アライメント、およびバッファ レイアウトに対して、同一の実装で対応することが可能になります。

画像バッファの正常なマッピングとピクセル アドレスの算出が完了し、ターゲット カーネルは、次のセクションで説明する特定用途向けの画像処理アルゴリズムを実行する準備が整いました。

4.4 スレッショルド アルゴリズムの実装

画像バッファを A72 のアドレス空間にマッピングしたら、ターゲット カーネルは標準的な C コードを使用して特定用途向けの画像処理を実行することができます。この例では、単純なスレッショルド処理を使用して、Process コールバック内でアルゴリズムがどのように実装されるかを示しています。

スレッショルド処理では、各入力ピクセルがユーザー定義のスレッショルド値と比較されます。スレッショルドより大きいピクセルには最大輝度値 (255) が割り当てられ、他のすべてのピクセルには 0 が割り当てられます。このアルゴリズムは、画像処理アルゴリズムそのものではなく、カスタム OpenVX ノード開発の全体的なプロセスに焦点が当てられるよう、意図的に単純化されています。

```
/* 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;
}
```

入力および出力行ポインタは、前のセクションで説明した画像アドレス情報を使用して計算されます。行ポインタを取得すれば、標準的な配列インデックスを用いて個々のピクセルにアクセスできるため、従来の C アプリケーションと同様の手法で画像処理アルゴリズムを実装することが可能になります。

この例ではスレッショルド処理を実装していますが、他の CPU ベースの画像処理アルゴリズムにおいても、全体的な開発フレームワークは変わりません。前のセクションで述べたホスト カーネルの実装、ターゲット カーネルの登録、および画像バッファへのアクセスは、変更することなく再利用できます。ほとんどの場合、Process コールバック内のアルゴリズム固有のピクセル処理コードのみを置き換える必要があります。

たとえば、同じフレームワークを使用して、画像フィルタリング、色空間変換、エッジ検出、特徴抽出、AI 前処理、あるいはその他のカスタム画像処理機能を実装することができます。フレームワーク固有の実装をアルゴリズム自体から分離することで、開発者は同じホストおよびターゲット カーネル インフラストラクチャを再利用しながら、新しい A72 ベースの OpenVX カーネルを効率的に作成できます。

4.5 カーネル パッケージの登録

ホスト カーネルとターゲット カーネルの両方を実装したら、最後のステップは、カスタム カーネルをパッケージして登録し、OpenVX アプリケーションで使用できるようにすることです。TIOVX フレームワークは、ホストとターゲットの登録を独立したカーネル パッケージに分離し、アプリケーションが初期化時に必要なモジュールのみをロードできるようにします。

パッケージ登録コードは、次のソース ファイルに実装されています。

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

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

Host パッケージはカスタム カーネル インターフェイスを OpenVX フレームワークに公開し、Target パッケージは A72 コアの実行コールバックを登録します。アプリケーションの初期化中、カスタム ノードをインスタンス化する前に、両方のパッケージがロードされます。

ホスト カーネル パッケージは、カスタム カーネルを OpenVX フレームワークに公開する役割を果たします。パッケージの初期化中、登録関数はセクション 4.1 で説明されているホスト カーネル登録ルーチンを呼び出します。

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

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

ターゲット カーネル パッケージは、セクション 4.2 で説明した実行コールバックを登録します。初期化中に、パッケージはカスタム カーネルを A72 コアに関連付け、OpenVX フレームワークはノード実行を正しいターゲットにディスパッチできるようにします。

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

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

両方のパッケージが登録されると、アプリケーションは初期化中にカスタム カーネル パッケージをロードします。パッケージがロードされた後、カスタム ノードを作成し、標準の OpenVX ノードと同様に OpenVX グラフに挿入できます。

```
/* 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);
```

アプリケーション開発者の観点からは、カスタム ノードは組み込みの OpenVX ノードとまったく同じ方法で使用されます。OpenVX フレームワークは、グラフ検証中にホスト カーネルを自動的に呼び出し、グラフ処理中に登録されたターゲット カーネルに実行をディスパッチします。この抽象化により、開発者は特定用途向けアルゴリズムの実装に集中すると同時に、TIOVX フレームワークに依存して、カーネルの登録、メモリ処理、異種コアでの実行を管理することができます。

5 カスタム ノードの vision_apps への統合

前章では、ホスト カーネル、A72 ターゲット カーネル、画像バッファへのアクセス、およびカーネル パッケージの登録を含む、カスタム OpenVX ノードの開発方法について説明しました。カスタム カーネルが実装されたら、vision_apps アプリケーションに統合し、標準的な OpenVX ノードと同じ方法で使用することができます。

この章では、app_custom_threshold リファレンス アプリケーションを使用した完全な統合プロセスについて説明します。すべての実装の詳細を説明するのではなく、アプリケーションがどのように OpenVX フレームワークを初期化し、カスタム カーネル パッケージをロードし、OpenVX グラフを作成および実行し、処理結果を検証するかに焦点を当てます。不要な実装の詳細は避けつつ、アプリケーション全体の流れを示すために、重要なコードを強調表示しています。

リファレンス アプリケーションは次のディレクトリにあります。

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

アプリケーションは、標準的な OpenVX 実行フローに従います。

- アプリケーションと OpenVX フレームワークを初期化します。
- カスタム カーネル パッケージをロードします。
- OpenVX グラフとカスタム ノードを作成します。
- グラフを検証して実行します。
- 出力画像を検証し、すべてのリソースを解放します。

以下のセクションでは、app_custom_threshold の例を使用して各ステップについて説明します。

5.1 app_custom_threshold サンプルの実装

app_custom_threshold アプリケーションは、カスタム スレッショルド ノードを vision_apps アプリケーションに統合する方法を示しています。カスタム カーネル パッケージが登録されると、カスタム ノードを作成し、標準の OpenVX ノードと同じプログラミング モデルを使用して実行することができます。

このアプリケーションは、標準的な OpenVX 実行フローに従います。フレームワークを初期化し、カスタム カーネル パッケージをロードし、処理グラフを作成し、グラフを実行し、出力画像を検証して、最後に割り当てられたすべてのリソースを解放します。以下のコードは、主要なアプリケーションの流れを強調しつつ、分かりやすさのために重要でないコードを省略しています。

```
/* 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();
```

アプリケーションは、最初に vision_apps フレームワークを初期化し、OpenVX コンテキストを作成することから始まります。その後、カスタム カーネル パッケージがロードされ、カスタム スレッショルド ノードがアプリケーションで使用可能になります。入力画像、出力画像、スレッショルド スカラーなど、必要な OpenVX オブジェクトを作成した後、第 4 章で紹介したパブリック ノード API を使用して処理グラフを作成します。

グラフが作成されると、ノード ターゲットは、**vxSetNodeTarget()** を使用して Arm Cortex-A72 コアに割り当てられます。その後、**vxVerifyGraph()** を呼び出してグラフが検証されます。この関数は、実行前にグラフ構成の妥当性をチェックします。検証が正常に完了すると、**vxProcessGraph()** はグラフを実行し、カスタム カーネル パッケージによって登録されたホストおよびターゲット カーネルのコールバックを呼び出します。

最後に、アプリケーションは生成された出力画像をリファレンス画像と比較して、処理結果を検証します。検証後、すべての OpenVX オブジェクトとカスタム カーネル パッケージが解放され、アプリケーションの実行が完了します。

5.2 実行ログの例

アプリケーションをビルドした後、ターゲットの Linux コンソールから vx_app_custom_threshold.out 実行ファイルを起動します。実行中に、アプリケーションは OpenVX フレームワークを初期化し、カスタム スレッショルド ノードを実行して、結果の出力画像を印刷して、実行ステータスを報告します。次のコンソール出力は、アプリケーションが正常に実行されたことを示しています。

```
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#
```

コンソール出力により、アプリケーションが OpenVX フレームワークを正常に初期化し、カスタム スレッショルド ノードを実行したことが確認できます。印刷されたピクセル値は、生成された出力画像に対応しており、スレッショルド処理の結果を簡単に検証できます。最後に、**APP_CUSTOM_THRESHOLD: PASS** メッセージは、出力が予想されるリファレンスと一致しており、カスタム ノードが vision_apps フレームワークに正常に統合され、Arm Cortex-A72 コアで実行されたことを示しています。

6 まとめ

このアプリケーション ノートでは、テキサス インスツルメンツのプロセッサ SDK RTOS が提供する TIOVX フレームワークを使用して Arm Cortex-A72 コアで実行される、カスタム OpenVX ノードを開発するための実用的なワークフローを紹介しました。カーネル インタフェイスの定義、ホスト カーネルの実装、A72 ターゲット カーネルの開発、画像バッファへのアクセス、カーネル パッケージの登録、vision_apps フレームワークへの統合など、開発プロセス全体を説明するためのリファレンス実装として、スレッショルド ノードを使用しました。

この例では、標準的な TIOVX 実行モデルを維持しながら、特定用途向けの処理を OpenVX グラフに組み込む方法を示しました。スレッショルド処理は単純さゆえに選ばれたものですが、ここで示した開発手法全体は、画像処理、センサ処理、AI の前処理や後処理といった幅広いカスタム アルゴリズムに適用可能です。Process() コールバック内で処理ロジックのみを変更すると、同じホスト カーネル、ターゲット カーネル、メモリ アクセス、グラフ統合フレームワークを再利用して、カスタム A72 ベースの OpenVX ノードの開発を迅速化できます。

ビジョン アプリケーションとエッジ AI アプリケーションの進化に伴い、特定用途向けの処理を異種コンピューティング プラットフォームに統合することがますます重要になっています。このアプリケーション ノートで提示しているワークフローは、既存のプロセッサ SDK RTOS ソフトウェア アーキテクチャとの互換性を維持しながら、カスタム機能を使用して TIOVX フレームワークを拡張するための実用的な土台を提供します。Arm Cortex-A72 コアの柔軟性と、グラフ ベースの実行モデル OpenVX を活用することで、開発者は幅広い組込みビジョンおよび AI アプリケーションのカスタム処理ノードの迅速なプロトタイプ製作、検証、導入を行うことができます。

7 参考資料

- テキサス インスツルメンツ、TDA4VE、製品ページ。
- テキサス インスツルメンツ、『J721S2、TDA4AL、TDA4VL、TDA4VE、AM68A テクニカル リファレンス マニュアル』、テクニカル リファレンス マニュアル。
- テキサス インスツルメンツ、『TDA4VE TDA4AL TDA4VL Jacinto™ プロセッサ、シリコン リビジョン 1.0 データシート』、データシート。
- テキサス インスツルメンツ、PROCESSOR-SDK-J721S2、製品ページ。

重要なお知らせと免責事項

TI は、技術データと信頼性データ (データシートを含みます)、設計リソース (リファレンス デザインを含みます)、アプリケーションや設計に関する各種アドバイス、Web ツール、安全性情報、その他のリソースを、欠陥が存在する可能性のある「現状のまま」提供しており、商品性および特定目的に対する適合性の黙示保証、第三者の知的財産権の非侵害保証を含むいかなる保証も、明示的または黙示的にかかわらず拒否します。

これらのリソースは、TI 製品を使用する設計の経験を積んだ開発者への提供を意図したものです。(1) お客様のアプリケーションに適した TI 製品の選定、(2) お客様のアプリケーションの設計、検証、試験、(3) お客様のアプリケーションに該当する各種規格や、その他のあらゆる安全性、セキュリティ、規制、または他の要件への確実な適合に関する責任を、お客様のみが単独で負うものとします。

上記の各種リソースは、予告なく変更される可能性があります。これらのリソースは、リソースで説明されている TI 製品を使用するアプリケーションの開発の目的でのみ、TI はその使用をお客様に許諾します。これらのリソースに関して、他の目的で複製することや掲載することは禁止されています。TI や第三者の知的財産権のライセンスが付与されている訳ではありません。お客様は、これらのリソースを自身で使用した結果発生するあらゆる申し立て、損害、費用、損失、責任について、TI およびその代理人を完全に補償するものとし、TI は一切の責任を拒否します。

TI の製品は、 TI の販売条件 、 TI の総合的な品質ガイドライン 、 ti.com または TI 製品などに関連して提供される他の適用条件に従い提供されます。TI がこれらのリソースを提供することは、適用される TI の保証または他の保証の放棄の拡大や変更を意味するものではありません。 TI がカスタム、またはカスタマー仕様として明示的に指定していない限り、TI の製品は標準的なカタログに掲載される汎用機器です。

お客様がいかなる追加条項または代替条項を提案する場合も、TI はそれらに異議を唱え、拒否します。

最終更新日：2025 年 10 月