https://doi.org/10.2352/EI.2025.37.15.AVM-111

TIOVX Apps – A new approach to development with OpenVX

Rahul Ravikumar, Abhay Chirania, Shyam Jagannathan, Villarreal Jesse; Embedded Processors Business, Texas Instruments

Abstract

OpenVX is an open standard for accelerating computer vision applications on a heterogeneous platform with multiple processing elements. OpenVX is accepted by Automotive industry as a go-to framework for developing performance-critical, power-optimized and safety compliant computer vision processing pipelines on real-time heterogeneous embedded SoCs. Optimizing OpenVX development flow becomes a necessity with ever growing demand for variety of vision applications required in both Automotive and Industrial market. Although OpenVX works great when all the elements in the pipeline is implemented with OpenVX, it lacks utilities to effectively interact with other frameworks. We propose a software design to make OpenVX development faster by adding a thin layer on top of OpenVX which simplifies construction of an OpenVX pipeline and exposes simple interface to enable seamless interaction with other frameworks like v4l2, OpenMax, DRM etc....

Introduction

A typical end-to-end computer vision pipeline starts with capture, followed by multiple levels of pre-processing data that feeds into a computer vision algorithm or a deep learning network, and finally visualize and display the result or take some decisions. In real use cases, multiple pipelines may be implemented on an SoC with heterogeneous architecture. For example, common modern ADAS systems have features like 360-degree surround view, driver monitoring, obstacle detection, camera mirrors, front camera and other advanced features. Such features are amalgamation of various capture and compute blocks including but not limited to, capture from multiple cameras with varying resolutions and framerates, vision processing and deep learning. These applications often have very strict latency and throughput requirements and are required to run on power and resource constrained embedded SoCs. Different parts of the pipeline need to be efficiently mapped to DSPs, Hardware Accelerators and Compute Cores to get the desired throughput. An automotive use case on such diverse compute landscape can take many man-months to realize and can prove to be quite an entry barrier for users to quickly evaluate and prototype. It also requires a ramp on learning the middleware to even put together a simple capture-inference-display chain. The proposed approach implements a layer on top of OpenVX [3], which makes the development much faster with simple APIs inspired by Gstreamer [2], a popular pipeline-based multimedia framework.

OpenVX Development Flow

Implementing a computer vision pipeline using OpenVX involves creating and connecting multiple nodes in a graph based on the processing elements in the pipeline and also managing the data exchange with the graph. A graph in OpenVX is set of nodes connected to each other in directed acyclic fashion to achieve a given functionality. A node in OpenVX is a processing element which takes in one or multiple inputs of certain type and generates one or multiple outputs of certain type. Inputs can come from either graph input interface or from another node output, similarly output can be consumed by user or given out via graph output interface.

Exemplars in OpenVX are sample data objects which represents the type of data that node will be handling and need to be provided by the application during node creation. To connect two node interfaces, application needs to provide same exemplar to both the nodes during creation. Nodes are instatiations of OpenVX kernels. Kernels implements a specific functionality and registered on targets on which the particular functionality can be run optimally.

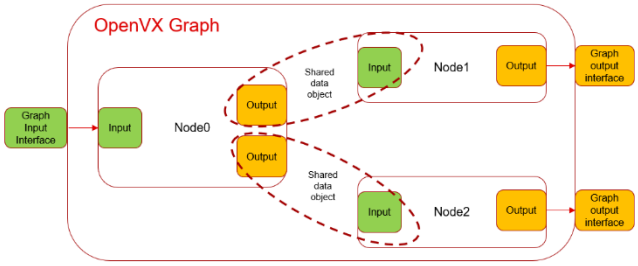
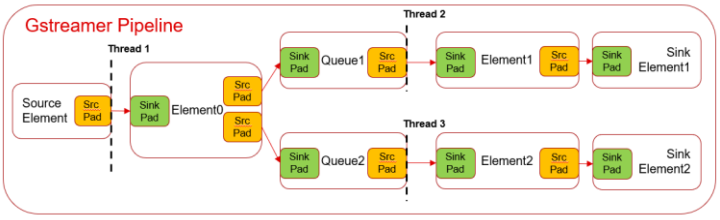


Figure 1. *Illustration of an OpenVX Graph*

Gstreamer Development Flow

Gstreamer is a framework for developing streaming applications centered around multimedia use cases. Due to its flexibility and ease of use it has gained traction in computer vision applications as well. A typical Gstreamer application involves creating a pipeline, which contains multiple elements connected to each other via pads. Element in Gstreamer represents an entity which produces or consumes data. Some elements can only produce data and are called source element, some elements only consume data and are called sink elements. Filters are the elements that can transform data. Pads in Gstreamer are entities via which data is exchanges between elements. There are two kinds of pads, Source pads and Sink pads. Source pads produces the data and Sink pad consumes the data. Each Sink pad can only be connected to one Source pad. Elements are instantiated using plugins. Each plugin implements a given functionality and defines the number of pads and caps they can handle. Caps in Gstreamer is used to specify the type of data a pad can handle. A source and sink pad should support at least one common set of caps for them to be able to connect.



- ***Figure 2.*** *Illustration of a Gstreamer Pipeline*

Similarities between OpenVX and Gstreamer

OpenVX and Gstreamer have lot of similarities in their design. Below table summarizes the similar concepts from both

OpenVX	Gstreamer
Graph	Pipeline
Node	Element
Kernel	Plugin
Shared Data Object	Caps and Pads
Graph Input/Output Interface	AppSrc and AppSink

OpenVX lacks some useful Gstreamer concept that makes development easier and interfacing with other framework simpler. For example, pads are not explicitly modeled in OpenVX, instead shared exemplars are used to connect nodes. Pads could simplify connecting and managing nodes in the application. The concept of a Buffer pool, is missing in OpenVX as well. Buffer pool helps in easily managing buffer allocation, acquiring/releasing buffers and pipeline the interaction between multiple frameworks.

It is possible that an OpenVX kernel might not be available for some kind of custom processing or a different framework is used for certain tasks. For example, v4l2 is used for camera capture and encode/decode, DRM is used for display, OpenGL is used for accessing GPU in Linux. In this case there are two options, first is wrapping these frameworks or custom code under custom OpenVX kernels, another is splitting the graphs and managing the data exchange in the application. Both of these options need a deeper understanding of OpenVX framework and might take quiet a bit of time and effort for development and optimization. Modeling these Gstreamer concepts like Pads, Buffer Pools, etc. can help tackle the problem in a better way.

TIOVX Apps Layer

The objective of TIOVX Apps Layer is to make it easier to realize complex pipelines involving custom code and different frameworks, which is optimized for memory and throughput and without involving any buffer copy. TIOVX Apps Layer provides a simplified solution for the construction and management of OpenVX pipelines by giving Gstreamer-like interface to the application. It imagines a new way of writing an OpenVX application by incorporating some of the best concepts from Gstreamer. It implements data structures and APIs which makes it simpler to create and connect compatible nodes, exchange input/output data via Pads, automatically create and manage shared buffer pools and provide a structured way of connecting custom processing blocks as Modules. The layer abstracts some of the steps required in traditional OpenVX application, to avoid writing lot of boilerplate code. At the same time, it gives access to all the underlying OpenVX data objects, in-case user wants to call some OpenVX API directly.

Key concepts implemented in the layer

Graph Object: Graph object is a wrapper around OpenVX Graph. Along with OpenVX graph it also stores additional information like list of nodes in the graph, graph parameter indexes, OpenVX context etc.

Node Object: Node object is a wrapper around OpenVX node. Along with OpenVX Node it will have a sink-pad for each input and source-pad for each output. Two node objects are connected via Pads.

Pads: Pads represents the input/output to a node. There are two types of pads, sink-pad - for input parameter, source-pad - for output parameter. Pad manages things like node parameter index, data objects for corresponding node parameter, number of channels etc. A source-pad can be linked to a sink-pad to connect two nodes or left floating to expose them as graph parameter, which can be managed by the application

Buffers and Buffer pools: Application need to enqueue/dequeue buffers for all the pads exposed as graph parameter. To make this easier, a pool of buffers is allocated for all floating pads. Application can acquire buffers form buffer pools and use it and release it back when done

Modules: Each module is a wrapper around an OpenVX kernel, which encapsulates the code for Creating data objects for inputs and outputs, Initialize Pads based on number of inputs and outputs Create the OpenVX node, and exposes a simple config data structure for user

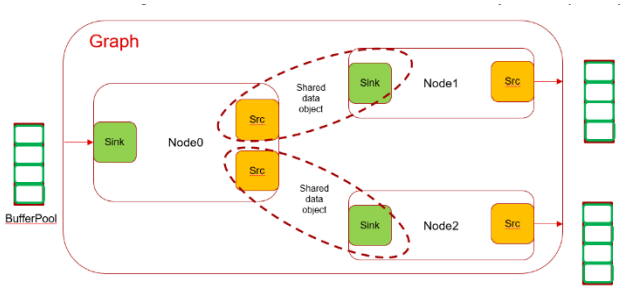


Figure 3. *Illustration of an OpenVX Application written using TIOVX Apps*

Application development Flow

The figure summarizes the application development flow with introduction of TIOVX Apps Layer on top of OpenVX

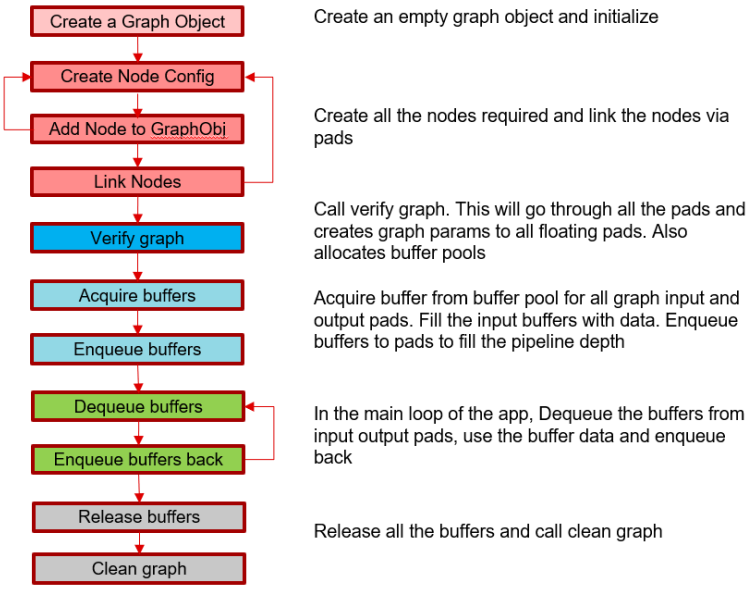


Figure 4. *Application development flow using TIOVX Apps*

Interaction using Pads and Buffer Pool

Let's take an example of simple Capture Display pipeline to showcase how we can use TIOVX Apps Layer to simplify the application code. As shown in the figure below, simple Capture Display pipeline involves 3 frameworks, v4l2 for capture, OpenVX for Image Signal Processing (ISP) and DRM for Display.

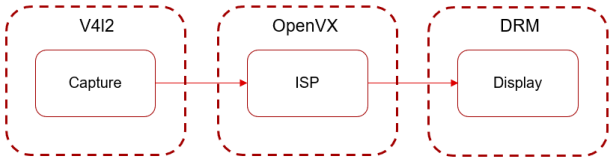


Figure 5. *Frameworks involved in Capture Display Pipeline*

Writing an optimized application for this pipeline involves managing the pipelining between v4l2, OpenVX and DRM by creating multiple threads for buffer enqueue/dequeue at each interface. The figure below showcases how this is made simpler with the TIOVX Apps layer

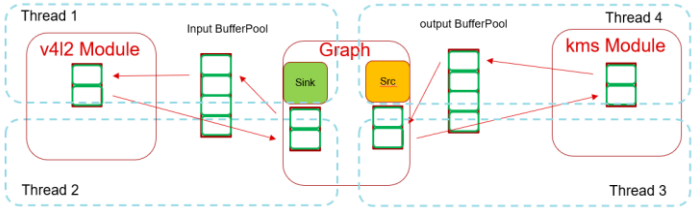


Figure 6. *Multiple frameworks interacting with TIOVX Apps Layer*

In this example, four threads are required for optimally realizing the pipeline, 1) Dequeue from v4l2 and enqueue input to Graph 2) Dequeue processed Graph input and release it to pool 3) Dequeue output from Graph and enqueue to DRM and 4) Dequeue rendered buffer from DRM and release to pool. Each thread just needs a Pad handle to enqueue/dequeue buffers, and concurrency is handled in buffer pool APIs, making buffer management easier for the users. Buffer pool helps in writing module for interfacing with non OpenVX frameworks, as it defines a simpler way to acquire and release buffers.

TEE Module for one to many connections

OpenVX allows connecting an output of the node to inputs of multiple nodes. When using OpenVX directly, this can be done using same shared exemplar for all node parameter that user wants to connect. TIOVX Apps Layer uses pads to connect nodes which supports only one to one connection i.e. one sink pad can only be connected to one source pad. To address this problem, TIOVX Apps introduces a special module called TEE, which can replicate a give sink pad into multiple sink pad that can be connected to multiple nodes. TEE module is not backed by an actual OpenVX kernel, it just ref counts the exemplar of input sink pad and shares it with the replicated source pad. Number of replicated pads is configured during node creation. Below figure illustrates the usage of TEE node.

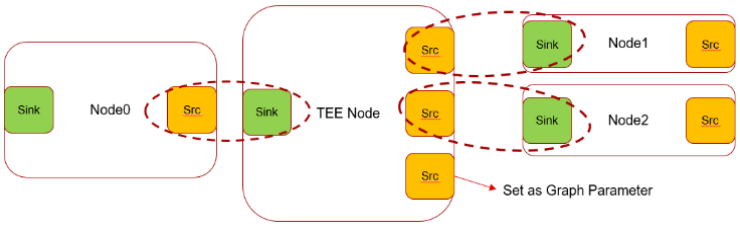


Figure 7. *Usage of TEE module*

In the figure above, output of node0 is connected to input of node1 and node2 via a TEE. A pad is also exposed as graph parameter which can be managed by application using pad handle.

Support for multi graph execution

Lot of times there is a need to split the OpenVX pipeline for some custom processing in between, such as in place transformation like text overlay, adding logo or even drop buffer based on some condition. For such cases, a seamless management of buffers needs to be implemented between two graphs for optimal performance. This requires a lot of additional effort while using OpenVX directly. TIOVX Apps layer makes it simpler with pads and buffer pools. Also, there is support for memory handle exchange between two buffers, which is useful for in-place update or dropping the buffer. The figure below illustrates the multi graph use case.

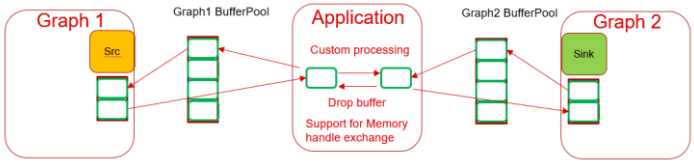


Figure 8. *Multi graph execution*

Results

We are comparing the proposed solution against development using direct OpenVX and the work done in [1], which proposes to wrap around OpenVX nodes as Gstreamer plugins. We are using performance, effort for automotive safety qualification, flexibility, operating system support and effort to add new kernel as metrics for comparison. We have taken multi-channel camera-based object detection application running on Sitara AM62A [4] SoC from Texas Instruments as reference to get the performance metrics. Pipeline is shown in the figure below.

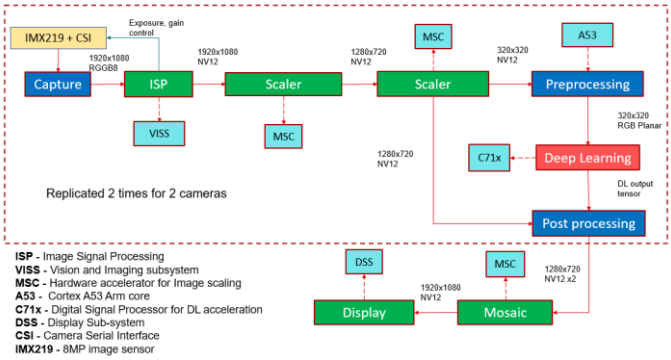


Figure 9. *Multi-channel object detection pipeline*

The table shows the comparison between the Gstreamer approach [1], TIOVX Apps Layer and OpenVX.

	Gstreamer [1]	**TIOVX Apps Layer**	**OpenVX**
Performance	A53 ~ 35% DDR BW ~ 2250 MB/s IPC Interrupts ~ 28433	A53 ~ 28% DDR BW ~ 2050 MB/s IPC Interrupts ~ 15944	A53 ~ 28% DDR BW ~ 2050 MB/s IPC Interrupts ~ 15944
Effort For safety qualification	Hard to get safety certified since • Open Source • large evolving code • Linux centric	Minimal extra effort required for safety certification as this is a thin layer on top of OpenVX	Little less effort, when compared to TIOVX Apps approach.
Flexibility	• Highly flexible, • Wide variety plugins for multimedia • Plugins for integration with other frameworks	• Fairly flexible, • Well defined interface for data exchange • Minimal effort for integrating a new framework	Lack of clearly defined interface for pipelining buffers with other frameworks, makes it challenging
Operating system	Linux	OS Agnostic	OS Agnostic
Effort to add custom kernel	Need to add support in multiple layers • EdgeAI Gst Plugins • EdgeAI TIOVX Modules • OpenVX	Need to add support in • TIOVX App modules • OpenVX	Need to add support in • OpenVX

Conclusion

In this paper we have proposed a new layer on top of OpenVX called TIOVX Apps Layer, which has enables faster development environment for OpenVX, with right balance between flexibility and control over the lower level components. Some of the key highlights of having this software design is, faster development compared to native OpenVX without impacting the performance, modular code and easy to debug, easier to realize and change the pipeline by adding, removing or modifying an element as compared to native OpenVX, simplified integration with other frameworks like v4l2, DRM, OpenMaxs, easier to safety qualify and go into production with as compared to Gstreamer. Proposed solution is implemented and tested on Sitara and Jacinto Analytics SoCs from Texas Instruments, source code is public and can be found here [5].

References

[1] Shyam Jagannathan, Vijay Pothukuchi, Villarreal Jesse, Kumar Desappan, Manu Mathew, Rahul Ravikumar, Aniket Limaye, Mihir Mody, Pramod Swami, Piyali Goswami, Embedded Processors Business, Texas Instruments, Carlos Rodriguez, Emmanuel Madrigal, Marco Herrera, "OpTIFlow – An optimized end-to-end dataflow for accelerating deep learning workloads on heterogeneous SoCs", AVM track, Electronic Imaging, 2023

[2] Gstreamer- https://gstreamer.freedesktop.org/documentation/applicationdevelopment/introduction/gstreamer.html

[3] OpenVx - https://www.khronos.org/openvx/

[4] AM62A - https://www.ti.com/tool/SK-AM62A-LP

[5] EdgeAI TIOVX Apps - https://github.com/TexasInstruments/edgeai-tiovx-apps

Author Biography

Rahul Ravikumar is a Software Engineering Manager working on SDKs for TI Jacinto devices at Embedded Processors Group, Texas Instruments. His domains of interests include Edge Analytics, Embedded Linux, Gstreamer, OpenVX, Yocto, RTOS. He received a master's degree from BITS Pilani in the field of Embedded Systems and been with Texas Instruments since 2021

Abhay Chirania is a Software Engineer working on accelerating deep learning on TI devices at Embedded Processors Group, Texas Instruments. His domain of interests includes Edge Analytics, Deep learning, TensorFlow, ONNX, Gstreamer. He received a bachelor's degree from SRM Institute of Science and Technology

Shyam Jagannathan is an EdgeAI architect and Senior Member of Technical Staff at Embedded Processors Group, Texas Instruments. His domains of interest include DSP architecture, SoC architecture, hardware accelerators, deep learning, perception, sensor fusion localization, path planning and overall system optimization He received a master's degree in the field of Signal Processing and Communications from Illinois Institute of Technology, Chicago in 2013

Jesse Villarreal is a software architect for TI's heterogeneous multicore SoCs and a Senior Member of Technical Staff (SMTS) at Embedded Processors Group, Texas Instruments. He received a master's degree from the University of Texas at Dallas in Computer Engineering and has been with Texas Instruments since 2001. His areas of interest include DSP software optimization, heterogeneous multicore middleware frameworks, vision and imaging hardware accelerators, and overall system software scalability, portability, and optimization