Novel OpenVX Implementation for Heterogeneous Multi-Core Systems

Kedar Chitnis, Jesse Villarreal*, Brijesh Jadav, Mihir Mody, Lucas Weaver*, Victor Cheng*, Kumar Desappan, Anshu Jain, Pramod Swami

Automotive Processor Business Unit, Texas Instruments, Bangalore, India & *Dallas, TX, USA
Email: {kedarc, jesse.villarreal, brijesh.jadav, mihir, l-weaver, victor_cheng, kumar.desappan, anshu.jain, pramods}@ti.com

***Abstract*— Heterogeneous multi-core systems (CPU, GPU, HWA, DSP) are becoming the de-facto norm for multiple computer vision applications across automotive, robotics, AR/VR, and industrial machine vision. This creates a need for a software framework which realizes high utilization of computing elements, low latency, real-time operation and ease of use. For specific applications, multiple proprietary solutions are offered to satisfy few of the above requirements. This paper proposes a solution based on the standard OpenVX specification to address heterogeneous systems. It introduces novel techniques of distributed graph execution across heterogeneous cores, data tiling to address diverse memory constraints and easy to use high-level graph description to describe the application. This novel solution is implemented on TI's TDA family of SoC for mono camera vision application with platform code generated from high-level graph description. The profiling confirms real time operation, low latency by reducing host CPU interaction and achieving 99% utilization across heterogeneous cores.**

Keywords: Software for Multi-core, Heterogeneous, Runtime systems, Computer vision, OpenVX

I. Introduction

Computer vision applications are becoming increasingly important in domains like autonomous driving, ADAS, augmented/virtual reality, machine vision, and robotics. In order to achieve high performance at low power and low cost, heterogeneous computing elements are used to implement such vision systems. Heterogeneous systems combine a general purpose CPU, like ARM or x86, vision optimized compute units like DSPs, Hardware accelerators (HWA), Vector Coprocessors (VCOP) and Graphics Processing Unit (GPU). Figure 1 shows a representative SoC having such heterogeneous compute elements. A computer vision application typically consists of low-level pixel pre-processing, mid-level feature computation and high level classification/analysis functions. In a true heterogeneous system different compute units are designated to run each such function depending on its suitability for that function. Low-level pixel processing like RAW sensor data conversion to YUV video data is typically well defined and well suited to be handled by a HWA like ISP (image signal processing). Mid-level functions like "Feature plane computation" have many different software implementations depending on the application, ex, HoG for pedestrian detection in ADAS use-cases. Therefore for such functions, a programmable vector co-processor with SIMD like features is well suited to give the right balance of flexibility and performance at low power. For higher level classification functions a general purpose CPU or DSP is better suited in order to run mix control functions and signal processing functions. Figure 2 shows the graph of a sample mono-camera analytics application implemented on TDA2x SOC. Here a Programmable Vector Co-processor (VCOP), a HWA engine, and multiple DSPs are used to realize the mono-camera analytics application. In order to implement such a graph an application developer needs an API which allows the expression of such a graph. Once the graph is specified, the underlying framework then needs to efficiently schedule such a graph such that all computes units are utilized effectively. In this paper we propose usage of OpenVX API in order to describe such a graph.

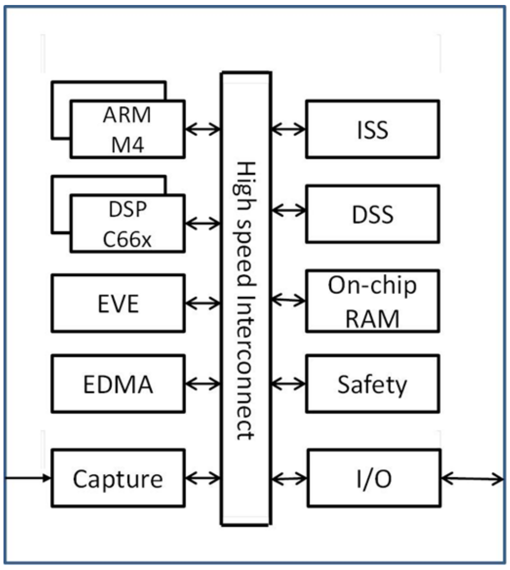


Figure 1: TDA2x ADAS Processor

OpenVX is an open, royalty-free standard for cross platform acceleration of computer vision applications. OpenVX enables performance and power-optimized computer vision processing, especially important in embedded and real-time use cases such as face, body and gesture tracking, smart video surveillance, advanced driver assistance systems (ADAS), object and scene reconstruction, augmented reality, visual inspection, robotics and more. The OpenVX API used to implement this application allows users to specify a system application as graph nodes connected via data objects as shown in the Figure 2. OpenVX API has two distinct phases, a non-real time create or verify phase and run-time execute phase. The verify phase typically happens once during system initialization and is marked by the execution of vxVerifyGraph API after which the run-time execution phase begins. During run-time a graph is typically executed repeatedly, by calling vxScheduleGraph / vxWaitGraph, in order to perform the required application function. The distinct phase of graph verification allows an implementation to perform decisions and optimization which later at run-time would allow efficient execution of the graph on the SoC.

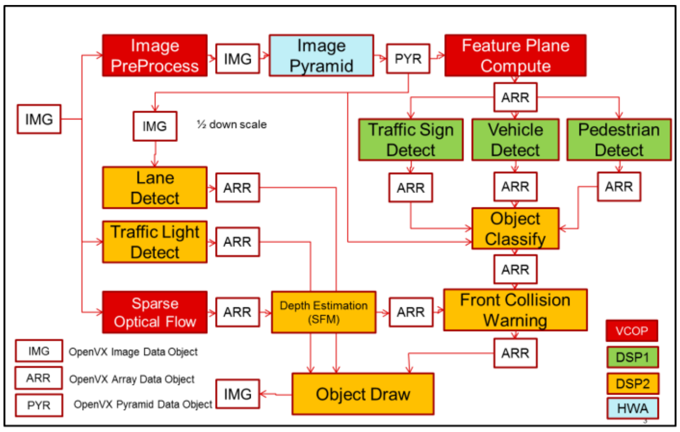


Figure 2: Mono-Camera Analytics Graph

While the OpenVX provides APIs to describe the graph, the underlying implementation needs to schedule the various nodes of the graph on different compute units in order to get maximum performance at lowest latency. Further the process of describing the graph and fine tuning it based on execution on the SoC can be a tedious time consuming effort. As the number of nodes and interconnections between the nodes increase, the number of lines of code increase, making debugging the application and changing the code harder over time.

In this paper we describe techniques used by TI's OpenVX implementation to schedule the graph in a distributed manner in order to fully utilize the underlying heterogeneous compute units. We also show how data tile/block based execution is used in order to pipeline steps within a larger function in order to reduce bandwidth at external memory. Next we introduce a tool for automated generation of OpenVX application code given a compact graph description. This tool allows users to visualize the graph and trap common programming mistakes even before executing the application code on a SoC, thus greatly increasing programmer productivity. Finally we present results obtained after using the techniques summarized above.

II. PROPOSED SOLUTION

The following novel methods are used to implement a high performance, low overhead, real-time, easy to use OpenVX framework on a true heterogeneous multi-core SoC

A. Distributed Graph Execution

Once a graph has been described by OpenVX API, assignment of nodes to different compute units must be decided. For some functions, implementation can do a direct mapping from a function to a compute unit. Ex, in TDA2x SoC, the fastest execution of image pyramid function can be done on a dedicated HWA. However some functions, like object classify, can run equally well on DSP1 or DSP2. The problem of scheduling a function among multiple available compute units is a function of multiple parameters like average execution time of the function, other functions running on that compute unit, interconnection of the function with other functions, and other graphs running in the system. Many of these parameters are not known to the implementation and any automated compute unit assignment may be sub optimal. In our implementation, we defer this decision to the application, wherein the application uses the OpenVX API, vxSetNodeTarget to select the compute unit on which the function should run. The implementation then focuses on the efficiently distributing the nodes across the statically determined compute units.

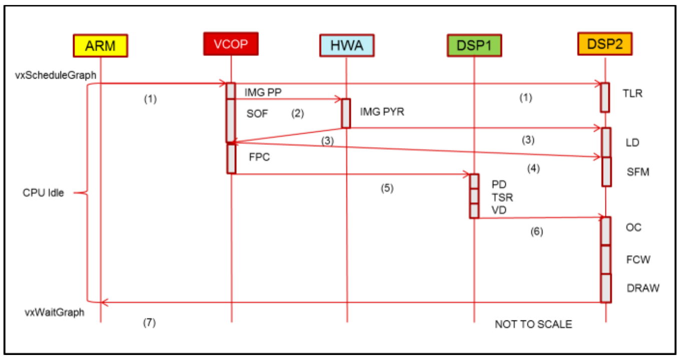


Figure 3: Distributed Graph Execution

A sample execution of the graph of Figure 2 is shown in Figure 3. In the proposed implementation, a data structure is maintained for each node, where for each node the child nodes and parent nodes are maintained. Additional information about each node like, position of the node within the graph (head, tail or intermediate), execution state of the node (not executed, executed) is also maintained within this data structure. This data structure is maintained in a coherent shared memory so that all compute units can access this data structure at run-time. The computation of this data structure is done once during vxVerifyGraph, i.e. before real-time execution of the application begins. Graph execution begins with HOST CPU (ARM in Figure 3) calling vxScheduleGraph, after which it blocks for graph completing by calling vxWaitGraph. On calling vxScheduleGraph, implementation triggers execution of the head nodes (i.e. nodes with no parents). Once a node finishes execution, it marks is state as "executed" and checks if any of the child nodes are ready for execution. Since a given node can have multiple parents, a child node is ready for execution only if all its parent nodes have executed. This process continues until a tail node (i.e. a node with no children) is reached. At this stage, the compute unit signals the HOST CPU. The HOST CPU unblocks from vxWaitGraph when all the tail nodes have signaled to the HOST. In this scheme, once a given node execution completes, it triggers the execution of its child nodes directly without intervention of the host CPU as shown in Figure 3. Further if a node has multiple child nodes then they get triggered at the same time. When the child nodes execute on different compute units, the execution of the child gets parallelized. This allows work to be distributed across available resources in a system and direct triggering of child nodes keeps the overheads associated with centralized control to a minimum. This keeps the graph execution overheads to the minimum at "run-time" (vxScheduleGraph), thus resulting in high utilization of underlying heterogeneous compute units.

B. Data Tile/Block based execution

In an OpenVX graph, typically, there are multiple interconnected nodes running on the same compute unit, say DSP. As part of vxVerifyGraph phase (which happens before real-time execution of graph), an implementation can combine graph nodes into a larger virtual node. OpenVX allows this by having user specify intermediate data objects as virtual objects. When a data object is specified as virtual, an implementation can choose to not physically represent this data object in memory. When nodes execute on the same compute units and are connected to each other via virtual objects, then the proposed implementation combines the nodes to form a larger virtual node.

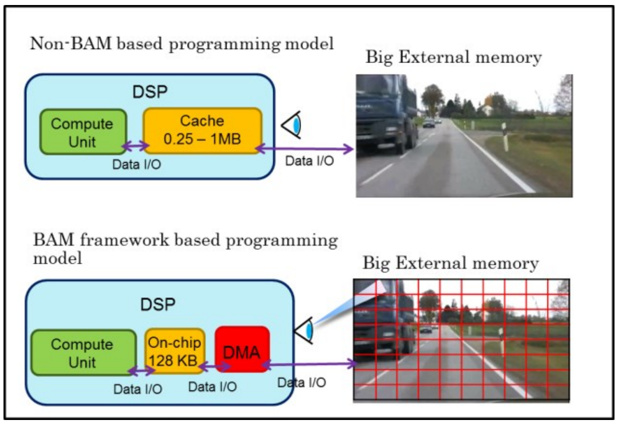
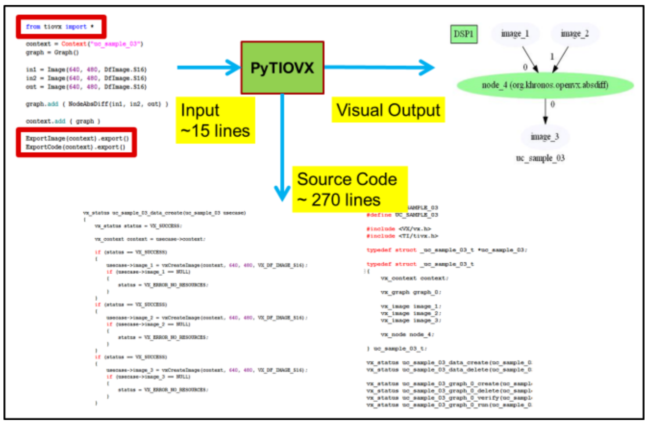


Figure 4: Block Access Manager (BAM)

Each compute unit, like DSP, usually has only a few 100KBs of on-chip memory. Typically the size of an input image frame is in MBs. Although an on-chip data-cache can help, it is usually not efficient in doing 2-D image accesses. In such a case, the proposed implementation divides an input frame into smaller 2-D blocks and pipelines the execution of these nodes on a "block" boundary using the BAM (Block Access Manager) framework. The intermediate results are buffered at the high-performance L2/L3 memory. Hence this scheme reduces the input/output accesses made in the external memory. Figure 4 shows an input image split into multiple blocks. Each block is fetched via DMA to L2 memory and multiple functions within the larger virtual node of the OpenVX graph operates on this block. The final output is DMAed back to the external memory. This operation is done in a double buffered (ping-pong) manner which allows concurrent execution of DMA and computation units. Such an implementation results in a reduced latency of node execution and overall system throughput is improved by avoiding memory bandwidth usage at the external memory. During this operation the intermediate data is never completely available at the external memory, hence the data object must be marked as virtual by the application in order for this optimization to take effect.

C. Ease of use via Automated code generation

A typical OpenVX graph can have a large number of nodes and data objects connected to each other. Users would also change these connections during development as new features are added. Writing the OpenVX API by hand and code maintenance for updates can be a non-trivial task, needing a lot of development time. In this process, possibilities of human errors is high, thus increasing the test and debug cycle. Further as shown in earlier section an implementation expects some directives from application so that the graph can execute efficiently. Ex, selecting a node to run on a specific target or marking a data object as virtual. It is time consuming to iteratively specify the directive or conversely correct the specification of a wrong directive until the desired effect is observed. Visualization of the graph as a whole with information about the distribution of compute units, location of virtual objects helps identify and fix such issues early in the development cycle.



```
from tiovx import *

context = Context("uc_sample_03")
graph = Graph()

in1 = Image(640, 480, DfImage.S16)
in2 = Image(640, 480, DfImage.S16)
out = Image(640, 480, DfImage.S16)

graph.add ( NodeAbsDiff(in1, in2, out) )

context.add ( graph )

ExportImage(context).export()
ExportCode(context).export()
```

```
vx_status uc_sample_03_data_create(uc_sample_03 usecase)
{
    vx_status status = VX_SUCCESS;

    vx_context context = usecase->context;

    if (status == VX_SUCCESS)
    {
        usecase->image_1 = vxCreateImage(context, 640, 480, VX_DF_IMAGE_S16);
        if (usecase->image_1 == NULL)
        {
            status = VX_ERROR_NO_RESOURCES;
        }
    }
    if (status == VX_SUCCESS)
    {
        usecase->image_2 = vxCreateImage(context, 640, 480, VX_DF_IMAGE_S16);
        if (usecase->image_2 == NULL)
        {
            status = VX_ERROR_NO_RESOURCES;
        }
    }
    if (status == VX_SUCCESS)
    {
        usecase->image_3 = vxCreateImage(context, 640, 480, VX_DF_IMAGE_S16);
        if (usecase->image_3 == NULL)
        {
            status = VX_ERROR_NO_RESOURCES;
        }
    }
```

```
SAMPLE_03
#define UC_SAMPLE_03

#include <VX/vx.h>
#include <TI/tivx.h>

typedef struct _uc_sample_03_t *uc_sample_03;

typedef struct _uc_sample_03_t
{
    vx_context context;

    vx_graph graph_0;

    vx_image image_1;
    vx_image image_2;
    vx_image image_3;

    vx_node node_4;

} uc_sample_03_t;

vx_status uc_sample_03_data_create(uc_sample_0
vx_status uc_sample_03_data_delete(uc_sample_0

vx_status uc_sample_03_graph_0_create(uc_sampl
vx_status uc_sample_03_graph_0_delete(uc_sampl
vx_status uc_sample_03_graph_0_verify(uc_sampl
vx_status uc_sample_03_graph_0_run(uc_sample_0
```

Figure 5: Ease of Use

As shown in Figure 5, the proposed implementation offers a convenient tool which allows a user to specify the graph via a compact graph description. The tool outputs a bitmap image of the graph for human visualization. The bitmap image color codes nodes based on the compute units and color codes virtual vs non-virtual data objects so that users can easily see how the described graph will be distributed on the underlying SoC. The tool also does a first level error analysis (ex, those that are done during vxVerifyGraph) offline so that mistakes can be corrected even before anything is executed on the SoC. Common mistakes like passing incompatible data objects or data formats, connecting incompatible nodes can be trapped by the tool. Finally, it outputs a complete "C" code with OpenVX APIs invoked with correct parameters and in the correct sequence. This code can be compiled and executed on the target SoC without need any further changes.

III. RESULTS

OpenVX is implemented for heterogeneous Multi-Core TDA2x SoC from TI. Mono-camera analytics graph in Figure 2 is implemented and standalone execution time of individual nodes is listed in Table 1. The execution time is normalized based on execution time of "Image Pre-process" function.

Function	**Compute Unit**	**Normalized execution time**
Image Pre-process	VCOP	1.0
Image Pyramid	HWA	3.0
Feature Plane Compute	VCOP	3.4
Sparse Optical Flow	VCOP	3.6
Traffic Sign Detect	DSP1	1.9
Vehicle Detect	DSP1	1.9
Pedestrian Detect	DSP1	1.9
Object Classify	DSP2	1.5
Lane Detect	DSP2	0.4
Traffic Light Detect	DSP2	0.6
Depth Estimation	DSP2	0.5
Front Collision Warning	DSP2	0.3
Object Draw	DSP2	0.5

Table 1: Normalized standalone performance

The execution time of the graph when executed serially vs when executed in a distributed manner is shown in Table 2.

Graph execution mode	**Normalized execution time**
Serial graph execution	20.4
Distributed graph execution	16.0

Table 2: Graph execution mode, serial vs distributed

The execution overhead of the framework is a function of number of messages or interrupts exchanged between the various compute units. This is good measure of the implementation overhead since a message or interrupt exchange from one compute unit to another needs additional cycles to setup the message data, trigger the HW mechanism to send a interrupt, handle the interrupt on the receive side, and context switch to the processing thread. The lesser the number of messages/interrupts exchanged, the lower the implementation overheads. In Table 3, the number of messages / interrupts exchanged in a distributed graph execution is compared against a centralized approach wherein after each node execution a message is sent to the host so that the host can decide the next set of nodes to execute. Practically, on target SoC, framework overhead of < 0.5% is observed when executing a graph using distributed execution.

Graph execution mode	**Number of messages exchanged**	**Remarks**
Centralized graph	26	13 nodes in the graph. 1 message to trigger node. 1 message to signal node completion
Distributed graph	9	

Table 3: Number of messages exchanged

Table 4 shows the effect of using BAM framework for the 40+ OpenVX v1.1 defined kernels. Finally, the entire OpenVX "C" code (~1000 LOC) for this application is generated using 100 lines of abstract graph description, thus improving developer productivity.

OpenVX v1.1 kernel	**Normalized execution time**	**Remarks**
With BAM	1	640x480 size image for supported image formats like U8, S16
Without BAM	2.3	

Table 4: BAM vs non-BAM performance

IV. CONCLUSION

Heterogeneous SoC consisting of CPU, DSP, HWA, VCOP compute units offer right mix of fixed function and programmable compute to allow users to implement computer vision applications on power and thermal constrained embedded platforms. OpenVX is a flexible API for such embedded computer vision applications and allows users to describe real world computer vision applications as abstract data dependency graphs. TI's proposed implementation exploits features in OpenVX specification in order to efficiently execute a graph across multiple compute units on a heterogeneous SoC. Distributed graph execution reduces processing latency by a factor of 22% and reduces implementation framework overhead by a factor of 2.9X for the targeted application. Automated code generation and offline graph visualization improves ease of use by allowing users to construct, verify and analyze the graph before executing on the SoC.

REFERENCES

[1] E. Rainey et al. "Addressing system-level optimization with OpenVX graphs." Proceedings of the IEEE Conference on CVPR. 2014.

[2] K. Yang et al. "Analysis for supporting real-time computer vision workloads using OpenVX on multicore+ GPU platforms." Proceedings of the 23rd Conference on Real Time and Networks Systems. ACM, 2015.

[3] Khronos Group, The OpenVX™ Specification, May 2016, version 1.1

[4] P. Viswanath et. al, "A Diverse Low Cost High Performance Platform for ADAS Applications", IEEE conference on CVPR Workshops, July 2016

[5] M. Mody et. al, "High performance front camera adas applications on ti's tda3x platform", IEEE High Performance Computing (HiPC), Dec 2015

[6] Z. Nikolic et. al, "A Scalable Heterogeneous Multicore Architecture for ADAS", IEEE Hotchips Conference, August 2015

[7] Texas Instruments ADAS & Automated Driving SoC, URL www.ti.com/adas

[8] K. Chitnis et. al, "Enabling Functional Safety for Autonomous Driving Software Systems", Electronic Imaging and System - Feb 2017