A Framework for the Design and Simulation of Embedded Vision Applications Based on OpenVX and ROS

Stefano Aldegheri, Nicola Bombieri, Nicola Dall'Ora, Franco Fummi, Simone Girardi, Marco Panato
Department of Computer Science
University of Verona
Email: name.surname@univr.it

***Abstract*—Customizing computer vision applications for embedded systems is a common and widespread problem in the cyber-physical systems community. Such a customization means parametrizing the algorithm by considering the external environment and mapping the Software application to the heterogeneous Hardware resources by satisfying non-functional constraints like performance, power, and energy consumption. This work presents a framework for the design and simulation of embedded vision applications that integrates the OpenVX standard platform with the Robot Operating System (ROS). The paper shows how the framework has been applied to tune the ORB-SLAM application for an NVIDIA Jetson TX2 board by considering different environment contexts and different design constraints.**

I. Introduction

Computer vision is a key component in modern cyber-physical systems. Its main goal is the use of digital processing and intelligent algorithms to interpret meaning from images or video. Due to the emergence of very powerful, low-cost, and energy-efficient processors, it has become possible to incorporate practical computer vision capabilities into embedded systems.

Nevertheless, developing and optimizing a computer vision application for an embedded system is far from straightforward and fast. Such a class of applications generally consists of communicating and interacting kernels, whose optimization spans across two dimensions: the kernel-level and the system-level optimizations. Kernel-level optimizations traditionally involve one-off or single function acceleration. This typically means that a developer re-writes, by using a language like OpenCL or CUDA, a computer vision function (e.g., image filter, geometric transform function) to be offloaded on a hardware accelerator like a GPU [1]. System-level optimizations target the overall power consumption, memory bandwidth loading, low-latency functional computing, and inter-processor communication overhead. These tasks are generally addressed through frameworks [2], as the application *knobs* cannot be set by using only compilers or operating systems.

OpenVX [3] is increasingly gaining consensus in the embedded vision community as API library for system-level optimizations. Such a platform is designed to maximize functional and performance portability across different hardware platforms, providing a computer vision framework that efficiently addresses different hardware architectures with minimal impact on software applications. Starting from a graph model of the embedded application, it allows for automatic system-level optimizations and synthesis on the target architecture by optimizing performance, power consumptionm and energy efficiency [4], [5], [6].

Nevertheless, the adoption of OpenVX targeting the application tuning and validation for a given embedded system has several limitations when both the tuning and validation have to consider other systems, which are external to the embedded board, and that interact with the application.

This paper presents a framework for the design and simulation of embedded video applications that integrates the OpenVX standard platform with the Robot Operating System (ROS) [7]. The goal of the framework is twofold. First, it aims at combining OpenVX, CUDA/OpenCL, and OpenMP to increase the parallelism and the portability of the embedded application code. Second, it aims at co-simulating and parametrizing the application by integrating different simulation kernels thorough the ROS API library.

The paper presents the results obtained by applying the proposed framework to customize the ORB-SLAM application [8] for an NVIDIA Jetson TX2 board by considering a camera sensor and a trajectory controller as external interacting systems.

The paper is organized as follows. Section II presents the background and the related work. Section III presents the key concepts of the framework. Section IV presents the experimental results, while Section V is devoted to the conclusions.

II. Background and related work

OpenVX relies on a graph-based software architecture to allow the embedded applications to be portable and optimized to different and heterogeneous architectures. It provides a library of primitives that are commonly used in computer vision algorithms and data objects like scalars, arrays, matrices and images, as well as high-level data objects like histograms, image pyramids, and look-up tables.

The developer defines a computer vision algorithm by instantiating kernels as nodes and data objects as parameters. Each node of the graph can use any processing units of target target heterogeneous board and, as a consequence, the

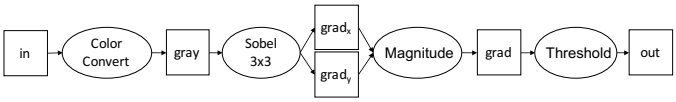

Fig. 1. OpenVX sample application (graph diagram)

```
1   vx_context c = vxCreateContext();
2   vx_graph g = vxCreateGraph(context);
3   vx_enum type = VX_DF_IMAGE_VIRT;
4   /* create data structures */
5   vx_image in = vxCreateImage(c, w, h, VX_DF_IMAGE_RGBX);
6   vx_image gray   = vxCreateVirtualImage(g, 0, 0, type);
7   vx_image gr_x = vxCreateVirtualImage(g, 0, 0, type);
8   vx_image gr_y = vxCreateVirtualImage(g, 0, 0, type);
9   vx_image gr   = vxCreateVirtualImage(g, 0, 0, type);
10  vx_image out = vxCreateImage(c, w, h, VX_DF_IMAGE_U8);
11  vx_threshold threshold = vxCreateThreshold(c,
     VX_THRESHOLD_TYPE_BINARY, VX_TYPE_FLOAT32);
12  /* read input image and copy it into "in" data object
     */
13  ...
14  /* construct the graph */
15  vxColorConvertNode(g, in, gray);
16  vxSobel3x3Node(g, gray, g_x, g_y);
17  vxMagnitudeNode(g, g_x, g_y, gr);
18  vxThresholdNode(g, gr, threshold, out);
19  /*verify the graph*/
20  status = vxVerifyGraph(g);
21  /*execute the graph*/
22  if (status == VX_SUCCESS)
23  status = vxProcessGraph(g);
24
```

Listing 1. OpenVX code of the example of Fig. 1

application graph can be be executed across different hardware accelerators (e.g., CPU cores, GPUs, DSPs). Fig. 1 and Listing 1 give an example of computer vision application and its OpenVX code, respectively. The programming flow starts by creating an OpenVX *context* to manage references to all used objects (line 1, Listing 1). Based on this context, the code builds the graph (line 2) and generates all required data objects (lines 4 to 11). Then, it instantiates the kernel as graph nodes and generates their connections (lines 15 to 18). The graph integrity and correctness is checked in line 20 (e.g., checking of data type coherence between nodes and absence of cycles). Finally, the OpenVX framework processes the graph (line 23). At the end of the code execution, it releases all created data objects, the graph, and the the context.

By adopting any vendor library that implements the graph nodes as Computer Vision primitives, OpenVX allows applying different mapping strategies between nodes and processing elements of the heterogeneous board, by targeting different design constraints (e.g., performance, power, energy efficiency).

Different works have been presented to analyse the use of OpenVX for embedded vision [4], [5], [6]. In [5], the authors present a new implementation of OpenVX targeting CPUs and GPU-based devices by leveraging different analytical optimzation techniques. In [6], the authors examine how OpenVX responds to different data access patterns, by testing three different OpenVX optimizations: kernels merge, data tiling and parallelization via OpenMP. In [4], the authors introduce ADRENALINE, a novel framework for fast prototyping and optimization of OpenVX applications for heterogeneous SoCs with many-core accelerators.

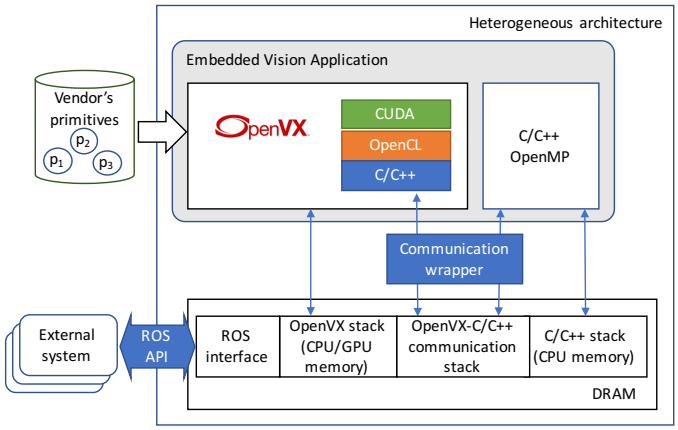

Fig. 2. Overview of the proposed framework

III. Method

Differently from all the work of the literature, the proposed framework implements an extension of the OpenvX environment to support: (i) multiple heterogeneous platforms to parallelize the software application, (ii) the customization of embedded vision applications that rely on concurrent and communicating blocks, and (iii) the application tuning through co-simulation with different simulation kernels based on the ROS API library.

Figure 2 shows the framework overview. Starting from a library of OpenVX primitives (e.g., NVIDIA VisionWorks [9], INTEL OpenVX [10], AMDOVX [11], Khronos OpenVX standard implementation [12]), developers can quickly and efficiently define a large variety of embedded vision applications. When a graph node is not implemented in the library or its implementation does not satisfy a given design requirement, the primitive can be re-implemented or optimized by the user (i.e., *user-defined primitives*) in CUDA or in OpenCL and integrated in the OpenVX environment. Then, the framework allows adopting any of the several solutions available at the state of the art to map the graph nodes to the processing elements of the board (e.g., [4], [5], [6]).

Nevertheless, OpenVX is limited to embedded vision algorithms that can be represented by a data-flow synchronous application. To extend its applicability, the proposed framework provides a communication wrapper that allows *concurrent blocks* of the application to interact and communicate with *data-flow* OpenVX blocks. Such concurrent blocks can be implemented in C/C++ and, to better exploit the multi-core processor tipically available in the hardware boards, they are parallelized through directive-based platforms (e.g., OpenMP). The main memory of the target hardware board is partitioned into different stacks, to support the concurrent execution of OpenVX, the parallel C/C++ code, and the communication between them.

The OpenVX runtime manager synchronizes the parallel execution of the data-flow synchronous blocks with those implemented in C/C++ and parallelized with OpenMP. The actual mapping between application blocks and processing

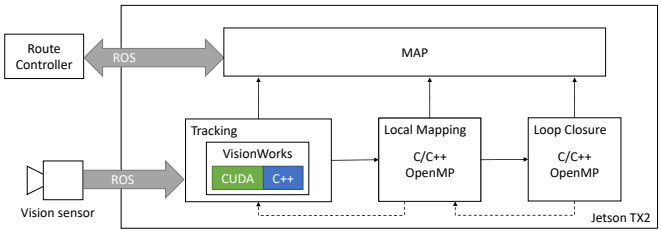

Fig. 3. Overview of ORB-SLAM implementation

elements of the board is first performed by the operating system and, with the remaining free processing units, by the OpenVX runtime manager. It is important to note that the optimization strategies adopted in OpenVX do not include the (parallel) C/C++ block implementations. As a consequence, the proposed framework allows performing a two-level mapping exploration: at system level, which includes the whole application blocks, and at OpenVX level, which is performed by the OpenVX runtime manager.

Finally, embedded vision applications require ad-hoc parametrizations by considering both the application environment (e.g., input streams, concurrent interactive systems) and the characteristics of the target platform. Thus, the framework adopts ROS to implement the communication between the application under analysis and external modules (e.g., sensors, controllers, etc.). ROS implements the messages passing among nodes by providing a *publish-subscribe* communication model. Every message sent through ROS has a *topic*, which is a unique string known to all the communication actors. The topic is assigned by the node that publishes the message and the receiving nodes subscribe to the topic.

The adoption of ROS, which relies on a message passing interface, provides different advantages. First, it allows the platform to model and simulate blocks running on different target devices. Then, it implements the inter-node communication in a modular way, thus guaranteeing the code portability. It also implements the inter-node communication by adopting a standard and widespread protocol. Finally, it implements the inter-node communication with minimum intervention or modifications to the original code of the embedded application.

IV. Experimental results

The proposed framework has been applied to customize the ORB-SLAM application [8]. ORB-SLAM aims at computing, in real-time, the camera trajectory and a sparse 3D reconstruction of the scene in a wide variety of environments, ranging from small hand-held sequences of a desk to a car driven around several city blocks. It aims at closing large loops and performing global re-localisation from wide baselines.

The application consists of three main blocks (see Figure 3). The *tracking* block has a twofold task: it updates the agent localization in the environment and it detects any significant discrepancies of the map w.r.t. the input stream. The *mapping* block updates the internal state of the algorithm and of the generated map by comparing new and already read frames.

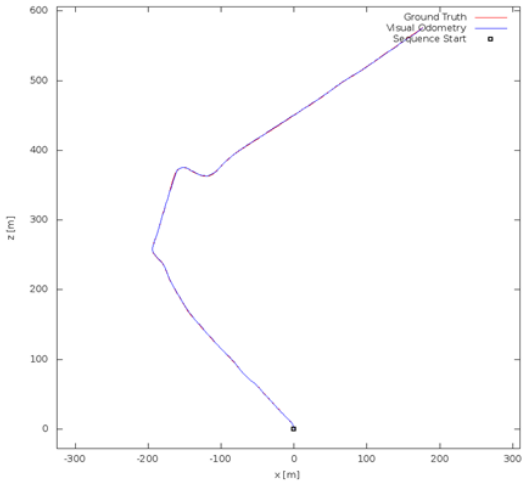

Fig. 4. KITTI sequence 11

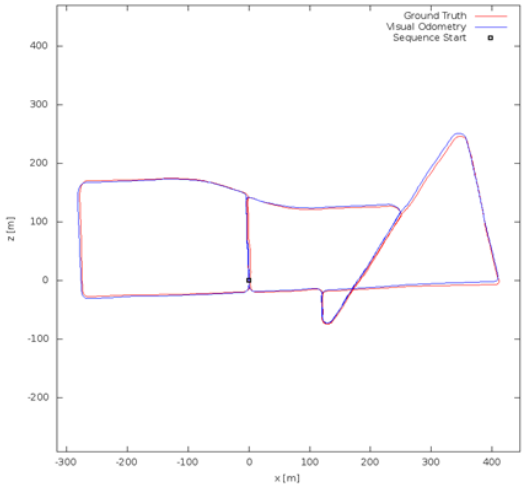

Fig. 5. KITTI sequence 13

The *loop closing* block aims at adjusting the scale drifting during the input analysis, which is unavoidable when using a monocular vision system.

The application receives the input stream by a camera thorough ROS. We tested the application on the KITTI dataset [15], which is a standard set of benchmarks for computer vision applications. For the sake of space, we present only the results obtained with the KITTI sequences 11 and 13 (10 frames per second -FPS- each). They have been chosen as they represent inputs with different workload on the three blocks. In particular, sequence 11 mostly relies on the tracking block while it never uses the mapping and loop closure blocks (see Fig. 4). In contrast, sequence 13 also runs the other two *computing intensive* blocks each time the trajectory returns on an already visited point in the map (see Fig. 5).

The application processes the stream and generates the map of the visited external environment. A route controller, which is run on an external board, queries the map generated by the ORB-SLAM application to elaborate trajectories toward a target position.

The application has been customized and run on an NVIDIA Jetson TX2 board. Such an heterogeneous low-power embedded system is equipped with a quad-core ARM CPU, and a 256 CUDA cores Pascal GPU.

Tables I and II report the obtained results in terms of number of CPU cores used by the application blocks, whether the GPU has been used by any application block, the average time required for processing one frame, the corresponding performance i.e., the maximum FPS, the average time required by the mapping phase, the total energy consumption for the whole stream analysis, the peak power of the board, and the result accuracy. The result accuracy expresses how many frames the application has been able to elaborate over the whole sequence and, among them, how many frames gave correct mapping information. For example, 54/675 accuracy with sequence 1 means that 675 over 921 frames have been elaborated and only 54 of them were useful. The result accuracy is exponentially related to the supported FPS.

Performance information has been collected through the CUDA runtime API to measure the execution time and through the `clock64()` device instruction for throughput values to ensure clock-cycle accuracy of time measurements. Power and energy consumption information have been collected through the *Powermon2* power monitoring device [13].

All these information are reported for four different versions of ORB-SLAM we developed and analysed. The first version is the original code retrieved from [8], which is run on a single core of the CPU. The second version is the original code with the multi-threading feature enabled (i.e., each block is mapped on a corresponding thread and run on a different CPU core). The third version is the original code enriched with OpenMP directives. In this version, the mapping and loop closure computations are parallelized when the map exceeds a given threshold and requires more computational power to be processed. The fourth version consists of the original code, in which the feature detection and ORB algorithms of the tracking block have been implemented by using OpenVX and CUDA, respectively. We adopted the NVIDIA VisionWorks primitive library for implementing the OpenVX blocks. We parallelized the CUDA code of ORB from scratch.

The table underlines that different customizations can be considered starting from the same applications, and that the best version depends on the selected design constraints and on the input stream. Considering sequence 11 and targeting performance, version OpenVX that adds the GPU computation to the multithreaded original version provides the best frame per second at the cost of the highest peak power. In this context, OpenMP does not provide any benefit as it is never applied (mapping and loop closing phases almost never run in seq. 11) while it introduces overhead. In contrast, OpenVX+OpenMP provides the best performance with sequence 13, in which the two phases are efficiently parallelized in the CPU cores. If the application is run in an energy-saving context, and considering the characteristics of the input stream (10 FPS), the multi-threaded version limited to three CPU cores provides the best energy efficiency and the most limited power consumption.

TABLE I
RESULTS WITH SEQUENCE 11 (921 FRAMES)

ORB-SLAM Version	#CPU cores	GPU us-age	Avg. time per frame (ms)	Avg. FPS	Avg map (ms)	Total En-ergy (J)	Peak power (W)	Accu-racy (frames/ frames)
Sequent.	1	N	133	7,5	335	488	9.4	54/675 (8%)
Multith.	3	N	105	9.5	250	534	9.5	586/837 (71%)
OpenMP	4	N	111	9.0	240	519	11.0	392/803 (49%)
OpenVX	4	Y	68	14,7	288	599	13,2	894/916 (98%)
OpenVX+ OpenMP	4	Y	70	14,3	292	601	13,5	894/916 (98%)

TABLE II
RESULTS WITH SEQUENCE 13 (3,281 FRAMES)

ORB-SLAM Version	#CPU cores	GPU us-age	Avg. time per frame (ms)	Avg. FPS	Avg map (ms)	Total En-ergy (J)	Peak power (W)	Accu-racy (frames/ frames)
Sequent.	1	N	137	7.3	336	1,758	9.8	95/2,336 (5%)
Multith.	3	N	122	8.2	338	1,815	9.9	750/2,617 (29%)
OpenMP	4	N	111	9.0	240	519	11.0	392/803 (49%)
OpenVX	4	Y	124	8.0	447	1,814	10.3	397/2,576 (16%)
OpenVX+ OpenMP	4	Y	83	12,1	320	2116	13,4	1,904/3,269 (59%)

This is due to the fact that this version, for both sequence 11 and 13 provides the maximunm FPS most similar to the FPS of the streams read in input.

V. CONCLUSION

This paper presented a framework for the design and simulation of embedded video applications that integrates OpenVX with ROS. Two contributions have been underlined: First, the framework extends the OpenVX environment to support the customization of embedded applications that rely on concurrent and communicating blocks. Second, the extended OpenVX environment has been integrated with ROS, in order to simulate and validate embedded vision applications by considering the environment in which they are embedded. The paper presented the results obtained by applying the framework to explore different configurations of ORB-SLAM for the NVIDIA Jetson TX2 by considering the performance, power, and energy consumption design constraints.

REFERENCES

[1] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Realtime computer vision with OpenCV," vol. 10, no. 4, 2012.

[2] E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill, "Addressing system-level optimization with OpenVX graphs," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 658–663.

[3] Khronos Group, "OpenVX: Portable, Power-efficient Vision Processing," https://www.khronos.org/openvx.

[4] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, "Adrenaline: An openvx environment to optimize embedded vision applications on many-core accelerators," in *International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, 2015, pp. 289–296.

[5] K. Yang, G. A. Elliott, and J. H. Anderson, "Analysis for supporting real-time computer vision workloads using openvx on multicore+gpu platforms," in *Proceedings of the 23rd International Conference on Real Time and Networks Systems*, ser. RTNS '15, 2015, pp. 77–86.

[6] D. Dekkiche, B. Vincke, and A. Merigot, "Investigation and performance analysis of openvx optimizations on computer vision applications," in *14th International Conference on Control, Automation, Robotics and Vision*, 2016, pp. 1–6.

[7] Open Source Robotics Foundation, "Robot Operating System," http://www.ros.org/.

[8] R. Mur-Artal, J. M. M. Montiel, and J. D. Tards, "Orb-slam: A versatile and accurate monocular slam system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, Oct 2015.

[9] NVIDIA Inc., "VisionWorks," https://developer.nvidia.com/embedded/visionworks.

[10] INTEL, "Intel Computer Vision SDK," https://software.intel.com/en-us/computer-vision-sdk.

[11] AMD, "AMD OpenVX - AMDOVX," http://gpuopen.com/compute-product/amd-openvx/.

[12] Khronos, "OpenVX lib," https://www.khronos.org/openvx.

[13] D. Bedard, M. Y. Lim, R. Fowler, and A. Porterfield, "Powermon: Fine-grained and integrated power monitoring for commodity computer systems," in *Proc. of IEEE SoutheastCon*, 2010, pp. 479–484.

[14] ——, "Powermon: Fine-grained and integrated power monitoring for commodity computer systems," in *Proceedings of the IEEE Southeast-Con*, 2010, pp. 479–484.

[15] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013.