Integrating Simulink, OpenVX, and ROS for Model-Based Design of Embedded Vision Applications

Stefano Aldegheri and Nicola Bombieri(✉)

Department of Computer Science, University of Verona, Verona, Italy
{stefano.aldegheri,nicola.bombieri}@univr.it

Abstract. OpenVX is increasingly gaining consensus as standard platform to develop portable, optimized and power-efficient embedded vision applications. Nevertheless, adopting OpenVX for rapid prototyping, early algorithm parametrization and validation of complex embedded applications is a very challenging task. This paper presents a comprehensive framework that integrates Simulink, OpenVX, and ROS for model-based design of embedded vision applications. The framework allows applying Matlab-Simulink for the model-based design, parametrization, and validation of computer vision applications. Then, it allows for the automatic synthesis of the application model into an OpenVX description for the hardware and constraints-aware application tuning. Finally, the methodology allows integrating the OpenVX application with Robot Operating System (ROS), which is the de-facto reference standard for developing robotic software applications. The OpenVX-ROS interface allows co-simulating and parametrizing the application by considering the actual robotic environment and the application reuse in any ROS-compliant system. Experimental results have been conducted with two real case studies: An application for digital image stabilization and the ORB descriptor for simultaneous localization and mapping (SLAM), which have been developed through Simulink and, then, automatically synthesized into OpenVX-VisionWorks code for an NVIDIA Jetson TX2 board.

1 Introduction

Computer vision has gained an increasing interest as an efficient way to automatically extract of meaning from images and video. It has been an active field of research for decades, but until recently has had few major commercial applications. However, with the advent of high-performance, low-cost, energy efficient processors, it has quickly become largely applied in a wide range of applications for embedded systems [1].

The term *embedded vision* refers to this new wave of widely deployed, practical computer vision applications properly optimized for a target embedded

Published by Springer Nature Switzerland AG 2019
M. Maniatakos et al. (Eds.): VLSI-SoC 2017, IFIP AICT 500, pp. 178–197, 2019.
https://doi.org/10.1007/978-3-030-15663-3_9

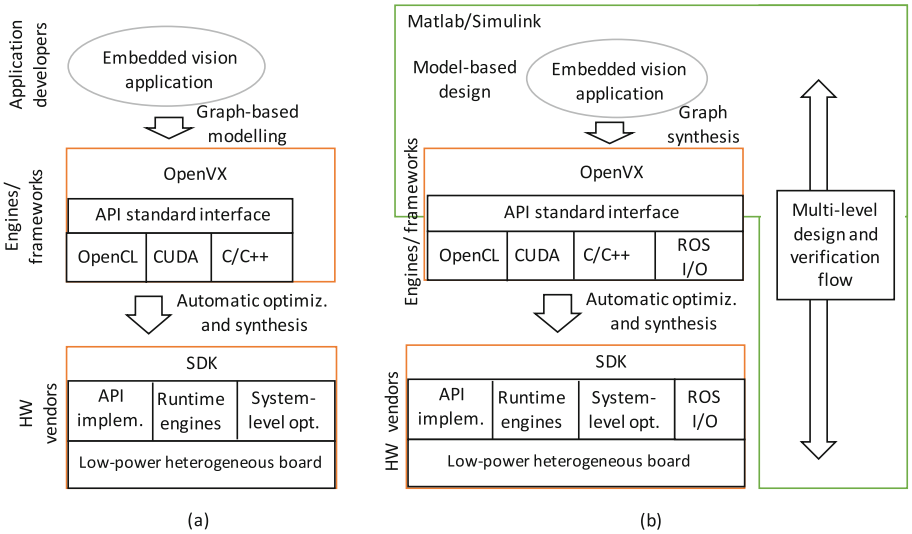

Fig. 1. The embedded vision application design flow: the standard (a), and the extended with the model-based design paradigm (b)

system by considering a set of design constraints. The target embedded systems usually consist of heterogeneous, multi-/many-core, low power embedded devices, while the design constraints, beside functional correctness, include performance, energy efficiency, dependability, real-time response, resiliency, fault tolerance, and certifiability.

Developing and optimizing a computer vision application for an embedded processor can be a non-trivial task. Considering an application as a set of communicating and interacting kernels, the effort for such application optimization goes over two dimensions: the single kernel-level optimization and the system-level optimization. Kernel-level optimizations have traditionally revolved around one-off or single function acceleration. This typically means that a developer rewrites a computer vision function (e.g., any filter, image arithmetic, geometric transform function) with a more efficient algorithm or offloads its execution to accelerators such as a GPU by using languages such as OpenCL or CUDA [2].

On the other hand, system-level optimizations pay close attention to the overall power consumption, memory bandwidth loading, low-latency functional computing, and Inter-Processor Communication overhead. These issues are typically addressed via frameworks [3], as the parameters of interest cannot be tuned with compilers or operating systems.

In this context, OpenVX [4] has gained wide consensus in the embedded vision community and has become the de-facto reference standard and API library for system-level optimization. OpenVX is designed to maximize functional and performance portability across different hardware platforms, providing a computer vision framework that efficiently addresses current and future 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 HW board targeting performance and power consumption design constraints [5–7].

Nevertheless, the definition of such a graph-based model, its parametrization and validation is time consuming and far from intuitive to programmers, especially for the development of medium-complex applications.

Embedded vision finds a large use in the context of Robotics, where cameras are mounted on robots and the results of the embedded vision applications are analysed for autonomous actions. Indeed, Computer vision allows robots to see what is around them and make decisions based on what they perceive. In this context, Robot Operating System (ROS) [8] has been proposed as a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms. It is become a de-facto reference standard for developing robotic applications. It allows for application re-use and easy integration of software blocks in complex systems.

This paper presents a comprehensive framework that integrates Simulink, OpenVX, and ROS for the model-based design of embedded vision applications (see Fig. 1). Differently from the standard approaches at the state of the art that require designers to manually model the algorithm through OpenVX code (see Fig. 1(a)), the proposed approach allows for a rapid prototyping, algorithm validation and parametrization in a model-based design environment (i.e., Matlab/Simulink). The framework relies on a multi-level design and verification flow (see Fig. 1(b)) by which the high-level model is then semi-automatically refined towards the final automatic synthesis into OpenVX code. The integration with ROS has two main goals: first, to allow co-simulating and parametrizing the application by considering the actual robotic environment, and then, to allow for application reuse in ROS-compliant systems.

The paper presents the results obtained by applying the proposed methodology for developing and tuning two real-case applications. The first is an algorithm for digital image stabilization for two different application contexts. The second is the application implementing the oriented fast and rotated brief (ORB) descriptor for simultaneous localization and mapping (SLAM). The paper presents the Simulink toolbox developed to support the NVIDIA OpenVX-VisionWorks library, and how it has been used in the design flow to synthesize OpenVX code for an NVIDIA Jetson TX2 embedded system board.

The paper is organized as follows. Section 2 presents the background and the related work. Section 3 explains in details the model-based design methodology. Section 4 presents the experimental results, while Sect. 5 is devoted to the conclusions.

2 Background and Related Work

OpenVX relies on a graph-based software architecture to enable efficient computation on heterogeneous computing platforms, including those with GPU accelerators. It provides a set of primitives (or kernels) that are commonly used in

computer vision algorithms. It also provides a set of data objects like scalars, arrays, matrices and images, as well as high-level data objects like histograms, image pyramids, and look-up tables. It supports customized user-defined kernels for implementing customized application features.

The programmer defines a computer vision algorithm by instantiating kernels as nodes and data objects as parameters. Since each node may use the mix of the processing units in the heterogeneous platform, a single graph may be executed across CPUs, GPUs, DSPs, etc. Figure 2 and Listing 1.1 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 1.1). 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 graph is processed by the OpenVX framework (line 23). At the end of the code execution, all created data objects, the graph, and the context are released.

The definition of algorithms through primitives has two benefits: First, it allows defining the application in an abstract way while preserving an efficient implementation. Then, it allows enabling system-level optimizations, like inter-node memory transfers, pipelining, concurrent and overlapped node execution. To utilize the different accelerators on the board, data transfer management needs to be addressed. Each operation requires time and power resources, and this has to be considered in the mapping process. Pipelining and tiling techniques can be efficiently utilized together to achieve better memory locality. This greatly reduces the data transfer overhead between global and scratchpad memory [9].

Different works have been presented to analyse the use of OpenVX for embedded vision [5–7,11]. In [6], the authors present a new implementation of OpenVX targeting CPUs and GPU-based devices by leveraging different analytical optimization techniques. In [7], 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 [5], the authors introduce ADRENALINE, a novel framework for fast prototyping and optimization of OpenVX applications for heterogeneous SoCs with many-core accelerators. The authors in [10] implemented a graphic interface that allows computer vision developers to create visual algorithms in OpenVX. The framework then automatically generates the corresponding OpenVX code, with a translation back-end that creates all the glue code needed to correctly run the OpenVX environment. This work extends the preliminary implementation of the model-based design presented in [11] by including the interface towards ROS. This allows co-simulating the OpenVX application with the external application environment (e.g., input streams, concurrent interactive systems, etc.) and, as a consequence, tuning more efficiently the SW parametrization. Results on a more advanced Robotic application (ORB descriptor) underlines that making any application ROS-compliant is strategic for IP-reuse.

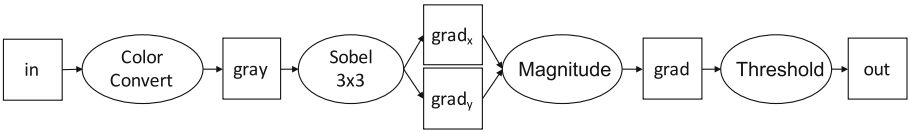

Fig. 2. OpenVX sample application (graph diagram)

```
vx_context c = vxCreateContext();
vx_graph g = vxCreateGraph(context);
vx_enum type = VX_DF_IMAGE_VIRT;
/* create data structures */
vx_image in = vxCreateImage(c, w, h, VX_DF_IMAGE_RGBX);
vx_image gray   = vxCreateVirtualImage(g, 0, 0, type);
vx_image grad_x = vxCreateVirtualImage(g, 0, 0, type);
vx_image grad_y = vxCreateVirtualImage(g, 0, 0, type);
vx_image grad   = vxCreateVirtualImage(g, 0, 0, type);
vx_image out = vxCreateImage(c, w, h, VX_DF_IMAGE_U8);
vx_threshold threshold = vxCreateThreshold(c, VX_THRESHOLD_TYPE_BINARY,
     VX_TYPE_FLOAT32);
/* read input image and copy it into "in" data object */
...
/* construct the graph */
vxColorConvertNode(g, in, gray);
vxSobel3x3Node(g, gray, grad_x, grad_y);
vxMagnitudeNode(g, grad_x, grad_y, grad);
vxThresholdNode(g, grad, threshold, out);
/*verify the graph*/
status = vxVerifyGraph(g);
/*execute the graph*/
if (status == VX_SUCCESS)
  status = vxProcessGraph(g);
```

Listing 1.1. OpenVX code of the example of Fig. 2

Differently from all the work of the literature, this paper presents an extension of the OpenvX environment to the model-based design paradigm. Such an extension aims at exploiting the model-based approach for the fast prototyping of any computer vision algorithm through a Matlab/Simulink model, its parametrization, validation, and automatic synthesis into an equivalent OpenVX code representation.

3 The Model-Based Design Approach

Figure 3 depicts the overview of the proposed design flow. The computer vision application is firstly developed in Matlab/Simulink, by exploiting a computer vision oriented toolbox of Simulink[1]. Such a block library allows developers to define the application algorithms through Simulink blocks and to quickly simulate and validate the application at system level. The platform allows specific

[1] In this work, we selected the *Simulink Computer Vision* toolbox (CVT), as it represents the most widespread and used toolbox in the computer vision community. The methodology, however, is general and can be extended to other Simulink toolboxes.

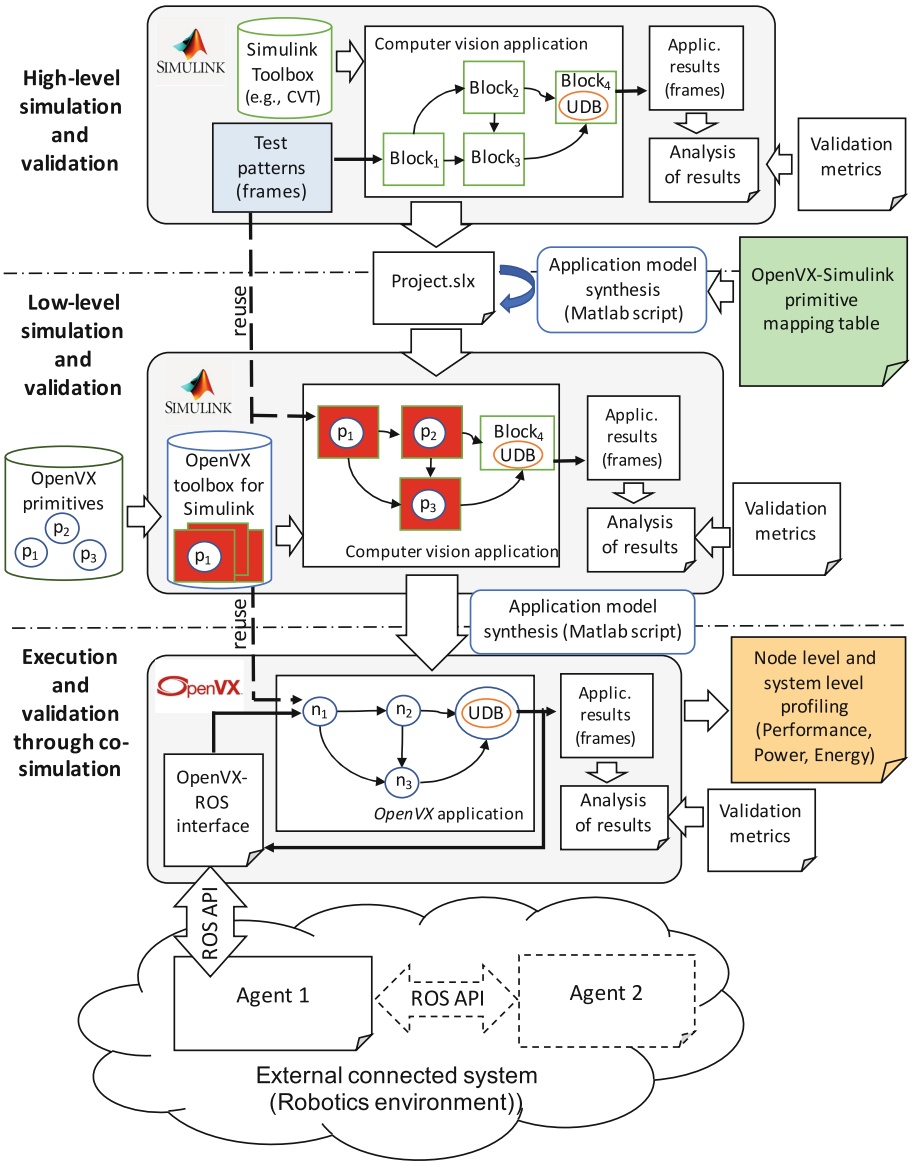

Fig. 3. Methodology overview

and embedded application primitives to be defined by the user if not included in the toolbox through the Simulink *S-Function* construct [12] (e.g., user-defined block UDB *Block*$_4$ in Fig. 3). Streams of frames are given as input stimuli to the application model and the results (generally represented by frames or streams of frames) are evaluated by adopting any ad-hoc validation metrics from the

computer vision literature (e.g., [13]). Efficient test patterns are extrapolated, by using any technique of the literature, to assess the quality of the application results by considering the adopted validation metrics.

The high-level application model is then automatically synthesized for a low-level simulation and validation through Matlab/Simulink. Such a simulation aims at validating the computer vision application at system-level by using the OpenVX primitive implementations provided by the HW board vendor (e.g., NVIDIA VisionWorks) instead of Simulink blocks. The synthesis, which is performed through e Matlab routine, relies on two key components:

1. *The OpenVX toolbox for Simulink.* Starting from the library of OpenVX primitives (e.g., NVIDIA VisionWorks [14], INTEL OpenVX [15], AMDOVX [16], Khronos OpenVX standard implementation [17]), such a toolbox of blocks for Simulink is created by properly wrapping the primitives through Matlab *S-Function*, as explained in Sect. 3.1.
2. *The OpenVX primitives-Simulink blocks mapping table.* It provides the mapping between Simulink blocks and the functionally equivalent OpenVX primitives, as explained in Sect. 3.2.

As explained in the experimental results, we created the OpenVX toolbox for Simulink of the NVIDIA VisionWorks library as well as the mapping table between VisionWorks primitives and Simulink CVT blocks. They are available for download from https://profs.sci.univr.it/bombieri/VW4Sim.

The low-level representation allows simulating and validating the model by reusing the test patterns and the validation metrics identified during the higher level (and faster) simulation.

The low-level Simulink model is synthesized, through a Matlab script, into an OpenVX model, which is executed and validated on the target embedded board. At this level, all the techniques of the literature for OpenVX system-level optimization can be applied. The synthesis is straightforward (and thus not addressed in this paper for the sake of space), as all the key information required to build a stand-alone OpenVX code is contained in the low-level Simulink model. Both the test patterns and the validation metrics can be re-used for the node-level and system-level optimization of the OpenVX application.

The proposed design flow also allows the embedded vision application to be refined by considering the external Robotics system, which is supposed to be implemented as ROS-compliant application. The OpenVX model is interfaced to ROS through a set of interface templates, which implement the OpenV-ROS communication based on message passing. A lightweight *target* I/O module is responsible to handle the information sent by the system (e.g., sensors, controllers, etc.) and to translate it into an OpenVX data structure. A similar I/O module implements the *initiator* interface, which allows sending information from OpenVX (generally the results of a computation) to the ROS system. By relying on the ROS communication protocol, the embedded vision application can easily interact with multiple external actors, allowing an easy integration and reuse into real Robotics systems.

```
1  function s_colorConvert(block)
2    setup(block);
3
4    function setup(block)
5      % Number of ports and parameters
6      block.NumInputPorts  = 1;
7      block.NumOutputPorts = 1;
8
9      block.RegBlockMethod('Start', @Begin);
10     block.RegBlockMethod('Stop', @End);
11     block.RegBlockMethod('Outputs', @Outputs);
12   function begin(block)
13     %create vx_image
14     gray = m_vxCreateImage();
15   function end(block)
16     %destroy vx_image
17     m_vxReleaseImage(gray);
18   function outputs(block) //computation phase:
19     in = block.InputPort(1).Data;
20     ret_val = m_vxColorConvert(in, gray);
21     block.OutputPort(1).Data = gray;
```

Listing 1.2. Matlab S-function code for the color converter node

3.1 OpenVX Toolbox for Simulink

The generation of the OpenVX toolbox for Simulink relies on the *S-function* construct, which allows describing any Simulink block functionality through C/C++ code. The code is compiled as *mex file* by using the Matlab *mex utility* [18]. As with other *mex* files, *S-functions* are dynamically linked subroutines that the Matlab execution engine can automatically load and execute. *S-functions* use a special calling syntax (i.e., *S-function API*) that enables the interaction between the block and the Simulink engine. This interaction is very similar to the interaction that takes place between the engine and built-in Simulink blocks.

We defined a *S-function* template to build OpenVX blocks for Simulink that, as for the construct specifications, consists of four main phases (see the example in Listing 1.2, which represents the *Color Converter* node of Fig. 2):

- *Setup phase* (lines 4–11): it defines the I/O block interface in terms of number of input and output ports and the block internal state (e.g., point list for tracking primitives).
- *Begin phase* (lines 12–14): It allocates data structure in the Simulink memory space for saving the results of the block execution. Since the block executes OpenVX code, this phase implementation relies on a *data wrapper* for the OpenVX-Simulink data exchange and conversion.
- *End phase* (lines 15–17): It deallocates the created data structures at the end of the simulation (after the computation phase).
- *Computation phase* (lines 18–20): it reads the input data and executes the code implementing the block functionality. It makes use of a *primitive wrapper* to execute OpenVX code.

Three different wrappers have been defined to allow communication and synchronization between the Simulink and the OpenVX environments. They are

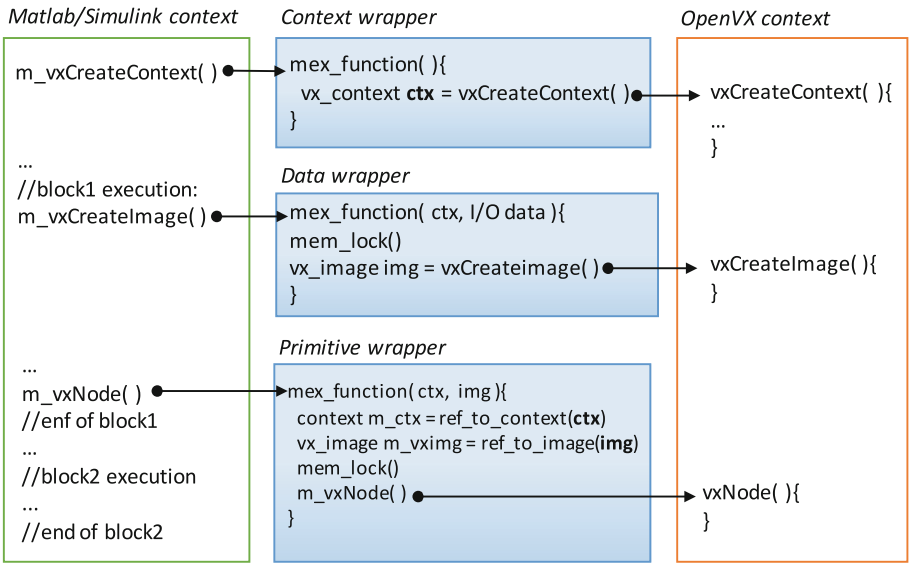

Fig. 4. Overview of the Simulink-OpenVX communication

summarized in Fig. 4. The *context wrapper* allows creating the OpenVX context (see line 1 of Listing 1.1), which is mandatory for any OpenVX primitive execution. It is run once for the whole system application. The *data wrapper* allows creating the OpenVX data structures for the primitive communication (see *in*, *gray*, $grad_x$, $grad_y$, *grad*, and *out* in the example of Fig. 2 and lines 4–11 of Listing 1.1). It is run once for each application block. The *primitive wrapper* allows executing, in the Simulink context, each primitive functionality implemented in OpenVX. To speed up the simulation, the wrapped primitives work through references to data structures, which are passed as function parameters during the primitive invocations to the OpenVX context. To do that, the wrappers implement memory locking mechanisms (i.e., through the Matlab *mem_lock()/mem_unlock()* constructs) to prevent data objects to be released automatically by the Matlab engine between primitive invocations.

3.2 Mapping Table Between OpenVX Primitives and Simulink Blocks

To enable the application model synthesis from the high-level to the low-level representation, mapping information is required to put in correspondence the built-in Simulink blocks and the corresponding OpenVX primitives. In this work, we defined such a mapping table between the Simulink CVT Toolbox and the NVIDIA OpenVX-VisionWorks library. The table, which consists of 58 entries in the current release, includes primitives for image arithmetic, flow and depth, geometric transforms, filters, feature and analysis operations. Table 1 shows, as an example, a representative subset of the mapped entries.

Table 1. Representative subset of the mapping table between Simulink CVT and NVIDIA OpenVX-VisionWorks

Simulink block	Visionworks primitive	Notes to the developer
CVT/AnalysisAnd -Enhancement/ EdgeDetection	vxuCannyEdgeDetector	If Simulink EdgeDetection set as Canny
CVT/AnalysisAnd -Enhancement/ EdgeDetection	vxuSobel3x3	If Simulink EdgeDetection set as Sobel
CVT/AnalysisAnd -Enhancement/ EdgeDetection	vxuConvolve	If filter size different from 3x3
CVT/Morphological operation/Opening	vxuErode3x3 + vxuDilate3x3	
CVT/Filtering/Median Filter	vxuMedianFilter3x3	
CVT/Filtering/Median Filter	vxuNonLinearFilter	If filter size different from 3x3
Math Op./Subtract + Math Op./Abs	vxuAbsoluteDifference	
CVT/Conversion/Color space conversion	vxuColorConvert	
CVT/Statistics/2D Mean	vxuMeanStdDev	Only mean and standard deviation of the entire image supported
CVT/Statistics/2D StandardDev		
Simulink/Math operations/Real/ComplexTo -Imag	vxuMagnitude	Gradient magnitude computed through complex numbers
Simulink/Math operations/Real/Imag to Magnitude		

We implemented three possible mapping strategies:

1. 1-to-1: the Simulink block is mapped to a single OpenVX primitive (e.g., color converter image arithmetic).
2. 1-to-n: the Simulink block functionality is implemented by a concatenation of multiple OpenVX primitives (e.g., the opening morphological operation).
3. n-to-1: a concatenation of multiple Simulink blocks are needed to implement a single OpenVX primitive (e.g., subtract + absolute blocks).

For some entry, the mapping also depends on the Simulink block setting. As an example, the OpenVX primitive for edge detection is selected depending on the setting of the corresponding CVT block. The setting includes the choice of the filter algorithm (i.e., Canny or Sobel) and the filter size.

The blocks listed in the left-most column of the table form the OpenVX toolbox for Simulink. Any Simulink model built from them can undergo the proposed automatic refinement flow. In addition, user-defined Simulink blocks implemented in C/C++ are supported and translated into OpenVX user kernels. They are eventually loaded and included in the OpenVX representation as graph nodes. To do that, we defined the wrapper represented in Listing 1.3, which follows the node implementation directives required by the standard OpenVX for importing user kernels[2]. The wrapper invocation (i.e., $vx\_userNode()$) is

[2] www.khronos.org/registry/openVX/specs/1.0/html/da/d83/group__group__user__kernels.html.

```
vx_userNode(){
  vx_status processingOpenVX(vx_node node, const vx_reference *parameters,
      vx_uint32 num)
  {
    //convert data in internal representation
    SimulinkBlockFunctionality(); //C/C++ code of the UDB functionality
    return VX_SUCCESS;
  }
  vx_status validationOpenVX(vx_node node, const vx_reference parameters
     [], vx_uint32 num,   vx_meta_format metas[]))
  {
    //insert parameter validation
    return VX_SUCCESS;
 }

  vx_status singleShotProcessing(vx_context context, parameters)
  {
    //create graph and execute it
  }

  vx_status registerCustomKernel(vx_context context)
  {
    vx_status = vxAddUserKernel(context, ...);//register kernel in context
    return VX_SUCCESS;
  }
}
```

Listing 1.3. Overview of wrapper for user-defined Simulink block implementations

similar to the invocation of any built-in OpenVX node (i.e., $vxNode()$) in the OpenVX context through the previously presented *context wrapper* (see the rightmost side of Fig. 4).

Finally, some restrictions on the Simulink block interfaces are required to allow the Simulink/OpenVX communication as well as the model synthesis. The set of data types and data structures available for the high-level model is reduced to the subset supported by OpenVX, whereby each I/O port of the Simulink blocks consists of:

- *Dimension* $d \in \{1D, 2D, 3D, 3D + AlphaChannel\}$, e.g., greyscale, RGB or YUV, and alpha channel for transparency.
- *Size* $s \in \{N \times M \times 1, N \times M \times 3, N \times M \times 4\}$.
- *Type* $t \in \{uint8, float\}$, where $uint8$ is generally used for representing data (pixels, colours, etc.) while $float$ is generally used for representing interpolation data.

3.3 ROS Integration

The adoption of ROS 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 and by adopting a standard and widespread protocol, thus guaranteeing code portability.

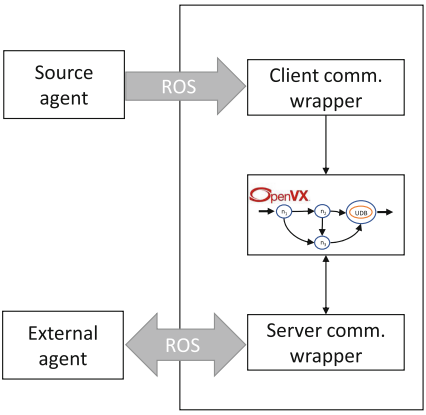

Fig. 5. The OpenVX-ROS communication through the server and client models

```
bool function_service(
    ServerType::Request  &req,
    ServerType::Response &res  )
{
  // compute the results
  res.output1 = openvx2ros(
    wrapper_openvx
      (ros2openvx(req.input1),
       ros2openvx(req.input2))
      );
  if(errors) return false;
  else       return true;
}

int process_init()
{
  ros::init(0, [],
        "service_server");
  ros::NodeHandle n;
  // Inform that this server is up
  ros::ServiceServer service =
      n.advertiseService(
        "topic_service",
        function_service
      );
  ros::spin();
}
```

(a)

```
int process_init()
{
  ros::init(0, [],
        "service_client");
  ros::NodeHandle n;

  ros::ServiceClient client =
      n.serviceClient<ServerType>(
        "topic_service"
      );
  ros::Publisher pub =
    n.advertise<DataType>("
     topic_out", 10);
  ServerType srv;
  //fill input data(opt. parameter)
  srv.req.input1 = ...;
  if (client.call(srv))
  {
    // processing went good
    n.publish(
      openvx2ros(wrapper_openvx(
        ros2openvx(srv.res.output1)
  )));
  }
  else
  {
    // processing fails
  }
}
```

(b)

Fig. 6. Skeleton implementation for the (a) server model and the (b) client model

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 by all the communicating nodes. An initiator node assigns a topic to publish a message, and the receiving nodes subscribe to the topic.

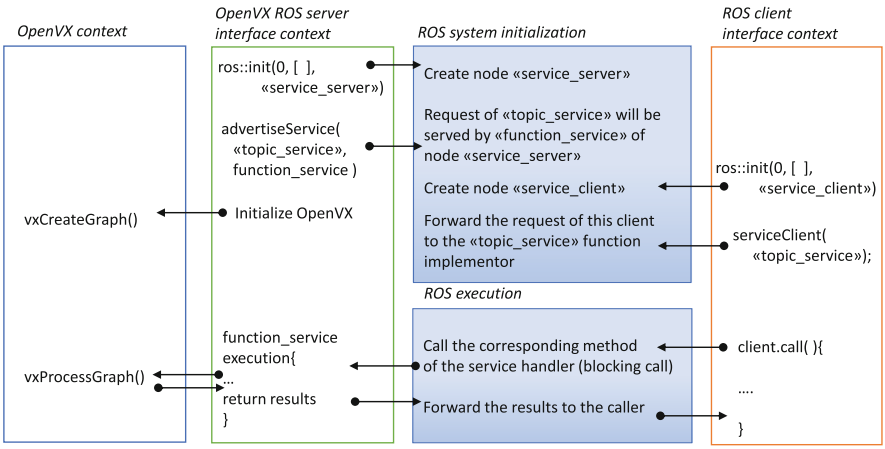

Fig. 7. Server model time evolution

Based on such a message passing interface, the proposed design flow relies on two communication models:

- *Client model:* The OpenVX application actively fetches inputs from a particular ROS node. It relies on a client communication wrapper, as shown in the upper side of Fig. 5. It is particularly suited for intensive yet synchronous communication (e.g., data acquisition of the OpenVX application from an input sensor).
- *Server model:* It allows the OpenVX application to be run on-demand. The external environment, which is implemented as ROS node, sends an execution request through an input data structure. The OpenVX application executes and returns the result as a response packet. It relies on a server communication wrapper, as shown in the bottom side of Fig. 5. It is well suited to implement sporadic communication (e.g., interpretation of the map built by a SLAM application by an external agent).

Figure 6(a) shows the skeleton implementation of the server interface. The *process_init* function is responsible to perform the node initialization in the ROS framework. It adds the current process to the ROS node list in the master server (lines 17–18). This node is sensitive to the topic specified in line 23. Line 24 specifies the function that will be called on the server invocation. Lines 1–13 provide the invocation of the OpenVX application. Two parameters are necessary to the function: the request, which contains the input data, and the response, which will be updated by the computing function. Conversion functions are defined to convert the data format between ROS and OpenVX. Finally, line 26 implements the busy waiting until the ROS framework shuts down all the nodes. Figure 7 shows the temporal evolution of the OpenVX-ROS communication based on the server model of Fig. 6(a).

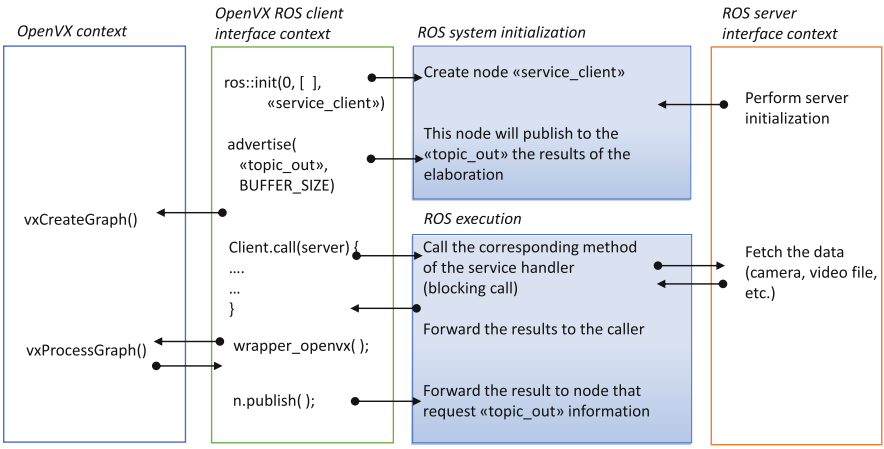

Fig. 8. Client model time evolution

Figure 6(b) depicts the skeleton for the client interface. After adding the process to the list of ROS nodes (lines 3–4), the system informs the ROS framework that the client requests need to be forwarded to the *topic_service* listener (lines 7–10). The wrapper creates the object to write the results of the computation (lines 11–12). Parameters are filled in line 15, and the call to request data is performed in line 16. In case of positive message receiving, the OpenVX computation is called (lines 20–21), through ad-hoc functions to convert the data format between ROS and OpenVX. The system publishes the results back to the network (line 19). Figure 8 shows the temporal evolution of such a client model process.

4 Experimental Results

We applied the proposed model-based design flow for the development of two embedded software: the first one implements a digital image stabilization algorithm for camera streams, while the latter calculate the ORB descriptor.

4.1 Image Stabilization

Figure 9 shows an overview of the algorithm, which is represented through a dependency graph. The input stream (i.e., sequence of frames) is taken from a high-definition camera, and each frame is converted to the grayscale format to improve the algorithm efficiency without compromising the quality of the result. A *remapping* operation is then applied to the resulting frames to remove fish-eye distortions. A sparse optical flow is applied to the points detected in the previous frame by using a feature detector (e.g., Harris or Fast detector). The resulting points are then compared to the original point to find the homography

matrix. The last N matrices are then combined by using a Gaussian filtering, where N is defined by the user (higher N means more smoothed trajectory a the cost of more latency). Finally, each frame is inversely warped to get the final result. Dashed lines in Fig. 9 denote inter-frame dependencies, i.e., parts of the algorithm where a temporal window of several frames is used to calculate the camera translation.

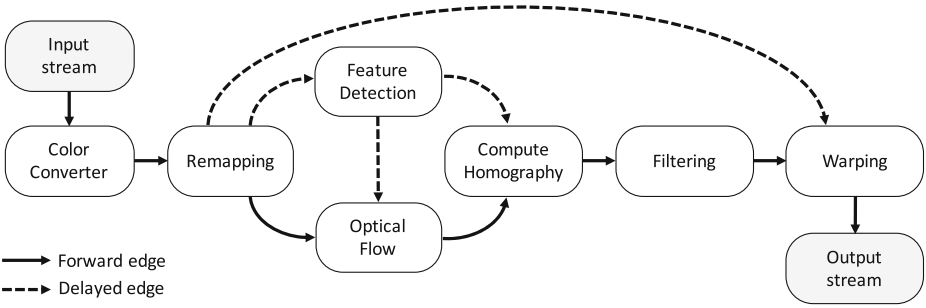

Fig. 9. Digital image stabilization algorithm

We firstly modelled the algorithm application in Simulink (CVT toolbox). The nodes *Optical flow* and *Filtering* have been inserted as user-defined blocks, since they implement customized functionality and are not present in the CVT toolbox. We conducted two different parametrizations of the algorithm, and in particular of the feature detection phase: For an indoor and for an outdoor application context. The first targets a system for indoor navigation of an Unmanned aerial vehicle (UAV), while the second targets a system for outdoor navigation of an Autonomous Surface Crafts (ASCs) [19].

We validated the two algorithm configurations starting from input streams registered by different cameras at 60 FPS with 1280×720 (1080P) and 1920×1080 wide angle resolution, respectively. Table 2 reports the characteristics of the input streams (columns *Video real time* and *#Frames*) and the time spent for simulating the high-level model on such video streams in Simulink (*Model simulation time*). Starting from the original video streams, we extrapolated a subset of test patterns, which consist of the minimal selection of video streams necessary to validate the model correctness by adopting the Smith et al. validation metrics for light field video stabilization [13]. The table reports the characteristics of such selected test patterns (sequences of frames).

We then applied the Matlab synthesis script to translate the high-level model into the low-level model by using the OpenVX toolbox for Simulink generated from the NVIDIA VisionWorks v1.6 [14] and the corresponding Simulink CVT-NVIDIA OpenVX/VisionWorks mapping table, as described in Sects. 3.1 and 3.2, respectively. In particular, the low level simulation in Simulink allowed us to validate the computer vision application implemented through the primitives

Table 2. Experimental results: High-level simulation time in Simulink

Context	Original input stream			Selected test patterns		
	Video real time (min)	Model simulation time (min)	Frames (#)	Video real time (min)	Model simulation time (min)	Frames (#)
Indoor	364	492	1.296.278	20.5	30.5	72.112
Outdoor	192	263	648.644	11.0	13.0	36.935

provided by the HW board vendor (e.g., NVIDIA OpenVX-VisionWorks) instead of Simulink blocks.

Finally, we synthesized the low-level model into pure OpenVX code, by which we run the real time analysis and validation on the target embedded board (NVIDIA Jetson TX1). Table 3 reports a comparison among the different simulation time (real execution time for the OpenVX code) spent to validate the embedded software application at each level of the design flow. At each refinement step, we reused the selected test patterns to verify the code over the adopted validation metrics [13] for both the contexts and by assuming a maximum deviation of 5%. The results underline that the higher level model simulation is faster as it mostly relies on built-in Simulink blocks. It is recommended for functional validation, algorithm parametrization, and test pattern selection. It provides all the benefits of the model-based design paradigm, while it cannot be used for accurate timing analysis, power, and energy measurements. The low level model simulation is much slower since it relies on actual primitive implementation and many wrapper invocations. However, it represents a fundamental step as it allows verifying the functional equivalence between the system-level model implemented through blocks and the system-level model implemented through primitives. Finally, the validation through execution on the target real device allows for accurate timing and power analysis, in which all the techniques at the state of the art for system-level optimization can be applied.

Table 3. Experimental results: Comparison of the simulation time spent to validate the software application at different levels of the design flow. The board level validation time refers to real execution time on the target board

Validation level	Sim./Exec. time (min)	
	Indoor	Outdoor
Simulink high-level model	30.5	13.0
Simulink low level model	59.0	26.5
Software application on target embedded system device	20.5	11.0

4.2 ORB Descriptor

In computer vision, *visual descriptors* or *image descriptors* represent visual features of image or video contents captured by an input video sensor. One of the most adopted is ORB [20], which is generally integrated in complex localization and mapping systems (i.e., ORB-SLAM [21]). The ORB-SLAM algorithm performs ORB computation at different levels, to detect both fine and coarse features of images. The inputs generally consists of a gray-scale image, the number of such levels, and the number of features to be analysed per level.

We applied the proposed model-based design flow to define the ORB algorithm, which is depicted in Fig. 10. For the sake of space, the figure shows the implementation of a single level in the Simulink environment. The data-flow oriented algorithm consists of 5 main steps: The input image is resized according to the scale level with the nearest-neighbors interpolation. The interesting points are detected by using the FAST corner detection algorithm with a specified threshold. Then, the computed keypoints are divided into a regular grid where a pruning is applied to each cell by using an higher threshold. This step is applied only to the cells where the results are non-empty (i.e., there exist at least one keypoint for each cell). The algorithm organizes the keypoints in a *quadtree data structure*, which allows achieving a uniform sampling of the keypoints in the image, carrying out the final pruning, and obtaining final keypoints, where as a result, an angle is computed (i.e., since FAST detector is not oriented). The algorithm applies a gaussian blur operation to the resized image, in order to improve the descriptor quality and to avoid artifacts that can be introduced by the nearest neighbor interpolation. Finally, the ORB descriptor is computed for each keypoint. The final coordinates of the keypoints are rescaled to the corresponding location in the original image.

Along the design flow, we measured the execution time of the algorithm implementations at different refinement steps, by using the KITTI dataset [22], which is a standard set of benchmarks for SLAM and computer vision applications. We also adopted the ROS interfaces described in Sect. 3.3 to receive the video stream (i.e., based on the slave model) and to integrate an external agent that reads the ORB result (i.e., based on the server model).

Table 4 reports the execution time we obtained by running the application on the input sequence 11 of the KITTI dataset at different refinement levels. We applied the semi-automatic translation process from Simulink to the final implementation as explained in Sect. 4.1, targeting an NVIDIA Jetson TX2 embedded board. We observed a slightly reduced execution time for the Simulink low-level model execution with respect to the high-level model despite the wrapper usage. This is due the fact that the algorithm implementation in Simulink required specialized MATLAB code that was not available with Simulink CVT library as native blocks. As a consequence, we developed custom code in MATLAB to meet the requirements, and imported such a code as user-defined Simulink blocks using the level-2 S-functions. As for the model-based design flow, the main focus of the Simulink implementation was to target the functional verification of the embedded application, with little effort on performance optimizations. On the

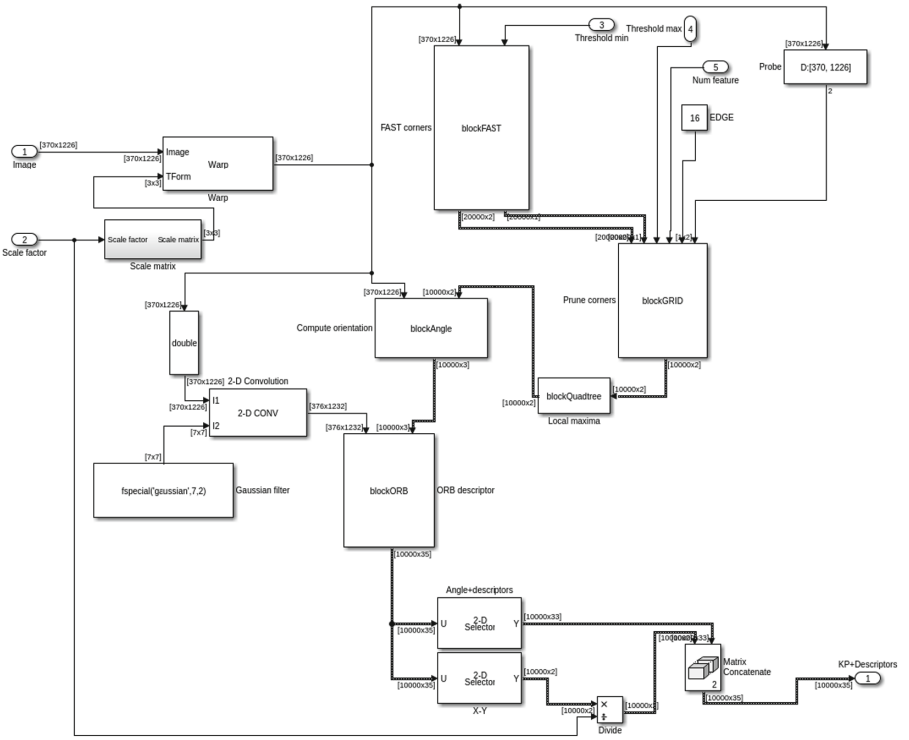

Fig. 10. ORB design in Simulink

Table 4. Experimental results: Comparison of the simulation time spent to validate the software application at different levels of the design flow. The board level validation time refers to real execution time on the target board including the ROS communication overhead

Validation level	Sim./Exec. time (seconds)
Simulink high-level model	804.0
Simulink low level model	762.0
Software application on target embedded system device (with accelerators) and ROS interface overhead	30.0

other hand, such user-defined blocks were available in the OpenVX-Vision Works library thorough GPU-accelerated primitives.

5 Conclusion

This paper presented a methodology to integrate model-based design to OpenVX. It showed how such a design flow allows for fast prototyping of any

computer vision algorithm through a Matlab/Simulink model, its parametrization, validation, and automatic synthesis into an equivalent OpenVX code representation. The paper presented the experimental results obtained by applying the proposed methodology for the development of two embedded software. The first implements a digital image stabilization, while the second implements an ORB descriptor for SLAM applications. The applications have been modelled and parametrized through Simulink for different application contexts. In particular, the ORB application has been validated by considering an external typical and dynamic Robotics environment. This has been done through the OpenVX-ROS interface generated with the proposed design flow, which allows co-simulating the OpenVX application with the external application environment (e.g., input streams, concurrent interactive systems, etc.) and, as a consequence, tuning more efficiently the SW parametrization. Both the applications have been automatically synthesized into OpenVX-VisionWorks code for an NVIDIA Jetson TX2 board.

References

1. Embedded Vision Alliance: Applications for Embedded Vision. https://www.embedded-vision.com/applications-embedded-vision
2. Pulli, K., Baksheev, A., Kornyakov, K., Eruhimov, V.: Real-time computer vision with OpenCV. Commun. ACM **55**(6), 61–69 (2012)
3. Rainey, E., Villarreal, J., Dedeoglu, G., Pulli, K., Lepley, T., Brill, F.: Addressing system-level optimization with OpenVX graphs. In: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, pp. 658–663 (2014)
4. Khronos Group: OpenVX: Portable, Power-efficient Vision Processing. https://www.khronos.org/openvx
5. Tagliavini, G., Haugou, G., Marongiu, A., Benini, L.: Adrenaline: an OpenVX environment to optimize embedded vision applications on many-core accelerators. In: International Symposium on Embedded Multicore/Many-core Systems-on-Chip, pp. 289–296 (2015)
6. Yang, K., Elliott, G.A., Anderson, J.H.: 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, RTNS 2015, pp. 77–86 (2015)
7. Dekkiche, D., Vincke, B., Merigot, A.: Investigation and performance analysis of OpenVX optimizations on computer vision applications. In: 14th International Conference on Control, Robotics and Vision, Automation, pp. 1–6 (2016)
8. Open Source Robotics Foundation: Robot Operating System. http://www.ros.org/
9. Popp, M., van Son, S., Moreira, O.: Automatic control flow generation for OpenVX graphs. In: 2017 Euromicro Conference on Digital System Design (DSD), pp. 198–204, August 2017
10. Syschikov, A., Sedov, B., Nedovodeev, K., Ivanova, V.: OpenVX integration into the visual development environment. Int. J. Embed. Real-Time Commun. Syst. **9**(1), 20–49 (2018). www.scopus.com
11. Aldegheri, S., Bombieri, N.: Extending OpenVX for model-based design of embedded vision applications. In: Proceedings of 2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC), pp. 1–6 (2017)

12. Simulink: S-Functions. https://it.mathworks.com/help/simulink/s-function-basics.html
13. Smith, B.M., Zhang, L., Jin, H., Agarwala, A.: Light field video stabilization. In: International Conference on Computer Vision, pp. 341–348 (2009)
14. NVIDIA Inc.: VisionWorks. https://developer.nvidia.com/embedded/visionworks
15. INTEL: Intel Computer Vision SDK. https://software.intel.com/en-us/computer-vision-sdk
16. AMD: AMD OpenVX - AMDOVX. http://gpuopen.com/compute-product/amd-openvx/
17. Khronos: OpenVX lib. https://www.khronos.org/openvx
18. Matlab: MEX functions. https://it.mathworks.com/matlabcentral/fileexchange/26825-utilities-for-mex-files
19. Aldegheri, S., Bloisi, D.D., Blum, J.J., Bombieri, N., Farinelli, A.: Fast and power-efficient embedded software implementation of digital image stabilization for low-cost autonomous boats. In: Hutter, M., Siegwart, R. (eds.) Field and Service Robotics. SPAR, vol. 5, pp. 129–144. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-67361-5_9
20. Rublee, E., Rabaud, V., Konolige, K., Bradski, G.: ORB: an efficient alternative to SIFT or SURF. In: 2011 International Conference on Computer Vision, pp. 2564–2571, November 2011
21. Mur-Artal, R., Montiel, J.M.M., Tardós, J.D.: ORB-SLAM: a versatile and accurate monocular SLAM system. IEEE Trans. Rob. **31**(5), 1147–1163 (2015)
22. Geiger, A., Lenz, P., Stiller, C., Urtasun, R.: Vision meets robotics: the KITTI dataset. Int. J. Rob. Res. (IJRR) **32**, 1231–1237 (2013)