Extending OpenVX for Model-based Design of Embedded Vision Applications

Stefano Aldegheri, Nicola Bombieri
Dept. Computer Science
University of Verona - Italy
Email: name.surname@univr.it

***Abstract*—Developing computer vision applications for low-power heterogeneous systems is increasingly gaining interest in the embedded systems community. Even more interesting is the tuning of such embedded software for the target architecture when this is driven by multiple constraints (e.g., performance, peak power, energy consumption). Indeed, developers frequently run into system-level inefficiencies and bottlenecks that can not be quickly addressed by traditional methods. In this context OpenVX has been proposed as the standard platform to develop portable, optimized and power-efficient applications for vision algorithms targeting embedded systems. Nevertheless, adopting OpenVX for rapid prototyping, early algorithm parametrization and validation of complex embedded applications is a very challenging task. This paper presents a methodology to integrate a model-based design environment to OpenVX. The methodology 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. Experimental results have been conducted with an application for digital image stabilization developed through Simulink and, then, automatically synthesized into OpenVX-VisionWorks code for an NVIDIA Jetson TX1 board.**

I. 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 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 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 re-writes

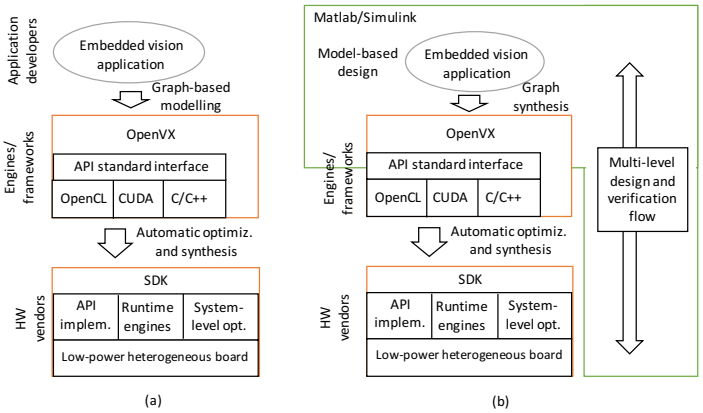


Fig. 1. The embedded vision application design flow: the standard (a), and the extended with the model-based design paradigm (b)

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], [6], [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.

This paper presents a framework that extends OpenVX to

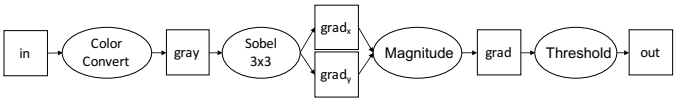


Fig. 2. OpenVX sample application (graph diagram)

the model-based design paradigm (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 by which the high-level model is then semi-automatically refined towards the final automatic synthesis into OpenVX code. The paper presents the results obtained by applying the proposed methodology for developing and tuning an algorithm for digital image stabilization for two different application contexts. 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 TX1 embedded system board.

The paper is organized as follows. Section II presents the background and the related work. Section III explains the model-based design methodology. 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 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 constructs 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.. Fig. 2 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 graph is processed by the OpenVX framework (line 23). At the end of the code execution, all created data objects, the graph, and the the context are released.

Different works have been presented to analyse the use of OpenVX for embedded vision [5], [6], [7]. In [6], the authors

```
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 grad_x = vxCreateVirtualImage(g, 0, 0, type);
8  vx_image grad_y = vxCreateVirtualImage(g, 0, 0, type);
9  vx_image grad   = 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, grad_x, grad_y);
17 vxMagnitudeNode(g, grad_x, grad_y, grad);
18 vxThresholdNode(g, grad, threshold, out);
19 /*verify the graph*/
20 status = vxVerifyGraph(g);
21 /*execute the graph*/
22 if (status == VX_SUCCESS)
23   status = vxProcessGraph(g);
```

Listing 1. OpenVX code of the example of Fig. 2

present a new implementation of OpenVX targeting CPUs and GPU-based devices by leveraging different analytical optimzation 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.

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.

III. The model-based design approach

Fig. 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 and embedded application primitives to be defined by the user if not included in the toolbox through the Simulink *S-Function* construct [8] (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., [9]). Efficient test patterns are extrapolated, by using any technique of the literature, to

[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.

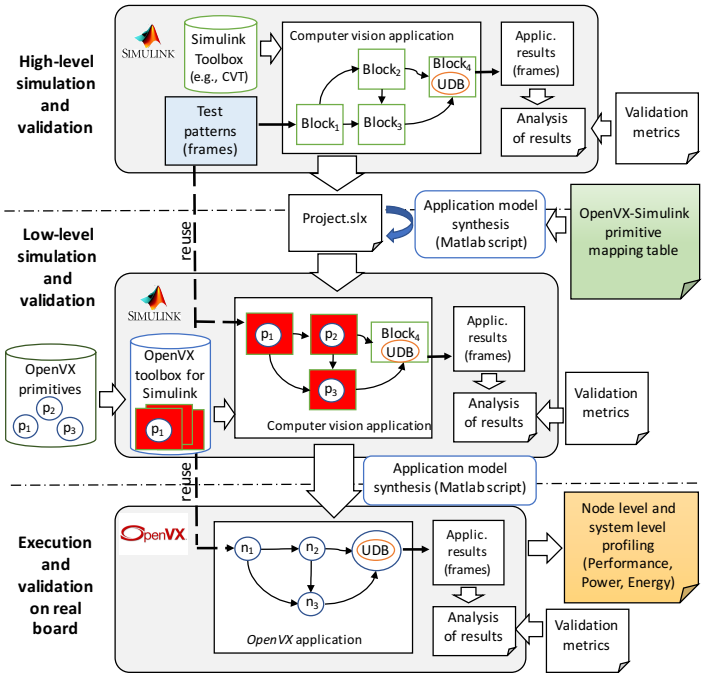


Fig. 3. Methodology overview

asses 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 [10], INTEL OpenVX [11], AMDOVX [12], Khronos OpenVX standard implementation [13]), such a toolbox of blocks for Simulink is created by properly wrapping the primitives through Matlab $S\text{-}Function$, as explained in Section III-A.
2) *The OpenVX primitives-Simulink blocks mapping table*. It provides the mapping between Simulink blocks and the functionally equivalent OpenVX primitives, as explained in Section III-B.

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.

Finally, the low-level Simulink model is synthesized, through a Matlab script, into an OpenVX model, which is

```
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 2. Matlab S-function Code for the Color Converter node.

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 are be re-used for the node-level and system-level optimization of the OpenVX application.

A. 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* [14]. 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 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).

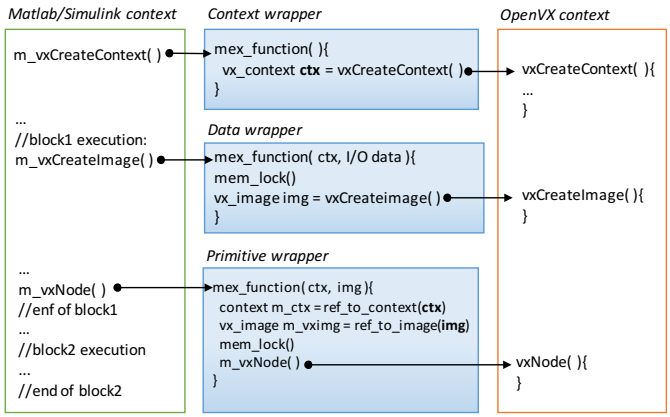


Fig. 4. Overview of the Simulink-OpenVX communication

- *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 summarized in Fig. 4. The *context wrapper* allows creating the OpenVX context (see line 1 of Listing 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). 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.

B. 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 I shows, as an example, a representative subset of the mapped entries.

We implemented three possible mapping strategies:

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		

TABLE I
REPRESENTATIVE SUBSET OF THE MAPPING TABLE BETWEEN SIMULINK CVT AND NVIDIA OPENVX-VISIONWORKS

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 3, which follows the node implementation directives required by the standard OpenVX for importing user kernels[2]. The wrapper invocation (i.e.,

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

```
1  vx_userNode(){
2    vx_status processingOpenVX(vx_node node, const
        vx_reference *parameters, vx_uint32 num)
3    {
4      //convert data in internal representation
5      SimulinkBlockFunctionality(); //C/C++ code of the UDB
        functionality
6      return VX_SUCCESS;
7    }
8    vx_status validationOpenVX(vx_node node, const
        vx_reference parameters[], vx_uint32 num,
        vx_meta_format metas[]))
9    {
10     //insert parameter validation
11     return VX_SUCCESS;
12   }
13
14   vx_status singleShotProcessing(vx_context context,
        parameters)
15   {
16     //create graph and execute it
17   }
18
19   vx_status registerCustomKernel(vx_context context)
20   {
21     vx_status = vxAddUserKernel(context, ...);//register
        kernel in context
22     return VX_SUCCESS;
23   }
24 }
```

Listing 3. Overview of wrapper for user-defined Simulink block implementations

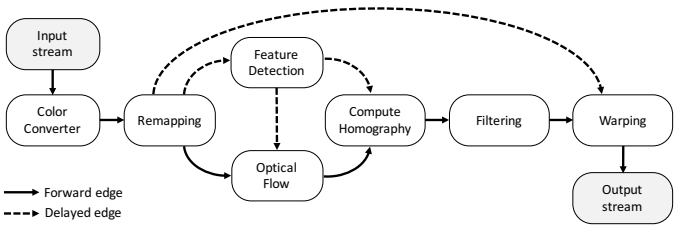


Fig. 5. Digital image stabilization algorithm.

$vx\_userNode()$) is 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 righ-most 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.

IV. Experimental Results

We applied the proposed model-based design flow for the development of the embedded software implementing a digital image stabilization algorithm for camera streams.

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

TABLE II
Experimental results: High-level simulation time in Simulink

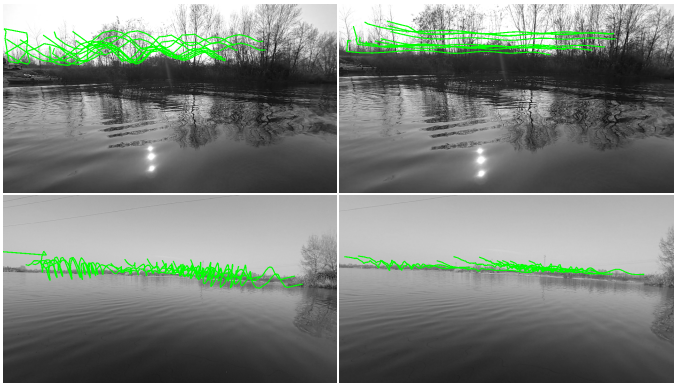
Fig. 5 shows an overview of the algorithm, which is represented through a dependency graph. The input stream (i.e., sequence of frames) is taken from a high-definition camera, and each frame is converted to the grayscale format to improve the algorithm efficiency without compromising the quality of the 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. 5 denote inter-frame dependencies, i.e., parts of the algorithm where a temporal window of several frames is used to calculate the camera translation.

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)[3].

We validated the two algorithm configurations starting from input streams registered by different cameras at 60 FPS with 1280x720 (1080P) and 1920x1080 wide angle resolution, respectively. Table II 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 [9]. The table reports the characteristics of such selected test patterns (sequences of frames), while Fig. 6 shows some of the application results obtained for the outdoor context.

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

[3]H2020 EU project INTCATCH - www.intcatch.eu



(a) (b)

Fig. 6. Video stabilization results for the outdoor context. (a) A frame in the unstabilized video overlayed with lines representing point trajectories traced over time. (b) The stabilization results satisfying the validation metrics in [9].

VisionWorks v1.6 [10] and the corresponding Simulink CVT-NVIDIA OpenVX/VisionWorks mapping table, as described in Sections III-A and III-B, respectively. In particular, the low level simulation in Simulink allowed us to validate the computer vision application implemented through the primitives 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 III 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 [9] 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.

Validation level	**Sim./Exec. time (min)**	
	Indoor	**Outdoor**
Simulink High-Level model	30.5	13.0
Simulink Low Level model	59.0	26’
Software application on target embedded system device	20.5	11.0

TABLE III
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.

V. 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 an embedded software implementing the digital image stabilization, which has been modelled and parametrized through Simulink for different application contexts and, then, automatically synthesized into OpenVX-VisionWorks code for an NVIDIA Jetson TX1 board.

ACKNOWLEDGMENT

The authors would like to thank Domenico Daniele Bloisi, Jason Joseph Blum, and Alessandro Farinelli for providing the datasets and for supporting the analysis of the quality results.

REFERENCES

[1] Embedded Vision Alliance, “Applications for Embedded Vision,” https://www.embedded-vision.com/applications-embedded-vision.
[2] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, “Realtime computer vision with OpenCV,” vol. 10, no. 4, 2012.
[3] 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.
[4] Khronos Group, “OpenVX: Portable, Power-efficient Vision Processing,” https://www.khronos.org/openvx.
[5] 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.
[6] 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.
[7] 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.
[8] Simulink, “S-Functions,” https://it.mathworks.com/help/simulink/s-function-basics.html.
[9] B. M. Smith, L. Zhang, H. Jin, and A. Agarwala, “Light field video stabilization,” in *International Conference on Computer Vision*, 2009, pp. 341–348.
[10] NVIDIA Inc., “VisionWorks,” https://developer.nvidia.com/embedded/visionworks.
[11] INTEL, “Intel Computer Vision SDK,” https://software.intel.com/en-us/computer-vision-sdk.
[12] AMD, “AMD OpenVX - AMDOVX,” http://gpuopen.com/compute-product/amd-openvx/.
[13] Khronos, “OpenVX lib,” https://www.khronos.org/openvx.
[14] Matlab, “mex functions,” https://it.mathworks.com/matlabcentral/fileexchange/26825-utilities-for-mex-files.