Rapid Prototyping of Embedded Vision Systems: Embedding Computer Vision Applications into Low-Power Heterogeneous Architectures

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

***Abstract*—Embedded vision is a disruptive new technology in the vision industry. It is a revolutionary concept with far reaching implications, and it is opening up new applications and shaping the future of entire industries. It is applied in self-driving cars, autonomous vehicles in agriculture, digital dermascopes that help specialists make more accurate diagnoses, among many other unique and cutting-edge applications. The design of such systems gives rise to new challenges for embedded Software developers. Embedded vision applications are characterized by stringent performance constraints to guarantee real-time behaviours and, at the same time, energy constraints to save battery on the mobile platforms. In this paper, we address such challenges by proposing an overall view of the problem and by analysing current solutions. We present our last results on embedded vision design automation over two main aspects: the adoption of the model-based paradigm for the embedded vision rapid prototyping, and the application of heterogeneous programming languages to improve the system performance. The paper presents our recent results on the design of a localization and mapping application combined with image recognition based on deep learning optimized for an NVIDIA Jetson TX2.**

***Index Terms*—Embedded vision, Heterogeneous architectures, OpenVX, GPU, ORB-SLAM, Jetson TX2**

I. Introduction

Embedded vision refers to the integration of an input data sensor (typically a camera), an embedded processing board, and a software application. The design and development process of such systems strongly rely on programming *embedded vision applications*, which is far from simple and immediate. Pushed by the need for extreme energy efficiency, embedded vision systems are embracing architectural heterogeneity, in which a multi-core host processor is coupled with programmable accelerators specialized for various domains (e.g., GPUs, DSPs, multicore CPUs, FPGAs) [1], [2]. Indeed, beside functional correctness, programmers have to face non-functional constraints like performance, power consumption, reliability, and real-time [3], [4].

Software development of embedded vision systems is architecture dependent and needs optimizations over two main dimensions: block-level and system-level. The first is more intuitive and applies to single functions (also called *kernels*). This involves the re-implementation or parallelization of single kernels for the target board accelerators through specific languages or programming environments like CUDA, OpenCL, or OpenCV. The system-level optimization targets the overall system power consumption, memory bandwidth, and inter-process communication overhead. OpenVX [5] has been proposed to help developers in both cases, and it is gaining consensus as the reference standard, programming environment, and API library. OpenVX aims at maximizing functional and performance portability of vision applications across different hardware platforms. It enables hardware vendors to implement and optimize low-level image processing primitives, with strong focus on mobile and embedded systems. In addition, it addresses code optimization for different hardware architectures with minimal impact on the software applications. Starting from a graph model of the embedded application, OpenVX allows for automatic system-level optimizations and synthesis on the target architecture by optimizing performance, power consumption and energy efficiency [6]–[9].

In this context, we consider two of the main still open issues. First, the definition of the graph-based model, its parametrization and validation is time consuming and far from intuitive to programmers, especially for the development of medium-complex applications. Second, due to the limitation of OpenVX to model complex applications through data-flow graphs and to the incompleteness of the OpenVX primitive library, any real embedded vision application requires the integration of OpenVX with user-defined C/C++ code. On the one hand, the user-defined code can benefit from parallelization techniques for multi-cores, thus providing heterogeneous parallel environments (i.e., multi-core + GPU parallelism). On the other hand, due to the private and not user-controlled memory stack of OpenVX, such an integration leads to the sequentialization of the different execution environments, with a consequent strong impact on the system-level optimization.

This paper presents an overall platform that aims at addressing these main challenges. The first contribution is a methodology that extends OpenVX to the model-based design paradigm. Differently from the standard approaches at the state of the art that require designers to manually model the algorithm through OpenVX code, the proposed approach allows for a rapid prototyping, algorithm validation and parametrization through 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.

Then the paper shows how the platform allows combining different programming environments, i.e., OpenMP, PThreads, OpenVX, OpenCV, and CUDA to best exploit different levels of parallelism while guaranteeing the semi-automatic customization of the embedded vision applications.

The paper presents the results obtained by applying the proposed framework to develop and verify a mapping and localization application (ORB-SLAM) for an NVIDIA Jetson TX2 board. The paper shows the impact of the several versions of the code obtained through the heterogeneous programming on the non-functional constraints when combined with an image recognition system based on Deep Learning running on the same board.

The paper is organized as follows. Section II presents and analysis of the state of the art and of the related work. Section III presents the model-based design flow. Section IV presents the techniques to combine heterogeneous programming languages for the design of embedded vision applications. Section V presents the experimental results, while Section VI is devoted to the concluding remarks.

II. Related work

During the performance optimization of a computer vision system, developers frequently run into platform-level bottlenecks and inefficiencies that cannot be addressed by traditional methods. OpenVX has been proposed to address such system-level issues by meas of a graph-based paradigm [10]. Graphs are used to specify a computing method. They are constructed, then verified for correctness, consistency, and connectedness, and finally processed.

The target embedded system (for computer vision applications) can have on-chip resources (computational, power, area, etc.) as large a an autonomous car or as small as a battery operated device. In both cases, the final goal for developers is maximizing performance while decreasing power consumption. Different works have been presented to optimize OpenVX in this direction. *JANUS* [11] is a compilation system for OpenVX that can analyse and optimize the graph to take advantage of parallel resources in many-core systems or FPGAs. Using a database of prewritten OpenVX kernels, it automatically adjusts the image tile size and relies on kernel duplication and coalescing to meet a defined area target, or to meet a specified throughput target.

Dekkiche et al. [8] investigated on how OpenVX responds to different data access patterns. They tested optimizations like kernel merge, data tiling, and parallelization via OpenMP. They also proposed an approach to target both system-level and kernel-level optimizations on different hardware architectures. The approach consists in merging OpenVX and the *Numerical Template ToolBox* (NT^2) library [12]. In this way, OpenVX addresses system-level optimizations, while NT^2 targets single kernels acceleration on different processing elements with minimal cost of code rewriting.

One of the main bottlenecks that can be addressed with OpenVX is the memory bandwidth limits imposed by the architectural constraints. Tagliavini et al. strongly investigated on this issue. They first proposed *ADRENALINE* [6], which is a framework for graph analysis and image tiling to accelerate the execution of image processing on cluster-based many-core accelerators. They then refined the approach by proposing tiling techniques optimized for different data access patterns [13]–[15].

Glenn et al. [16] presented a variant of OpenVX that is amenable to real-time analysis. They presented some graph transformation techniques to eliminate graph cycles due to back edges and to enable pipelining. These transformations enable real-time constraints to be validated. In particular, the specific constraint they consider is that end-to-end graph response times are provably bounded [7].

Yang et al. [17] proposed a much more fine-grained approach for scheduling OpenVX graphs. The approach is designed to enable additional parallelism and to eliminate schedulability-related processing-capacity loss that arises when programs execute on both CPUs and GPUs. They presented a response-time analysis for this new approach and the evaluation of its efficacy.

Implementing or porting OpenVX for different hardware architectures has been the focus of many research groups in the last years [18]–[20].

Differently from all the work of the literature, this paper presents and integrated framework with a twofold contribution: An extension of the OpenvX environment to the model-based design paradigm, and the application of different programming languages to best exploit the heterogeneous nature of the target architectures. All the state-of-the-art optimization techniques can be applied and combined to the proposed platform.

III. Rapid System Protoyping Through the Model-based Paradigm

Fig. 1 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 [21] (e.g., user-defined block UDB $Block_4$ in Fig. 1). They can also model vision blocks for which the developer has the corresponding kernel optimized for parallel execution (e.g., C/C++ with OpenMP pragmas, C/C++ with PThread, CUDA).

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., [22]). Efficient test patterns are extrapolated, by using any technique

[1]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.

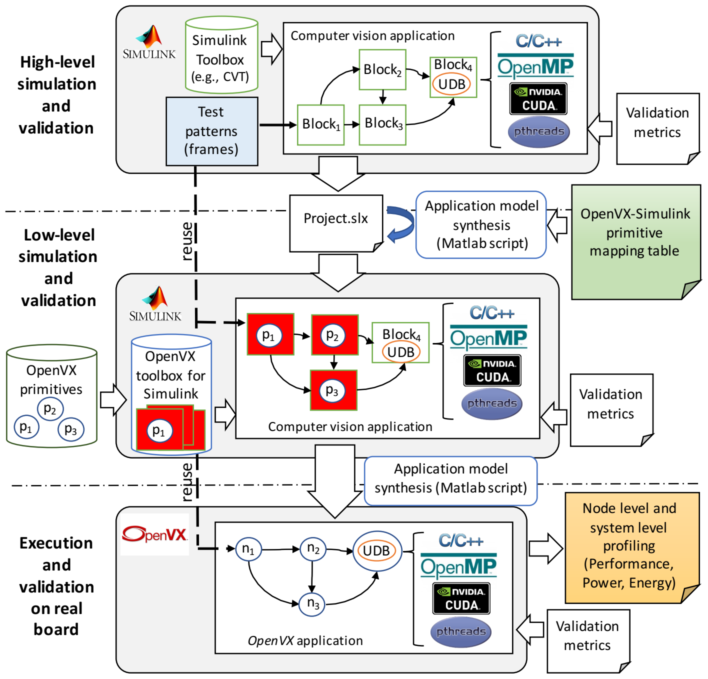
High-level simulation and validation
Simulink Toolbox (e.g., CVT)
Test patterns (frames)
Computer vision application
Block1
Block2
Block3
Block4
UDB
C/C++
OpenMP
CUDA
pthreads
Validation metrics
Project.slx
Application model synthesis (Matlab script)
OpenVX-Simulink primitive mapping table
reuse
Low-level simulation and validation
OpenVX primitives
OpenVX toolbox for Simulink
Execution and validation on real board
Node level and system level profiling (Performance, Power, Energy)
OpenVX application

Fig. 1. The model-based design flow

of the literature, to 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:

-) *The OpenVX toolbox for Simulink*. Starting from the library of OpenVX primitives (e.g., NVIDIA VisionWorks [23] in the current implementation), we created a toolbox of blocks for Simulink by properly wrapping the primitives through Matlab $S\text{-}Function$. Such function allows describing any Simulink block functionality through C/C++ code. The code is compiled as *mex file* by using the Matlab *mex utility* [24]. As with other *mex* files, *S-functions* are dynamically linked subroutines that the Matlab execution engine automatically loads and executes. *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. Such a wrapping-based methodology can be applied to implement the toolbox of Simulink blocks for other hardware vendor primitives (e.g., INTEL OpenVX [25], AMDOVX [26], Khronos OpenVX standard implementation [27]).

-) *The OpenVX primitives-Simulink blocks mapping table*. It provides the mapping between Simulink blocks and the functionally equivalent OpenVX primitives. 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 or, if any user-defined primitive, a heterogeneous model implemented in OpenVX, OpenMP, Pthreads, and CUDA. The interaction of the different programming languages is addressed in the following section. The program is executed and validated on the target embedded board. At this level, all the techniques of the literature for 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 re-used for the node-level and system-level optimization of the OpenVX application.

IV. Enhancing Performance Through Heterogeneous Language Programming

Beside OpenVX, we consider five different languages and parallel programming environments (*environments* in the following): C/C++, Pthreads, OpenMP, OpenCV, and CUDA. The environment heterogeneity allows implementing different application blocks with the most appropriate style, such as C/C++ for control parts, Pthreads for concurrent execution functions on the CPUs, OpenMP for directive-based automatic parallelization of code chunks, CUDA for any kernel (if available) acceleration on GPU, and OpenVX for primitive-based parallelization of data-flow routines. OpenCV has been chosen to implement standard I/O communication protocols of computer vision applications through standard data-structures and APIs. This allows the embedded vision applications to be portable and efficiently integrated to any other application compliant to the standard.

For the sake of clarity and without loss of generality, we consider, as a running example, the widespread and most popular NVIDIA Jetson TX2 as the target platform. Such an embedded board relies on a shared-memory architecture, in which two different clusters of CPUs (four cores Cortex-A57 CPUs and two cores Denver CPUs) and a GPU with two symmetric multiprocessors share an unified memory space.

The top of Figure 2 depicts the stack layer involved by the concurrent execution of each environment. It relies on two main parts:

- *The user-controlled stack*, which allows for shared memory-based communication among processes running on different CPUs. They include C/C++ processes, OpenCV APIs, Pthreads, and processes generated by OpenMP.

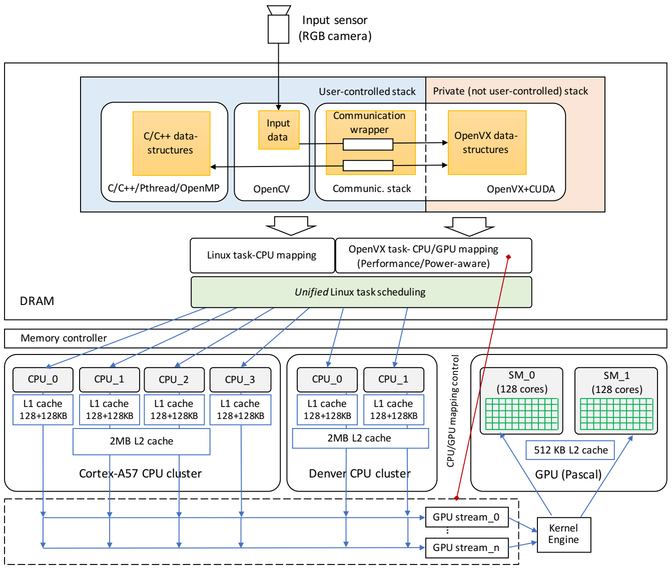

Fig. 2. Framework overview: memory stack, task mapping, and task scheduling layers of an embedded vision application developed with the proposed method on the NVIDIA Jetson TX2 board.

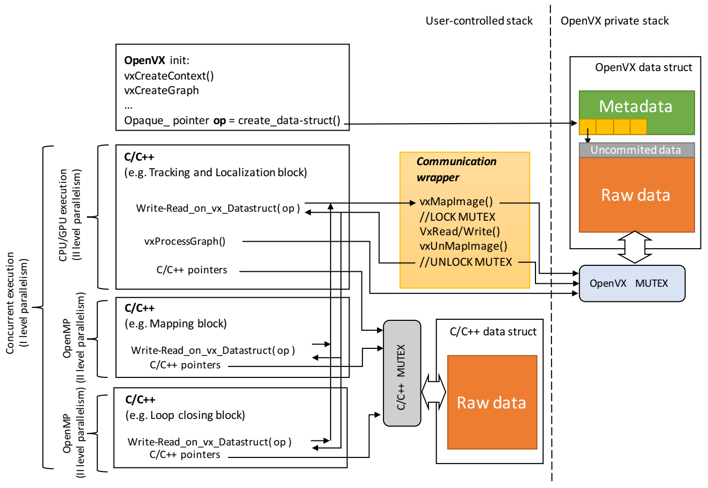

Fig. 3. Overview of the communication wrapper

- *The private (not user-controlled) stack*, which is created and handled by OpenVX and allows for communication between OpenVX graph nodes running on different CPUs or on the GPU.

The tasks related to the user-controlled stack are mapped to the CPU cores by the operating system (i.e., Linux Ubuntu for the NVIDIA Jetson). The OpenVX tasks are mapped to the CPU cores or GPU multiprocessors by the OpenVX runtime system.

To enable the full concurrency of the two parts, to avoid sequentialization of the two sets of tasks, and to avoid the consequent synchronization overhead, we associate the two parts to a single *unified* scheduling engine. This allows all the tasks mapped to the CPU cores (of both stack parts) to be scheduled by the operating system, while the GPU task scheduling, the CPU-to-GPU communication and synchronization (i.e., GPU stream and kernel engine) to be controlled by the OpenVX runtime system. To do that, we propose a C/C++-OpenVX template-based communication wrapper, which allows for memory accesses to the OpenVX data structures on the private stack and for full control of the OpenVX context execution by the C/C++ environment.

Figure 3 gives an overview of the wrapper and its integration in the system. The OpenVX initialization phase generates the graph context and allocates the private data structures. Such allocation returns *opaque pointers* to the allocated memory segments, i.e., pointers to private memory areas which layout is unknown to the programmer.

OpenVX read and write primitives (`Write-Read_on_vx_Datastructure()` in the Figure) have been defined to access the private data structures through the opaque pointers. The primitives are invoked from the C/C++ context and, through the communication wrapper APIs, they set a mutex mechanism to safely access the OpenVX data structures. The same mutex is shared with the OpenVX runtime system for the overall graph processing (`vxProcessGraph()` in the Figure). As a consequence, the mechanism guarantees synchronization during the accesses to the shared data structures between the OpenVX and C/C++ contexts when run concurrently on multicores. It is important to note that the invocation of the overall graph processing, which is performed in the C/C++ environment, starts the execution of the data-flow oriented OpenVX code. As shown in Figure 3, such an invocation can be performed concurrently by different C/C++ threads, and each invocation involves a mapping and scheduling of the corresponding graph instance. The proposed communication wrapper and mutex system allow for synchronization among the different concurrent OpenVX graph executions and the C/C++ calling environments.

Standard mutex mechanisms are adopted to synchronize all the other C/C++ based contexts belonging to the user-controlled stack, when accessing shared data structures.

The mutex-based communication wrapper allows for multi-level parallel execution of the application. For example, a first level of parallelism can be implemented by Pthreads, which run application blocks on different CPU cores. Then, other blocks can be implemented in OpenVX and run on a CPU core and on the GPU. The parallel implementation of the graph nodes offloaded on the GPU is provided by the OpenVX library vendor (i.e., NVIDIA VisionWorks for our case study) and are optimized for the specific GPU architecture. In case two nodes of the OpenVX graph are independent, they are executed concurrently. Finally, OpenMP provides another level of parallelism when a block is enriched with parallel directives. Each of these blocks is executed in parallel by the threads generated automatically by the compiler, which run on the available CPU cores.

V. Putting all together: the Mapping, Localization, and Image Recognition Case Study

We applied the proposed platform to develop a mapping and localization application (ORB-SLAM) [28] for an NVIDIA Jetson TX2 board. We developed and optimized such application considering its utilization possibly concurrent to an image recognition system based on Deep Learning [29] running on the same board (see Figure 4).

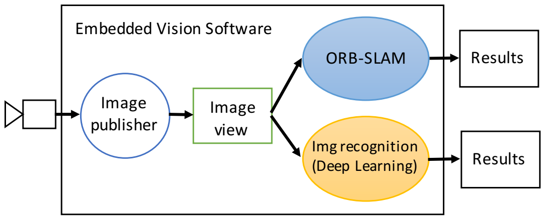

Fig. 4. Inter-application communication

ORB-SLAM solves the simultaneous localization and mapping problem when RGB camera sensors are adopted. It computes, 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 builds a 3D map starting from an input stream and/or it performs localization by considering the current map. The application consists of three main blocks (see Figure 5):
- The *tracking and localization* block computes visual features, it localizes the agent in the environment, and, in case of significant discrepancies between an already saved map and the input stream, it communicates updating information of the map to the mapping block. The processing rate (i.e., the supported frame rate per second) and the main power consumption of the whole application strongly depend on this block performance.
- The *mapping* block updates the environment map by using information (map changes) sent by the localization block.
- The *loop closing* block aims at adjusting the scale drift error accumulated during the input analysis. When a loop in the agent pathway is detected, this block updates the mapped information through a high latency heavy computation, during which the first two blocks must be suspended. This can lead the agent to loose tracking and localization information and, as a consequence, the agent to get temporary lost. The computation efficiency of this block (run on-demand) is crucial for the quality of the whole application results.

We first applied the model-based design flow to define the mapping and localization algorithm as explained in Section III. Along the design flow, we measured the execution time of the algorithm implementations at different refinement steps, by using the KITTI dataset [30], which is a standard set of benchmarks for SLAM and computer vision applications.

Table I reports the execution time we obtained at different refinement levels. Starting from the original video streams, we extrapolated a subset of test patterns, which consist of

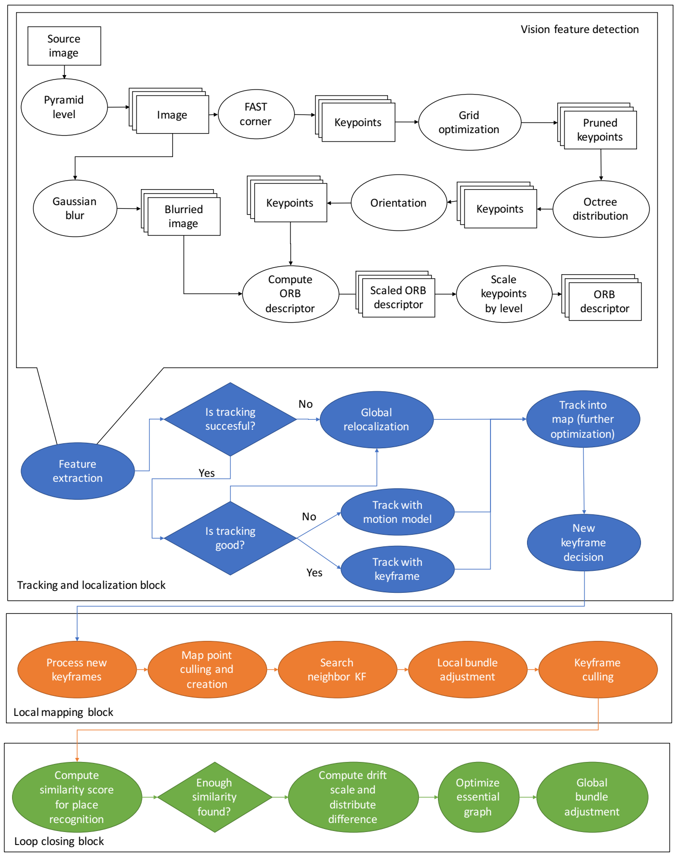

Fig. 5. Overview of the ORB-SLAM use case.

Validation level	Sim./Exec. time (sec)
Simulink High-Level model	804.0
Simulink Low Level model	762.0
Software application on target embedded system device (with accelerators) - *Version 3*: Pthreads+OpenVX	30.0

TABLE I
Simulation (in Simulink) and execution (on real board) times

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 [22].

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 [23] and the corresponding Simulink CVT-NVIDIA OpenVX/VisionWorks mapping table.

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 TX2) with the different code versions generated through the heterogeneous language programming.

We observed a slightly reduced execution time for the Simulink low-level model execution with respect to the high-level model despite the overhead introduced by the wrappers.

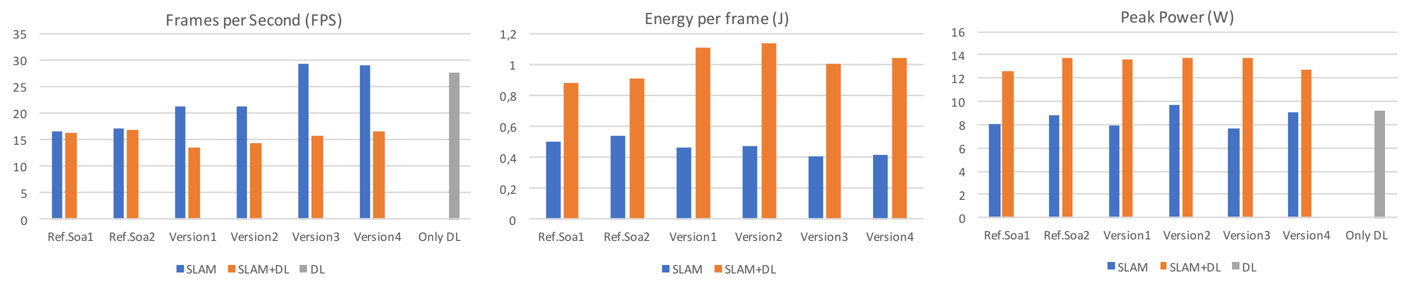

Fig. 6. Evaluation of non-functional properties

This is due to the fact that the algorithm implementation in Simulink required specialized MATLAB code that was not available with Simulink CVT library as native blocks. We developed custom code in MATLAB to meet the requirements, and imported such a code as user-defined Simulink blocks using 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 other hand, such user-defined blocks were available in the OpenVX-Vision Works library thorough GPU-accelerated primitives.

At each refinement step, we reused the selected test patterns to verify the code over the adopted validation metrics [22] 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 and power analysis.

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, we run the validation through execution on the target real device for both functional and non-functional verification. In particular, we evaluated performance, power consumption and energy efficiency of the different code versions generated thanks to the heterogeneous language programming presented in Section IV (*Version 1, 2, and 3* in the following). We compared their non-functional properties with those provided by the most efficient parallel implementations at the state of the art [31] of the same algorithm (*Reference SoA 1 and 2* in the following). In particular, we consider:

- *Reference SoA 1 (Pthreads)*: It is the state of the art version [31], in which the three main blocks (Tracking and localization, Mapping, and Loop closing blocks) are run concurrently by Pthreads on the CPU cores.
- *Reference SoA 2 (Phtreds+OpenMP)*: It extends *Reference SoA 1* by enabling OpenMP parallelism. In particular, it parallelizes the bundle adjustment task, both local in the mapping block and global in the loop closing block.
- *Version 1* (OpenVX+Pthreads): It is the first version generated with the proposed framework. It implements the tracking sub-block in OpenVX, while the other two blocks are implemented in C/C++ and run concurrently through Pthreads.
- *Version 2* (OpenVX+Pthreads+OpenMP): It extends *Version 1* by enabling also OpenMP in the Mapping and Loop closing blocks.
- *Version 3* (OpenVX+CUDA+Pthreads): starting from *Version 1*, we reused a CUDA kernel that implements the *gaussian blur* primitive in the tracking sub-block. We modularly replaced the corresponding OpenVX VisionWork primitive with such a more optimized kernel.
- *Version 4* (OpenVX+CUDA+Pthreads+OpenMP): It extends version 3 by enabling also OpenMP.

The Pthreads guarantee the minimum level of parallelism, by enabling one CPU core per block. OpenMP has been set to use the maximum number of available CPU cores (i.e., 4+2 in the Jetson). The GPU is enabled only by OpenVX/CUDA.

Figure 6 summarizes the results. The first plot (FPS) reports information about the ORB-SLAM application performance. *FPS* represents the maximum number of frames per second supported by the embedded system. It has been measured in two system configurations: with the only ORB-SLAM application running on the board (all board resources available for the developed application) and with ORB-SLAM running concurrently with the image recognition -DL- system (board resources shared). The results show the benefit of the heterogeneous language programming, by which Version 3 and Version 4 almost increase the performance by 100% with respect to the parallel versions for multicore at the state of the art, and by 50% with respect to Version 1 and 2 (without CUDA) generated by the proposed flow. On the other hand, the plot also shows that the versions that rely on the only multi-core CPUs are slightly better when run concurrently with GPU-hungry applications (i.e., DL). This is due to the fact that scheduling ORB-SLAM tasks on GPUs in these cases causes more overhead (for resource contention) than benefits.

The second and third plots report the energy efficiency and peak power consumption of the system. The results show that Version 3 provides the best results by guaranteeing up to 20% and 15% of energy and peak power reduction, respectively, w.r.t. the other versions in the first configuration. The plots show that, when the DL application is switched on, the Ref. Soa implementations are the most energy efficient while the peak power is fairly the same.

In general, the results show that, as expected, exploiting the heterogeneous characteristics of the board allows reaching the best performance and energy efficiency of the system. This underlines the benefits of the proposed method, by which the different computing elements are fully exploited by the different programming environments. On the other hand, we found that adopting all the possible environments is not always the best solution. Version 6 is an example, in which switching on the OpenMP parallelism does not provide better performance than the Pthread+OpenVX+CUDA version while it increases the peak power consumption.

In conclusion, the experimental results show how the different versions provide a very large mapping space to be explored (which is part of our future work). Such a space can provide the best solution for each of the considered design constraints like performance, power consumption, and energy efficiency.

VI. Conclusion

This paper presented an overall platform with two main characteristics: It extends OpenVX to the model-based design paradigm, and it allows for heterogeneous language programming. Differently from the standard approaches, the proposed platform allows for a rapid prototyping, algorithm validation and parametrization in a model-based design environment (i.e., Matlab/Simulink). The platform allows combining different programming environments, i.e., OpenMP, PThreads, OpenVX, OpenCV, and CUDA to best exploit different levels of parallelism while guaranteeing the semi-automatic customization of the embedded vision applications. The experimental results showed how the different versions provide a very large mapping space to be explored by considering different system configurations (available resources) and non-functional design constraints like performance, power consumption, and energy efficiency.

References

[1] C. Ttofis, C. Kyrkou, and T. Theocharides, "A hardware-efficient architecture for accurate real-time disparity map estimation," *ACM Transactions on Embedded Computing Systems*, vol. 14, no. 2, p. 36, 2015.

[2] S. Aldegheri, N. Bombieri, N. Dall'Ora, F. Fummi, S. Girardi, and M. Panato, "A framework for the design and simulation of embedded vision applications based on OpenVX and ROS," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018.

[3] N. Sawasaki, M. Nakao, Y. Yamamoto, and K. Okabayashi, "Embedded vision system for mobile robot navigation," in *Proc. of IEEE International Conference on Robotics and Automation*, 2006, pp. 2693–2698.

[4] Y. Tang and N. Verma, "Energy-efficient pedestrian detection system: Exploiting statistical error compensation for lossy memory data compression," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1301–1311, 2018.

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

[6] 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.

[7] 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.

[8] 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.

[9] S. Aldegheri and N. Bombieri, "Extending OpenVX for model-based design of embedded vision applications," in *Proceedings of the 18th IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC*, 2017, pp. 1–6.

[10] 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.

[11] H. Omidian and G. Lemieux, "Janus: A compilation system for balancing parallelism and performance in OpenVX," *Journal of Physics: Conference Series*, vol. 1004, no. 1, 2018.

[12] P. Estrie, J. Falcou, M. Gaunard, J.-T. Laprest, and L. Lacassagne, "The numerical template toolbox: A modern c++ design for scientific computing," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3240–3253, 2014.

[13] G. Tagliavini, G. Haugou, and L. Benini, "Optimizing memory bandwidth in openvx graph execution on embedded many-core accelerators," vol. 2015-May, 2014.

[14] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, "A framework for optimizing openvx applications performance on embedded manycore accelerators," 2015, pp. 125–128.

[15] ——, "Optimizing memory bandwidth exploitation for openvx applications on embedded many-core accelerators," *Journal of Real-Time Image Processing*, vol. 15, no. 1, pp. 73–92, 2018.

[16] G. A. Elliott, K. Yang, and J. H. Anderson, "Supporting real-time computer vision workloads using openvx on multicore+GPU platforms," in *Real-Time Systems Symposium*, 2015, pp. 273–284.

[17] M. Yang, T. Amert, K. Yang, N. Otterness, J. H. Anderson, F. D. Smith, and S. Wang, "Making OpenVX really real time," 2018.

[18] G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, "Enabling OpenVX support in mw-scale parallel accelerators," in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2016*, 2016.

[19] Z. Guo, J. Han, and T. Li, "Implementing OpenVX on a polymorphous array processor," in *International Conference on Communication Technology Proceedings, ICCT*, vol. 2016-February, 2016, pp. 598–601.

[20] Z. Guo, J. Han, F. Che, and T. Li, "Parallel implementation of OpenVX on firefly2 gpu," vol. 1, 2016, pp. 218–221.

[21] Simulink, "S-Functions," https://it.mathworks.com/help/simulink/s-function-basics.html.

[22] B. M. Smith, L. Zhang, H. Jin, and A. Agarwala, "Light field video stabilization," in *International Conference on Computer Vision*, 2009, pp. 341–348.

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

[24] Matlab, "mex functions," https://it.mathworks.com/matlabcentral/fileexchange/26825-utilities-for-mex-files.

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

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

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

[28] 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.

[29] NVIDIA, "Deep-learning inference networks and deep vision primitives with TensorRT and NVIDIA Jetson," https://github.com/dusty-nv/jetson-inference.

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

[31] G. Klein and D. Murray, "Parallel tracking and mapping for small ar workspaces," in *2007 6th IEEE and ACM International Symposium on Mixed and Augmented Reality*, Nov 2007, pp. 225–234.