Enhancing Performance of Computer Vision Applications on Low-Power Embedded Systems Through Heterogeneous Parallel Programming

Stefano Aldegheri, Silvia Manzato, Nicola Bombieri
Department of Computer Science
University of Verona
Email: name.surname@univr.it

***Abstract*—Enabling computer vision applications on low-power embedded systems gives rise to new challenges for embedded SW developers. Such applications implement different functionalities, like image recognition based on deep learning, simultaneous localization and mapping tasks. They are characterized by stringent performance constraints to guarantee real-time behaviors and, at the same time, energy constraints to save battery on the mobile platform. Even though heterogeneous embedded boards are getting pervasive for their high computational power at low power costs, they need a time consuming customization of the whole application (i.e., mapping of application blocks to CPU-GPU processing elements and their synchronization) to efficiently exploit their potentiality. Different languages and environments have been proposed for such an embedded SW customization. Nevertheless, they often find limitations on complex real cases, as their application is mutual exclusive. This paper presents a comprehensive framework that relies on a heterogeneous parallel programming model, which combines OpenMP, PThreads, OpenVX, OpenCV, and CUDA to best exploit different levels of parallelism while guaranteeing a semi-automatic customization. The paper shows how such languages and API platforms have been interfaced, synchronized, and applied to customize an ORB-SLAM application for an NVIDIA Jetson TX2 board.**

I. Introduction

Computer vision is becoming pervasive in modern cyber-physical systems [1], [2]. Its main goal is the use of digital processing and intelligent algorithms to interpret meaning from images or video streams. Computer vision applications, in the context of cyber-physical systems, generally consists of several computational-intensive kernels that implement different functionalities, ranging from image recognition to simultaneously mapping and localization (SLAM).

Developing and optimizing such applications for an embedded system is not an immediate and simple task. Beside functional correctness, developers have to deal with non-functional aspects like performance, power consumption, energy efficiency and real time constraints. The optimization is architecture dependent and spans across two main dimensions: block-level and system-level. The first is more intuitive and involves the re-implementation/parallelization of single kernels for the target board accelerators (e.g., GPU, DSP, or multi-cores) through specific languages or programming environments like CUDA, OpenCL, Pthreads or OpenMP. The system-level optimization targets the overall system power consumption, memory bandwidth, and inter-process communication overhead. Mapping space exploration means exploring the different strategies to map each of such kernels to the right processing elements of the board and analyzing the corresponding impact on the design constraints.

OpenVX [3] is increasingly gaining consensus in the embedded vision community as programming environment and API library for system-level optimizations [4]. Such a platform is designed to maximize functional and performance portability across different hardware platforms, providing a computer vision framework that efficiently addresses different hardware architectures with minimal impact on software applications. Starting from a graph model of the embedded application, it allows for automatic system-level optimizations and synthesis on the target architecture by optimizing performance, power consumption and energy efficiency [5], [6], [7], [8].

Nevertheless, 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 a framework for heterogeneous parallel programming of embedded vision applications. It 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.

The paper presents an analysis of the limitations found by applying the state-of-the-art parallel programming environments to customize a modern SLAM application for the

978-1-5386-4756-1/18/$31.00

widespread NVIDIA Jetson TX2 board. Finally, it presents the results of the mapping space exploration we performed by considering performance, power consumption, energy efficiency, and result quality design constraints. The paper is organized as follows. Section II presents an introduction on OpenVX and the related work. Section III presents an analysis of parallel programming environments for embedded vision applications through a case study. Section IV presents the proposed framework. Section V presents the experimental results, while Section VI is devoted to the concluding remarks.

II. Background and related work

OpenVX is a framework to develop and optimize embedded vision applications by considering different design constraints (i.e., performance, power consumption). It relies on a graph-based model to define a high-level and architecture independent representation of the application. Such a representation is modularly built by the user through the use of a set of primitives, which are provided by the framework and that represent the most commonly used functionalities and data objects in computer vision algorithms, such as scalars, arrays, matrices and images, as well as high-level data objects like histograms, image pyramids, and look-up tables.

The high-level representation (i.e., the *graph*) is then automatically optimized and embedded thanks to the libraries of architecture-oriented implementation of the primitives and data-structures provided by the board vendor.

The developer defines a computer vision algorithm by instantiating kernels as nodes and data objects as parameters (see the example in Fig. 1). Each node of the graph is identified as a function kernel that can run on any processing unit of target heterogeneous board. Indeed, the application graph represents the partitioning of the whole application into blocks, which can be be executed across different hardware accelerators (e.g., CPU cores, GPUs, DSPs).

The programming flow starts by creating an OpenVX *context* to manage references to all used objects. Based on this context, the code builds the graph and generates all required data objects. Then, it instantiates the kernel as *graph nodes* and generates their connections. The framework first checks the graph integrity and correctness (e.g., checking of data type coherence between nodes and absence of cycles) and, finally, it processes the graph. At the end of the code execution, it releases all created data objects, the graph, and the context.

In the example of the Fig. 1, the application computes the gradient magnitude and gradient phase from a blurred input image. The *Magnitude* and *Phase* nodes are independently computed, in that each does not depend on the output of the other. OpenVX does not mandate that they are run simultaneously or in parallel, but it allows the runtime manager of the board vendor to decide on the mapping and execution strategy.

By adopting any vendor library that implements the graph nodes as Computer Vision primitives, OpenVX allows applying different mapping strategies between nodes and processing

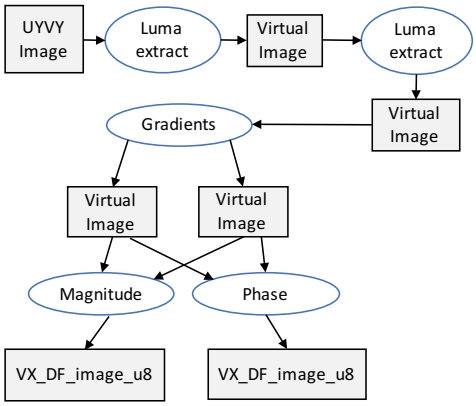

Fig. 1. OpenVX sample application (graph diagram)

elements of the heterogeneous board, by targeting different design constraints (e.g., performance, power, energy efficiency).

Different works have been presented to analyze the use of OpenVX for embedded vision [6], [7], [5], [8]. In [6], the authors 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. In [8], we proposed 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.

III. Analysis of parallel programming environments for embedded vision applications through the ORB-SLAM case study

In order to understand the limitations of the state-of-the-art environments for parallel programming embedded vision applications and the contribution of the proposed framework, we first present the case study, which will be used as a model in the subsequent sections. The case study, ORB-SLAM [9], represents a typical real embedded application, which is applied in different contexts, ranging from automotive to robotic systems. NVIDIA Jetson TX2, which is a widespread and low-cost embedded board, is the target platform.

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

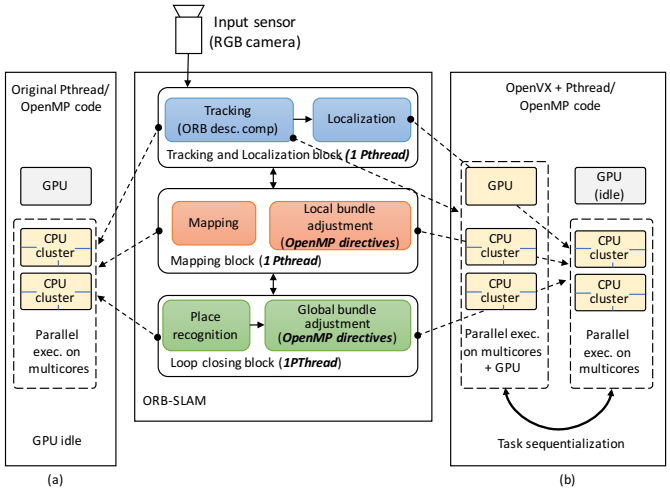

Fig. 2. Overview of ORB-SLAM application and execution models: (a) the original code (parallelized for multicore), (b) the state-of-the-art OpenVX implementation.

by considering the current map. The application consists of three main blocks (see Figure 2):

- 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 (detected map changes) sent by the localization block. In case of a well consolidated map, this module can be shut down to save system resources.
- The *loop closing* block aims at adjusting the scale drift error accumulated during the input analysis, which is unavoidable when adopting a monocular vision system (i.e., RGB camera). 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. As a consequence, the computation efficiency of this block (run on-demand) is crucial for the quality of the whole application results.

In the best ORB-SLAM implementation at the state of the art [9], due to their concurrent execution model, the three blocks are implemented to be run in parallel through PThreads on shared-memory multiprocessors. In addition, since the bundle adjustment task, both local in the mapping block and global in the loop closing block, can have long latencies, it is a primary target for parallelization. Its nested and data-independent loops well apply for directive-based *automatic* parallelization. Thus, the state of the art code is available with OpenMP directives for parallel execution on multi-cores. No block is originally considered for parallel execution on GPU (see Figure 2(a)).

The manual implementation of any sub-block for GPU is out of the scope of this work. Rather, due to the complexity of such a parallelization task for this application class yet considering different design constraints (power consumption and energy efficiency beside performance), we consider the semi-automatic *embedding* of the application through OpenVX.

We rely on standard libraries of computer vision functions, which are provided by the target board vendors (i.e., VisionWorks [10] for NVIDIA boards). The library can be extended through user-defined or third-party CUDA kernels, which are integrated in the OpenVX implementation as *custom nodes*.

On the other hand, due to the limitation of OpenVX to model complex applications through data-flow graphs and to the incompleteness of the vendor library, the OpenVX application has to be often integrated to standard C/C++ code. In the ORB-SLAM case study, only the tracking sub-block can be modelled through a data-flow graph and is worth to be optimized for CPU/GPU execution. Even though the rest of the code can still run on multicores, the two environment execution (OpenVX+CUDA and the rest) is sequentialized to allow for communication and synchronization, as explained in Section IV. The proposed method aims at integrating the two environments. Such an integration involves several advantages, such as, multi-level parallel execution of the application and better mapping space between tasks and processing elements to be explored.

IV. Method

Figure 3 depicts the overview of the proposed framework. We consider six different languages and parallel programming environments (*environments* in the following): C/C++, Pthreads, OpenMP, OpenCV, OpenVX, 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.

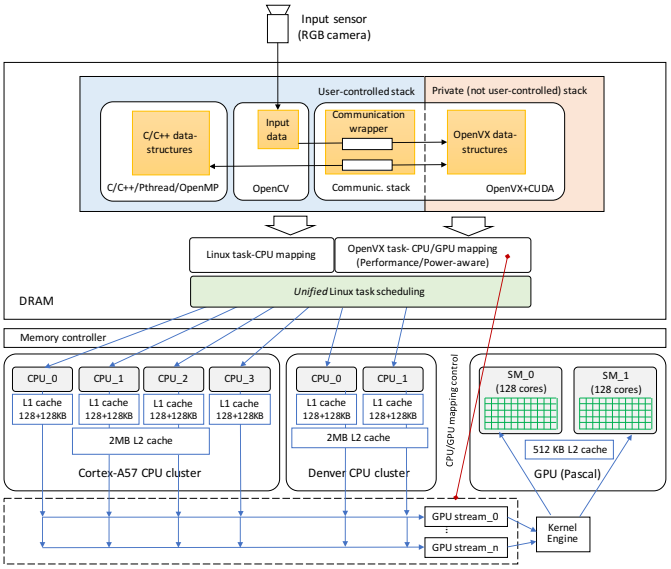

Fig. 3. Framework overview: memory stack, task mapping, and task scheduling layers of an embedded vision application developed with the proposed method on the NVIDIA Jetson TX2 board.

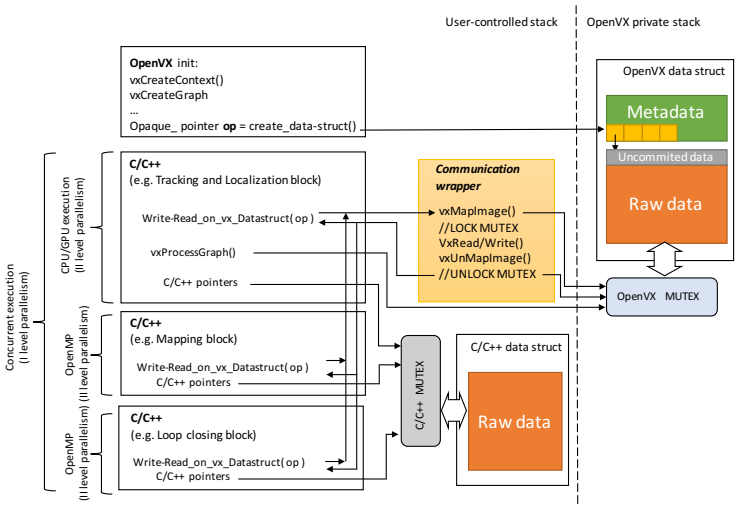

Fig. 4. Overview of the communication wrapper and its integration in the system.

The top of Figure 3 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.
- *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 4 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 safety 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 4, 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. Considering for example the ORB-SLAM case study, the first level of parallelism is implemented by the Pthreads, which run the three main modules of the application on different CPU cores.

Then, the tracking block of the first module is 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 (see the the example of Fig. 1), they are executed concurrently.

Finally, OpenMP provides another level of parallelism when a block is enriched with parallel directives (e.g., Mapping and Loop closing blocks in the example). Each of these blocks is executed in parallel by the threads generated automatically by the compiler, which run on the available CPU cores.

V. Experimental results

The framework has been applied to embed the ORB-SLAM application on the Jetson TX2 incrementally. We started from the most efficient parallel implementation at the state of the art [11]. We then integrated modularly the different parallel environments supported by the framework as follows:

- *Version 1 (Pthreads)*: It is the starting version [11], in which the three main blocks (Tracking and localization, Mapping, and Loop closing blocks) are run concurrently by Pthreads on the CPU cores.
- *Version 2 (Phtreds+OpenMP)*: It extends version 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 3* (Pthreads+OpenVX): It extends version 1 (i.e., with Pthreads, without OpenMP parallelism) by implementing the tracking sub-block in OpenVX.
- *Version 4* (Pthreads+OpenMP+OpenVX): It extends version 3 by enabling also OpenMP.
- *Version 5* (Pthreads+OpenVX+CUDA): starting from version 3, we reused a CUDA kernel that implements the *ORB* primitive in the tracking sub-block. We modularly replaced the corresponding OpenVX VisionWork primitive with such a more optimized kernel.
- *Version 6* (Pthreads+OpenMP+OpenVX+CUDA): It extends version 5 by enabling also OpenMP.

We validated and evaluated all the versions by using the *KITTI* dataset [12], which is a standard and widespread benchmark for vision applications. The dataset consists of video streams captured by driving around a car equipped with RGB camera in the mid-size city of Karlsruhe, in rural areas and on highways. For the sake of space, we present the results obtained on the sequence number 13, since it is the most meaningful to show the variance of workload in all the three blocks of ORB-SLAM and the corresponding effects on the design constraints.

For the evaluation, we set the Jetson TX2 board with two different configurations: medium frequency (75%) and maximum frequency (100%). They represent the frequency setting of 4 board components, i.e., the four cores Cortex-A57 cluster (1.42 GHz and 2.035 GHz as medium and maximum frequency, respectively), the two cores Denver cluster (1.42 GHz and 2.035 GHz), the GPU (1.032 GHz and 1.3 GHz), and the memory (1.062 GHz and 1.866 GHz).

Tables I and II show the results for the medium and maximum frequency setting, respectively. The best results are reported in bold. The mapping columns report the number of processing elements used by the different versions during computation. The Pthreads guarantee the minimum level of parallelism, by enabling one core per block. OpenMP has been set to use the maximum number of available CPU cores (6). The GPU is enabled only by OpenVX/CUDA.

Columns *FPS* and *Time per frame* report information about the application performance, and, in particular, *FPS* represents the maximum number of frames per second supported by the embedded system. The columns underline how each level of parallelism influences the overall performance.

To understand the effect of the different versions on power and energy efficiency, the tables report the total energy spent for the computation of the whole stream, the average and peak power, and the average energy per frame.

Finally, the tables report information about the quality of service (QoS) results. It includes the number of frames correctly processed against those skipped for the overloading of the processing elements. Frame skipping is caused by the mapping and loop closing blocks that run the bundle adjustment computation and, due to the work overload, their latency prevent the tracking block in analysing new frames. The maximum number of frames skipped tolerated is a design constraints, since it involves the QoS of the application like the number of times the system gets lost (see Section III).

The tables underline that, as expected, the performance (FPS) provided by the different versions are strictly correlated with the power consumption. Enabling all the processing elements through the different levels of parallelism leads to the best performance at the cost of the higher peak power. However, we found that OpenMP allows improving the performance in the overall heterogeneous context not in all cases. 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. On the other hand, version 6 provides better QoS in the maximum frequency configuration. This is due to the fact that OpenMP is strictly involved by the bundle adjustment phases, which affect the frame skipped while they do not affect the supported FPS. This does not happen in the medium frequency setting as the CPU frequency in which such a kernel is run does not allow the tracking block to respect the real time constraints.

We found that, for the medium frequency configuration, version 3 is the most energy efficient and provides the best QoS results. Version 5 provides the best performance and does not involve the worst power consumption. Version 6 provides the best performance, and it pays the almost best QoS (99.8%) with the highest power consumption.

TABLE I
AVERAGE FPS AND TIME PER FRAME VALUES ON KITTI, SEQUENCE 13, 75% OF THE FREQUENCIES

Version	Mapping			FPS	Time per frame (ms)	Energy (J)	Avg Power (W)	Peak power (W)	% frame processed	Energy per frame (J)
	A57	Denver	GPU SM							
Version 1	3	-	-	13.1	76.4	1,205	**3.37**	**4.05**	3,097/3,281 (94.4%)	**0.357**
Version 2	4	2	-	12.9	77.2	1,125	3.43	5.24	3,021/3,281 (92.1%)	0.372
Version 3	3	-	2	18.3	54.7	**1,039**	3.78	5.62	**3,279/3,281 (99.9%)**	0.378
Version 4	4	2	2	19.8	50.6	1,242	5.79	7.85	3,260/3,281 (99.4%)	0.381
Version 5	3	-	2	**23.2**	**43.2**	1,184	3.61	5.46	3,269/3,281 (99.0%)	0.362
Version 6	4	2	2	**23.2**	**43.2**	1,197	5.65	7.92	**3,271/3,281 (99.8%)**	0.366

TABLE II
AVERAGE FPS AND TIME PER FRAME VALUES ON KITTI, SEQUENCE 13, 100% OF THE FREQUENCIES

Version	Mapping			FPS	Time per frame (ms)	Energy (J)	Avg Power (W)	Peak power (W)	% frame processed	Energy per frame (J)
	A57	Denver	GPU SM							
Version 1	3	-	-	16.9	59.1	1,917	**5.84**	**8.54**	3,264/3,281 (99.5%)	0.587
Version 2	4	2	-	17.2	58.2	1,967	6.00	9.61	3,269/3,281 (99.6%)	0.602
Version 3	3	-	2	21.2	47.2	1,954	5.96	9.54	3,276/3,281 (99.8%)	0.597
Version 4	4	2	2	21.7	46.0	1,945	7.93	11.01	**3,280/3,281 (100%)**	0.593
Version 5	3	-	2	**29.3**	**34.1**	1,895	5.98	9.65	3,274/3,281 (99.0%)	0.579
Version 6	4	2	2	**29.2**	**34.3**	**1,843**	7.62	11.70	**3,279/3,281 (99.9%)**	**0.562**

For the maximum frequency configuration, version 5 provides the best tradeoff in terms of performance and power consumption, while version 6 provides the best tradeoff in terms of performance, energy efficiency, and QoS.

In conclusion, the experimental results show how the different versions and, for each of them a frequency configuration of the single processing elements, provide a very large mapping space to be explored (which is out of the scope of this work). Such a space can provide the best solution for each of the considered design constraints like performance, power consumption, energy efficiency, and quality of service.

VI. CONCLUSION

This paper presented a framework for heterogeneous parallel programming of embedded vision applications. The paper first presented an analysis of the actual limitations of the most meaningful and used environments for parallel programming embedded vision applications at state of the art when applied singularly. The paper then presented how the framework allows combining such environments, i.e., OpenMP, PThreads, OpenVX, OpenCV, and CUDA to best exploit different levels of parallelism while guaranteeing the semi-automatic customization. The paper presented the analysis and framework through a real case of study, i.e., an ORB-SLAM application, which has been customized for an NVIDIA Jetson TX2 embedded board. The paper finally showed that, thanks to the larger mapping space generated and the multi-level parallelism provided by such a heterogeneous parallel programming, we obtained sensibly better results over different design constraints i.e., performance, power consumption, energy efficiency, and result quality.

REFERENCES

[1] B. Meus, T. Kryjak, and M. Gorgon, "Embedded vision system for pedestrian detection based on hog+svm and use of motion information implemented in zynq heterogeneous device," vol. 2017-September, 2017, pp. 406–411.

[2] B. Z. C. Z. K. M. jizhong zhao; Nanning Zheng, "Hierarchical and parallel pipelined heterogeneous soc for embedded vision processing," vol. 99, no. 99, 2017.

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

[4] S. Aldegheri, D. D. Bloisi, J. J. Blum, N. Bombieri, and A. Farinelli, "Fast and power-efficient embedded software implementation of digital image stabilization for low-cost autonomous boats," in *Proc. of 11th International Conference Field and Service Robotics (FSR)*, 2017, pp. 129–144.

[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] S. Aldegheri and N. Bombieri, "Extending OpenVX for model-based design of embedded vision applications," in *Proceedings of the 2010 18th IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC*, 2017, pp. 1–6.

[9] 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.

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

[11] 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.

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