UNIVERSITY OF VERONA

DEPARTMENT OF COMPUTER SCIENCE

GRADUATE SCHOOL OF NATURAL SCIENCES AND ENGINEERING

DOCTORAL PROGRAM IN COMPUTER SCIENCE

CYCLE XXXII

A model-based design flow for embedded vision applications on heterogeneous architectures

S.S.D. ING-INF/05

Coordinator: _____
Prof. Massimo Merro

Tutor: _____
Prof. Nicola Bombieri

Doctoral Student: _____
Dott. Stefano Aldegheri

Abstract

The ability to gather information from images is straightforward to human, and one of the principal input to understand external world. Computer vision (CV) is the process to extract such knowledge from the visual domain in an algorithmic fashion.

The requested computational power to process these information is very high. Until recently, the only feasible way to meet non-functional requirements like performance was to develop custom hardware, which is costly, time-consuming and can not be reused in a general purpose. The recent introduction of low-power and low-cost heterogeneous embedded boards, in which CPUs are combine with heterogeneous accelerators like GPUs, DSPs and FPGAs, can combine the hardware efficiency needed for non-functional requirements with the flexibility of software development. *Embedded vision* is the term used to identify the application of the aforementioned CV algorithms applied in the embedded field, which usually requires to satisfy, other than functional requirements, also non-functional requirements such as real-time performance, power, and energy efficiency.

Rapid prototyping, early algorithm parametrization, testing, and validation of complex embedded video applications for such heterogeneous architectures is a very challenging task. This thesis presents a comprehensive framework that:

- is based on a *model-based paradigm*. Differently from the standard approaches at the state of the art that require designers to manually model the algorithm in any programming language, 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 the target hardware device.
- relies on a *polyglot parallel programming model*. The proposed model combines different programming languages and environments such as C/C++, OpenMP, PThreads, OpenVX, OpenCV, and CUDA to best exploit different levels of parallelism while guaranteeing a semi-automatic customization.
- optimizes the application performance and energy efficiency through a novel algorithm for the mapping and scheduling of the application

tasks on the heterogeneous computing elements of the device. Such an algorithm, called exclusive earliest finish time (XEFT), takes into consideration the possible multiple implementation of tasks for different computing elements (e.g., a task primitive for CPU and an equivalent parallel implementation for GPU). It introduces and takes advantage of the notion of *exclusive overlap* between primitives to improve the load balancing.

This thesis is the result of three years of research activity, during which all the incremental steps made to compose the framework have been tested on real case studies.

Acknowledgements

My primary acknowldegment goes to my advisor, prof. Nicola Bombieri, for both the academic and personal support.

The second thought reaches Sara, who provided the emotional support in the most difficult moments.

I want to acknowledge Barbara for helping me to keep pushing my limits, forbidding the rest on laurers.

I can't forget to give gratitude to my family, especially the remarkable patience of my mother, who understood deadlines importance and delayed the dinner preparation.

The almost final acknowledgment goes to the people in my office, with whom I spent the last three years, for the continuous and prolific information exchange colored with personal relationship.

Finally, I want to thanks my friends for the warm closeness showed to me, especially Gianluca, Alessandra and Emanuele.

Thank you all for helping me reaching this goal.

Contents

List of Figures

List of Tables

1

Introduction

Computer vision has gained an increasing interest as an efficient way to automatically extract of meaning from images and video. It has been an active field of research for decades, but until recently has had few major commercial applications. With the advent of high-performance, low-cost, energy efficient processors, computer vision 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, beside functional correctness, include performance, energy efficiency, dependability, real-time response, resiliency, fault tolerance, and certifiability.

Developing and optimizing a computer vision application for an embedded processor can be a non-trivial task. Considering an application as a set of communicating and interacting kernels, the effort for such application optimization goes over two dimensions: the single kernel-level optimization and the system-level optimization. Kernel-level optimizations have traditionally revolved around one-off or single function acceleration. This typically means that a developer re-writes 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. Differently from the standard design flows that require users to manually customize the application on the target hardware device, the OpenVX-based design flow starts from a graph model of the embedded application and it allows for automatic system-level optimizations and synthesis on the device targeting performance, and energy efficiency (see Fig. 1.1(a) and Fig. 1.1(b)) [5]–[7].

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

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

As a consequence, the integration of embedded vision applications with ROS-compatible systems is mandatory to guarantee code reuse, portability and system modularity.

This thesis proposes a model-based design flow and a comprehensive development framework for the design of embedded vision applications that addresses all the issues underlined above.

The framework relies on three main concepts, which are the main contributions of this thesis: A *model-based design flow*, a *polyglot parallel programming and integration model*, and an *algorithm for scheduling and mapping* of application tasks for performance enhancement.

Model-based design flow

Differently from the standard OpenVX-based design flow that require designers to manually model the algorithm through OpenVX code, the proposed design flow 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 on the target device (see Fig. 1.1(c)). The programmer starts to develop the application in a high-level language, possibly visual. At this stage, only the functional verification is needed. The developer can use all the enviroment functionalities to add the required validation, which will be carried through next iterations. The second step aims to perform an equivalent substitution of the primitives to match OpenVX primitives. After the conversion, the functional correctness of the program can be assessed by the previously

configured assertion. In the last step, the application is transferred to the real embedded board, and a final validation run ensure the behaviour match the requirements through the same acceptance test defined in the first step. The proposed design flow is presented in detail in Section 3.

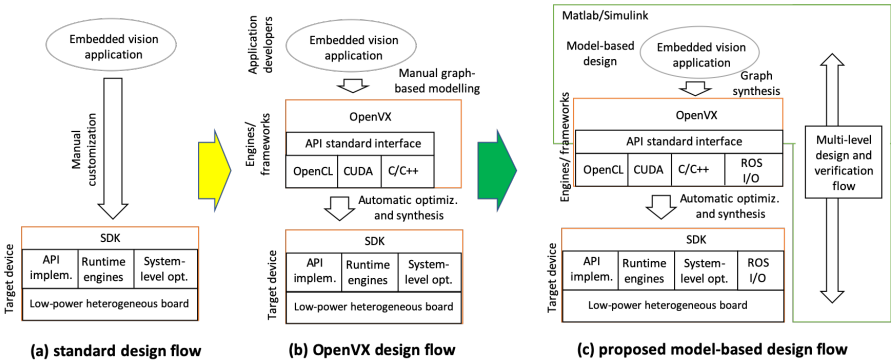


Fig. 1.1: The paradigm shift from manual customization, through OpenVX-based design flow, to the proposed model-based design flow.

Heterogeneous parallel programming model and integration

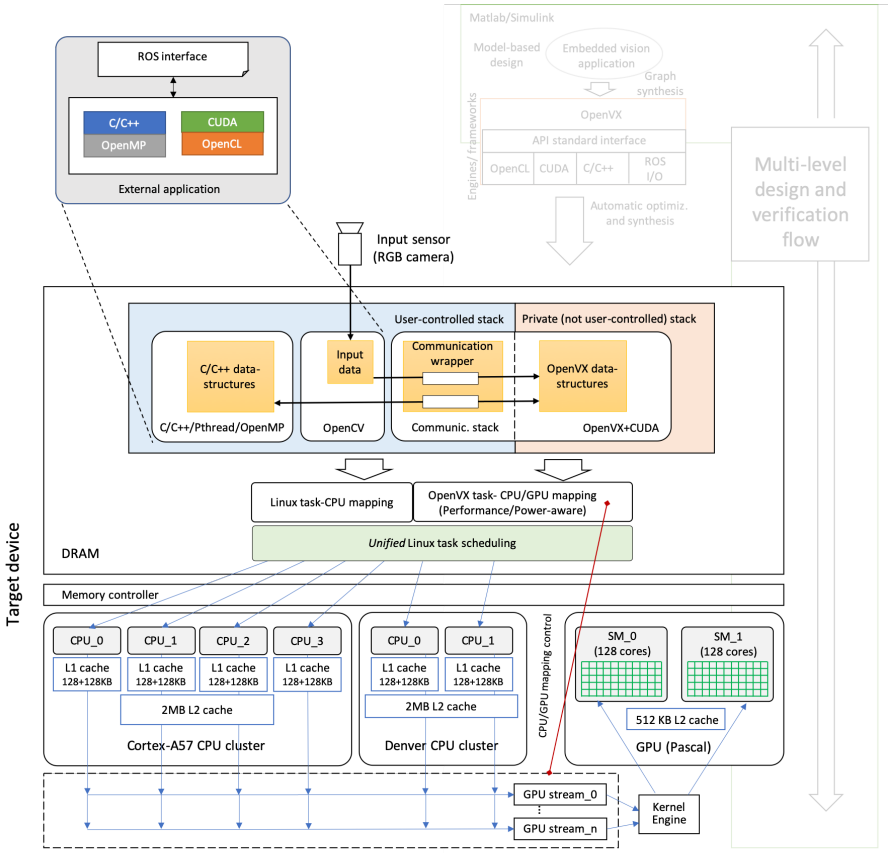


Fig. 1.2: An overview of the heterogeneous programming model the corresponding memory stack on the target device.

OpenVX 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. Nevertheless, it can be adopted to model only applications that can be represented through data-flow graphs. For this reason and for the fact that the vendor-provided libraries of OpenVX primitive are often incomplete, any real embedded vision application requires the integration of OpenVX with user-defined code (e.g., C/C++, CUDA, OpenCL, etc). In general, user-defined code can benefit from parallelization techniques for multi-cores through heterogeneous parallel environments

(i.e., multi-core + GPU parallelism). Nevertheless, such an integration is not straightforward since the memory stack of OpenVX is private and not user-controllable. At the state of the art, the only possibility to integrate OpenVX code with user-defined code is to sequentialize the different execution environments, with a consequent strong impact on the system-level optimization. To address this issue, the second main contribution of this thesis is a model to integrate polyglot parallel programming environments for efficient execution on the target device (see Fig. 1.2). The model allows combining multiple programming environments, i.e., OpenMP, PThreads, OpenVX, OpenCV, and CUDA to best exploit different levels of parallelism while guaranteeing the semi-automatic customization. The model is presented in detail in Section 4.

An algorithm for scheduling and mapping of application tasks for performance enhancement

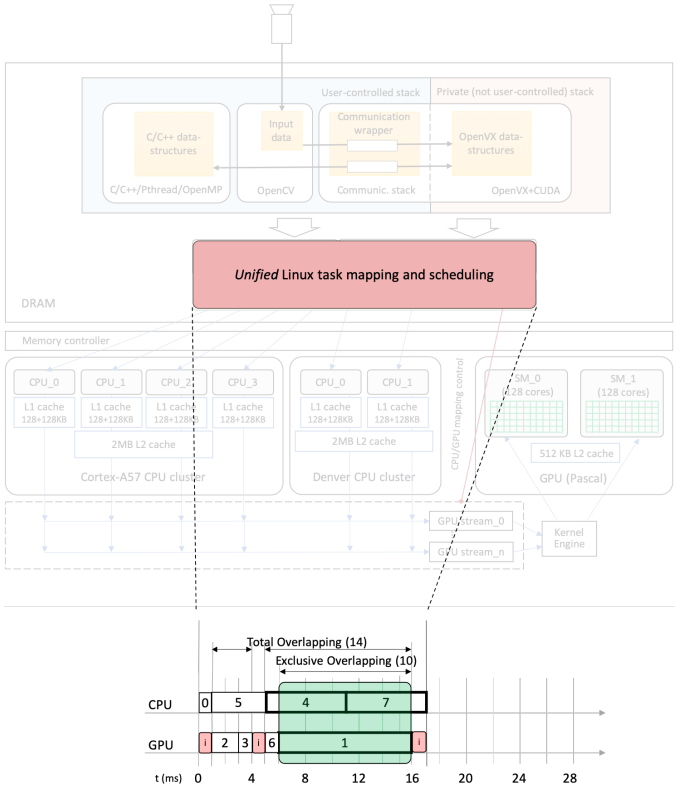


Fig. 1.3: The proposed algorithm for scheduling and mapping of application tasks and its integration in the design flow.

Prior research efforts attempted to optimize the performance of the code generated from OpenVX toolchains. They proposed techniques to implement different data access patterns such as DAG node merge, data tiling, and parallelization via OpenMP. There also have been efforts to make the OpenVX task scheduling deliver real-time guarantees. Nevertheless, there is no prior work that focuses on efficient mapping strategies and its corresponding scheduling of OpenVX (DAG-based) applications for heterogeneous architectures. Prior approaches that propose mapping strategies for OpenVX considered each DAG node to have only one exclusive implementation (e.g., either GPU or CPU), and the mapping is driven by the availability of the node's implementation in the library: if a node has a GPU implementation then it is mapped on the GPU. Otherwise it is mapped on a CPU core.

To take into consideration the heterogeneity of the target architectures, the possible multiple implementations of DAG nodes, and the problem complexity, the heterogeneous earliest finish time (HEFT [9]) heuristic for static mapping and scheduling of OpenVX applications has been implemented. As confirmed in the experimental analysis, such a HEFT implementation sensibly outperforms (i.e., up to 70% of performance gain) the state-of-the-art solution currently adopted in one of the most widespread embedded vision systems (i.e., NVIDIA VisionWorks on NVIDIA Jetson TX2). Nevertheless, this thesis experimentally proves that such a heuristic, when applied to DAG graphs for which not every node has multiple implementations, can lead to idle periods for the computing elements (CEs). Since not having multiple implementations for all nodes happens in a majority of real embedded vision contexts, this work proposes, as the third main contribution, an algorithm called XEFT that reorganizes the HEFT ranking to improve the load balancing.

The main idea of the XEFT is to schedule, in the same timespan, primitives which have the implementation exclusively for one accelerator, called *exclusive nodes*. It starts with the original ranking generated by HEFT, and it re-organizes such exclusive nodes by putting them close together forming various clusters, respecting DAG precedence topology. Due the load-balancing property of HEFT, the probability such ordering will enhance the *exclusive overlapping* (i.e., the overlapping of exclusive nodes) is higher. Fig. 1.3 shows an example of a scheduling with the various overlapping highlighted. As confirmed by the experimental results conducted on a very large set of real and synthetic benchmarks, XEFT can improve the system performance up to 33% over HEFT, and 82% over the state of the art approaches. XEFT is presented in detail in Section 5.

2

Background and related work

In this thesis, three major points will be analyzed: an enhanced model-based design flow, the integration of several applications and languages, and finally a new scheduler specifically suited for the lacking of implementation for specific accelerators. This chapter serves to summarize state-of-the-art techniques, and to show the possible improvement analyzed later in the respective section.

2.1 Model-based

Usually, vision applications are built using libraries. *OpenCV* is a popular open source library of primitives for computer vision. It comprises a comprehensive set of over 2,500 functions ranging from simple building-block functions such as matrix arithmetic functions to substantial computer vision modules such as object detection and image stabilization. OpenCV enables developers to quickly implement and test sophisticated computer vision algorithms. A subset of primitives are implemented in CUDA or OpenCL to be accelerated on a GPU.

Since the target for this thesis is the embedded platform NVIDIA Jetson TX2, a closed-source porting of such library called *OpenCV4Tegra* was used, provided by NVIDIA specifically optimized for the Tegra architecture. However, during the performance optimization of a computer vision system, platform-level bottlenecks require a lot of work to be identified, and the solution is platform-specific. Traditional methods are not well suited to address these issues. OpenVX has been proposed to address such system-level issues by means of a graph-based paradigm [3]. 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* [10] 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. [7] 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 [11]. 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* [5], 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 [12]–[14].

Glenn et al. [15] 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 [6].

Yang et al. [16] 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 [17]–[19]. *VisionWorks* is the NVIDIA closed-source porting to Tegra architecture. While it provide excellent kernel-level acceleration, experimental evaluations show a simple scheduling mechanism: if a primitive has a GPU implementation it is mapped there, otherwise it is executed on the CPU. Since the GPU implementation for computer vision primitives are more efficient than the CPU one, it is a good strategy if the nodes are executed sequentially, like in the VisionWorks runtime. However, there are a level of parallelism between CPU and GPU that could be exploited to increase the application performance.

2.2 Polyglot framework and integration

In this thesis, the integration is analyzed in two directions. The first one focuses on intra-application combination of multiple languages. The second one highlight the communications between two applications that have to exhange data.

2.2.1 Polyglot programming languages for heterogeneous accelerators

Embedded applications requires to pack efficient solutions with a limited power budget. The key to achieve these results is to use several accelerators to increase the performance-per-watt metric. However, each computing element must be correctly programmed and fine-tuned to express the maximum efficiency. Due to different programming paradigm and architectural changes, an ad-hoc language is more efficient than a general one [20]–[22]. The choice of the programming language is thus dependant on the available accelerators and OS support. In this thesis, other than the basic C language used to program the CPU, three others parallel languages were used:

- *Pthread*. It implements a MIMD (Multiple Instruction on Multiple Data) architecture. It is targeted for CPU-level parallelism, and it is useful to perform several different tasks at once. For example, a core could sort an array while another one is computing the least square analysis on another set of points. Albeit being a powerful tool, deadlock situations could happen if synchronization is not managed properly.
- *OpenMP*: this parallel framework distribute work in a shared memory system. Up to version 4.5, it could only target CPU multi-core parallelism. From this release, it can also target GPU offloading. It consists in code-level annotation to mark sections that can be parallelized. During execution, a runtime system will dinamically allocate and map threads based on available resources. [23]
- *CUDA*: in 2007, NVIDIA introduced this proprietary C-like language to exploit their hardware for General Purpose GPU (GPGPU) programming and not only for graphics processing [24]. GPU architectures are very efficient to implement SIMD instructions, since they were designed for this use case. Albeit having less powerful cores than a CPU, they usually outnumber them by 2 or 3 order of magnitude. This solution is useful for problems which are highly parallelizable.
- *OpenCL*: OpenCL language that aims to provide an homogeneous code-base for various accelerators [25]. It provides a C-like language targeting both CPUs and GPUs. There are two reasons this language is not used in this thesis. First of all, at the moment of the writing it is not officially supported on the Tegra architecture, which is the target of the experiments. Second reason is performance related: programs written in CUDA are more efficient than OpenCL ones [21]. This is not surprising, since CUDA is released by NVIDIA that can adapt it to best express in a high-level language the hardware capability of their architecture.

2.2.2 Communication between modules

To connect two different applications, an interface must be specified to delimit input/output boundaries. ROS [8] is the de-facto standard for the decentralized communication in robotic systems. The data exchange is performed by a message passing mechanism and a publish-subscribe communication using a unique string as identifier called *topic*. A *node* is the ROS object which is responsible to feed or consume the data served over a topic. Each node registers itself on a *roscore*, which is a

network-reachable service which is responsible to manage the nodes requests. When a node want to publish data over a topic, a request is sent to the roscore. It notifies the pending nodes that requested the subscription for that topic. It then provides to both the nodes the information to open a direct connection. Using this approach, the orchestration is centralized, but the communication itself is decentralized.

The adoption of ROS provides different advantages. First, it allows the platform to model and simulate blocks running on different target devices. Then, it implements the inter-node communication in a modular way and by adopting a standard and widespread protocol, thus guaranteeing code portability.

2.3 Scheduling on heterogeneous architectures

There has been extensive prior research in task scheduling for multi/many cores at different levels of abstractions over the last decade. We kindly refer the reader to an extensive overview of mapping and scheduling strategies that can be found in [26]. Considering our target applications (i.e., embedded vision), we limit our focus on the class of static scheduling for heterogeneous architectures, for which we summarize the most recent and related works next.

TETRiS [27] is a run-time system for static mapping of *multiple applications* on heterogeneous architectures. It takes advantage of compile-time information to map and migrate applications by preserving the predictable performance of using static mappings. Nevertheless, it does not apply to DAG-based applications, and it does not support the concept of *multiple* (i.e., one implementation of a node for each CE) and *exclusive* implementations (i.e., the implementation of a node for a given CE) of nodes for heterogeneous CEs (e.g., CPUs and GPUs).

We consider an application being represented by a directed acyclic graph (DAG), $G = (V, E)$, where V is the set of v tasks and E is the set of e edges between the tasks (we use the terms *task* and *node* interchangeably in the thesis). Each edge $(t, q) \in E$ represents the precedence constraint such that task t should complete its execution before task q starts.

In [28] first, and then in [29], the authors proposed an approach and its optimization to schedule DAG-based OpenVX applications for multi-cores and GPU architectures. Their approach allowed the application performance to be increased by overlapping sequential executions of the application. On the other hand, it does not consider the multiple implementations of DAG nodes, i.e., the mapping algorithm targets the best *local* solution: If there exists a GPU kernel for a DAG node then that node is mapped onto the GPU.

To support the mapping of each DAG node onto one CE among different heterogeneous possibilities we consider the heterogeneous earliest finish time (HEFT) algorithm [9]. HEFT schedules tasks in two phases. The first is the *task prioritizing phase*, in which the tasks are ranked according to each task's priority as follows[1]:

[1] In this work, we consider the *upward* ranking [9] since it has shown to provide the best results for our graph characteristics. However, the optimization based on the exclusive overlap is independent from any ranking methodology.

$$rank(t) = \overline{w_t} + \max_{q \in succ(t)} (\overline{c_{t,q}} + rank(q)), \quad (1)$$

where $succ(t)$ is the set of immediate successors of task t, $\overline{c_{t,q}}$ is the average communication cost of edge (t, q), and $\overline{w_t}$ is the average computation cost of task t.

The second is the *processor selection phase*, in which each task t on the rank list is mapped onto the CE that minimizes the finish time of t.

In [30], comparison between several heuristic has been proposed ranking them by several factor, especially for primitive numbers and core count. Due the fact this work is targeted for embedded platform with low number of core, HEFT has comparable performance to more complex heuristic like CEFT [31] or PEFT [32]. Even if HEFT suffers from the deep difference between CPUs and GPUs [33], optimizations shown its portability and potentiality for embedded multi/many-core architectures [34]–[36].

3

Model-based design flow

This section consists of three parts. First, it presents a quick overview about how to write a OpenVX program (Section 3.1). Then, it presents a real case study in which an OpenVX-based design flow has been applied, by underlining the advantage of such an approach versus the manual code refinement (see Section 3.2). Finally, it presents the proposed model-based design flow (see Section 3.3).

3.1 Overview of a OpenVX program

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. 3.1 and Listing 3.1 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 3.1). 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.

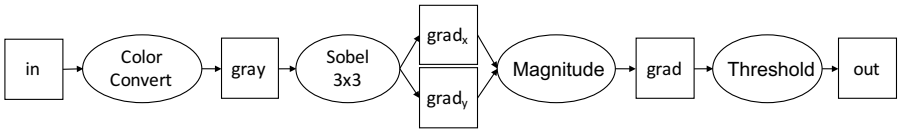


Fig. 3.1: OpenVX sample application (graph diagram)

```
1  vx_context c = vxCreateContext();
2  vx_graph g = vxCreateGraph(context);
3  vx_enum type = VX_DF_IMAGE_VIRT;
4  /* create data structures */
5  vx_image in = vxCreateImage(c, w, h, VX_DF_IMAGE_RGBX);
6  vx_image gray   = vxCreateVirtualImage(g, 0, 0, type);
7  vx_image 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 3.1: OpenVX code of the example of Fig. 3.1

3.2 The OpenVX-based design flow

To explain the advantages of adopting OpenVX in a design flow of embedded vision application this section introduces a real case study as a running example. Such a benchmark implements a stabilization algorithm for digital image streams. In particular, the application is applied to a visual stream captured by a camera mounted on a small ASV (Autonomous Surface Vehicles) [37], [38]. Since embedded hardware to damp the camera effects is generally expensive, a software implementation of the digital stabilization is preferable.

An unstabilized video stream is an image sequence that exhibits unwanted perturbations in the apparent image motion. The goal of digital video stabilization is to improve the video quality by removing unwanted camera motion while preserving the dominant motions in the image sequence. As an example, for obtaining an obstacle detection solution, stabilization is a crucial pre-processing step before performing higher-level processing like object tracking. Stabilization is necessary since the accuracy of predicted object trajectories can decrease in case of unstabilized image streams [15].

Fig. 3.2 shows an overview of the adopted video stabilization algorithm, which is represented through a dependency graph. The input sequence of frames is taken from a high-definition camera, and each frame is converted to the gray-scale 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 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. 3.2 denote inter-frame dependencies, i.e., parts of the algorithm where a temporal window of several frames is used to calculate the camera translation.

Although this algorithm does not represent particular challenges for the sequential implementation targeting CPU-based embedded systems, it presents a large design space to be explored when implemented for hybrid CPU-GPU systems. On the one hand, several primitives of the algorithm (graph nodes) can benefit from GPU acceleration while, on the other hand, their *offloading* on GPU involves additional memory-transfer overhead. The mapping exploration between nodes and computational elements (i.e., CPU or GPU) is thus crucial both for the performance and for the energy consumption.

To best explore correctness, performance, and energy consumption of the algorithm, we implemented the software in all the possible configurations (nodes vs. CPU/GPU) and by adopting both a standard design flow based on the OpenCV library and an OpenVX-based design flow.

3.2.1 Imperative versus data-flow implementation

The difference between OpenCV and OpenVX relies on their execution paradigm. OpenCV encourages an imperative programming approach, while OpenVX imposes

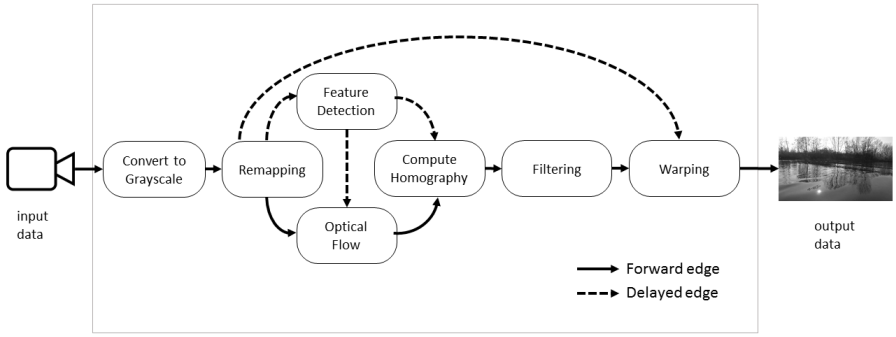


Fig. 3.2: Dependency graph of the video stabilization algorithm.

a data-flow paradigm. Although both models are semantically equivalent, standard design flows rely on the imperative approach. Moving to the data-flow paradigm can pose several limitations, as discussed in the next sections. The following two paragraphs present and compare the video stabilization implementation in the imperative and data-flow approaches, by underlining the results in terms of obtained performance.

An imperative implementation for standard OpenCV-based design flows

OpenCV provides the implementation for CPU of all the primitives of the video stabilization algorithm. In addition, it provides the GPU implementation of the following five nodes:

- Convert to Grayscale.
- Remapping.
- Feature detection (either based on Harris or FAST algorithm).
- Optical Flow.
- Warping.

The complete design space exploration of OpenCV consists of 32 configurations with the Harris-based feature detection plus 32 configurations with the FAST-based one. We exhaustively implemented and compared all the possible configurations. We also conducted the system-level optimization for each configuration, which, by adopting OpenCV, is a manual and time consuming task. Indeed, although any single function downloading on GPU requires a quite straightforward code intervention (i.e., a function signature replacement), the system-level optimization involves a more accurate and time consuming analysis of the CPU-GPU data dependency in the overall data flow. As an example, consider the three nodes *feature detection*, *compute homography*, and *optical flow*. Any configuration requiring the first mapped on the CPU and the others on the GPU involves one data transfer from the CPU main memory (the output of the feature detection) to the GPU main memory (as input for either the optical flow or homography). A second (useless) CPU-GPU data transfer leads, in this algorithm implementations, to a 15% performance loss. Finding such data dependency and optimizing all the CPU-GPU data transfer, in the current OpenCV release, is let to the programmer.

The profiling analysis of all these code versions underlines that the *feature detection* and *optical flow* nodes are the two most computational demanding functions of the algorithm. Fig. 3.3 depicts their OpenCV structure, by underlining how they are implemented (in terms of data exchange structures and the primitive signature) if run on the CPU or offloaded on the GPU. Their mapping on CPU or GPU involves the main important differences from the performance and power consumption point of view, as shown in section 3.2.2.

Even though OpenCV primitives for GPUs are implemented both in OpenCL and CUDA, only CUDA implementations can be adopted for the Jetson board.

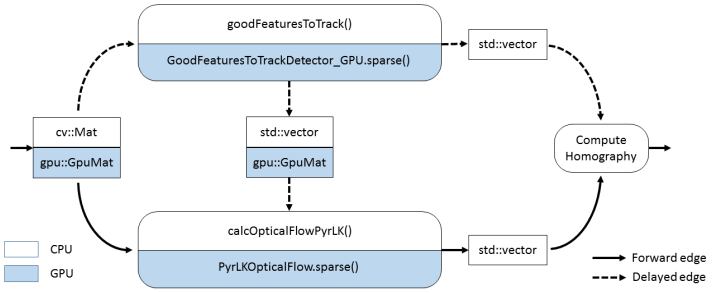


Fig. 3.3: OpenCV implementation of the most computational demanding nodes of the video stabilization algorithm.

A data-flow implementation for OpenVX-based design flows

Listing 3.2 shows the most important parts of the OpenVX code. Some primitives have been tested both with the version released in the standard OpenVX library and in the VisionWorks library.

```
1  vx_context context = vxCreateContext();
2  /* create data structure */
3  vx_image gray = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_U8);
4  vx_image rect_image = vxCreateVirtualImage(graph, 0, 0, VX_DF_IMAGE_U8);
5  vx_array curr_list = vxCreateVirtualArray(graph, VX_TYPE_KEYPOINT, 1000);
6  vx_matrix homography = vxCreateMatrix(context, VX_TYPE_FLOAT32, 3, 3);
7  /* create graph and relative structure */
8  vx_graph graph = vxCreateGraph(context);
9  vxColorConvertNode(graph, frame, gray);
10 vx_node remap_node = vxRemapNode(graph, gray, rect_image,
       VX_INTERPOLATION_BILINEAR, remapped);
11 nvxCopyImageNode(graph, rect_image, out_frame, 0));
12 vxGaussianPyramidNode(graph, remapped, new_pyramid);
13 vx_node opt_flow_node = vxOpticalFlowPyrLKNode(graph,old_pyramid, new_pyramid,
       points, points, curr_list, VX_TERM_CRITERIA_BOTH, s_lk_epsilon,
       s_lk_num_iters, s_lk_use_init_est, s_lk_win_size);
14 nvxFindHomographyNode(graph, old_points, curr_list, homography,
       NVX_FIND_HOMOGRAPHY_METHOD_RANSAC, 3.0f, 2000, 10, 0.995f, 0.45f, mask);
15 homographyFilterNode(graph, homography, current_mtx, curr_list, frame, mask);
16 matrixSmootherNode(graph, matrices, smoothed);
17 truncateStabTransformNode(graph, smoothed, truncated, frame, s_crop);
18 vxWarpPerspectiveNode(graph, frame_out_sync, truncated,
       VX_INTERPOLATION_TYPE_BILINEAR, frame_stabilized);
19 nvxHarrisTrackNode(graph, rect_image, new_points, NULL, curr_list, 0.04, 3, 18,
        NULL);
20 /* force node to GPU */
21 vxSetNodeTarget(remap_node, NVX_TARGET_GPU, NULL);
22 /* force node to CPU */
23 vxSetNodeTarget(opt_flow_node, NVX_TARGET_CPU, NULL);
```

Listing 3.2: OpenVX Code Example

The programming flow starts by creating a context (line 1). Based on this context, the program builds the graph (line 8) and the corresponding data objects (lines 3-6). The whole algorithm is then finalized as a dataflow graph by linking data objects through nodes (lines 9-19). Lines 21 and 23 show how processing nodes can be manually forced to be mapped on specific processing elements.

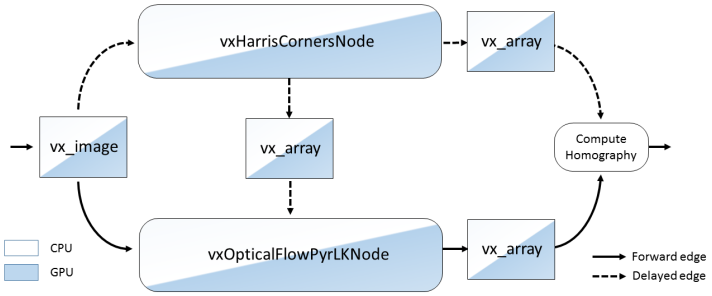


Fig. 3.4: OpenVX implementation of the most computational demanding nodes of the video stabilization algorithm.

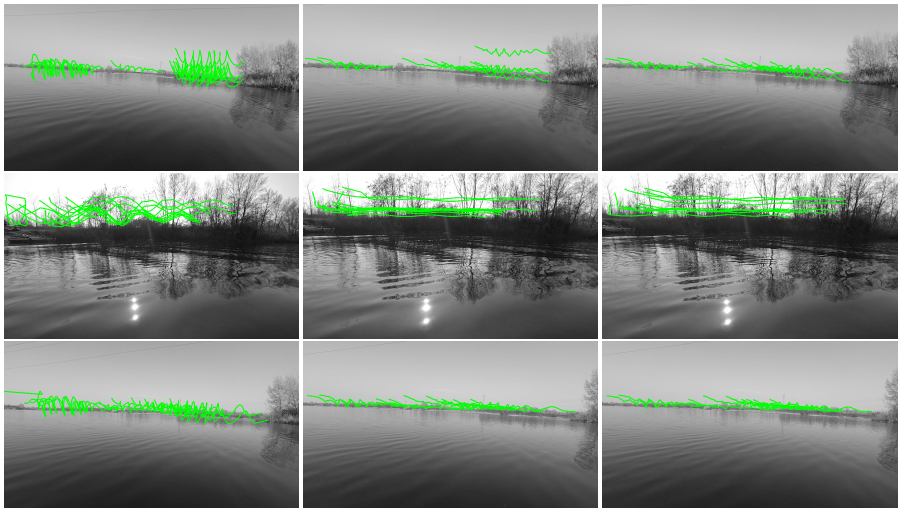
The OpenVX environment allows automatically changing the nodes-to-processing elements mapping and the corresponding data exchange system-level optimization. It also provides both the Harris-based and FAST-based feature detector, both available for CPU and GPU. In particular, it provides two different versions for each primitive: the first one is the standard "general purpose" OpenVX version, while the second one is provided in the VisionWorks library and is optimized for tracking algorithms.

In order to verify the best configuration targeting performance and to figure out the best one targeting power efficiency, we developed and tested all the possible configurations by forcing the nodes-to-processing elements mapping.

In the OpenVX mapping exploration, we considered the following differences from OpenCV:

1. Harris/FAST tracker: OpenVX/VisionWorks provides the Harris/FAST tracker, which optimizes the flow of data by giving priority to the points tracked in the previous frame instead of the new detected ones, if they are in the same area.
2. OpenVX relies on column-major matrices, while OpenCV relies on row-major matrices. This is important especially in the remap cases, where the OpenCV backend is used to build the coordinates of the remapped points.
3. VisionWorks relies on a delay object to store an array of temporal objects (e.g., N frames back). This is not possible in OpenCV.

Fig. 3.5: IntCatch 2020 project uses Platypus Lutra boats, about $1m$ long and $0.5m$ wide.



(a) (b) (c)

Fig. 3.6: Video stabilization results on sequences S1 (first row), S2 (second row), and S3 (third row). (a) A frame in the unstabilized video overlayed with lines representing point trajectories traced over time. (b) The corresponding frame in the OpenCV stabilized video. (c) The OpenVX stabilization results. Point trajectories are significantly smoother when the stabilization is activated.

3.2.2 Qualitative and quantitative results

Experiments have been carried out on real data collected in a small lake near Verona, Italy with a GoPro Hero 3 Black camera mounted on the bow of the Platypus Lutra boat (see Fig. 3.5). In particular, we analyzed three different image sequences (see Fig. 3.6), registered at 60 FPS with 1920×1080 wide angle resolution. Sequence S1 is particularly challenging due to large rolling movements, while Sequence S2 presents strong sun reflections on the water surface, and the boat is very close to the coast. The last Sequence S3 presents a similar view-point with respect to S1, but with lower rolling. The three sequences can be downloaded from the IntCach AI website[1], together with the source code of the different implementations that has been considered. Additional sequences, not analyzed in this work, can be downloaded from the *IntCatch Vision Data Set*[2].

Stabilization results. In order to show the importance of video stabilization as a necessary preprocessing step for horizon line detection, we have considered the three sequences in Fig. 3.6, using them as input for a feature tracking algorithm. Supplemental videos can be downloaded at `goo.gl/mg4fH1` for a clearer demonstration of our results.

[1] `http://profs.scienze.univr.it/~bloisi/intcatchai/intcatchai.html`

[2] http://profs.sci.univr.it/bloisi/intcatchvisiondb/intcatchvisiondb.html

Table 3.1: Stabilization quantitative results

Sequence	Number of tracked points	MAD		
		Unstabilized	OpenCV	OpenVX
S1	10	31.5	11.3	**10.4**
S2	10	25.0	8.2	**6.8**
S3	10	23.4	**10.4**	10.7

We used the well-known Kanade-Lucas-Tomasi feature tracker (KLT) for tracking points around the horizon line in the camera field of view. The obtained feature points are visualized by tracing them through time [39]. The first column in Fig. 3.6 contains the results when the input consists of images that have not been stabilized. The second and the third column show the results generated by the OpenCV and OpenVX based implementation, respectively.

We use the median absolute deviation with median as central point (MAD) to estimate the stabilization quality by measuring the statistical dispersion of the points generated by the KLT algorithm:

$$MAD = median\left(|x_i - median(X)|\right) \tag{3.1}$$

where $X = \{x_1, x_2, ..., x_n\}$ is the n original observations.

Table 3.1 shows the MAD values obtained with the unstabilized images and with the preprocessed data generated by OpenCV and OpenVX. The stabilization allows to obtain lower MAD values. It is worth noticing that the alternatives generated by the same environment (i.e., OpenCV or OpenVX) give comparable results in terms of MAD, while differ in terms of performance and power consumption as discussed in the next paragraph.

Computational load results. Multiple mappings between routines and processing elements have been evaluated using different metrics. Live capture has been simulated from recorded video stored in the Jetson internal memory by skipping the frames in accordance with the actual processing speed of the considered implementation.

Tables 3.2 and 3.3 show the obtained results in terms of frames per second (FPS) together with the following metrics:

$$P_{peak} = \max_t P(t) \tag{3.2}$$

Table 3.2: OpenCV implementation results

Remap	Cvt color	Warping	Optical Flow	Features	P_{avg} (W)	P_{peak} (W)	FPS	$E(T_{end})$ (J)
GPU	GPU	GPU	GPU	CPU/HARRIS	**5.1**	**11**	7.5	**772**
GPU	GPU	GPU	GPU	GPU/FAST	5.7	15	**16.3**	855
GPU	GPU	GPU	GPU	GPU/HARRIS	6.0	17	15.3	906
GPU	GPU	GPU	GPU	CPU/FAST	5.3	15	13.7	810

Table 3.3: OpenVX implementation results

	Remap	Cvt color	Warping	Optical Flow	Features	P_{avg} (W)	P_{peak} (W)	FPS	$E(T_{end})$ (J)
	GPU	GPU	GPU	CPU	GPU/HARRIS OPENVX	5.4	12	36.5	810
	GPU	GPU	GPU	CPU	GPU/FAST VISIONWORKS	5.1	**11**	15.3	760
	GPU	GPU	GPU	GPU	CPU/FAST VISIONWORKS	**4.3**	**11**	16.2	**639**
*	GPU	GPU	GPU	GPU	GPU/FAST OPENVX	5.3	12	**60**	804
*	GPU	GPU	GPU	GPU	GPU/HARRIS OPENVX	5.2	**11**	**60**	776
*	GPU	GPU	GPU	GPU	GPU/FAST VISIONWORKS	5.2	12	**60**	780
*	GPU	GPU	GPU	GPU	GPU/HARRIS VISIONWORKS	5.1	**11**	**60**	764

$$E(t) = \int_0^t P(t) \tag{3.3}$$

$$P_{avg} = \frac{E(T_{end})}{T_{end}} \tag{3.4}$$

where $P(t) = V(t)I(t)$ is the instant power usage and T_{end} is the total duration of the analyzed sequence in seconds. Table 3.3 has an asterisk at the beginning of rows which correspond to configurations that have been chosen by Visionworks automatically. The value $E(T_{end})$ denotes the absolute battery consumption measured in Joule (J), P_{avg} gives a measure of the average instant power usage during computation measured in Watt (W), and P_{peak} is the maximum instant power usage in Watt.

A Powermon board [40] has been used to measure the energy consumption. The absolute value of energy is an important measure that determines the battery capacity required for a mobile platform to perform the work. The number of processed frames is equally important to understand how well the algorithm performs. It is worth noticing that the OpenCV implementation is not able to achieve real-time performance with high resolution high frame rate data.

We investigated the maximum value of FPS that different configurations can reach and the corresponding power consumption. We also investigated how the power consumption scales with the performance. Since no OpenCV implementation allows achieving real-time performance (60 FPS), we used a high resolution timer to simulate the actual capture rate of the camera.

The results underline that the best OpenCV implementation allows reaching 16.3 FPS at the cost of 855 J energy consumption. They also underline that OpenCV does not allow selecting between performance-oriented or energy-saving mode, since the most appropriate energy saving configuration (-10% J) provides very insufficient performance. In contrast, OpenVX allowed us to implement four different config-

urations that guarantee appropriate performance (60 FPS) and energy scaling over FPS, as underlined in Fig. 3.7.

3.3 Extending OpenVX for Model-based Design of Embedded Vision Applications

Fig. 3.8 depicts the overview of the proposed design flow. The computer vision application is firstly developed in MATLAB/Simulink. The choice for the MATLAB platform is due to convenience, since it contains a huge amounts of vision-related functions that allows a quick prototyping of the application. Moreover, it contains several libraries for other areas (e.g. aeronautics), upon this methodology could be applied. For our use case, we exploit a computer vision oriented toolbox of Simulink[3]. 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 [41] (e.g., user-defined block UDB $Block_4$ in Fig. 3.8). 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., [39]). Efficient test patterns are extrapolated, by using any technique of the literature, to asses the quality of the application results by considering the adopted validation metrics.

[3] 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.

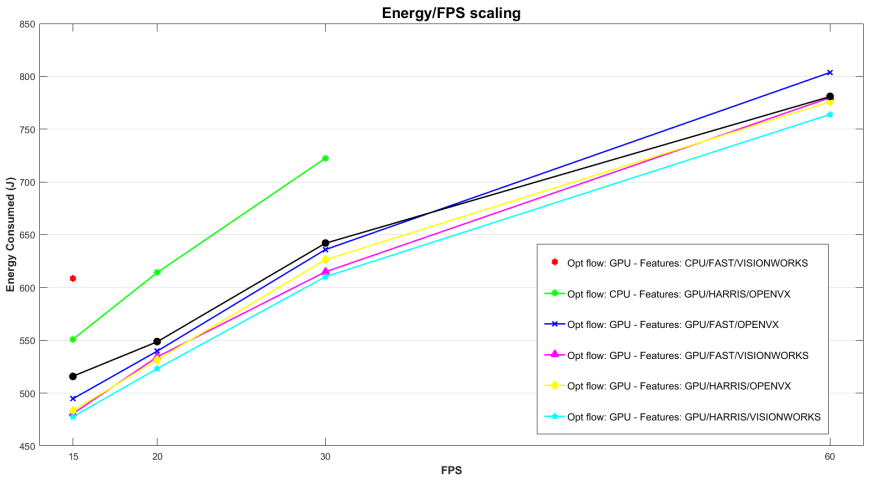


Fig. 3.7: OpenVX energy scaling per FPS

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 [42], INTEL OpenVX [43], AMDOVX [44], Khronos OpenVX standard implementation [45]), such a toolbox of blocks for Simulink is created by properly wrapping the primitives through Matlab *S-Function*, as explained in Section 3.3.2.

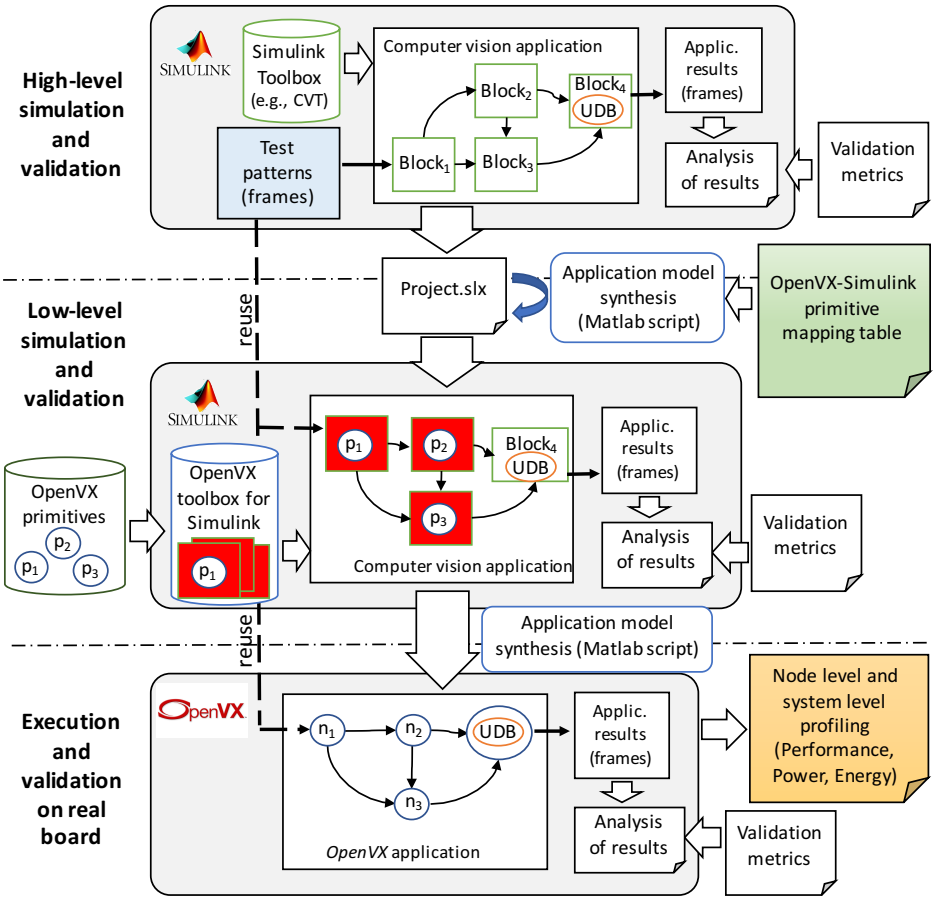


Fig. 3.8: Methodology overview

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 3.3.3.

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 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, 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.

3.3.1 From imperative to data-flow: example of ORB descriptor

In our syntax, we cannot use conditional and loop operators. Therefore, there are some algorithm that need to be rewritten to comply with this rule. There is not an automatic way to do such rewrite, thus we present, as an example, how we managed to convert an adaptive-threshold keypoints extraction algorithm from ORB-SLAM [46], which will be used later in section 4.1.

In computer vision, a descriptor is an object which represents a visual feature of an image. It could encode shape, color, or other meaningful values that are able to identify some sort of knowledge. In particular, the ORB descriptor [47] is a vector that encodes the neighborhood of a point in an image with a good speed in computation, and it is resilient to illumination and orientation changes. This allows the search of points correspondence in two images to calculate, for example, the triangulation.

One way to gather enough information to match two images and yet to reduce as much as possible the amount of points needed for the image matching, is to compute such descriptor on some interesting points, called keypoints or corners, spread across the image, computed on a recursively scaled image up to some level.

Being data-flow oriented, OpenVX does not allow the use of conditional paths or loops. Specifically, the implementation of the keypoint extraction in ORB-SLAM2 [46] follows the example depicted in Figure 3.9. The original image is split into several sub-images. In each of these tiles, a keypoint extraction procedure like the well-known (and OpenVX native block) FAST [48] is performed, using a pre-defined threshold. If no points are found, a lower one is applied specifically for that tile. This process allows to select a first high-value threshold to produce high-quality keypoints, if those are available. However, low quality keypoints are tolerated to allow an evenly spaced distribution among the image.

This procedure performs a loop over all the tiles and a conditional path is used to decide if the second threshold has to be applied. This process could be implemented in the OpenVX context by a custom definition for the FAST block, to allow an adaptive, per-cell threshold selection. On the other hand, this new approach could lead to the loss of the optimizations that have been made for the target board for the native FAST block. Therefore, it is preferable to express the needed functionality by re-using, as much as possible, the primitive implementations provided with the primitive library of the target device vendor.

In the FAST procedure, each keypoint has an associated strength. The routine generates all the keypoints that have a strength higher than a selected threshold. Such a property can be used to implement the aforementioned strategy by reversing the two steps: first, the list of all keypoints with the lower threshold are selected. Then,

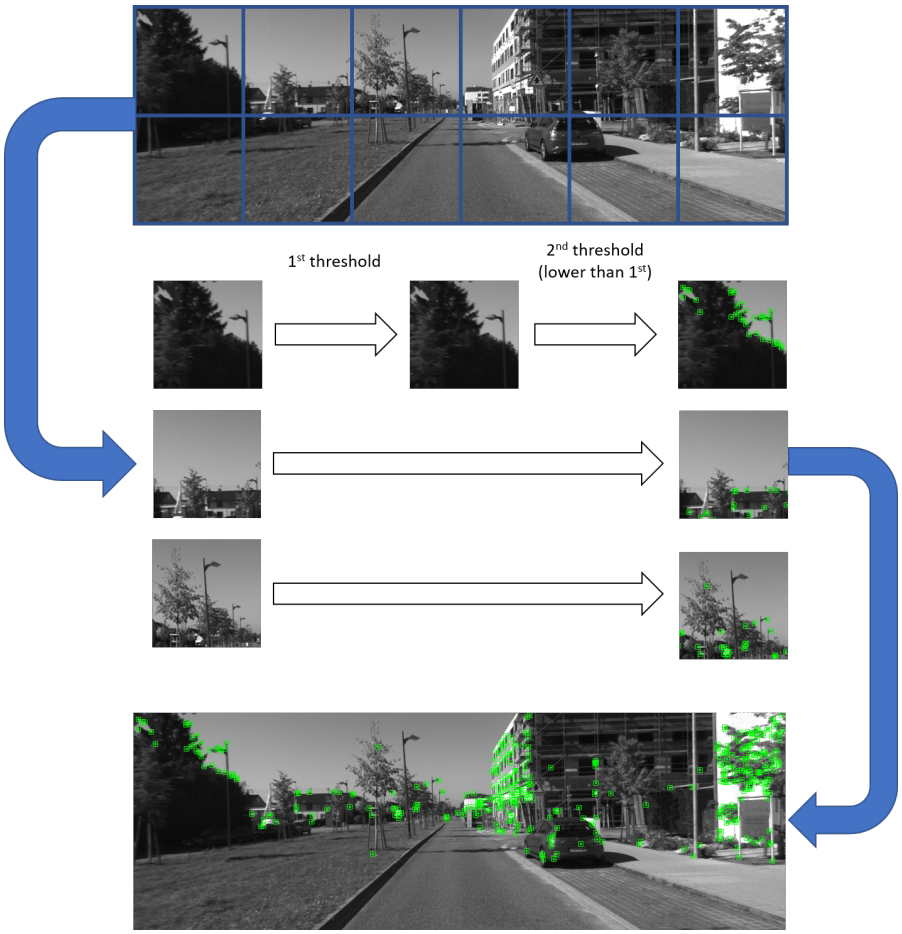


Fig. 3.9: Example for keypoints and ORB extraction, frame 20 of sequence KITTI06

a custom procedure prunes the keypoints with low quality if higher ones are present in the same window.

Figure 3.10 shows an example of such a mechanism applied to an image. The points indicated by red diamonds correspond to keypoints with lower reliability, while the green squares express the ones with higher strength. Using this structure to compute adaptive threshold, loops and conditional instructions can be applied inside a node that takes, as input, the output of the FAST routine. If a green point is detected in a sub-window, then all the red points in that image portion are filtered out. In Figure 3.10, only the top-left square contains no high reliability keypoints. The lower threshold is used only for that tile, granting an even distribution of keypoints across the image, while keeping high quality corners for the rest of the image.

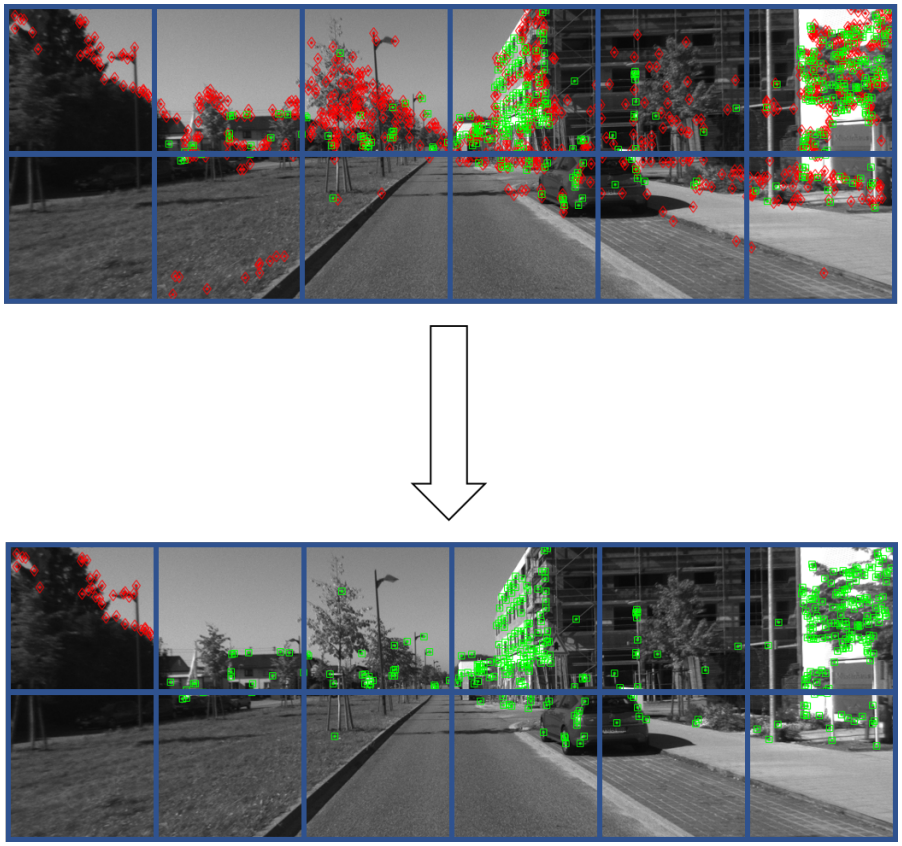
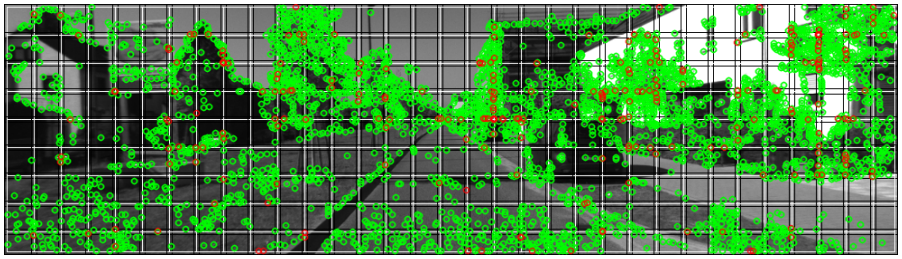


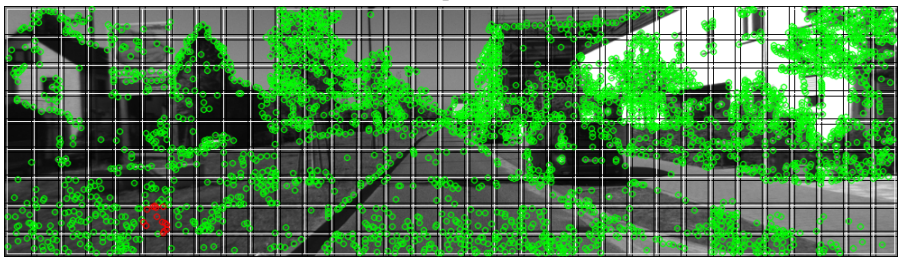
Fig. 3.10: Dataflow elaboration for keypoints and ORB extraction, frame 20 of sequence KITTI06

Differences between imperative and data-flow version

To avoid too much clutter in the keypoints, a procedure called non-maximum suppression has been used after the corner detection inside the FAST routine. This technique discards points which are too close to each other, keeping the one with the maximum value within a radius. Therefore, the keypoints found by the FAST procedure in a sub-section of image could differ from the keypoints present in that region given the whole image as input, especially around the edge of the sub-window. To illustrate this behavior, Figure 3.11 shows the difference between the execution of the imperative and data-flow approaches, by using low values for threshold window size to emphasize the differences. The green marks represent points present in both images, while the red ones indicate a corner present in only one image.



(a) Imperative



(b) Dataflow

Fig. 3.11: Difference between the two types of implementation

Points which are exclusively present in the imperative version are spread around the border between adjacent windows as depicted in Figure 3.11a. This effect is due to non-maximum suppression, which has not the information to prune very close keypoints when the stronger corner is outside of the reference window. The imperative version has the same keypoints of the data-flow implementation with the addition of other corners near window edges.

The only exception to this behavior can be observed at column 6 and the row before the last one in Figure 3.11b. Due to non-maximum suppression, a high reliability corner is pruned because another one is present and very close to it. However, this keypoint belongs to another window, thus marking that sub-window without any

high-reliability corner, and thus forcing the adaptive threshold to be set to the lowest value.

In the majority of cases, however, the two implementations are almost equivalent. When they are not, the selected corners are very close to each other, leading to no substantial difference between the two versions.

3.3.2 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* [49]. 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.

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

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 3.3, which represents the *Color Converter* node of Fig. 3.1):

- *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 exe-

cutes OpenVX code, this phase implementation relies on a *data wrapper* for the OpenVX-Simulink data exchange and conversion.

- *End phase* (lines 15-17): It deallocates the created data structures at the end of the simulation (after the computation phase).
- *Computation phase* (lines 18-21): it reads the input data and executes the code implementing the block functionality. It makes use of a *primitive wrapper* to execute OpenVX code.

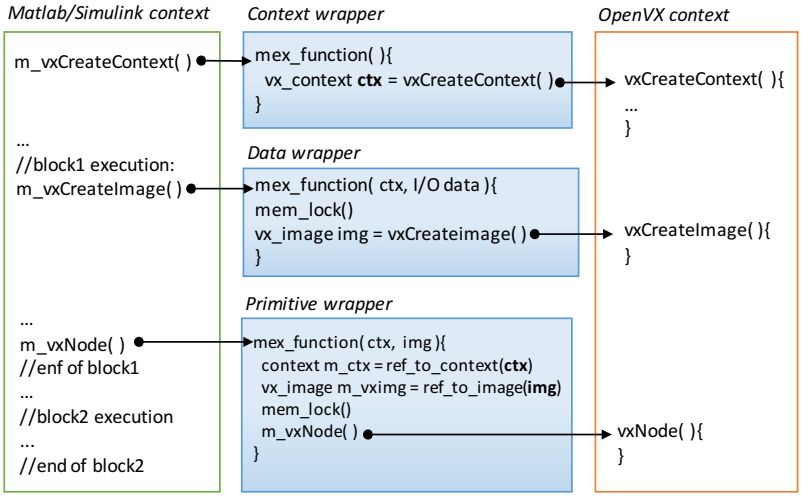


Fig. 3.12: Overview of the Simulink-OpenVX communication

Three different wrappers have been defined to allow communication and synchronization between the Simulink and the OpenVX environments. They are summarized in Fig. 3.12. The *context wrapper* allows creating the OpenVX context (see line 1 of Listing 3.1), which is mandatory for any OpenVX primitive execution. It runs 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. 3.1 and lines 4-11 of Listing 3.1). It runs once for each application block. The *primitive wrapper* allows executing, in the Simulink context, each primitive functionality implemented in OpenVX. To speed up the simulation, the wrapped primitives work through references to data structures, which are passed as function parameters during the primitive invocations to the OpenVX context. To do that, the wrappers implement memory locking mechanisms (i.e., through the Matlab *mem_lock()/mem_unlock()* constructs) to prevent data objects to be released automatically by the Matlab engine between primitive invocations.

3.3.3 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 3.4 shows, as an example, a representative subset of the mapped entries.

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 Standard-Dev		
Simulink/Math operations/Real/ComplexTo -Imag	vxuMagnitude	Gradient magnitude computed through complex numbers
Simulink/Math operations/Real/Imag to Magnitude		

Table 3.4: Representative subset of the mapping table between Simulink CVT and NVIDIA OpenVX-VisionWorks

We implemented three possible mapping strategies:

```
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 3.4: Overview of wrapper for user-defined Simulink block implementations

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 3.4, which follows the node implementation directives required by the standard OpenVX for importing user kernels[4]. The wrapper invocation (i.e., $vx\_userNode()$) 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. 3.12).

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

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 3.5: Experimental results: High-level simulation time in Simulink

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

3.3.4 Results

We applied the proposed model-based design flow for the development of the embedded software implementing the digital image stabilization algorithm represented as running example in Fig. 3.2.

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) [50].

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 3.5 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 [39]. The table reports the characteristics of such selected test patterns (sequences of frames).

Validation level	**Sim./Exec. time (min)**	
	Indoor	**Outdoor**
Simulink High-Level model	30.5	13.0
Simulink Low Level model	59.0	26.0
Software application on target embedded system device	20.5	11.0

Table 3.6: 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.

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 [42] and the corresponding Simulink CVT-NVIDIA OpenVX/VisionWorks mapping table, as described in Sections 3.3.2 and 3.3.3, 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 3.6 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 [39] 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.

4

Polyglot parallel programming model and integration

Due to the limitation of OpenVX to model complex applications through data-flow graphs and to the incompleteness of the OpenVX primitive libraries provided by the device vendors, 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 section presents a model that combines different programming environments, i.e., C/C++, OpenMP, PThreads, OpenVX, OpenCV, and CUDA to best exploit different levels of parallelism while guaranteeing the semi-automatic customization.

4.1 Analysis of polyglot programming environments in 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 [51], 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

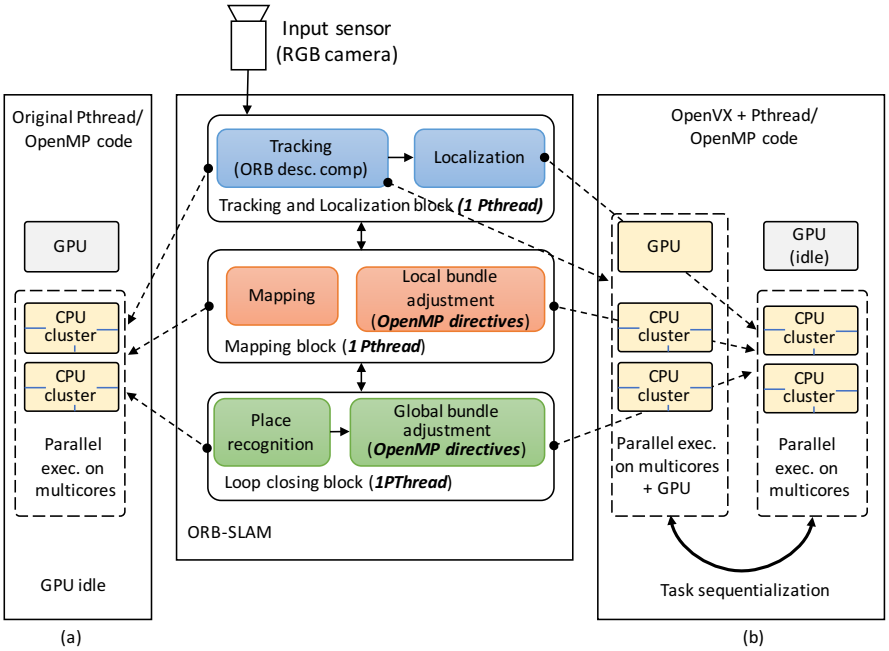


Fig. 4.1: Overview of ORB-SLAM application and execution models: (a) the original code (parallelized for multicore), (b) the state-of-the-art OpenVX implementation.

by considering the current map. The application consists of three main blocks (see Figure 4.17):
- 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 [51], 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 4.17(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 [42] 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 modeled through a data-flow graph and is worth to be optimized for CPU/GPU execution.

4.2 Polyglot framework for heterogeneous platforms

Figure 4.2 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 environments 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 4.2 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.
- *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.,

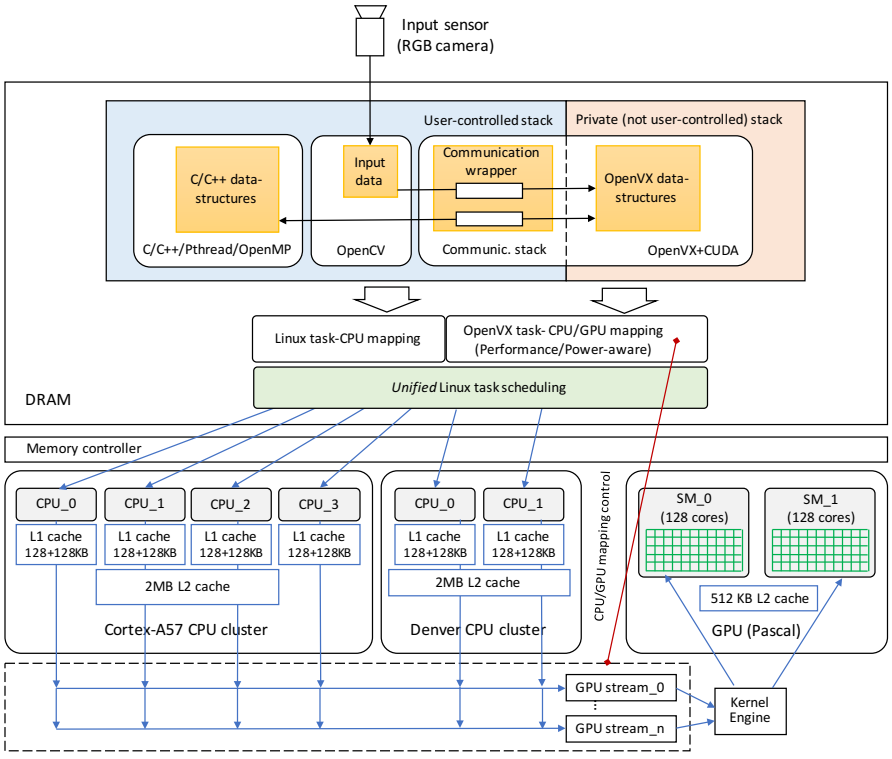


Fig. 4.2: Framework overview: memory stack, task mapping, and task scheduling layers of an embedded vision application developed with the proposed method on the NVIDIA Jetson TX2 board.

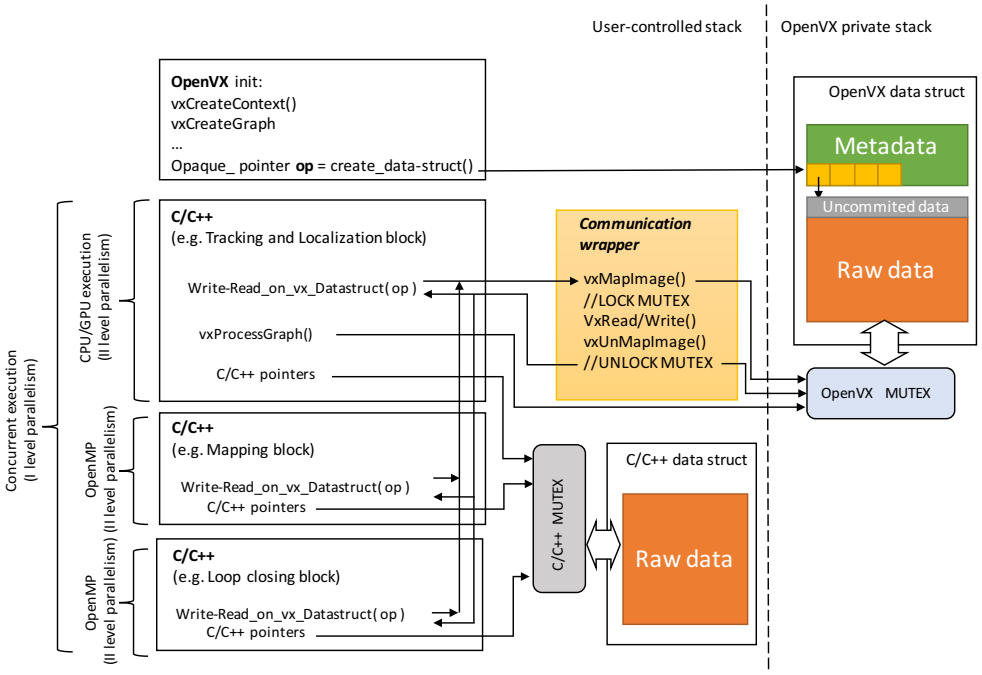


Fig. 4.3: Overview of the communication wrapper and its integration in the system.

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 4.3 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 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 4.3, 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. 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.

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.

4.3 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 [52]. We then integrated modularly the different parallel environments supported by the framework as follows:

- *Version 1 (Pthreads)*: It is the starting version [52], 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.

Table 4.1: 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	-	-	7.6	131.4	1,205	**3.37**	**4.05**	3,097/3,281 (94.4%)	**0.357**
Version 2	4	2	-	7.6	132.2	1,125	3.43	5.24	3,021/3,281 (92.1%)	0.372
Version 3	3	-	2	9.1	109.7	**1,039**	3.78	5.62	**3,279/3,281 (99.9%)**	0.378
Version 4	4	2	2	9.5	105.6	1,242	5.79	7.85	3,260/3,281 (99.4%)	0.381
Version 5	3	-	2	**11.3**	**88.2**	1,184	3.61	5.46	3,269/3,281 (99.0%)	0.362
Version 6	4	2	2	**11.3**	**88.4**	1,197	5.65	7.92	**3,271/3,281 (99.8%)**	0.366

Table 4.2: 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	-	-	8.8	114.1	1,917	**5.84**	**8.54**	3,264/3,281 (99.5%)	0.587
Version 2	4	2	-	8.8	113.2	1,967	6.00	9.61	3,269/3,281 (99.6%)	0.602
Version 3	3	-	2	9.8	102.2	1,954	5.96	9.54	3,276/3,281 (99.8%)	0.597
Version 4	4	2	2	9.9	101.0	1,945	7.93	11.01	**3,280/3,281 (100%)**	0.593
Version 5	3	-	2	**12.8**	**78.1**	1,895	5.98	9.65	3,274/3,281 (99.0%)	0.579
Version 6	4	2	2	**12.8**	**78.3**	**1,843**	7.62	11.70	**3,279/3,281 (99.9%)**	**0.562**

- *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 [53], which is a standard and widespread benchmark for SLAM 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 4.1 and 4.2 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.

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.

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.

4.4 Performance enhancing with multilevel parallelism

The original implementation of the algorithm [46] provides two levels of parallelism. The first level is given by the three main algorithm blocks (see Fig. 4.4), which are

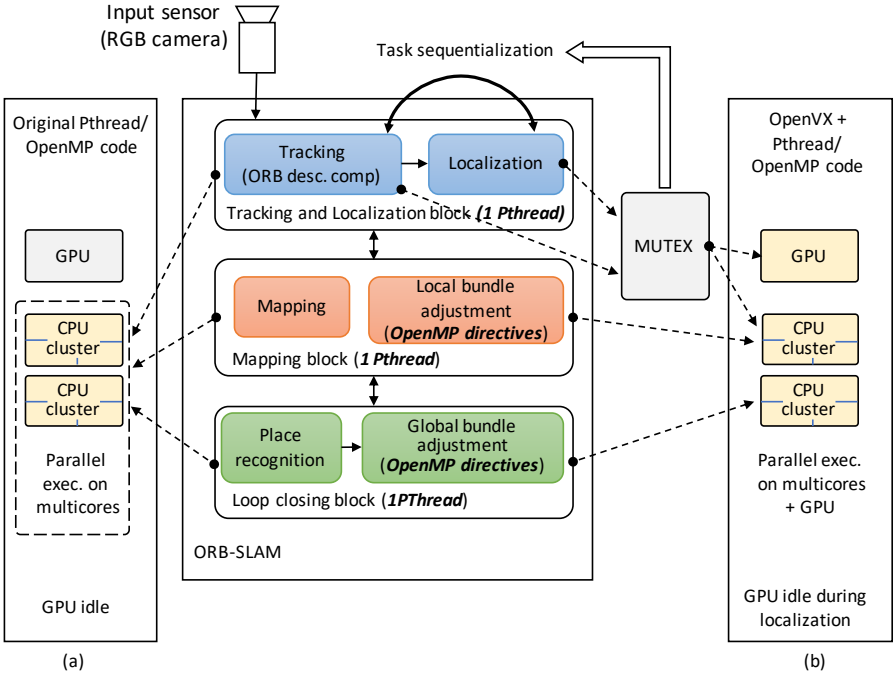


Fig. 4.4: Limitations of the ORB-SLAM application and execution models on the Jetson board: (a) the original code (no GPU use), (b) the OpenVX NVIDIA VisionWorks (sequentialization of tracking and localization tasks and no pipelining).

implemented to be run as parallel PThreads on shared-memory multi-core CPUs. The second level is given by the automatic parallel implementation (i.e., thorugh OpenMP directives) of the *bundle adjustment* sub-block, which is part both of the local mapping and loop closing blocks. This allows the parallel computation of such a long latency task on multi-core CPUs. No blocks or sub-blocks are considered for parallel execution on GPU in the original code (see Fig. 4.4(a)).

To fully exploit the heterogeneous nature of the target board (i.e., multi-core CPUs combined with many-core GPU), we added two further levels of parallelism. The first is given by the parallel implementation for GPU of a set of tracking sub-blocks (see Fig. 4.5). The second is given by the implementation of a 8-stage pipeline of such sub-blocks. We focused on the feature extraction block as, for the datasets analysed in this work (i.e., KITTI [53]), it is the most important bottleneck that characterizes the processing rate in terms of supported FPS. Other to these additions, we pipelined the execution of ORB feature extraction block and Localization block to further improve system throughput, leveraging PThreads parallelism level.

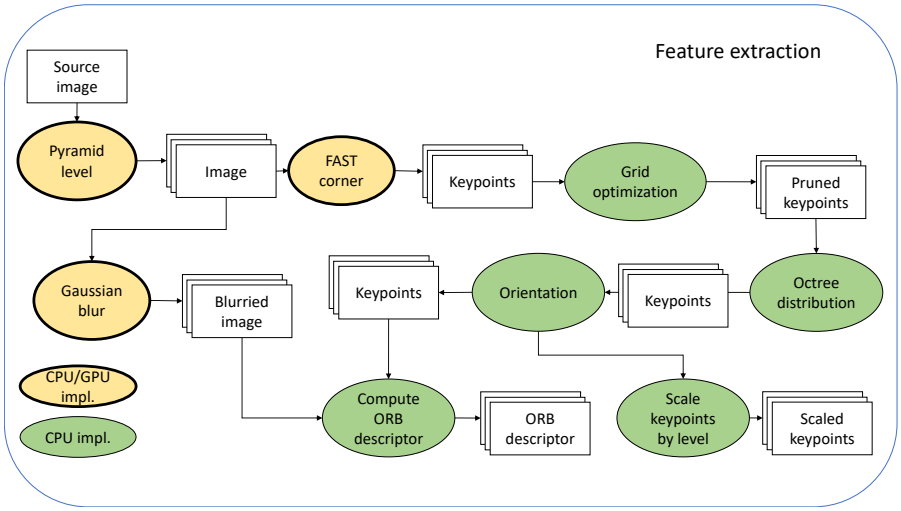


Fig. 4.5: DAG of the feature extraction block and the corresponding sub-block implementations (GPU vs. CPU).

To do that, we first re-designed the model of the feature extraction block as a direct acyclic graph (DAG) by adopting the OpenVX standard[1] as shown in Fig. 4.5. The transformation of the original implementation, which was originally conceived for CPUs only, into a CPU/GPU parallel execution requires a control on the communication between code running on the CPUs and on the GPU. In particular, the mapping phase is critical, requiring a temporal synchronization between the blocks of the algorithm to be successful.

NVIDIA provides *VisionWorks*, which extends the OpenVX standard through efficient implementations of embedded vision kernels and runtime system optimized for CUDA-capable GPUs. Nevertheless, such a toolkit has some limitations, which do not allow for the target multi-level parallelism. In particular, the VisionWorks runtime system, which manages synchronization and execution order among the DAG sub-blocks, implicitly sequentializes the tracking and localization block execution (see Fig. 4.4(b)). This is due to the fact that only the tracking sub-block can be modeled as DAG and, although the rest of the system can be integrated as C/C++/OpenCV code, their communication and synchronization is solved through a mutex-based mechanism (see Fig. 4.4). Such a sequentialization leads to the idle state of the GPU whenever the localization block is running. In addition, VisionWorks does not support pipelined execution among DAG sub-blocks.

Since VisionWorks is not open source, we re-implemented and made open source both an advanced runtime system targeting multi-level parallelism and a library of accelerated computer vision primitives compliant to OpenVX for the Jetson TX2 board.

[1] `https://www.khronos/openvx`

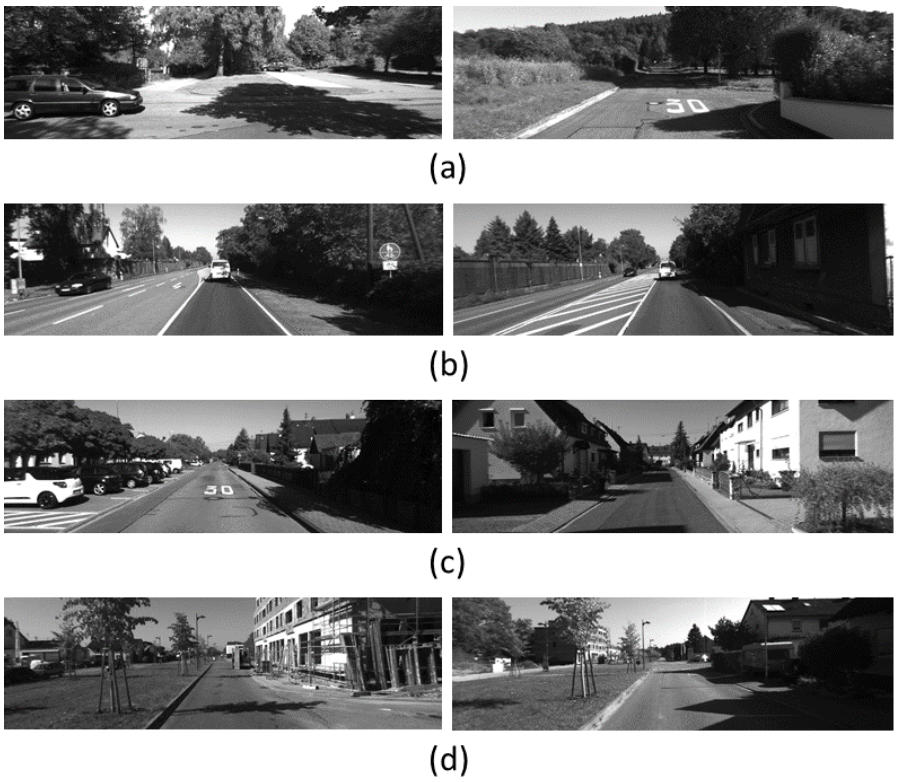


Fig. 4.6: Samples from the four sequences of the KITTI dataset used for evaluation. (a) Sequence 03. (b) Sequence 04. (c) Sequence 05. (d) Sequence 06.

4.5 Results with optimized version

To evaluate the results of our modified version of ORB-SLAM2, we used four sequences from the KITTI dataset (see Fig. 4.6) as done in the original paper by Mur-Artal and Tardos [46].

The KITTI dataset [53] contains sequences of 1,242×375 images recorded at 10 FPS from a car in urban and highway environments (see Fig. 4.6). We consider four sequences, namely 03, 04, 05, 06. Sequences 03 and 04 do not contain loops, while sequences 05 and 06 contain different numbers of loops. Public ground-truth is available for the considered sequences. We implemented and evaluated three different versions of ORB-SLAM2, in which the tracking block exploits:

1. CPU + pipelining
2. CPU + GPU
3. CPU + GPU + pipelining

Table 4.3: Runtime performance (FPS)

Sequence	CPU	CPU+pipelining	CPU+GPU	CPU+GPU+pipelining
03	6.99	15.23	17.90	**27.79**
04	7.47	15.96	20.70	**30.33**
05	6.54	15.56	18.52	**27.97**
06	6.68	16.03	19.66	**32.30**

4.5.1 Runtime Performance

Table 4.3 shows the performance of the original ORB-SLAM2 code[2] and our three different versions (publicly available on GitHub[3]) running on the Jetson TX2 board. The results have been generated using the same settings for comparing the different versions and by repeating five times the execution of each considered sequence.

The results in Table 4.3 highlight the different performance achieved by the original code and the three different versions in terms of supported FPS. The original code run on such an embedded and low-power board does not support real-time execution, by achieving in average 7 FPS. Both pipelining and heterogeneous CPU+GPU execution allow improving the performance by exploiting different kinds of parallelism. The CPU+GPU+pipelining version provides the best results thanks to the multi-level (combined) parallelism. It supports real-time executions with frame rates above 25 FPS.

4.5.2 Qualitative and Quantitative Evaluation and Metrics

Fig. 4.7 shows the qualitative results of our best implementation (i.e., CPU+GPU+pipelining).

For the quantitative evaluation of the result quality, we considered three different metrics: the root mean squared error for the absolute translation (RMSE ATE), the root mean squared error for the average relative pose error (RMSE RPE) [54]

[2] `https://github.com/raulmur/ORB_SLAM2`
[3] `https://github.com/xaldyz/dataflow-orbslam`

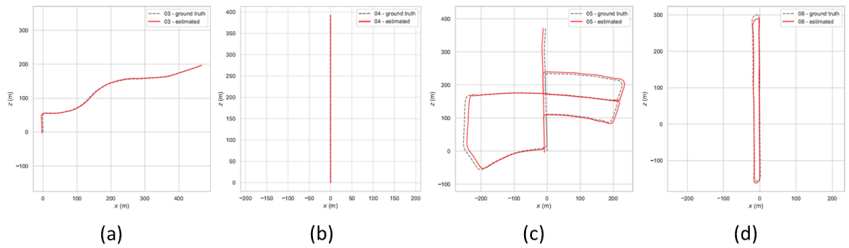


Fig. 4.7: Qualitative evaluation of the proposed ORB-SLAM application version CPU+GPU+pipelining on some parts of KITTI sequence 03 (a), sequence 04 (b), sequence 05 (c) and sequence 06 (d).

Table 4.4: Quantitative results

Version	Original			CPU + pipelining			CPU + GPU			CPU + GPU + pipelining		
Sequence	$ATE(m)$	$RPE(m)$	% cov.	$ATE(m)$	$RPE(m)$	% cov.	$ATE(m)$	$RPE(m)$	% cov.	$ATE(m)$	$RPE(m)$	% cov.
03	0.73	0.15	70.87	1.33	0.09	99.90	1.15	0.09	99.92	1.13	0.08	99.97
04	0.82	0.46	15.01	1.31	0.11	99.92	0.39	0.26	19.63	0.37	0.11	99.72
05	6.58	0.76	23.09	7.44	0.82	82.99	10.54	1.05	91.03	13.88	1.18	95.58
06	1.29	0.29	27.37	16.04	1.11	89.98	15.46	0.99	77.36	16.30	1.14	91.75

and the percentage of the reconstructed map. Measuring the absolute distances between the estimated and the ground truth trajectory using ATE provides a measure of the global consistency of the estimated trajectory. Moreover, ATE has an intuitive visualization that facilitates visual inspection (see Fig. 4.7). The RPE measure allows us to evaluate the *local* accuracy of the SLAM system, i.e., to measure the error related to two consecutive poses [54]. The percentage of reconstructed map is calculated from an initialization step. ATE and RPE are considered on the portion of the reconstructed map.

Table 4.4 shows the quantitative results. With the original implementation, the processing speed (around 7 FPS) fails to meet the 10 FPS requirement of the dataset. As a consequence, only a partial reconstruction is available. Since the metrics are defined only for the reconstructed portion of the map, the low map coverage reconstruction leads to a very low (misleading) absolute error. This behaviour is more evident in the ATE, while the RPE is comparable in all versions. The analysis underlines a slight quality degradation of the results provided by the implementations for GPU when run above 28 FPS. This is due to the different implementation and synchronization of the feature extraction primitives with respect to the original sequential version. In general, by considering both the performance and the quality of the results, the CPU+GPU+pipelining implementation provides the highest FPS, it gets lost sensibly less, and provides a negligible degradation of results w.r.t. the original sequential implementation.

4.6 Inter-application integration

To perform an easy integration between several modules, they have to share the same way to exchange data. In this context, ROS is the de-facto standard communication mechanism adopted in robotic applications. It provides a distributed system to perform decentralized communication using common defined interfaces, thus exposing the input/output boundary of a system.

4.6.1 ROS overview

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

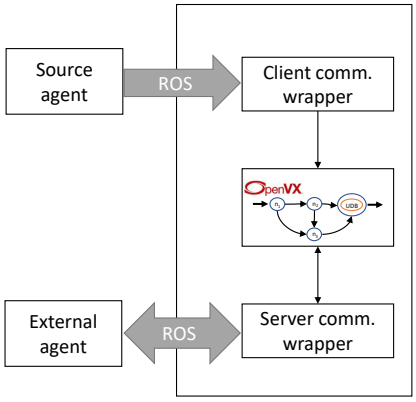


Fig. 4.8: The OpenVX-ROS communication thorough the server and client models

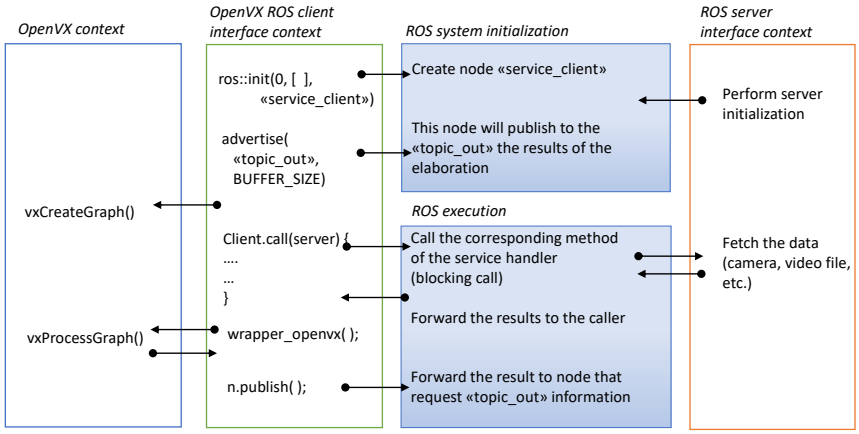


Fig. 4.9: Client model time evolution

- *Client model:* The OpenVX application actively fetches inputs from a particular ROS node. It relies on a client communication wrapper, as shown in the upper side of Figure 4.8. It is particularly suited for intensive yet synchronous communication (e.g., data acquisition of the OpenVX application from an input sensor).
- *Server model:* It allows the OpenVX application to be run on-demand. The external environment, which is implemented as ROS node, sends an execution request through an input data structure. The OpenVX application executes and returns the result as a response packet. It relies on a server communication wrapper, as shown in the bottom side of Figure 4.8. It is well suited to implement sporadic

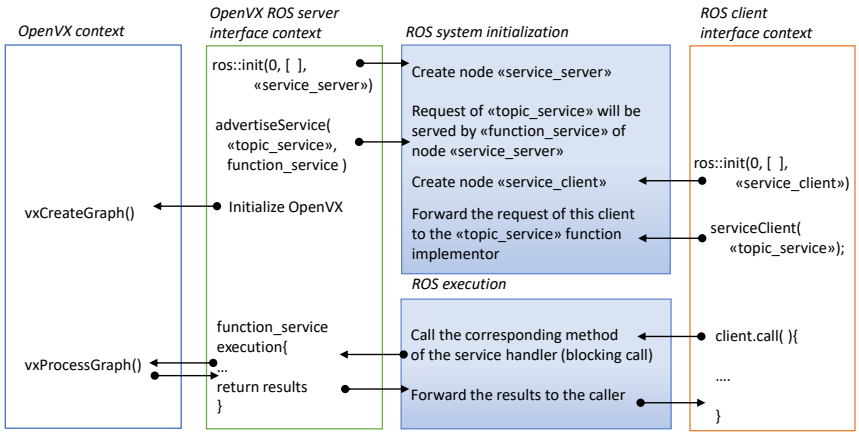


Fig. 4.10: Server model time evolution

Listing 4.1: Skeleton implementation for the (a) server model and the (b) client model

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

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

(a)

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

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

(b)

communication (e.g., interpretation of the map built by a SLAM application by an external agent).

Listing 4.1(a) shows the skeleton implementation of the server interface. The *process_init* function is responsible to perform the node initialization in the ROS framework. It adds the current process to the ROS node list in the master server (line 14). This node is sensitive to the topic specified in line 15. Line 16 specifies the function that will be called on the server invocation. Lines 1-7 provide the invocation of the OpenVX application. Two parameters are necessary to the function: the request, which contains the input data, and the response, which will be updated by the computing function. Conversion functions are defined to convert the data format between ROS and OpenVX. Finally, line 17 implements the busy waiting until the ROS framework shuts down all the nodes. Figure 4.10 shows the temporal evolution of the OpenVX-ROS communication based on the server model of Figure 4.1(a).

Listing 4.1(b) depicts the skeleton for the client interface. After adding the process to the list of ROS nodes (line 3), the system informs the ROS framework that the client requests need to be forwarded to the *topic_service* listener (lines 6-7). The wrapper creates the object to write the results of the computation (lines 8-11). Parameters are filled in line 14, and the call to request data is performed in line 15. In case of positive message receiving, the OpenVX computation is called (line 19-22), through ad-hoc functions to convert the data format between ROS and OpenVX. The system publishes the results back to the network (line 18). Figure 4.9 shows the temporal evolution of such a client model process.

An overview of the framework can be seen in fig. 4.11

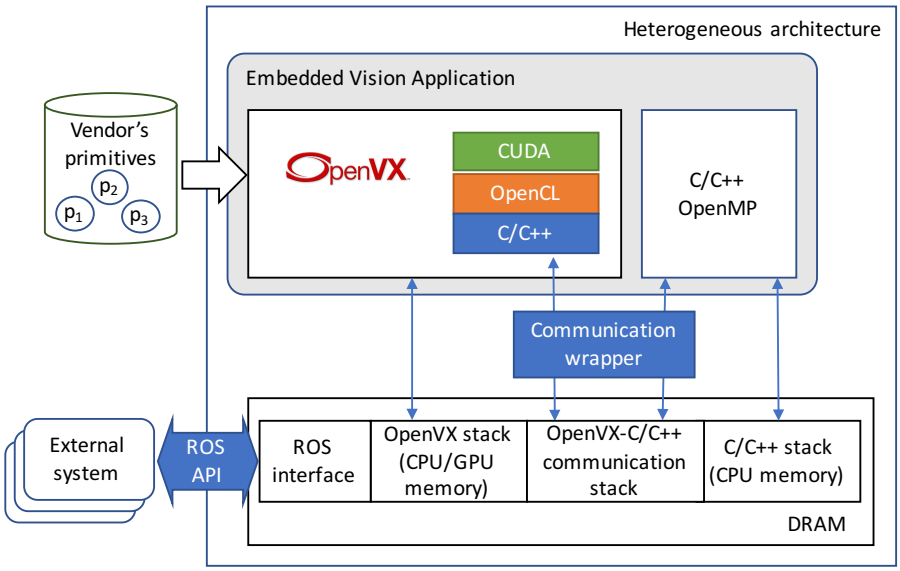


Fig. 4.11: Overview of the proposed framework

4.6.2 Results

The proposed OpenVX-ROS framework has been applied to customize the ORB-SLAM application [51]. The application receives the input stream by a camera through ROS as depicted in figure 4.12 We tested the application on the KITTI dataset [53], which is a standard set of benchmarks for computer vision applications. For the sake of space, we present only the results obtained with the KITTI sequences 11 and 13 (10 frames per second -FPS- each). They have been chosen as they represent inputs with different workload on the three blocks. In particular, sequence 11 mostly relies on the tracking block while it never uses the mapping and loop closure blocks (see Fig. 4.13). In contrast, sequence 13 also runs the other two *computing intensive* blocks each time the trajectory returns on an already visited point in the map (see Fig. 4.14).

The application processes the stream and generates the map of the visited external environment. A route controller, which is run on an external board, queries the map generated by the ORB-SLAM application to elaborate trajectories toward a target position. The whole package has been customized to run on an NVIDIA Jetson TX2 board. Such an heterogeneous low-power embedded system is equipped with a quad-core ARM CPU, and a 256 CUDA cores Pascal GPU.

Tables 4.5 and 4.6 report the obtained results in terms of number of CPU cores used by the application blocks, whether the GPU has been used by any application block, the average time required for processing one frame, the corresponding performance i.e., the maximum FPS, the average time required by the mapping phase, the total energy consumption for the whole stream analysis, the peak power of the board, and the result accuracy. The result accuracy expresses how many frames the application has been able to elaborate over the whole sequence and, among them, how many frames gave correct mapping information. For example, 54/675 accuracy with sequence 1 means that 675 over 921 frames have been elaborated and only 54 of them were useful. The result accuracy is highly correlated to the supported FPS.

Performance information has been collected through the CUDA runtime API to measure the execution time and through the `clock64()` device instruction for throughput values to ensure clock-cycle accuracy of time measurements. Power and

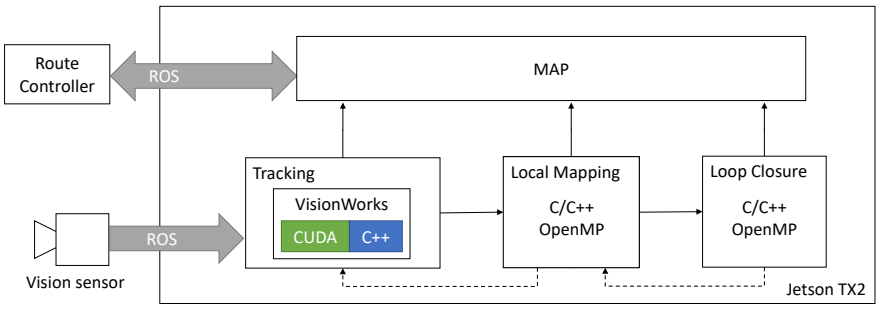


Fig. 4.12: Overview of ORB-SLAM implementation

Table 4.5: Results with sequence 11 (921 frames)

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

Table 4.6: Results with sequence 13 (3,281 frames)

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

energy consumption information have been collected through the *Powermon2* power monitoring device [55].

All these information are reported for four different versions of ORB-SLAM we developed and analysed. The first version is the original code retrieved from [51], which is run on a single core of the CPU. The second version is the original code with the multi-threading feature enabled (i.e., each block is mapped on a corresponding thread and run on a different CPU core). The third version is the original code enriched with OpenMP directives. In this version, the mapping and loop closure computations are parallelized when the map exceeds a given threshold and requires more computational power to be processed. The fourth version consists of the original code, in which the feature detection and ORB algorithms of the tracking block have been implemented by using OpenVX and CUDA, respectively. We adopted the NVIDIA VisionWorks primitive library for implementing the OpenVX blocks. We parallelized the CUDA code of ORB from scratch.

The table underlines that different customizations can be considered starting from the same applications, and that the best version depends on the selected design constraints and on the input stream. Considering sequence 11 and targeting performance, version OpenVX that adds the GPU computation to the multithreaded original version provides the best frame per second at the cost of the highest peak power. In this context, OpenMP does not provide any benefit as it is scarcely applied (loop closure phase never runs in seq. 11) while it introduces overhead. In contrast, OpenVX+OpenMP provides the best performance with sequence 13, in which the

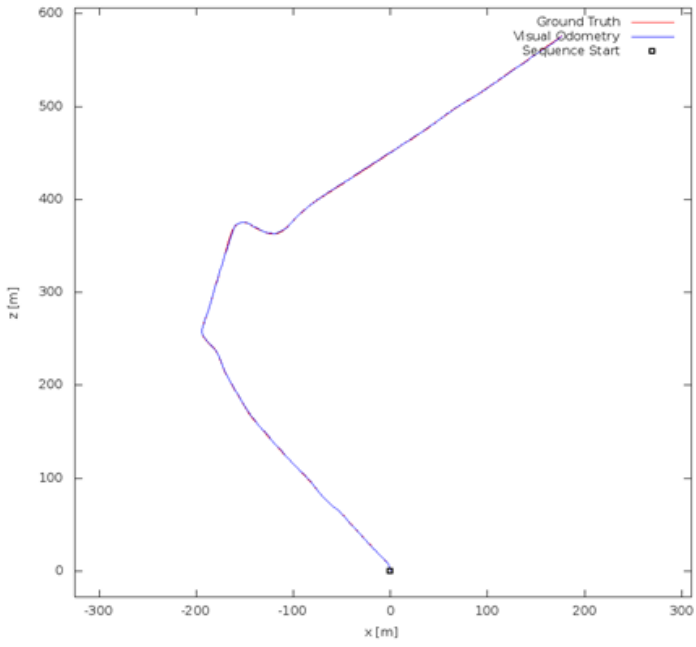


Fig. 4.13: KITTI sequence 11

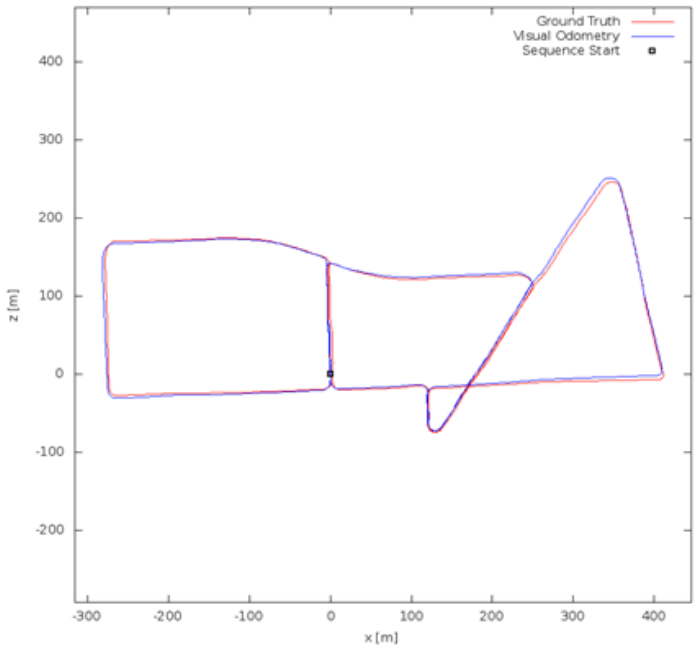


Fig. 4.14: KITTI sequence 13

two phases are efficiently parallelized in the CPU cores. If the application is run in an energy-saving context, and considering the characteristics of the input stream (10 FPS), the multithreaded version limited to three CPU cores provides the best energy efficiency and the most limited power consumption. This is due to the fact that this version, for both sequence 11 and 13 provides the maximunm FPS most similar to the FPS of the streams read in input.

4.7 Combining heterogeneous applications

The trend in the last years for object detection and recognition is through use of machine learning techniques. In case of images, a particular class of models called Convolutional Neural Networks (CNNs) are used to gather a lot of prior knowledge using labeled data in an automatic fashion. Their capacity can be controlled by varying their depth and breadth, and they also make strong and mostly correct assumptions about the nature of images (namely, stationarity of statistics and locality of pixel dependencies) [56].

Figure 4.15 shows an example of a model which is used to recognize handwritten numbers. The input image is processed by a set of filter, each one with his own parameters. A train phase is used to customize the parameters by using the labeled data. This step became feasible using large dataset only in recent years, due the GPGPU acceleration.

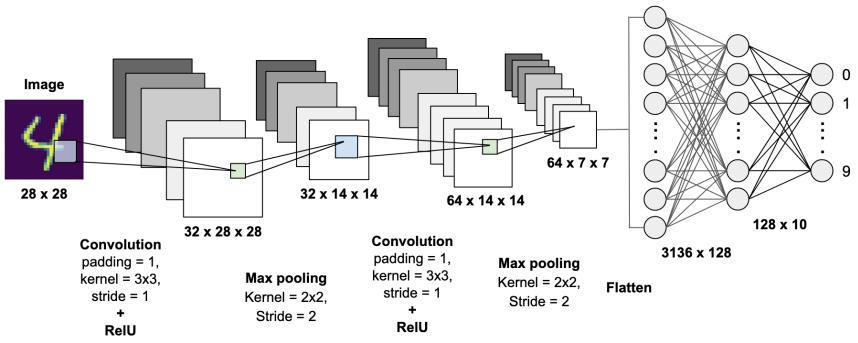


Fig. 4.15: CNN example for handwritten digit classification

We applied the proposed model-based design flow to develop and optimize the ORB-SLAM application combined with an image recognition system based on Deep Learning [57] running on the same board (see Figure 4.16).

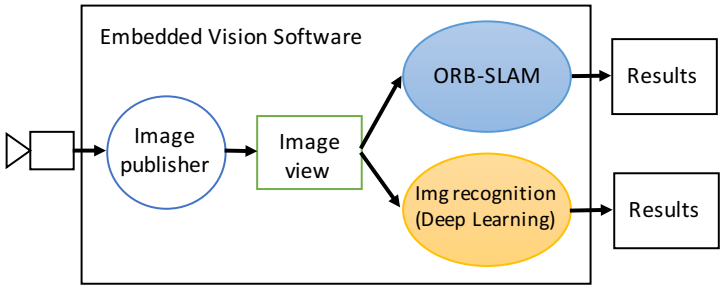


Fig. 4.16: Inter-application communication

Along the ORB-SLAM/DL-based image recognition design flow, we measured the execution time of the algorithm implementations at different refinement steps, by using the KITTI dataset[53], which is a standard set of benchmarks for SLAM and computer vision applications.

Table 4.7 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 the minimal selection of video streams necessary to validate the model correctness. TODO

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 [42] 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

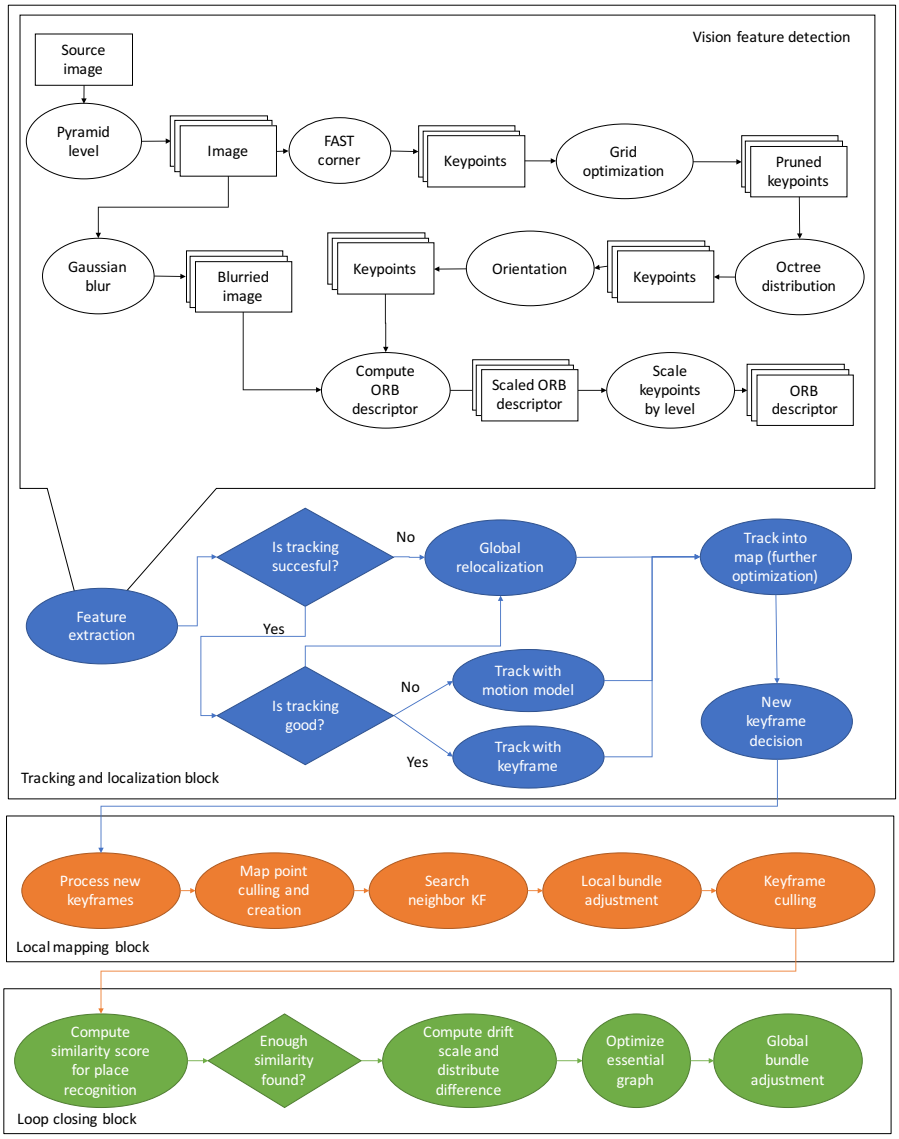


Fig. 4.17: Overview of the ORB-SLAM use case.

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. This is due to the fact that the algorithm implementation in Simulink required specialized MATLAB code that was not available with Simulink CVT library

Validation level	**Sim./Exec. time (sec)**
Simulink High-Level model	404.0
Simulink Low Level model	762.0
Software application on target embedded system device (with accelerators) - *Version 3*: Pthreads+OpenVX	30.0

Table 4.7: Simulation (in Simulink) and execution (on real board) times

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.

At each refinement step, we reused the selected test patterns to verify the code over the adopted validation metrics [39] 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 3.3 (*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 [52] 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 [52], 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 4.18 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. Such a space can provide the best solution for each of the considered design constraints like performance, power consumption, and energy efficiency.

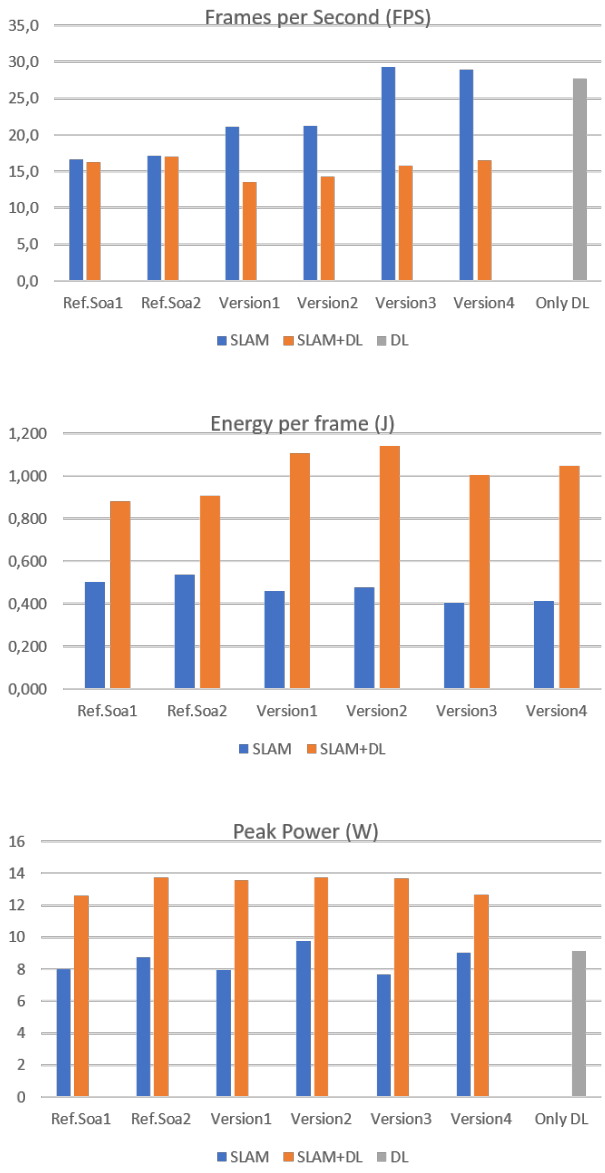


Fig. 4.18: Evaluation of non-functional properties

5

An algorithm for scheduling and mapping of application tasks for performance enhancement

Prior research efforts have used OpenVX for embedded vision [5], [7], [58], and attempted to optimize the performance of the generated code. They proposed techniques to implement different data access patterns such as DAG *node merge*, *data tiling*, and parallelization via OpenMP. There also have been efforts to make the OpenVX task scheduling deliver real-time guarantees [29]. Nevertheless, to the best of our knowledge, there is no prior work that focuses on efficient mapping strategies and its corresponding scheduling of OpenVX (DAG-based) applications for heterogeneous architectures. Prior approaches that propose mapping strategies for OpenVX considered each DAG node to have only one exclusive implementation (e.g., either GPU or CPU), and the mapping is driven by the availability of the node's implementation in the library: If a node has a GPU implementation then it is mapped on the GPU. Otherwise it is mapped on a CPU core.

What is missing is a mapping strategy that targets the system throughput rather than kernel throughput. The fact that there can be multiple implementations for nodes (e.g. one that is executable on a GPU and another on CPU), which are often available in the OpenVX libraries [42]–[45], can allow for additional mapping flexibility, and as a consequence, for better load balancing at the application level (e.g., a CPU implementation that is slower at kernel level can lead to a faster application at system level). However, such a combined mapping and scheduling problem is similar to the *Quadratic Assignment Problem*, a well-known NP-hard problem. Finding an optimal solution satisfying all the given DAG constraints is difficult. Thus, heuristics based on the application domain knowledge need to be employed to find a near-optimal solution.

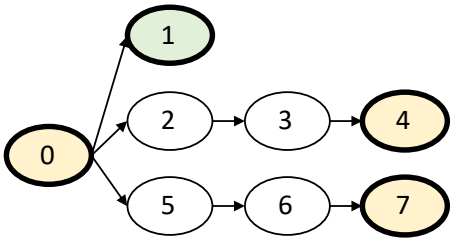
To take into consideration the heterogeneity of the target architectures, the multiple implementations of DAG nodes, and the problem complexity, this thesis proposes an implementation of the heterogeneous earliest finish time (HEFT) heuristic [9] for *static* mapping and scheduling of OpenVX applications. We show that the HEFT implementation sensibly outperforms (i.e., up to 70% of performance gain) the state-of-the-art solution currently adopted in one of the most widespread embedded vision systems (i.e., NVIDIA VisionWorks on NVIDIA Jetson TX2). Then, we show that such a heuristic, when applied to DAG graphs for which *not every* node has multiple

implementations, can lead to idle periods for the computing elements (CEs). Since not having multiple implementations for all nodes happens in a majority of real embedded vision contexts, this thesis proposes an algorithm that reorganizes the HEFT ranking to improve the load balancing. The algorithm aims at generating sequences of nodes with the single implementation (which we call *clusters* of *exclusive nodes*) in the ranking with the objective of reducing idle times caused by the combination of DAG constraints and exclusive implementation.

This section presents the results on a very large set of synthetic benchmarks and on a the ORB-SLAM application combined with an NVIDIA image recognition application based on deep-learning.

5.1 HEFT overview

In this section we first propose an implementation of the task mapping and scheduling for OpenVX, which is based on the HEFT heuristic with up-word ranking and max functions [9]. To our knowledge, the application of HEFT on such platforms has not been considered in prior work. We adapted the algorithm in order to support the exclusive implementation of DAG nodes. We then propose an optimization of HEFT that, starting from a given ranking list, it reorganizes the list to improve the task overlapping and the overall application performance. It is important to note that the proposed optimization is independent from the HEFT variants, since it applies to any ranking list.

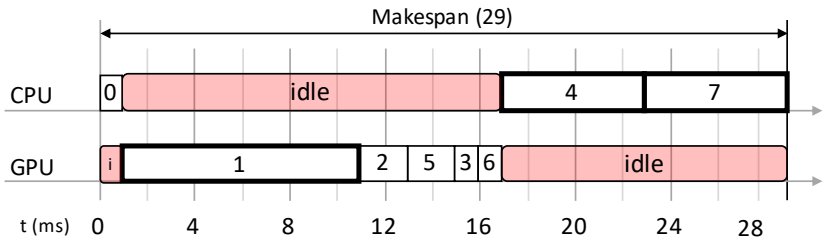


node #	t_{CPU} (ms)	t_{GPU} (ms)	HEFT rank (max)
0	1	-	15
1	-	10	10
2	4	2	14
3	4	1	10
4	6	-	6
5	4	2	14
6	4	1	10
7	6	-	6

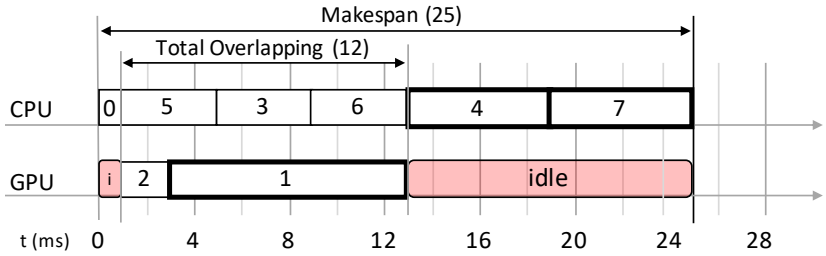
Fig. 5.1: Example of DAG, execution time of tasks mapped on CPU/GPU, and the corresponding HEFT ranking.

The limitation of current state of the art can be stated with the example in Fig. 5.1, which represents the DAG model of an application to be deployed on a heterogeneous CPU/GPU board, and for which there exists a library of primitives for the node implementations. The library provides the *exclusive* implementation for CPU of node #0 (which is the application starting point), of node #4 and of node #7, the exclusive implementation for GPUs (i.e., GPU kernel) of node #1, while it provides

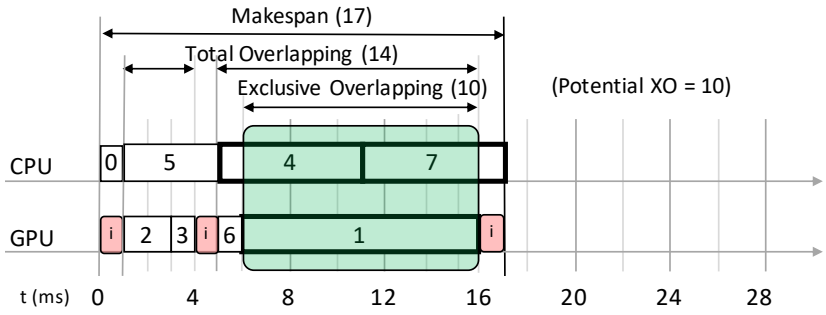
the multiple implementation (CPU implementation and an equivalent GPU kernel) of nodes #2, #3, #5, and #6. The table in Fig. 5.1 summarizes the execution time of each primitive when executed in isolation on the corresponding CEs.



(a): NVIDIA VisionWorks **standard** task scheduling order: 0, **1**, 2, 5, 3, 6, **4**, **7**



(b): **HEFT** task scheduling order: 0, 2, 5, **1**, 3, 6, **4**, **7**



(c): **Optimized HEFT** task scheduling order: 0, 2, 5, 3, 6, **1**, **4**, **7**

Fig. 5.2: Task scheduling algorithms of the DAG of Fig. 5.1: native NVIDIA VisionWorks (a), HEFT (b), and the proposed optimized HEFT (c).

For brevity, we assume the CPU-GPU data transfer time to be negligible in this example (it has been considered in the methodology and in our experimental anal-

ysis), and we consider the target heterogeneous system to consist of one CPU core and one GPU. We also assume task executions to be non-preemptive.

Fig. 5.2(a) represents the task mapping and scheduling of the application implemented by the NVIDIA VisionWorks runtime system. A similar approach is implemented by the AMD OpenVX (AMDOVX) runtime system. The mapping relies on the best *local optimization*, that is, a node is mapped on the GPU if there exists the corresponding GPU kernel in the library. The scheduling relies on the topological order of tasks in the DAG, and honours the topological order constraints among nodes. However, this approach does not implement overlapping among tasks.

Fig. 5.2(b) shows the mapping and scheduling of the proposed HEFT implementation for OpenVX, which takes advantage of both task overlapping and the mapping optimized at *system-level*. Starting from a task ranking generated as described in equation (1), one node at a time is mapped onto the CE that involves a better *system* execution time. This means that a node can be mapped on a CE that leads to a higher execution time at task level (see tasks of nodes #5, #3, and #6 in the example of Fig. 5.2(b)). Assuming that the nodes have the multiple implementation and not necessarily all the GPU kernels outperform the corresponding CPU primitives, the HEFT algorithm heuristically provides load balancing on the CEs by *overlapping* the task execution. As confirmed by our experimental results, this leads to improvements to the system performance (i.e., reduction to the application makespan). However, there are nodes that do not have multiple implementations. In this case, the iterative nature of task mapping and balancing of HEFT can lead to large idle periods. An example is the idle period on the GPU in Fig. 5.2(b), which could be avoided (or reduced) by an implementation of node #7 for GPU.

In general, the main limitation of HEFT in the context of heterogeneous architectures is that, by following the rank order, it maps one task at a time by guaranteeing the best load balancing at each iteration. It does not consider the single or multiple implementation of the nodes.

5.2 The proposed scheduling and mapping algorithm

Our idea is that the load balancing can be improved by prioritizing the overlapping between exclusive nodes, which we call *exclusive overlapping*. Considering the standard definition of *overlapping* between two tasks t and q as follows:

$$O(t,q) = \max\left(0, \min\left(t_{end}, q_{end}\right) - \max\left(t_{start}, q_{start}\right)\right), \tag{2}$$

where t_{start} and t_{end} are the starting and ending times of t, respectively. We define *exclusive overlapping (XO)* between two tasks t and q running on different CEs as follows:

$$XO(t,q) = \begin{cases} O(t,q), & \text{if } (\nexists\, t_{CPU} \wedge \nexists\, q_{GPU}) \\ & \quad \vee (\nexists\, t_{GPU} \wedge \nexists\, q_{CPU)}\ , \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

where t_{CPU} (t_{GPU}) represents the CPU implementation (GPU implementation) of task t.

Exclusive overlapping applies to nodes that cannot compete for the same CE due to exclusive implementations. It is a subset of the standard overlapping. We define the total overlapping and the total exclusive overlapping between tasks *of an application* A as follows:

$$O(A) = \sum_{t,q \in A} O(t,q), \tag{4}$$

$$XO(A) = \sum_{t,q \in A} XO(t,q), \tag{5}$$

Fig. 5.2(c) shows the exclusive overlapping between the tasks of the example. The main idea is that increasing XO can reduce idle times caused by the combination of DAG constraints and exclusive implementation (see for example the idle time on the GPU between instants 13 and 25 in Fig. 5.2(b)). Our experimental analysis shows that increasing XO corresponds to an increase of the standard overlapping and to an improvement of performance.

To increase XO, we propose an algorithm (*Algorithm* 1) that, starting from a given ranking list, it reorganizes the list to identify and generate *clusters* of exclusive nodes, i.e., sequences of exclusive nodes that are strictly consecutive in the ranking (see nodes #1, #4, and #7 in Fig. 5.2(c)).

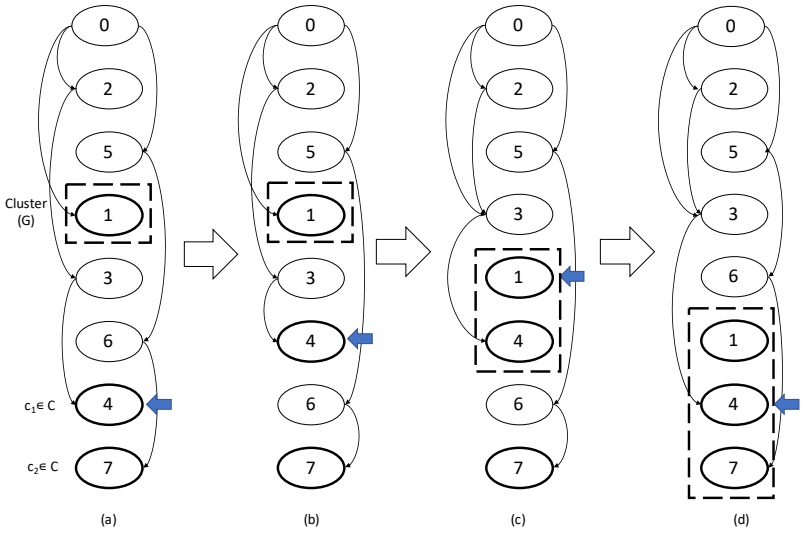


Fig. 5.3: Cluster generation step (`APPLY(rank, cluster)`) for the example in Fig. 5.1.

The algorithm starts by defining the standard HEFT ranking from the application graph (row 2). One node at a time, and for all nodes of the ranking (row 4), the algorithm identifies a new cluster starting from the next exclusive node of the

Algorithm 1 Cluster identification and generation

```
1:  procedure BUILDCLUSTER(graph)
2:      rank ← build_rank(graph)
3:      i ← 0
4:      while i < size(rank) do
5:          if rank[i] is exclusive then
6:              candidates ← rank[i]
7:              j ← i + 1
8:              while j < size(rank) do
9:                  if rank[j] is exclusive ∧∀p ∈ candidates, p ↛ rank[j] then
10:                     candidates ← candidates ∪ rank[j]
11:                 j ← j + 1
12:             total_cpu ← reduce_sum(candidates, t_cpu)
13:             total_gpu ← reduce_sum(candidates, t_gpu)
14:             C ← exclusive CPU nodes in candidates
15:             G ← exclusive GPU nodes in candidates
16:             if total_gpu < (total_cpu / n_cores) then
17:                 cluster ← G
18:                 for all c ∈ C do
19:                     t ← reduce_sum(cluster, t_cpu)
20:                     if |total_gpu − (t+c_cpu)/n_cores| < |total_gpu − t/n_cores| then
21:                         cluster ← cluster ∪ c
22:             else
23:                 for all g ∈ G do
24:                     t ← reduce_sum(cluster, t_gpu)
25:                     if |total_cpu/n_cores − t + g_gpu| < |total_cpu/n_cores − t| then
26:                         cluster ← cluster ∪ g
27:             APPLY(rank, cluster)
28:             i ← cluster_end + 1
29:         else
30:             i ← i + 1
        return rank
```

list (row 6). It searches among all the next nodes in the ranking that are exclusive and that do not have topological constraints in the DAG with the current cluster nodes (i.e., that are not in the same DAG path). The second condition is necessary to avoid serialization among the cluster nodes. The algorithm then calculates the total makespan of the cluster nodes for each CE (rows 12 and 13). The shortest among the calculated makespans characterizes the maximum XO of the cluster under generation. For the sake of clarity, in *Algorithm* 1, we considered two possible cluster makespans ($total_{cpu}$ and $total_{gpu}$). The algorithm completes the cluster identification by including all nodes (for the same CE) that give the shortest makespan and, incrementally, with the exclusive nodes that bring to a comparable makespan on the other CEs (rows 16-26). The first node of any other CE that causes makespan unbalancing starts a new cluster in the following iteration of the algorithm.

The algorithm implements the cluster generation by moving either the identified nodes up on the ranking or the cluster down on the ranking. All the identified nodes (i.e., *candidates* in *Algorithm* 1) and the cluster can be moved and made adjacent since, for the condition in row 9, they cannot have topological constraints against each other.

The `APPLY(rank,cluster)` function implements such a node shift in the ranking by considering the identified cluster. For the sake of clarity we show the two shift types for the running example (see Fig. 5.3). Considering that nodes #1, #4,

#7 have not topological constraints, G = $\{\#1\}$, C = $\{\#4, \#7\}$, $total_{cgu}$ = 10, and $total_{cpu}$ = 12, the cluster starts by node #1 (Fig. 5.3(a)). The algorithm annexes the CPU node #4 to the cluster in two steps. First (Fig. 5.3(b)) by moving node #4 upword in the ranking since it has not topological constraints with the switching nodes (node #6 in the example). Then (Fig. 5.3(c)) by moving downword the cluster since node #4 has topological constraints with node #3. For the same concept, the algorithm annexes node #7 to the cluster by moving downword the cluster (Fig. 5.3(d)), since node #7 has a topological constraint with node #6.

5.3 Experimental results

We evaluated the proposed algorithms by considering two categories of benchmarks. The first is the ORB-SLAM application [46] combined with the image recognition system based on Deep Learning (DL) [57]. We considered three different versions of the applications: *Monocular* with a 41 node DAG, *stereo* (81 nodes), and *4-stereo* (161 nodes). We used the standard KITTI input dataset[53] for the evaluation, which consists of video streams taken by a car driven around city blocks.

The second category is a set of synthetic DAGs. We designed a parametric DAG generator that generated, for our evaluation, around 40,000 DAGs with different characteristics: Size (from 20 to 250 nodes), node degree, execution times of CPU and GPU node implementations, exclusive implementation vs. multiple implementation of nodes, CPU/GPU speedup for nodes with multiple implementation. We collected the generated DAGs in two classes: *Tree*, for DAGs with high average degree, medium small diameters, and low standard deviation. *Linear* for the rest. The idea was to classify the DAGs depending on the average level of topological constraints among nodes.

For all benchmarks, we used the NVIDIA Jetson TX2 as target architecture, which is a prevalent low-power heterogeneous device used in industrial robots, machine vision cameras, and portable medical equipment. It consists of a dual-core Denver2 64-bit CPU + quad-core ARM A57 complex, and a 256-core Pascal GPU. We evaluated the embedding, mapping and scheduling process for 2, 3, 4, and 5 CPUs core + 1 GPU. We assigned one CPU core for the OpenVX runtime system and CPU/GPU synchronization.

Table 5.1 presents the results obtained by running the different configurations of ORB-SLAM+DL on the target architecture with the different CPU/GPU scenarios (i.e., #CPU cores enabled beside the GPU). The table shows the comparison of the different mapping and scheduling approaches considered in this thesis, i.e., NVIDIA VisionWorks (VW) [42], standard HEFT, and the optimized HEFT (XEFT). The schedule length ratio (SLR) normalizes the makespan over the maximum critical path.

Our results show that, as expected, HEFT sensibly improves the application performance w.r.t. the scheduling system currently released with the NVIDIA Vision Work library. The improvement ranges from a minimum of 33.7% to a maximum

ORB-SLAM version	CPU cores (#)	DL time (ms)	Max XO (ms)	Overlap (ms)		XO (ms)		Idle time (%)		XO / Max_XO (%)		Makespan (ms)			Speedup			SLR	
				HEFT	XEFT	HEFT	XEFT	HEFT	XEFT	HEFT	XEFT	VW	HEFT	XEFT	HEFT on VW	XEFT on VW	XEFT on HEFT	HEFT	XEFT
Monocular	2	15	30.0	43.1	46.0	10.7	24.5	45.9%	27.6%	35.8%	81.8%	74.7	39.9	**31.7**	46.6%	57.5%	**20.4%**	2.45	**1.95**
Monocular	3	15	44.1	62.1	70.2	18.7	32.5	31.5%	20.7%	42.3%	73.7%	74.7	30.2	**25.3**	59.6%	66.2%	**16.3%**	1.86	**1.56**
Monocular	4	15	44.1	76.0	80.2	26.5	29.4	28.2%	40.5%	60.0%	66.6%	74.7	26.5	**25.3**	64.6%	66.2%	**4.4%**	1.63	**1.56**
Monocular	5	15	44.1	90.1	82.0	37.8	31.3	21.8%	40.9%	85.6%	70.9%	74.7	**23.0**	24.4	69.2%	67.4%	-5.7%	**1.42**	1.50
Monocular	2	20	40.0	51.6	55.2	7.0	32.5	43.9%	15.5%	17.6%	81.2%	79.7	46.0	**32.7**	42.3%	59.0%	**29.0%**	2.16	**1.54**
Monocular	3	20	44.1	72.3	80.2	19.1	32.5	31.9%	30.8%	43.4%	73.7%	79.7	35.4	**30.3**	55.6%	62.0%	**14.3%**	1.66	**1.43**
Monocular	4	20	44.1	92.9	98.2	31.3	34.4	21.2%	30.1%	71.0%	77.9%	79.7	29.5	**28.7**	63.0%	64.0%	**2.6%**	1.39	**1.35**
Monocular	5	20	44.1	103.2	100.4	41.0	37.4	23.1%	35.1%	93.0%	84.7%	79.7	**26.9**	27.9	66.3%	65.0%	-3.8%	**1.26**	1.31
Monocular	2	25	44.1	60.3	70.4	0.3	36.8	42.9%	14.4%	0.7%	83.5%	84.7	52.8	**35.3**	37.7%	58.3%	**33.2%**	2.01	**1.34**
Monocular	3	25	44.1	84.9	92.0	16.6	34.2	29.3%	25.1%	37.6%	77.4%	84.7	40.0	**34.4**	52.8%	59.5%	**14.1%**	1.52	**1.31**
Monocular	4	25	44.1	107.3	109.4	32.9	37.4	20.2%	31.2%	74.7%	84.7%	84.7	33.6	**32.9**	60.3%	61.2%	**2.2%**	1.28	**1.25**
Monocular	5	25	44.1	119.2	119.2	42.3	42.3	22.0%	22.0%	95.9%	95.9%	84.7	30.6	**30.6**	63.9%	63.9%	**0.0%**	1.16	**1.16**
Stereo	2	15	30.0	66.4	67.8	5.7	21.1	50.7%	43.8%	18.9%	70.3%	118.8	67.4	**60.3**	43.3%	49.2%	**10.5%**	4.15	**3.71**
Stereo	3	15	45.0	92.5	96.6	8.8	25.5	40.7%	31.3%	19.6%	56.7%	118.8	52.0	**46.8**	56.3%	60.6%	**9.9%**	3.20	**2.88**
Stereo	4	15	60.0	115.4	121.4	17.2	45.1	32.8%	15.3%	28.7%	75.2%	118.8	42.9	**35.9**	63.9%	69.8%	**16.5%**	2.64	**2.21**
Stereo	5	15	75.0	133.0	137.0	15.6	41.5	32.4%	25.8%	20.8%	55.3%	118.8	39.4	**34.2**	66.9%	71.2%	**13.1%**	2.42	**2.11**
Stereo	2	20	40.0	74.2	76.6	9.8	30.4	48.8%	38.7%	24.4%	75.9%	123.8	72.4	**62.5**	41.5%	49.5%	**13.7%**	3.41	**2.94**
Stereo	3	20	60.0	101.9	110.0	6.4	46.1	42.2%	17.6%	10.7%	76.9%	123.8	58.8	**44.5**	52.6%	64.1%	**24.3%**	2.76	**2.09**
Stereo	4	20	80.0	122.1	127.1	12.4	50.6	39.3%	19.0%	15.5%	63.2%	123.8	50.3	**39.2**	59.3%	68.3%	**22.1%**	2.37	**1.85**
Stereo	5	20	88.2	146.3	147.0	16.0	41.5	34.2%	35.2%	18.1%	47.0%	123.8	44.4	**39.2**	64.1%	68.3%	**11.7%**	2.09	**1.85**
Stereo	2	25	50.0	81.0	87.0	0.0	36.7	51.2%	31.7%	0.0%	73.4%	128.8	83.0	**63.7**	35.6%	50.5%	**23.2%**	3.16	**2.43**
Stereo	3	25	75.0	110.8	123.1	0.0	57.3	44.2%	10.0%	0.0%	76.4%	128.8	66.2	**45.6**	48.6%	64.6%	**31.1%**	2.52	**1.74**
Stereo	4	25	88.2	134.1	147.1	6.2	50.6	40.7%	28.1%	7.0%	57.3%	128.8	56.6	**44.2**	56.1%	65.7%	**21.9%**	2.16	**1.68**
Stereo	5	25	88.2	163.4	167.8	13.0	44.9	33.4%	36.4%	14.8%	50.9%	128.8	49.1	**43.3**	61.9%	66.4%	**11.8%**	1.87	**1.65**
4-stereo	2	15	30.0	112.7	114.0	1.8	1.0	57.3%	54.3%	6.1%	3.2%	207.0	131.9	**124.7**	36.3%	39.8%	**5.5%**	5.63	**5.32**
4-stereo	3	15	45.0	155.5	159.6	8.3	0.9	47.8%	43.2%	18.5%	2.1%	207.0	99.3	**93.7**	52.1%	54.8%	**5.6%**	4.23	**3.99**
4-stereo	4	15	60.0	189.2	195.5	10.5	0.0	44.1%	39.0%	17.5%	0.0%	207.0	84.6	**80.2**	59.1%	61.3%	**5.2%**	3.61	**3.42**
4-stereo	5	15	75.0	217.8	228.7	12.4	13.3	42.0%	35.0%	16.5%	4.3%	207.0	75.1	**70.4**	63.7%	66.0%	**6.3%**	3.20	**3.00**
4-stereo	2	20	40.0	120.3	126.7	0.0	16.6	56.3%	48.6%	0.0%	41.4%	212.0	137.7	**123.2**	35.0%	41.9%	**10.6%**	5.87	**5.25**
4-stereo	3	20	60.0	164.2	179.5	7.8	21.8	48.1%	34.3%	13.0%	36.4%	212.0	105.5	**91.0**	50.2%	57.1%	**13.7%**	4.50	**3.88**
4-stereo	4	20	80.0	202.2	215.6	9.7	16.1	43.8%	32.9%	12.2%	20.1%	212.0	89.9	**80.3**	57.6%	62.1%	**10.6%**	3.83	**3.43**
4-stereo	5	20	100.0	237.0	248.6	13.7	61.4	40.1%	29.1%	13.7%	61.4%	212.0	79.1	**70.1**	62.7%	67.0%	**11.4%**	3.37	**2.99**
4-stereo	2	25	50.0	130.7	137.3	0.0	33.4	54.6%	43.9%	0.0%	66.8%	217.0	144.0	**122.4**	33.7%	43.6%	**15.0%**	5.48	**4.66**
4-stereo	3	25	75.0	181.4	188.8	0.0	27.1	45.1%	32.9%	0.0%	36.1%	217.0	110.1	**93.8**	49.3%	56.8%	**14.8%**	4.20	**3.57**
4-stereo	4	25	100.0	222.4	236.0	3.8	89.0	40.3%	22.9%	3.8%	89.0%	217.0	93.2	**76.6**	57.1%	64.7%	**17.8%**	3.55	**2.92**
4-stereo	5	25	125.0	258.3	276.5	8.1	114.4	38.1%	17.4%	6.4%	91.5%	217.0	83.5	**66.9**	61.5%	69.2%	**19.8%**	3.18	**2.55**

Table 5.1: Experimental results with ORB-SLAM+DL on Jetson TX2

of 69.2%. Then, the table shows that XEFT provides an exclusive overlapping degree that is higher than that provided by HEFT in almost all cases (see double column XO). The only case of XO reduction is with the simpler application versions (monocular) run on a large number of CPU cores (i.e., 5). For this reason, in these two contexts, also the overall performance improvement provided by XEFT against HEFT is slightly negative (-5.7% and -3.8%).

The idle time values show that this category of benchmarks scheduled with HEFT suffers from load imbalance. The efficiency of HEFT to provide exclusive overlapping is reported in column XO/Max_XO and it is higher with simpler applications and with low levels of maximum potential XO. The clustering effect of XEFT is a reduction of the idle times in the CEs and an increase of the XO efficiency. The two values improve by increasing the application complexity.

In general, XEFT provides a performance improvement w.r.t. HEFT up to 33.2% and, more importantly, it provides the same or better performance of HEFT with less architectural resources (e.g., with one less CPU core).

Figures 5.4 and 5.5 show the performance comparison between XEFT and HEFT on the synthetic benchmarks. The two figures identify each benchmark on the plot by considering two metrics evaluated with HEFT: the XO and the idle time. As an example, the rightmost side and top side of the plot group the benchmarks for which HEFT provides the highest XO and the highest idle time, respectively. For each benchmark,

the circle and the cross represent the improvement and loss of performance, respectively. The size of circles(crosses) represents the improvement(loss) measure.

The figures aim at understanding the correlation between the XEFT efficiency w.r.t. HEFT and the DAG characteristics. For both the synthetic classes, the results underline that XEFT generally outperforms HEFT with benchmarks for which (i) HEFT suffers from idle time, and (ii) the XO efficiency of HEFT is low. This class of benchmarks are grouped, in both figures, in the leftmost top side. XEFT cannot improve the HEFT performance if the benchmark, with HEFT, is already well balanced (low idle time) or already presents high XO efficiency. In these cases, XEFT can lead to a loss of performance up to 19%.

Figure 5.4 shows that HEFT already performs well with the main parts of DAGs of class *Tree*. This is due to the fact that they provide high potential overlapping, less node constraints and, as a consequence, HEFT generates low idle time. Since the potential XO is also related to the number of exclusive nodes, which has been generated with a Gaussian distribution for the analysis, XEFT improves the performance mostly in benchmarks with less than 80% XO of HEFT. Figure 5.5 shows that the rest of the benchmarks are more distributed over the space. Here it is evident the main contribution of XEFT on the leftmost top side of the plot, to which also the ORB-SLAM+DL benchmarks belong.

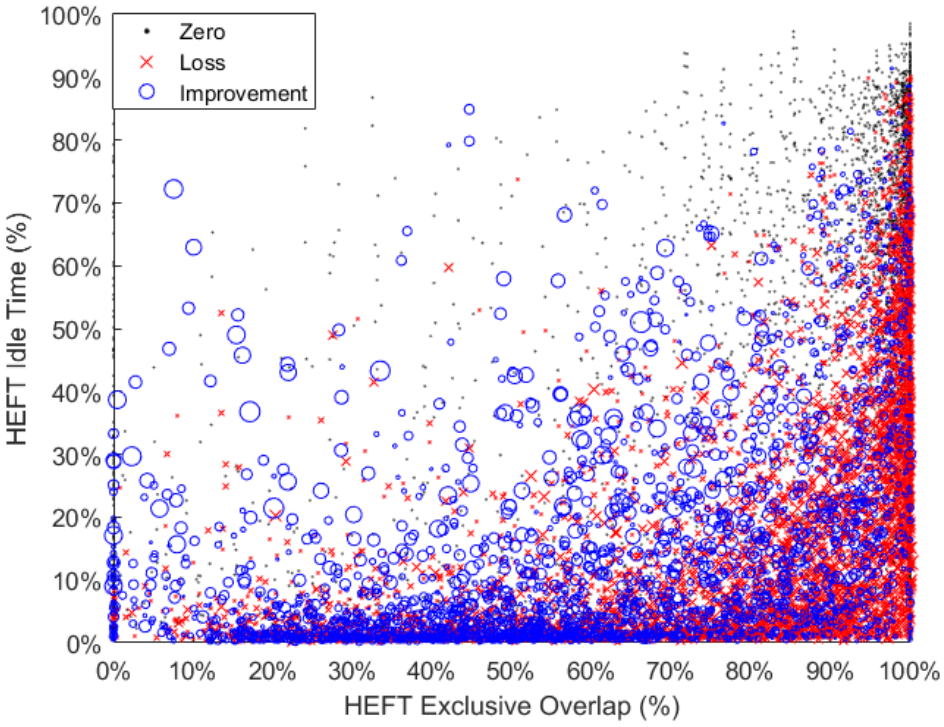


Fig. 5.4: Experimental results with the *Tree* class of synthetic DAGs on the Jetson TX2

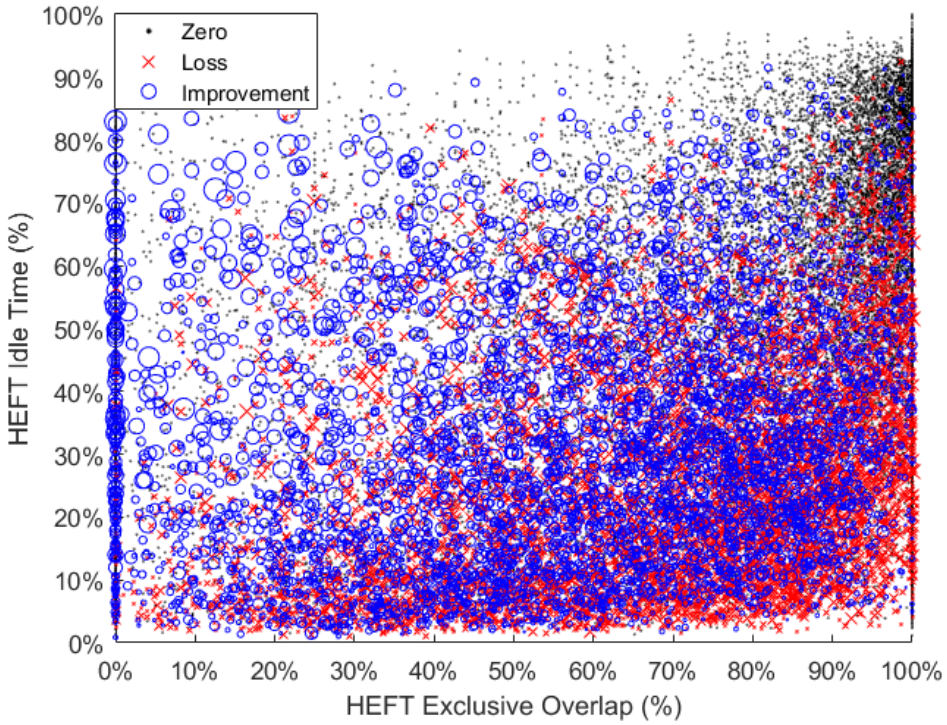


Fig. 5.5: Experimental results with the *Linear* class of synthetic DAGs on the Jetson TX2

6

Conclusion and future work

In this thesis, a thorough design flow has been presented to develop embedded vision applications. Specifically, three areas have been covered:

6.1 Summary of the proposed approach

- **Model-based design flow**. Unlike conventional software development, the algorithm of an embedded vision application has to be thought in a data-flow manner. This change leads to some refactor in primitive usages, but the verification of changes is made easy with the introduction of multi-level verification flow.
- **Polyglot parallel programming model and integration**. To fully utilize all the heterogeneous resources of the target device, several languages must co-exist. This thesis proposed a methodology that combines polyglot programming languages and environments to develop efficient applications in a very short amount of time, allowing portability across different hardware architectures.
- **An algorithm for scheduling and mapping of application tasks for performance enhancement**. This part is responsible to apply device-specific optimizations. The thesis presented the XEFT algorithm to overcome the HEFT limitations when applied in embedded devices. XEFT has been tested in a real-world case study, and it allowed a real-time execution with a 4x speedup w.r.t. the original implementation.

6.2 Directions for future research

While the presented design flow covers the full toolchain for the development of embedded vision applications, there are two areas which could be extended. The first resides in the modeling of conditional and loop instructions that are not present in the OpenVX semantic, to provide a smoother transition from the traditional development. The second is the extension of the methodology to more than two types of

accelerators in case, such as DSPs of FPGAs. A higher number of accelerators increases the heterogeneity level of primitives provided by vendors. This can increase the need of exclusive overlapping to reduce idle times in the application scheduling.

Summary of the proposed innovative contributions

This chapter reports the innovative contributions to the *State of the Art* by the work proposed in this thesis. The division resembles chapters organization. The contributions inside each group are in chronological ordered.

Model-based design flow

1. S. Aldegheri, D. D. Bloisi, J. J. Blum, *et al.*, "Fast and power-efficient embedded software implementation of digital image stabilization for low-cost autonomous boats," in *Field and Service Robotics*, M. Hutter and R. Siegwart, Eds., Cham: Springer International Publishing, 2018, pp. 129–144, ISBN: 978-3-319-67361-5
2. S. Aldegheri and N. Bombieri, "Extending OpenVX for model-based design of embedded vision applications," in *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2017, pp. 1–6. DOI: 10.1109/VLSI-SoC.2017.8203457
3. S. Aldegheri, S. Manzato, and N. Bombieri, "Enhancing performance of computer vision applications on low-power embedded systems through heterogeneous parallel programming," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2018, pp. 119–124. DOI: 10.1109/VLSI-SoC.2018.8644937

Polyglot parallel programming model and integration

1. S. Aldegheri, N. Bombieri, N. Dall'Ora, *et al.*, "A framework for the design and simulation of embedded vision applications based on OpenVX and ROS," in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5. DOI: 10.1109/ISCAS.2018.8351514
2. S. Aldegheri and N. Bombieri, "Integrating Simulink, OpenVX, and ROS for model-based design of embedded vision applications," in *VLSI-SoC: Opportunities and Challenges Beyond the Internet of Things*, M. Maniatakos, I. A. M.

Elfadel, M. Sonza Reorda, *et al.*, Eds., Cham: Springer International Publishing, 2019, pp. 178–197, ISBN: 978-3-030-15663-3
3. S. Aldegheri and N. Bombieri, "Rapid prototyping of embedded vision systems: Embedding computer vision applications into low-power heterogeneous architectures," in *2018 International Symposium on Rapid System Prototyping (RSP)*, 2018, pp. 63–69. DOI: 10.1109/RSP.2018.8631995

An algorithm for scheduling and mapping of application tasks for performance enhancement

1. S. Aldegheri, D. D. Bloisi, N. Bombieri, *et al.*, "Data flow ORB-SLAM for real-time performance on embedded GPU boards," in *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2019, pp. 1–6
2. S. Aldegheri, N. Bombieri, and H. Patel, "On the task mapping and scheduling for DAG-based embedded vision applications on heterogeneous multi/many-core architectures," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020, pp. 1–4

Bibliography

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 *Proc. of IEEE International Symposium on Embedded Multicore/Many-core SoCs*, 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 *Int. Conf. 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 *Proc. of IEEE International Conference on Control, Automation, Robotics and Vision*, 2016, pp. 1–6.

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

[9] H. Topcuoglu, S. Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[10] 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. DOI: 10.1088/1742-6596/1004/1/012011.

[11] P. Estérie, J. Falcou, M. Gaunard, J.-T. Lapresté, 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.

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

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

[14] ——, "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.

[15] 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.

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

[17] 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.

[18] 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.

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

[20] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande, "Accull: An openacc implementation with cuda and opencl support," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 871–882, ISBN: 978-3-642-32820-6.

[21] C. Su, P. Chen, C. Lan, L. Huang, and K. Wu, "Overview and comparison of opencl and cuda technology for gpgpu," in *2012 IEEE Asia Pacific Conference on Circuits and Systems*, 2012, pp. 448–451.

[22] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Computing*, vol. 38, no. 8, pp. 391 –407, 2012, APPLICATION ACCELERATORS IN HPC, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2011.10.002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001335.

[23] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[24] C. Nvidia, "Nvidia cuda c programming guide," *Nvidia Corporation*, vol. 120, no. 18, p. 8, 2011.

[25] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS)*, IEEE, 2009, pp. 1–314.

[26] A. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *Proc. od ACM/IEEE Design Automation Conference*, 2013.

[27] A. Goens, R. Khasanov, J. Castrillon, M. Hähnel, T. Smejkal, and H. Härtig, "TETRiS: A Multi-Application Run-Time System for Predictable Execution of Static Mappings," in *Proc. of ACM International Workshop on Software and Compilers for Embedded Systems*, 2017, pp. 11–20.

[28] G. Elliott, K. Yang, and J. Anderson, "Supporting Real-Time Computer Vision Workloads Using OpenVX on Multicore+GPU Platforms," in *Proceedings - Real-Time Systems Symposium*, 2016, pp. 273–284.

[29] M. Yang, T. Amert, K. Yang, N. Otterness, J. Anderson, F. Smith, and S. Wang, "Making OpenVX Really 'Real Time'," in *Proc of IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 80–93.

[30] A. Maurya and A. Tripathi, "Performance Comparison of HEFT, Lookahead, CEFT and PEFT Scheduling Algorithms for Heterogeneous Computing Systems," in *Proc. of ACM International Conference on Computer and Communication Technology*, 2017, pp. 128–132.

[31] M. A. Khan, "Scheduling for heterogeneous systems using constrained critical paths," *Parallel Computing*, vol. 38, no. 4, pp. 175 –193, 2012, ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2012.01.001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819112000105.

[32] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2014.

[33] S. AlEbrahim and I. Ahmad, "Task scheduling for heterogeneous computing systems," *Journal of Supercomputing*, vol. 73, no. 6, pp. 2313–2338, 2017.

[34] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, "DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm," in *Proc. of Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 27–34.

[35] H. Zhao and R. Sakellariou, "An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm," in *Euro-Par 2003 Parallel Processing*, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 189–194, ISBN: 978-3-540-45209-6.

[36] K. R. Shetti, S. A. Fahmy, and T. Bretschneider, "Optimization of the HEFT Algorithm for a CPU-GPU Environment," in *Proc. of International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2013, pp. 212–218.

[37] T. El-Gaaly, C. Tomaszewski, A. Valada, P. Velagapudi, B. Kannan, and P. Scerri, "Visual obstacle avoidance for autonomous watercraft using smartphones," in *Autonomous Robots and Multirobot Systems workshop*, 2013.

[38] J. Kalwa, M. Carreiro-Silva, F. Tempera, J. Fontes, R. S. Santos, M. C. Fabri, L. Brignone, P. Ridao, A. Birk, T. Glotzbach, M. Caccia, J. Alves, and A. Pascoal, "The morph concept and its application in marine research," in *MTS/IEEE OCEANS*, 2013, pp. 1–8.

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

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

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

[42] NVIDIA Inc., "VisionWorks," https://developer.nvidia.com/embedded/ vision-works.

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

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

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

[46] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: An open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017. DOI: 10.1109/TRO.2017.2705103.

[47] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *2011 International Conference on Computer Vision*, 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[48] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Computer Vision – ECCV 2006*, A. Leonardis, H. Bischof, and A. Pinz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 430–443, ISBN: 978-3-540-33833-8.

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

[50] 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 *Field and Service Robotics*, M. Hutter and R. Siegwart, Eds., Cham: Springer International Publishing, 2018, pp. 129–144, ISBN: 978-3-319-67361-5.

[51] R. Mur-Artal, J. M. M. Montiel, and J. D. TardÃ³s, "Orb-slam: A versatile and accurate monocular slam system," *IEEE Transactions on Robotics*, vol. 31, no. 5, pp. 1147–1163, 2015. DOI: 10.1109/TRO.2015.2463671.

[52] G. Klein and D. Murray, "Parallel tracking and mapping for small ar workspaces," in *2007 6th IEEE and ACM International Symposium on Mixed and Aug-*

mented Reality, 2007, pp. 225–234. DOI: 10.1109/ISMAR.2007.4538852.

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

[54] A. Kasar, "Benchmarking and comparing popular visual SLAM algorithms," *CoRR*, vol. abs/1811.09895, 2018. arXiv: 1811.09895. [Online]. Available: http://arxiv.org/abs/1811.09895.

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

[56] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

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

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