Data Flow ORB-SLAM for Real-time Performance on Embedded GPU Boards

Stefano Aldegheri[1], Nicola Bombieri[1], Domenico D. Bloisi[2], and Alessandro Farinelli[1]

***Abstract*— The use of embedded boards on robots, including unmanned aerial and ground vehicles, is increasing thanks to the availability of GPU equipped low-cost embedded boards in the market. Porting algorithms originally designed for desktop CPUs on those boards is not straightforward due to hardware limitations. In this paper, we present how we modified and customized the open source SLAM algorithm ORB-SLAM2 to run in real-time on the NVIDIA Jetson TX2. We adopted a data flow paradigm to process the images, obtaining an efficient CPU/GPU load distribution that results in a processing speed of about 30 frames per second. Quantitative experimental results on four different sequences of the KITTI datasets demonstrate the effectiveness of the proposed approach. The source code of our data flow ORB-SLAM2 algorithm is publicly available on GitHub.**

I. INTRODUCTION

Navigation is the main task for an autonomous mobile robot. In order to move from the current position A to a desired destination B, a mobile robot needs a map, to know its position on the map, and to have a plan to get from A to B, possibly selecting the most appropriate from a number of alternative routes. Simultaneous Localization and Mapping (SLAM) aims at processing data coming from robot sensors to build a map of the unknown operational environment and, at the same time, to localize the sensors in the map (also getting the trajectories of the moving sensors).

Many different types of sensors can be integrated in SLAM algorithms such as laser range sensors, encoders, inertial units, GPS, and cameras. In recent years, SLAM using cameras only has been actively discussed because cameras are relatively cheap with respect to other sensor types and their configuration requires the smallest sensor setup [1]. When the input for SLAM is visual information only, the technique is specifically referred to as visual SLAM (vSLAM).

vSLAM algorithms can be grouped according to three categories, namely *feature-based*, *direct*, and *RGB-D* approaches. In feature-based methods, geometric information from images is estimated by extracting a set of feature observations from the image in input and by computing the camera position and scene geometry as a function of these feature observations only [2]. Direct (or featureless) approaches aims at optimizing the geometry directly on the image intensities and use photometric consistency over the whole image as an error measurement. In RGB-D SLAM, dense depth enables the detection of planes that have no textures [3].

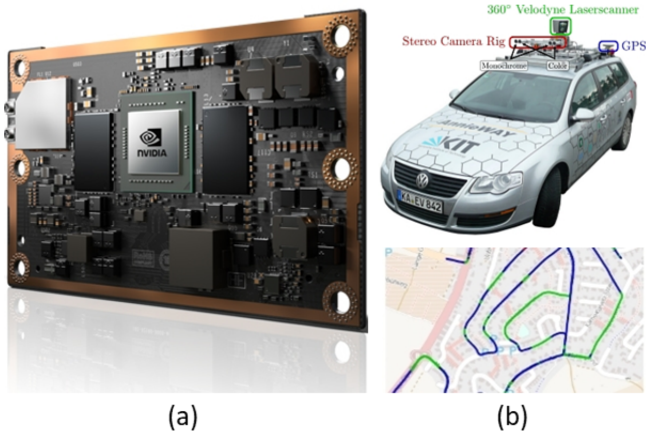

Fig. 1. (a) NVIDIA Jetson TX2 module. (b) The KITTI dataset [5]

One of the main challenges in vSLAM is to achieve real-time processing. Direct and RGB-D methods are computationally demanding and requires GPU computation to run in real-time. ORB-SLAM2 [4] is, at the moment, the most complete feature-based vSLAM system [1]. It works in real-time on standard CPUs, but not on embedded boards.

In this paper, we present a modified version of ORB-SLAM2 that runs in real-time on an NVIDIA Jetson TX2 embedded board (see Fig. 1). According to [6], ORB-SLAM2 is the package that provides the best result in terms of accuracy among the most popular sparse methods. However, it is a high-demanding algorithm in terms of CPU and memory usage [7], thus a careful computational load distribution is required to obtain real-time performance with hardware limitations.

The contributions of this work are three-fold:

1) We use a data flow paradigm to obtain a representation of the original algorithm as a graph, which allows to subdivide efficiently the computational load between CPU and GPU.
2) Experimental results demonstrate that, by balancing CPU/GPU usage, it is possible to achieve real-time performance on four different sequences of the KITTI dataset while maintaining good accuracy.
3) We provide on GitHub the complete source code optimized for real-time use on the NVIDIA Jetson TX2.

[1]Stefano Aldegheri, Nicola Bombieri, and Alessandro Farinelli are with the Department of Computer Science, University of Verona, Strada le Grazie 15 - 37134 Verona, Italy `nicola.bombieri@univr.it`

[2]Domenico D. Bloisi is with the Department of Mathematics, Computer Science, and Economics, University of Basilicata, Viale dell'Ateneo Lucano, 10 - 85100 Potenza, Italy `domenico.bloisi@unibas.it`

The remainder of the paper is structured as follows. Related work is discussed in Section II. The proposed method is presented in Section III. Experimental evaluation is shown in Section IV. Finally, conclusions are drawn in Section V.

II. Background and Related Work

ORB-SLAM2 [4] is a SLAM system that can work with data coming from monocular, stereo, and RGB-D cameras. The system consists of the following three main blocks (see Fig. 2):

Tracking and localization. This block is in charge of computing visual features, localizing the robot in the environment, and, in case of significant discrepancies between an already saved map and the input stream, communicating updated map information to the mapping block. The frames per second (FPS) that can be computed by the whole system strongly depends on the performance of this block.

Mapping. It updates the environment map by using the information (map changes) sent by the localization block. It is a computational time consuming block and its execution rate strictly depends on the agent speed. However, considering the actual agent speed of the KITTI datasets analysed in this work [5], it does not represent a system bottleneck.

Loop closing. It aims at adjusting the scale drift error accumulated during the input analysis. When a loop in the robot pathway is detected, this block updates the mapped information through a high latency heavy computation, during which the first two blocks are suspended. This can lead the robot to loose tracking and localization information and, as a consequence, the robot to get temporary lost. The computation efficiency of this block (running on-demand) is crucial for the quality of the final results.

The system is organized on three parallel threads, one per block. The use of parallel threads allows for obtaining real-time processing on an Intel Core i7 desktop PC with 16GB RAM [4].

The NVIDIA Jetson TX2[1] embedded board is an ideal platform to be mounted on research vehicles having a dual-core NVIDIA Denver2 plus quad-core ARM Cortex-A57, 8GB RAM, and integrated 256-core Pascal GPU. There are few examples of SLAM algorithms able to run in real-time on embedded boards: Abouzahir et al. [8] present a study of processing times of four different SLAM algorithms (i.e., Monocular FastSLAM2.0, ORB-SLAM, RatSLAM and Linear SLAM) under different embedded architectures (i.e., iMX6, panda ES, XU4 and TX1), finding that all the considered SLAM algorithms do not achieve real-time performance on those architectures. Nguyen et al. [9] report that ORB-SLAM takes 190 ms per frame on the NVIDIA Jetson TX1 embedded platform. Moreover, low-cost embedded board, such as Raspberry or Odroid, are not powerful enough for running ORB-SLAM2. For example, Nitsche et al. [10] were not able to run ORB-SLAM2 on the Odroid XU4 since tracking was quickly lost due to high processing time for each frame.

[1]https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/?section=jetsonTX2

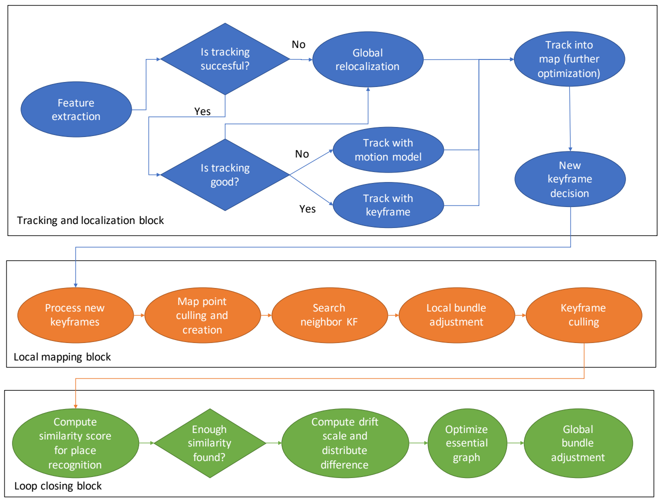

Fig. 2. Main blocks of the ORB-SLAM2 algorithm.

Instead of running SLAM onboard, it is possible to carry out the computation on a remote desktop PC and to send the results to the embedded board on the robot. An example is the work by Van Opdenbosch et al. [11]. They present an approach for remote vSLAM where local binary features are extracted at the robot, compressed and sent over a network to a centralized processing node. However, remote vSLAM is not usable in the absence of a reliable network, which is often the case in field deployments.

Giubilato et al. [7] analyse the challenges in implementing SLAM systems on embedded boards and highlight that most of the existing reviews and analysis of SLAM systems does not take into account the consequences of the implementation on an embedded computing platform. In particular, they compare the performance of ORB-SLAM2, SPTAM, and RTAB-MAP on a Jetson TX2 finding that ORB-SLAM2 shows a great robustness in challenging scenarios. However, they were not able to run ORB-SLAM2 in real time on the TX2 board. Vempati et al. [12] describe a system for autonomous spray painting using an unmanned aerial vehicle (UAV) equipped with a Jetson TX2. They use a depth-based vSLAM method achieving an onboard computation at 60 FPS. However, they do not provide a comparison of their method with other approaches, thus it is not possible to evaluate if their approach can be used to applications different from the considered spray painting use-case.

In this paper, we describe a method for optimizing the CPU/GPU computational load to achieve real-time performance for ORB-SLAM2 on the TX2 board. In particular, we provide a method for fully exploiting the potentiality of recent embedding boards using an heterogeneous (i.e., CPU + GPU) implementation of ORB-SLAM2.

III. Methods

We started the embedding process from the open source ORB-SLAM2 implementation proposed in [4], which orig-

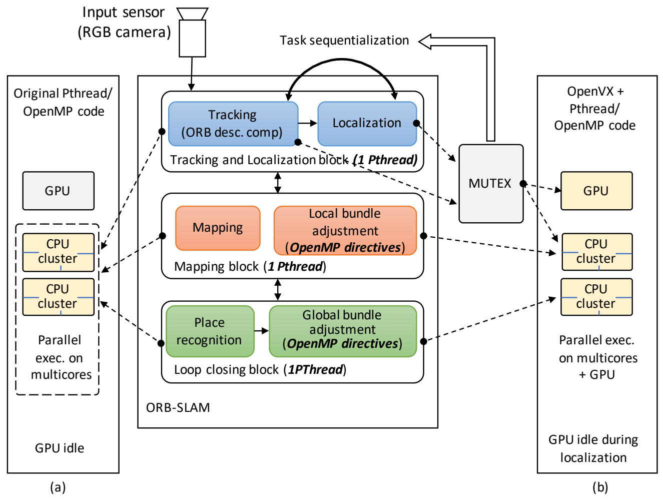
Input sensor
(RGB camera)
Task sequentialization
Original Pthread/ OpenMP code
GPU
CPU cluster
CPU cluster
Parallel exec. on multicores
GPU idle
Tracking (ORB desc. comp)
Localization
Tracking and Localization block (1 Pthread)
Mapping
Local bundle adjustment (OpenMP directives)
Mapping block (1 Pthread)
Place recognition
Global bundle adjustment (OpenMP directives)
Loop closing block (1PThread)
ORB-SLAM
MUTEX
OpenVX + Pthread/ OpenMP code
GPU
CPU cluster
CPU cluster
Parallel exec. on multicores + GPU
GPU idle during localization
(a)
(b)

Fig. 3. 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).

inally provides two levels of parallelism. The first level is given by the three main algorithm blocks (see Fig. 2), which are 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. 3a).

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). 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 [5]), it is the most important bottleneck that characterizes the processing rate in terms of supported FPS.

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[2] as shown in Fig. 4. 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

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

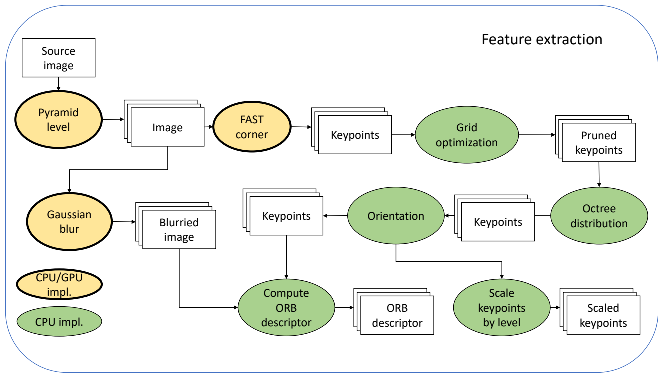

Fig. 4. DAG of the feature extraction block and the corresponding sub-block implementations (GPU vs. CPU).

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. 3b). 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. 3). 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 (see Section IV for information about our code on GitHub).

A. The Heterogeneous Implementation of ORB-SLAM

Fig. 5 shows the overview of the proposed software approach mapped into the Jetson TX2 architecture. To fully exploit the potential of the TX2 board, we combined different languages and parallel programming environments. In particular, we implemented control parts in C/C++, concurrent blocks on CPU cores through Pthreads, code chunks with parallelization directives in OpenMP, kernels for GPU computing in CUDA, while primitive-based parallelization of data-flow routines in OpenVX. We adopted OpenCV to implement the I/O communication protocols through standard data-structures and APIs. This allows the developed ORB-SLAM application to be portable and easy to be plugged into any other application that uses OpenCV.

The Jetson TX2 board is a shared-memory architecture that combines two CPU clusters with two symmetric GPU multiprocessors. The CPUs and the GPU share an unified memory space.

The stack layer involved in the concurrent execution of each software module consists of two main parts:

- *User-controlled stack.* It is a programmable stack

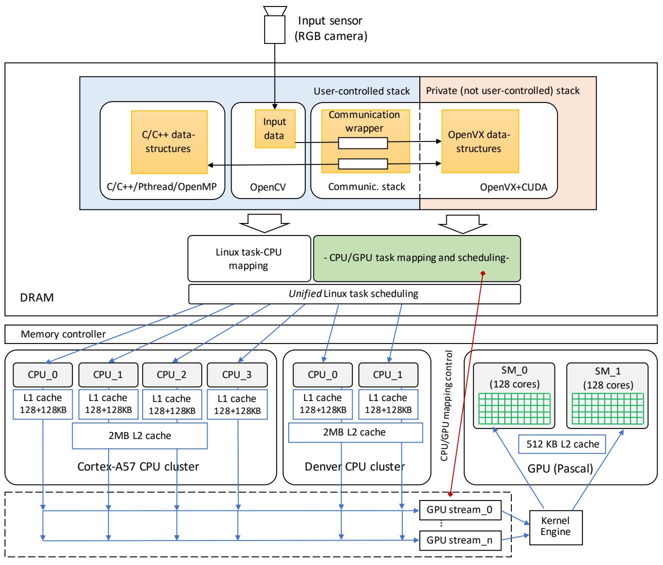

Fig. 5. System architecture: memory stack, task mapping, and task scheduling layers of an embedded vision application developed with the proposed method on the NVIDIA Jetson TX2 board.

that allows processes running on different CPUs (e.g., C/C++ processes, OpenCV APIs, Pthreads, and OpenMP processes) to communicate through shared memory.

- *Private stack*. It is generated and managed by the OpenVX runtime system, not programmable by the user, and it allows the communication among OpenVX graph nodes running on different CPUs or on the GPU.

The top of Fig. 5 shows the user-controlled and private stacks in our architecture.

The Linux Ubuntu operating system natively running on the NVIDIA Jetson maps the tasks related to the user-controlled stack to the CPU cores. The proposed runtime system maps the OpenVX tasks to the CPU cores or to GPU multiprocessors.

The two parts are associated into a single unified scheduling engine in order to:

1) Enable the full concurrency of the two parts;
2) Avoid sequentialization of the two sets of tasks;
3) Avoid synchronization overhead.

In this way, the operating system can schedule all the tasks mapped to the CPU cores (of both stack parts), while the OpenVX runtime system can control the GPU task scheduling, the CPU-to-GPU communication, and the CPU-to-GPU synchronization (i.e., GPU stream and kernel engine). To do that, we have developed 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.

Fig. 6 shows 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.,

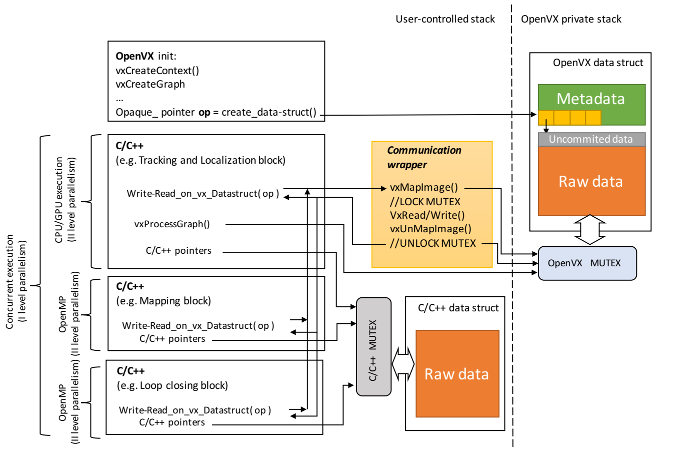

Fig. 6. The communication wrapper and its integration in the system.

pointers to private memory areas which layout is unknown to the programmer.

OpenVX read and write primitives (`Write-Read_on_vx_Datastructure()` in Fig. 6) have been defined to access the private data structures through the opaque pointers. The primitives are invoked from the C/C++ context and, through the communication wrapper APIs, they set a mutex mechanism to safely access the OpenVX data structures. The same mutex is shared with the OpenVX runtime system for the overall graph processing (`vxProcessGraph()` in Fig. 6). As a consequence, the mechanism guarantees synchronization during the accesses to the shared data structures between the OpenVX and C/C++ contexts when running concurrently on multicores. It is worth noting that the invocation of the overall graph process, which is performed in the C/C++ environment, starts the execution of the data-flow oriented OpenVX code. As shown in Fig. 6, 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. In conclusion, the proposed mutex-based communication wrapper allows for multi-level parallel execution of the application. In particular, it is crucial for synchronization among sub-blocks in their pipelined execution, as well as the simultaneous application of the different levels of parallelism.

IV. EXPERIMENTAL RESULTS

To evaluate the results of our modified version of ORB-SLAM2, we used four sequences from the KITTI dataset (see Fig. 7) as done in the original paper by Mur-Artal and Tardos [4].

The KITTI dataset [5] contains sequences of 1,242×375 images recorded at 10 FPS from a car in urban and highway

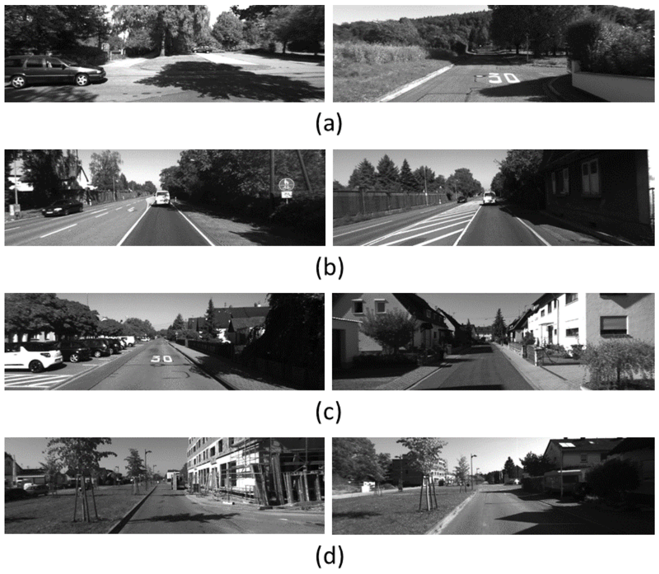

Fig. 7. Samples from the four sequences of the KITTI dataset used for evaluation. (a) Sequence 03. (b) Sequence 04. (c) Sequence 05. (d) Sequence 06.

environments (see Fig. 7). 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 (see Section III-A):

1) CPU + pipelining
2) CPU + GPU
3) CPU + GPU + pipelining

A. Runtime Performance

Table I shows the performance of the original ORB-SLAM2 code[3] and our three different versions (publicly available on GitHub[4]) 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 I 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.

B. Qualitative and Quantitative Evaluation and Metrics

Fig. 8 shows the qualitative results of our best implementation (i.e., CPU+GPU+pipelining). For the sake of space, we report only some parts of the analysed KITT sequences.

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) [13] 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. 8). 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 [13]. The percentage of reconstructed map is calculated from an initialization step. ATE and RPE are considered on the portion of the reconstructed map.

Table II 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.

V. CONCLUSIONS

Visual SLAM systems can achieve real-time performance on commercial desktop PCs. However, running vSLAM methods on embedded boards in real-time requires a modification of the original approaches to fully exploit the potential of the recent embedded boards equipped with GPU accelerators. In this paper, we presented a modified version of the ORB-SLAM2 algorithm that can achieve real-time performance on the NVIDIA Jetson TX2 board. Experimental results, conducted on three different publicly available datasets, demonstrate that:

- The proposed implementation is up to 4.5 times faster than the original code when running on the TX2 board;
- The accuracy of the modified version is comparable with the results generated by the original code.

We have released the source code of our system on GitHub, with examples and instructions so that it can be easily used by other researchers. As future work, we intend to mount a Jetson TX2 board on a mobile robot to test the use of our ORB-SLAM2 implementation combined to other computer vision applications for navigation tasks.

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

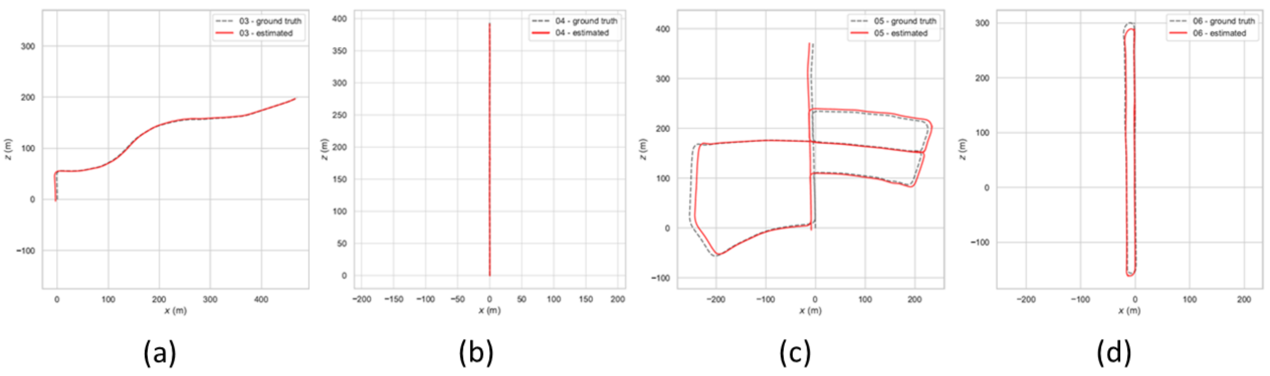

Fig. 8. Qualitative evaluation of the proposed ORB-SLAM application version CPU+GPU+pipelining on some parts of KITTI sequence 03 (a), sequence 04 (b), sequence 05 (c) and sequence 06 (d).

TABLE I
RUNTIME PERFORMANCE (FPS)

Sequence	03				04				05				06			
Resolution	1242×375				1226×370				1226×370				1226×370			
Input FPS	10				10				10				10			
version	CPU	CPU+ pipelining	CPU+ GPU	CPU+ GPU+ pipelining	CPU	CPU+ pipelining	CPU+ GPU	CPU+ GPU+ pipelining	CPU	CPU+ pipelining	CPU+ GPU	CPU+ GPU+ pipelining	CPU	CPU+ pipelining	CPU+ GPU	CPU+ GPU+ pipelining
FPS	6.99	15.23	17.90	**27.79**	7.47	15.96	20.70	**30.33**	6.54	15.56	18.52	**27.97**	6.68	16.03	19.66	**32.30**

TABLE II
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

ACKNOWLEDGMENT

This research work has been partially supported by the project ”Dipartimenti di Eccellenza 2018 2022” funded by the Italian Ministry of Education, Universities and Research (MIUR) and by GNCS-INDAM (Nicola Bombieri).

REFERENCES

[1] T. Taketomi, H. Uchiyama, and S. Ikeda, “Visual SLAM algorithms: a survey from 2010 to 2016,” *IPSJ Transactions on Computer Vision and Applications*, vol. 9, no. 1, 2017. [Online]. Available: https://doi.org/10.1186/s41074-017-0027-2

[2] J. Engel, T. Schöps, and D. Cremers, “LSD-SLAM: Large-scale direct monocular SLAM,” in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds., 2014, pp. 834–849. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-10605-2_54

[3] L. Ma, C. Kerl, J. Stckler, and D. Cremers, “CPA-SLAM: Consistent plane-model alignment for direct RGB-D SLAM,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1285–1291. [Online]. Available: https://ieeexplore.ieee.org/document/7487260

[4] 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. [Online]. Available: https://ieeexplore.ieee.org/document/7946260

[5] A. Geiger, P. Lenz, and R. Urtasun, “Are we ready for autonomous driving? the kitti vision benchmark suite,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[6] A. Quattrini Li, A. Coskun, S. M. Doherty, S. Ghasemlou, A. S. Jagtap, M. Modasshir, S. Rahman, A. Singh, M. Xanthidis, J. M. O’Kane, and I. Rekleitis, “Experimental comparison of open source vision-based state estimation algorithms,” in *2016 International Symposium on Experimental Robotics*, 2017, pp. 775–786. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-50115-4_67

[7] R. Giubilato, S. Chiodini, M. Pertile, and S. Debei, “An experimental comparison of ros-compatible stereo visual slam methods for planetary rovers,” in *2018 5th IEEE International Workshop on Metrology for AeroSpace (MetroAeroSpace)*, 2018, pp. 386–391. [Online]. Available: https://ieeexplore.ieee.org/document/8453534

[8] M. Abouzahir, A. Elouardi, R. Latif, S. Bouaziz, and A. Tajer, “Embedding SLAM algorithms: Has it come of age?” *Robotics and Autonomous Systems*, vol. 100, pp. 14 – 26, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889017301963

[9] D. Nguyen, A. Elouardi, S. A. R. Florez, and S. Bouaziz, “HOOFR SLAM system: An embedded vision SLAM algorithm and its hardware-software mapping-based intelligent vehicles applications,” *IEEE Transactions on Intelligent Transportation Systems*, pp. 1–16, 2018. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8556398

[10] M. A. Nitsche, G. I. Castro, T. Pire, T. Fischer, and P. D. Cristoforis, “Constrained-covisibility marginalization for efficient on-board stereo SLAM,” in *2017 European Conference on Mobile Robots (ECMR)*, 2017, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8098655

[11] D. V. Opdenbosch, M. Oelsch, A. Garcea, T. Aykut, and E. Steinbach, “Selection and compression of local binary features for remote visual slam,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 7270–7277. [Online]. Available: https://ieeexplore.ieee.org/document/8463202

[12] A. S. Vempati, I. Gilitschenski, J. Nieto, P. Beardsley, and R. Siegwart, “Onboard real-time dense reconstruction of large-scale environments for uav,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 3479–3486. [Online]. Available: https://ieeexplore.ieee.org/document/8206189

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