An FPGA Overlay for Efficient Real-Time Localization in 1/10th Scale Autonomous Vehicles

Andrea Bernardi, Gianluca Brilli, Alessandro Capotondi, Andrea Marongiu, Paolo Burgio
University of Modena and Reggio Emilia, Italy
name.surname@unimore.it

***Abstract*—Heterogeneous systems-on-chip (HeSoC) based on reconfigurable accelerators, such as Field-Programmable Gate Arrays (FPGA), represent an appealing option to deliver the performance/Watt required by the advanced *perception* and *localization* tasks employed in the design of *Autonomous Vehicles*. Different from software-programmed GPUs, FPGA development involves significant hardware design effort, which in the context of HeSoCs is further complicated by the system-level integration of HW and SW blocks. High-Level Synthesis is increasingly being adopted to ease hardware IP design, allowing engineers to quickly prototype their solutions. However, automated tools still lack the required maturity to efficiently build the complex hardware/software interaction between the *host* CPU and the FPGA accelerator(s). In this paper we present a fully integrated system design where a *particle filter* for LiDAR-based localization is efficiently deployed as FPGA logic, while the rest of the compute pipeline executes on programmable cores. This design constitutes the heart of a fully-functional *1/10th-scale* racing autonomous car. In our design, accelerated IPs are controlled locally to the FPGA via a *proxy core*. Communication between the two and with the *host* CPU happens via shared memory banks also implemented as FPGA IPs. This allows for a scalable and easy-to-deploy solution both from the hardware and software viewpoint, while providing better performance and energy efficiency compared to state-of-the-art solutions.**

I. Introduction

Autonomous Vehicles (AV) must elaborate in real-time an enormous amount of information from different sensors such as RGB cameras, LiDARs, radars and IMUs [1]–[3] to safely take informed decisions. To implement on-board *Domain Controllers*, embedded accelerators based on Field Programmable Gate Arrays (FPGAs) are becoming an appealing choice due to their flexibility, throughput and energy-efficiency [4].

The main limiting factor in adopting FPGAs is the complex development process. One promising technique for improving design productivity is to use a virtual hardware representation that overlays the original FPGA fabric, referred to as an *overlay* architecture [5]. Overlays are programmable, coarse-grained hardware abstraction layers on top of the FPGA hardware, abstracting the underlying hardware details as a software-managed task, which can be hooked to standard APIs for heterogeneous compute platform programming.

In this work, we explore how such technology can be adapted and tailored to deploy a fully functional AV stack in a simple and effective – but most important *scalable* – manner. Our target is a state-of-the-art heterogeneous FPGA SoC, the Xilinx Zynq Ultrascale+, employed as the *Domain Controller* of a 1/10th[1] scale racing vehicle prototype [6]. We provide a solution for offloading the *localization* component of a typical AV driving stack, where the vehicle matches the data coming from perception modules (i.e., from a LiDAR sensor) with a pre-built map for precise localization. We employ a well-known Monte-Carlo method called *Particle Filter* (PF) [7], which is a perfect candidate for acceleration on a highly parallel co-processor. We first describe a design space exploration of the PF accelerator IP, discussing the trade-offs among performance, accuracy and area occupation. This exploration enables fast performance tuning under different racing scenarios and vehicle hardware configurations.

Second, we show how to deploy a fully functional system where the entire application is considered. Besides the PF, the rest of the AV driving software stack is typically deployed on general-purpose cores in heterogeneous FPGA SoCs. This type of partitioning is *per-se* a complex task but also entails repeated movement of data between the host CPU and the accelerator, which hinders performance. We discuss how the proposed overlay architecture can optimize memory transfers among the two subsystems, thanks to a local core to the acceleration logic, which we call the *Proxy Core*. We present the integrated AV design step by step, highlighting how the overlay methodology simplifies the deployment.

An extensive experimental campaign illustrates the various aspects of the performance improvement enabled by hardware acceleration and the Proxy Core compared to the pure SW implementation running on the host cores, targeting two state-of-the-art embedded SoC from the Xilinx Ultrascale+ family. While FPGA acceleration of PF alone speeds up CPU execution by a factor of $2.5\times$, employing the overlay and the Proxy Core provides an additional $\approx 2\times$ speedup. Experiments were conducted on real-life racetracks, representing relevant corner cases for the targeted application domain.

This paper is structured as follows. Section II shows our reference hardware platform and the target AV software stack. Section III shows our novel methods both for the host and the accelerated platform. Section IV validates the effectiveness of our approaches, in terms of top/average speed and throughput achieved within the PF component. Section V highlights the state-of-the-art in the field of perception and localization on FPGA-based SoCs. Section VI concludes the paper.

[1] https://f1tenth.org/

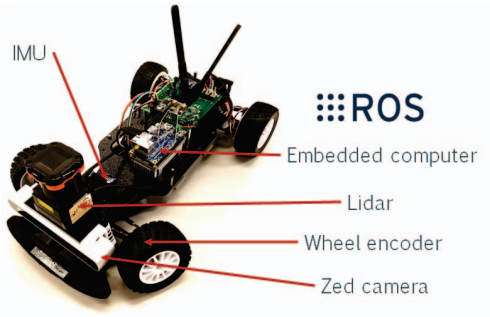

Fig. 1: The F1/10 vehicle prototype.

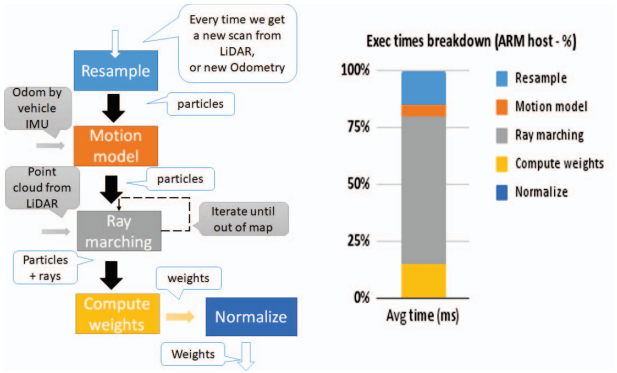

Fig. 2: Scheme and timing breakdown of the reference Particle Filter executed on a Xilinx ZCU102 board.

II. Background

Our target AV software stack is partitioned in a typical *perception-plan-act* design for racing vehicles. The main component is the LiDAR sensor that is used to localize the car in a known map through a method called *Particle Filter* (PF) [7] (see Figure 1). The optimal trajectory of the car is calculated offline via *Path Planning* [8]. The control loop is closed by a well-known *Pure Pursuit* [9] algorithm that proves itself an optimal tradeoff between complexity and effectiveness at the speeds our vehicle is expected to race.

Figure 2 shows the software implementation [7] of our reference Particle Filter in its building blocks. It is composed of five main stages, as follows: i) *Resample*: a high number of particles (i.e., position candidates) is randomly generated. ii) *Motion Model*: updates the particles with data from odometry. This data might not be accurate, and typically Gaussian noise is added. iii) *Ray Marching*: creates rays[2] centered in each particle, representing their $\langle x, y, \theta \rangle$ poses given the measured ranges from the LiDAR scanner. This step is iterated until rays cover all the map. iv) *Compute Weights*: associates a weight to each particle depending on how much it matches the "world" as seen by the LiDAR sensors. v) *Normalize*: flattens the weights within the [0,1] interval.

[2]The number of rays depends on the LiDAR's resolution (1081 in our Hokuyo UST-10LX).

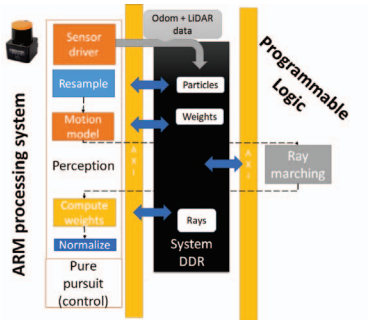

Fig. 3: Simple Particle Filter Partitioning between ARM host and Programmable Logic

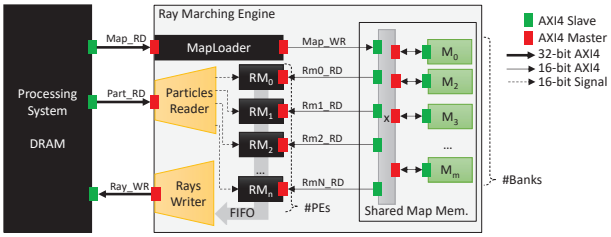

Fig. 4: System organization and Ray Marching Engine internal architecture, acting as co-processor for the PS.

Figure 2 also shows a timing breakdown of the software-only implementation of the algorithm running on a host core (ARM Cortex-A53) of a Xilinx ZCU102. For 1000 particles and 1081 rays, the system sustains a rate of *4 FPS*, still not close enough to the *40 Hz* of a common vehicle LiDAR. Being a Monte-Carlo method, it is highly data-parallel because all stages but the latter work on each particle independently. This makes the PF the primary candidate for FPGA acceleration.

III. FPGA-Overlay for Real-Time Localization

To improve end-to-end latency of the whole compute pipeline we need to tackle two problems: how to partition the computation among the host CPUs and the Programmable Logic, and how to design the chosen HW accelerator(s).

Driven by the profiling shown in Figure 2 we first design an HW Accelerator for the Ray Marching (which takes up to 90% of the latency of the whole PF), following "typical" partitioning strategies and FPGA design flows. Second, after assessing the performance gain of accelerating Ray Marching alone, we present a system-level solution, based on an FPGA *overlay* which integrates the Marching accelerator with a soft-core – called *Proxy Core* [10] – used to implement in a resource-efficient way the additional particle filter components.

Ray Marching Compute Engine (RME). Figure 3 shows a simple partitioning strategy for accelerating the Particle Filter, where the Ray Marching algorithm is implemented in hardware and the four remaining stages execute on one host processor. Figure 4 shows the Ray Marching Engine (RME) internal design and its interfaces.

An RME Engine has three AXI4 Master Interfaces: *Map_RD* port for initializing the map; *Part_RD* port used

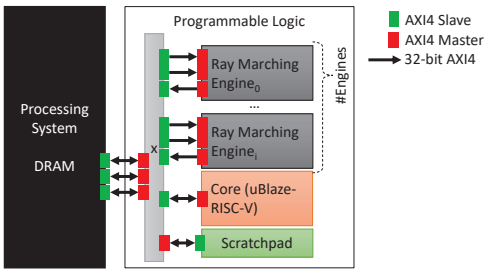

Fig. 5: Proposed FPGA-overlay architecture.

for reading the particles from the DRAM; and *Ray_WR* port used for writing out the calculated rays. The main internal components of a single RME are:

- A *MapLoader* for initializing the *Shared Map Memory*. This process is only done once at boot time;
- A multi-banked *Shared Map Memory* for storing the map of the environment;
- A parametric number of *Processing Elements (PEs)*. PEs execute in parallel the calculations for a single particle;
- *Particle Reader* and *Rays Writer* FSM for dispatching and collecting input and outputs.

The number of PEs and memory banks of the *Shared Map Memory* is parametric and can be adjusted according to the required latency and available resources. Moreover, as shown in Figure 4, more than one RME can be instantiated in a system connecting them through a simple crossbar.

The RME is implemented using Vivado 2019.2 HLS, exploits internally 16-bit fixed-point arithmetic, and supports up to 16 PEs for each engine. The number of RME is only limited by the resources available in the programmable logic. If more than one RME is employed, the accelerators can work asynchronously on a different set of particles.

Accelerating the sole Ray Marching stage would imply a delay in transferring data back and forth between the host and the accelerator. Executing more stages of the pipeline in the programmable logic and sharing the data structure containing particles and rays would through these stages improves the performance, as all the computation is done on a memory that is local to the accelerator. On the other hand, considering the profiling shown in the previous section it does not make sense to implement a HW IP for every other stage of the pipeline, as this would inefficiently use PL area. That's where our overlay and *Proxy Core* come into play, allowing for the execution of critical PF stages local to the memory where the HW accelerator has generated its output.

FPGA-Overlay Architecture. The proposed overlay design is composed of three main components: the Ray Marching Engine, as described above; a RISC soft-core deployed on the Programmable Logic; and a shared *Scratchpad* memory used for storing temporary buffers. Figure 5 shows how the design is connected to the PS subsystem. As an embodiment of our proxy core we rely on a Xilinx uBlaze Microcontroller. Open-source cores, such as RISC-V [10], [11] might also be considered.

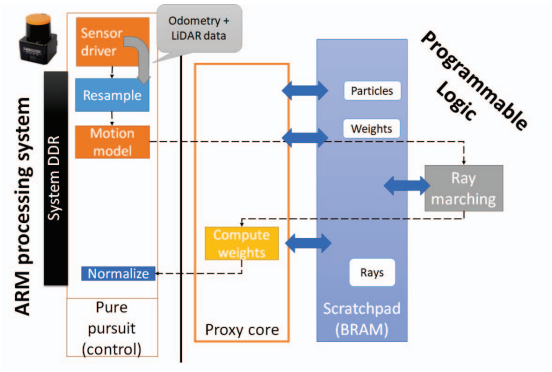

Fig. 6: Particle Filter Partitioning between ARM host, Proxy Core and RM Engine.

The proxy core enables offloading the Compute Weights stage into the Programmable Logic, removing the need to move data out of the Ray Marching. The new HW/SW partitioning of the particle filter algorithms is described in Figure 6. Our design still requires some data to be exchanged between the PL modules and the host. These data are the vehicle odometry, LiDAR point-cloud, and the particle filter structure (≈1KB), which we got by sensor drivers (namely, Ethernet and CAN) of the ARM GNU/Linux, and from the Resample step. It takes significantly less time to transfer than the whole particles+rays data structure, as said, 4KB.

IV. Experimental Results

A. Experimental Setup

We target two development boards, namely Avnet Ultra96 and Xilinx ZCU102. They have the same processing system (PS) but are distinguished mainly by the available resources of the programmable logic[3]. Since we target racing vehicles, our goal is to achieve the maximum *average speed* on a single lap, to minimise the lap time. At the same time, we also wish to maximise our *peak speed* because it is helpful in head-to-head situations, e.g., when overtaking. Hence, we adopt these two metrics, and more in detail, we want to show how close we can get to the *ideal* speeds, as computed by the planning algorithm. We focus on the localisation component, whose performance has room for improvement with HW acceleration. In the Particle Filter method, the critical parameter is the number of random particles/candidate positions that we can process before a new LiDAR frame is captured (at 40Hz). Intuitively, the higher the number of particles, the more precise we can localise the vehicle and hence the higher the speed we can sustain. At the same time, if we increase the computational workload, we might not be able to process a high(er) number of particles in time and lose localisation at high speeds. There is a minimum number of particles under which the localisation algorithm fails. Large tracks and tracks with long straights or wide lanes inherently require spawning a higher number

[3]The Ultra96 is equipped with a Xilinx Zynq UltraScale+ MPSoC ZU3EG, while the Xilinx ZU102 features a more powerful, ZU9EG SoC.

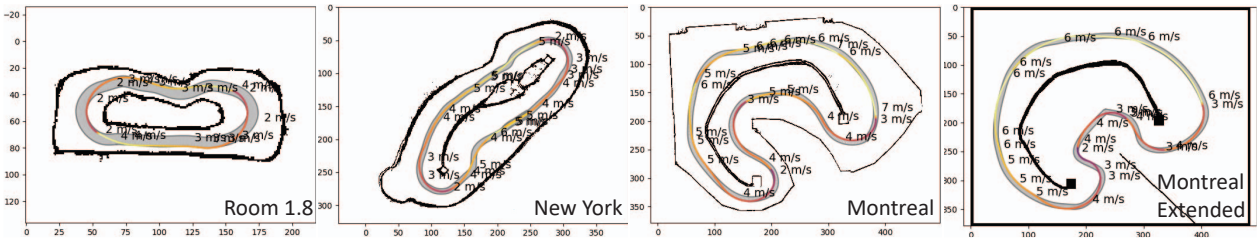

Fig. 7: Racetracks maps (dimensions in pixels), with ideal trajectory and speeds.

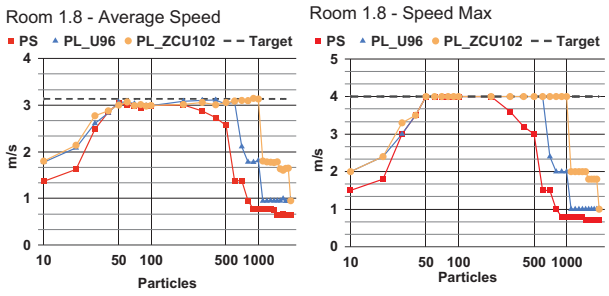

Fig. 8: Average and Max speed for *Room 1.8* Dataset.

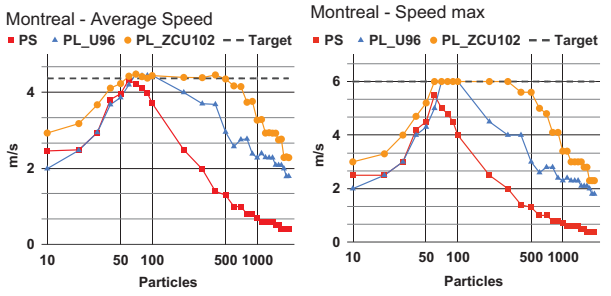

Fig. 10: Average and Max speed for *New York* Dataset.

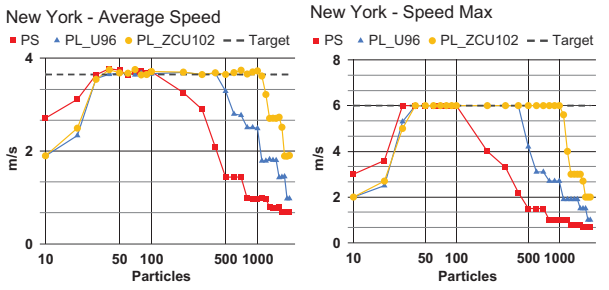

Fig. 9: Average and Max speed for *New York* Dataset.

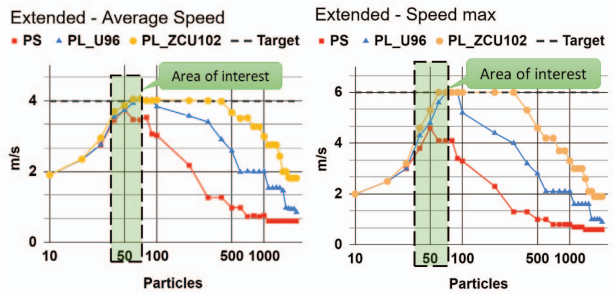

Fig. 11: Average and Max speed for *Extended Montreal*.

of particles. For this reason, we repeated our experiments on four tracks with increasing complexity. The first one (Figure 7, on the left) is a track we built in a room in our department (*Room 1.8*). The second and third ones, *New York* and *Montreal* come from the official F1/10 repositories from two actual races (Figures 7, on the centre). The fourth one, called *Extended* (Figure 7 on the right), is a modified version of the *Montreal* track, which we extend to a limit where the highest possible number of particles is required. This is used to stress corner-case behaviors. The variety of the maps shows, in a realistic scenario, the performance/accuracy boundaries of the PF method.

B. Performance improvement with RME

Charts in Figure 8 show how close we can get to the ideal speed (dashed line) for an increasing number of particles (x-axis) in the system running only on PS (red curve) and in that accelerated on Ultra96 and ZCU102 boards (blue and yellow curves, respectively). The points in the figures show the maximum speed achievable by our car. Above that speed and with that number of particles, the vehicles can no longer compute a safe trajectory, eventually smashing into the walls. All the three series (strategies/boards) can reach ideal max and average speed for ≈ 50 particles because this track is "simple" for the PF. Using the RME component onto the PL improves the FPS of the whole localisation step. Hence, we can safely process more particles before the next LiDAR frame. Intuitively, among the two boards, the more powerful ZCU102 outperforms the tiny Ultra96, which has less resources. Charts in Figure 9 are from the real *New York* F1/10 track. This track is even simpler than our *Room 1.8*, and the localisation succeeds for approximately 40 particles, with PS slightly outperforming PL-based approaches. However, running localisation in pure software cannot meet the LiDAR frequency for more than ≈ 100 particles, while, for instance, the ZCU102 SoC can process ten times that number of particles before missing this deadline.

The *Montreal* track (Figure 10) is more challenging and requires at least ≈ 70 particles to correctly localize and run at ideal speed. The PS version of the algorithm cannot reach the ideal Max speed, as shown on the right in Figure 10 while Ultra96 and ZCU102 can process up to 100 and 400 particles,

TABLE I: Latency breakdown expressed in milliseconds of Particle Filter Algorithm, varying the number of particles (p).

	SW-Only			RME			RME+Overlay		
p	**60**	**70**	**80**	**60**	**70**	**80**	**60**	**70**	**80**
Resample	0.65	0.99	1.09	0.65	0.99	1.09	0.65	0.99	1.09
Motion Model	0.45	0.52	0.58	0.45	0.52	0.58	0.45	0.52	0.58
Ray Marching	16.82	20.79	22.56	6.15	6.76	6.95	**1.05**	**1.20**	**1.38**
Compute Weights	0.91	1.06	1.24	0.91	0.91	1.24	*1.83*	*2.16*	*2.46*
Normalization	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01
Total (ms)	18.83	23.37	25.48	8.16	9.34	9.87	3.98	4.88	5.52
FPS	53.11	42.78	39.25	122.50	107.04	101.33	**250.95**	**204.92**	**181.09**

TABLE II: Resource Usage of different designs

	LUT		LUTRAM		FF	
Board	**Ultra96**	**ZCU102**	**Ultra96**	**ZCU102**	**Ultra96**	**ZCU102**
RME	23473	175540	781	3396	27929	186731
RME+Overlay	/	186430	/	3927	/	197154
Available	70560	274080	28800	144000	141120	548160
%RME	33.27	64.05	2.71	2.36	19.79	34.07
%RME+Overlay	/	68.02	/	2.73	/	35.97

	BRAM		DSP		BUFG	
Board	**Ultra96**	**ZCU102**	**Ultra96**	**ZCU102**	**Ultra96**	**ZCU102**
RME	207	614	58	452	3	3
RME+Overlay	/	667	/	457	/	6
Available	216	912	360	2520	196	404
%RME	**95.83**	67.32	16.11	17.94	1.53	0.74
%RME+Overlay	/	73.14	/	18.13	/	1.49

respectively. The *Extended Montreal* track, with its wide racing lane and bigger size, is the most complex one, where (see Figure 11) the pure SW implementation never reaches the ideal speed plateau. However, the performance for PL-based approaches keeps scaling with the number of particles, and we can safely keep the pace of LiDAR frames for up to 100 and 500 particles, respectively, for Ultra96 and ZCU102.

C. Proxy Core

We compare our proposed design (Figure 5) to one that only offloads the Ray Marching stage on the programmable logic (Figure 4). We focus on a small number of particles, i.e., the area of the previous charts where we are close to the performance plateau, hence where we can localise the vehicle effectively (see Figure 11). We plot performance profiles of the three design configurations and for 60, 70 and 80 particles, respectively, fixing the optimal number of 1081 rays. Results are shown in Table I, for the different stages of the Particle Filter component, namely (from left to right) for i) a pure SW implementation running on the ARM core; ii) a system where only the Ray Marching is implemented in FPGA PL, and iii) our design with the Proxy core. The two accelerated systems require data movements to copy in/out particles and data structures to a non-paged area of the DRAM to properly feed the accelerator. These transfers take approximately 0.03ms. Moreover, the design (ii) also requires particles and rays to be copied out from the Ray Marching stage, which takes approximately 30ms. In contrast, our design (iii) only needs to move out the results of the entire Particle Filter/Normalise stage, i.e., the particle weights. With this design, we further improve the end-to-end performance of the Particle Filter by $\approx$2-3$\times$, compared to a design with the RME only.

We undergo a detailed assessment of our HW/SW partitioning and show end-to-end latency (in ms) and throughput (FPS) metrics. The latter is representative of the needs of modern AD systems. The LiDAR is the primary sensor for the perception stage, while the former gives an intuition of the reaction time of the algorithm. Camera-based perception systems are also an interesting option. Still, their effectiveness in the localisation task has not surpassed that of the LiDAR-based ones yet, so we don't consider them in this work. Table II shows the area occupancy of our design as per the output of the Vivado tools. In the Ultra96 board, where the FPGA resources are limited, the IP alone occupies 96% of the BRAM. There is no room for the Proxy Core. On the larger ZCU102 our complete design with the HW IP and overlay uses 75% of the BRAM.

A final important thing to note is that while the Ray Marching IP could be improved, the primary outcome is that **this design we propose is completely scalable** because the Proxy Core is a completely programmable RISC architecture.

V. Related work

Particle Filter is one of the most used methodologies for implementing the localization task on an AD software stack. However, there are only two practical algorithmic approaches, one based on Bresenham [12] ray casting algorithm, while the other is based on ray marching [7]. The latter is, on average faster, because it makes longer steps along the query ray, thereby avoiding unnecessary memory reads. In particular, with 1000 particles and 61 rays, the reference Python implementation of Walsh et al. [7] reaches 1.47 FPS. It doesn't exploit any hardware acceleration. Also, it features a highly-optimized data structure called Compressed Directional Distance Transform (CDDT) to represent the value range for each discrete state $\langle x, y, \theta \rangle$ and particles. Unlike them, we compute the ranges of each particle dynamically and not statically by accessing a data structure made offline, allowing a precise localization. We accelerate on FPGA the additional dynamic calculation of every ray for every particle. There are also a few approached to ray marching based on deep learning, such as Lin Bai's [13]. Our deterministic sensor model functions are based on statistical calculations for comparing particle weights, and their precision can be increased by simply adding more particles and rays.

The distinctive aspect of our work is that our acceleration strategy employs a mix of hardware IPs and soft proxy cores in the accelerator logic. There are multiple literatures that partition the application between "hard" cores (such as the ARM complex of the Zynq SoC) and FPGA IPs. In

Section IV we show how our approach removes unnecessarily big data movements and outperforms such a design. D. Giri et al. [14]–[16] propose *Embedded Scalable Platform (ESP)* a novel approach to SoC design and reconfigurability that a programmable logic-based system can offer, proving excellent flexibility on a basic ASIC design and hardware abstraction.

K. Sugiura and H. Matsutani present an efficient hardware implementation for Simultaneous Localization and Mapping (SLAM) methods [17], sustaining a relatively small number of particles, despite a very efficient data compression strategy. The speed up the process of matching scanning on the FPGA in 2D LiDAR SLAM method called GMapping is based on the Rao-Blackwellized Particle Filter algorithm.

Eisoldt et. al [18] propose an approach to integrate reconfigurable SoCs into Robot Operating System (ROS). This approach is very similar to the one we use to make the accelerated part on FPGA interact with the rest of the driving stack. They use Kernel-space drivers. Instead, we work at the user-space level, increasing software flexibility and modularity thanks to 3D-party libraries.

Similarly to us, A. Kurth et al. [19], [20] proposed a heterogeneous platform combining ARM Cortex-A host coupled with a cluster of RISC-V cores, they use homogeneous clusters while in this case it is hardware accelerator.

VI. Conclusion

In this work, we proposed an original system design for next-generation automotive domain controllers based on reconfigurable embedded accelerators, such as the Xilinx Ultrascale+ SoC. With our design, one can deploy computational-intensive tasks for autonomous driving systems efficiently and in a scalable manner, with a high degree of flexibility and power efficiency. We target a real software stack for Autonomous Vehicles and prove how it is possible to offload the most timing consuming part of it, namely a localization kernel based on a *particle filter* Montecarlo method, in such a way that it outperforms both pure software solutions and traditional mixed hardware/software solutions even in with a non-optimal (HLS-based) implementation of the hardware module. Finally, the proposed system can be used to drive a 1/10th scale model of a car efficiently. In the future, we plan to use it on a real vehicle and explore its applicability to more complex modules, such as object detection with deep learning accelerators.

VII. Acknowledgements

The authors have received funding from the ECSEL-JU projects COMP4DRONES (No. 826610) and AI4CSM (No. 101007326).

References

[1] NVIDIA, "NVIDIA Jetson AGX Xavier Delivers 32 TeraOps for New Era of AI in Robotics," 2018. [Online]. Available: https://devblogs.nvidia.com/nvidia-jetson-agx-xavier-32-teraops-ai-robotics/

[2] J. Redmon, "YOLO: Real-Time Object Detection." [Online]. Available: https://pjreddie.com/darknet/yolov2/

[3] Z. Zhu, D. Liang, S. Zhang, X. Huang, B. Li, and S. Hu, "Traffic-sign detection and classification in the wild," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2110–2118.

[4] M. Verucchi, G. Brilli, D. Sapienza, M. Verasani, M. Arena, F. Gatti, A. Capotondi, R. Cavicchioli, M. Bertogna, and M. Solieri, "A systematic assessment of embedded neural networks for object detection," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1. IEEE, 2020, pp. 937–944.

[5] X. Li, A. K. Jain, D. L. Maskell, and S. A. Fahmy, "A time-multiplexed fpga overlay with linear interconnect," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 1075–1080.

[6] M. O'Kelly, V. Sukhil, H. Abbas, J. Harkins, C. Kao, Y. V. Pant, R. Mangharam, D. Agarwal, M. Behl, P. Burgio *et al.*, "F1/10: An open-source autonomous cyber-physical platform," *arXiv preprint arXiv:1901.08567*, 2019.

[7] C. H. Walsh and S. Karaman, "Cddt: Fast approximate 2d ray casting for accelerated localization," *Computer Science, Engineering, Mathematics 2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[8] T. Lipp and S. Boyd, "Minimum-time speed optimisation over a fixed path," *International Journal of Control*, vol. 87, no. 6, pp. 1297–1311, 2014.

[9] M. Samuel, M. Hussein, and M. Binti, "A review of some pure-pursuit based path tracking techniques for control of autonomous vehicle," *International Journal of Computer Applications*, vol. 135, pp. 35–38, 02 2016.

[10] G. Bellocchi, A. Capotondi, F. Conti, and A. Marongiu, "A RISC-V-based FPGA Overlay to Simplify Embedded Accelerator Deployment," in *2021 24th Euromicro Conference on Digital System Design (DSD)*. IEEE, 2021, pp. 9–17.

[11] D. Rossi, F. Conti, A. Marongiu, A. Pullini, I. Loi, M. Gautschi, G. Tagliavini, A. Capotondi, P. Flatresse, and L. Benini, "PULP: A parallel ultra low power platform for next generation IoT applications," in *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 2015, pp. 1–39.

[12] L. Honglin, "Research and implementation of the fundamental algorithms of computer graphics based on vc," *Proceedings of the 2016 6th International Conference on Management, Education, Information and Control (MEICI 2016)*, 2016.

[13] L. Bai, Y. Lyu, X. Xu, and X. Huang, "Pointnet on fpga for real-time lidar point cloud processing," *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2020.

[14] D. Giri, K.-L. Chiu, G. Di Guglielmo, P. Mantovani, and L. Carloni, "ESP4ML: Platform-based design of systems-on-chip for embedded machine learning," *Design, Automation and Test in Europe Conference Exhibition (DATE), Grenoble, France, 2020*, 2020.

[15] D. Giri, K.-L. Chiu, G. Eichler, P. Mantovani, N. Chandramoorthy, and L. Carloni, "Ariane + NVDLA: Seamless Third-Party IP Integration with ESP," *Fifth Workshop on Computer Architecture Research Directions. CARD 2019*, 2019.

[16] D. Giri, P. Mantovani, and L. Carloni, "Runtime reconfigurable memory hierarchy in embedded scalable platform," *ASPDAC '19: Proceedings of the 24th Asia and South Pacific Design Automation Conference*, 2019.

[17] K. Sugiura and H. Matsutani, "An fpga acceleration and optimization techniques for 2d lidar slam algorithm," *ArXiv*, vol. abs/2006.01050, 2020.

[18] M. Eisoldt, S. Hinderink, M. Tassemeier, M. Flottmann, J. Vana, T. Wiemann, J. Gaal, M. Rothmann, and M. Porrmann, "ReconfROS: Running ROS on Reconfigurable SoCs," in *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, 2021, pp. 16–21.

[19] A. Kurth, V. Pirmin, A. Capotondi, A. Marongiu, and L. Benini, "HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA," *First Workshop on Computer Architecture Research with RISC-V (CARRV 2017)*, 2017.

[20] A. Kurth, A. Capotondi, V. Pirmin, L. Benini, and A. Marongiu, "Hero: an open-source research platform for hw/sw exploration of heterogeneous manycore systems," *2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems, ANDARE 2018 - A Workshop part of PACT 2018 Conference.*, 2018.