Building the Computing System for Autonomous Micromobility Vehicles: Design Constraints and Architectural Optimizations

Bo Yu
PerceptIn
bo.yu@perceptin.io

Wei Hu
PerceptIn
wei.hu@perceptin.io

Leimeng Xu
PerceptIn
leimeng.xu@perceptin.io

Jie Tang
South China University of Technology
cstangjie@scut.edu.cn

Shaoshan Liu
PerceptIn
shaoshan.liu@perceptin.io

Yuhao Zhu
University of Rochester
yzhu@rochester.edu

***Abstract*—This paper presents the computing system design in our commercial autonomous vehicles, and provides a detailed performance, energy, and cost analyses. Drawing from our commercial deployment experience, this paper has two objectives. First, we highlight *design constraints* unique to autonomous vehicles that might change the way we approach existing architecture problems. Second, we identify *new architecture and systems problems* that are perhaps less studied before but are critical to autonomous vehicles.**

***Keywords*—Autonomous vehicles; self-driving cars;**

I. Introduction

Micromobility is a rising transport mode wherein lightweight vehicles cover short trips that massive transit ignore. According to US Department of Transportation, 60% of vehicle traffic is attributed to trips under 5 miles [1], [2]. Transportation needs in short trips are disproportionally under-served by current mass transit systems due to high cost, which affects the society profoundly. For instance, people who have difficulties getting to and from transit stations—the perennial first-mile/last-mile challenge [3], [4], [5]—forego job opportunities, accesses to healthy food, and preventative medical care [6]. Micromobility bridges transit services and communities' needs, driving the rise of Mobility-as-a-Service [7].

Over the past three years, we have built and commercialized an autonomous vehicle system for micromobility. Our vehicles so far have been deployed in the US, Japan, China, and Switzerland. Drawing from the real product that is deployed and currently operating in the field, this paper presents the opportunities and challenges of building *on-vehicle computing systems* for autonomous vehicles. We introduce our workloads and their characteristics, describe the design constraints we face, explain the trade-offs we make driven by the constraints, and highlight optimization opportunities through characterizing the end-to-end system. This paper complements prior academic studies [8], and provides contexts to compare autonomous vehicles with other autonomous machines such as mobile robots and unmanned aerial vehicles [9], [10], [11], [12], [13], [14].

Designing the on-vehicle computing system for an autonomous vehicle faces a myriad of constraints such as latency, throughput, energy, cost, and safety. While these constraints also govern conventional computing system design, we must now analyze them in the broad context of an end-to-end vehicle. To that end, we introduce generic analytical models that let us reason about the constraints in any autonomous vehicle, and use concrete data measured from our vehicles to highlight the constraints we face (Sec. III).

Leveraging these quantitative models, the rest of the paper presents and characterizes our on-vehicle processing system, which we dub "Systems-on-a-Vehicle" (SoV). The SoV processes sensory information to generate control commands, which are sent to the actuator to control the vehicle.

We first introduce the software pipeline exercised by the SoV in our deployed vehicles (Sec. IV). While individual algorithmic building blocks such as perception and planning have been explored before in the literature [15], [9], [10], [13], we focus on the data-flow across the building blocks and their inherent task-level parallelisms (TLP). We also describe the hybrid proactive-reactive design in our software pipeline, which ensures safety with low compute cost.

Based on the software pipeline, we then describe the computing platform design (Sec. V). We quantitatively demonstrate the limitations of two obvious options for building the computing platform: off-the-shelf mobile Systems-on-a-chip (SoCs) and existing automotive chips (Sec. V-A). In light of the limitations, we introduce our custom computing platform, taking into account the inherent TLPs, programmability, ease of deployment, and cost in the broad context of the SoV (Sec. V-B). We present a performance, power, and cost breakdown of the computing platform (Sec. V-C), which shows several counter-intuitive observations. For instance, sensing contributes significantly to the end-to-end latency, whereas motion planning is insignificant to us.

While the computing platform is critical, hyper-optimizing it may not lead to the best overall solution. In cases where computing could be directly replaced by sensing, accelerating the computing algorithms provides little benefits. The interactions of computing and sensing in the SoV provides new optimization opportunities that lead to greater overall gains. We discuss two concrete instances where we co-design sensing with computing reduces the compute cost

while improving the navigation quality and safety (Sec. VI).

To this day, autonomous driving is still largely an obscure segment with a dearth of concrete data and technological details publicly available. We hope that by anatomizing a complete commercial system and reporting our end-to-end design with unobscured, unnormalized data, we could promote open-source hardware platforms for autonomous vehicles in the decade to come, which empowers everyone to innovate on algorithms and build competitive products, much like how machine learning (ML) platforms today have led to an explosion of ML-enabled applications and services. In summary, this paper makes the following contributions:

- We introduce real autonomous vehicle workloads, and present unobscured, unnormalized data (power, cost, latency) measured from our deployed vehicles that future research can build on.
- We present simple, generic models of latency, energy, and cost constraints for designing autonomous vehicles' computing systems. We then use concrete parameters measured from our vehicles to highlight the design decisions we make. For instance, adding an additional on-vehicle server to the SoV would reduce driving time by at least 3%.
- We present a detailed performance characterization of our vehicles. The latency of the on-vehicle computing system contributes to 88% of the end-to-end latency, and the rest is attributed to mechanical latency. Sensing, while less-studied, constitutes almost 50% of the SoV latency, and is a lucrative target for optimization.
- We highlight that the computing system shouldn't be optimized alone. We present two case-studies to show that co-optimizing the sensory and computational components is sometimes more effective.
- Hardware acceleration on FPGA improves perception latency by 1.6×, translating to 23% end-to-end latency reduction on average. Runtime partial reconfiguration provides a cost-effective solution to balance programmability and efficiency.

II. Background and Context

We first briefly introduce our products and deployment in order to provide contexts for the rest of the paper. We then overview the end-to-end infrastructure that we deploy to support its autonomous vehicles.

II-A. Micromobility and Our Vehicles

McKinsey estimates that the micromobility market will grow to up to $500 billion in 2030, about one quarter of the entire autonomous-driving market [2]. We see micromobility at the forefront of adopting and popularizing autonomous vehicle technologies, because micromobility trips usually take place in constrained environments, wherein deploying autonomous vehicles meets less pushback.

We provide two autonomous vehicles designs: 2-seater pods targeting private transportation experiences, and 8-seater shuttles targeting public autonomous driving transportation services. Both designs are capped at 20 mph to suit the unique needs of micromobility. Our vehicles are fully autonomous without requiring human driver's intervention, complying with level 4 autonomous vehicle defined by the US Department of Transportation's National Highway Traffic Safety Administration (NHTSA) [16].

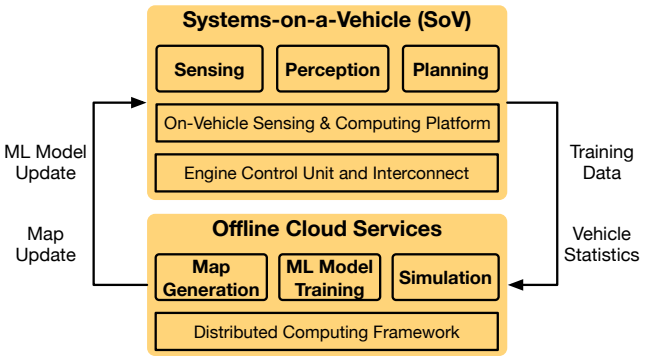


Fig. 1: Our autonomous driving infrastructure. This paper focuses on the on-vehicle processing system, i.e., SoV.

The unique scenarios and use-cases let us build commercial autonomous vehicles at a reasonable cost. Our vehicles are sold at a price of $70,000, over 10 × lower than what is commonly believed to be possible for commercial autonomous vehicles [17]. Our vehicles currently operate in the city of Fishers, Indiana (US), tourist sites at Nara and Fukuoka (Japan), an industrial park at Shenzhen (China), and a university campus at Fribourg (Switzerland) for commercial uses. Our vehicles have accumulated over 200,000 miles since launched in 2018.

II-B. Autonomous Driving Infrastructure

Our autonomous driving technologies combine both online tasks processed in real-time on the vehicle and offline tasks hosted as cloud services. Fig. 1 illustrates our end-to-end infrastructure. While the rest of the paper focuses on the on-vehicle processing system, this section briefly introduces our end-to-end system to provide a comprehensive context.

On-Vehicle Processing On-vehicle processing tasks generally fall under three main categories: sensing, perception, and planning. Our vehicle relies on a wide variety of sensors, including (stereo) cameras, GPS, IMU, Radars, and Sonars. The sensing data feeds into the perception module, which localizes the vehicle itself in the global environment and understands the surroundings such as object positions. The perceptual understandings are used by the planning module, whose goal is to generate a safe and efficient action plan for the vehicle in real-time. Sec. V through Sec. VI describe on-vehicle processing in detail.

Cloud Services While executed offline, cloud services are essential to supporting an autonomous vehicle [18]. Our cloud workloads include map generation, simulation, and machine learning (ML) model training. Over time, the new ML models, algorithms, and maps are updated to the vehicles, which in turn continuously provide real-world

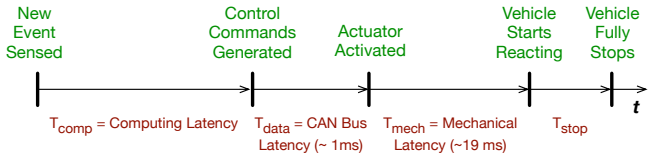


Fig. 2: The end-to-end latency model. The computing latency, T_{comp}, is the main optimization target.

observations and statistics to the cloud tasks. Similar to Tesla, we also use a pre-constructed map that marks lanes. In particular, we use OpenStreetMap (OSM) [19], and Tesla uses Google Maps [20]. We frequently annotate OSM with semantic information of the environment.

Due to the limitation of communication bandwidth, the only data we upload to the cloud in real-time is the condensed operational log (once an hour), which is very small in size (a few KB). The raw training data (e.g., images) is enormous even after compression (as high as 1 TB per day) and, thus, the raw data is stored in the on-vehicle SSD and manually uploaded to the cloud at the end of each operational day, a practice widely adopted in the industry.

III. Design Constraints

We introduce general models that let us reason about the constraints and requirements in any autonomous vehicle. These models help architects evaluate their design decisions for the end-to-end system rather than for isolated components. We then use concrete data from our vehicles to highlight the constraints and requirements we face.

III-A. Performance Requirements

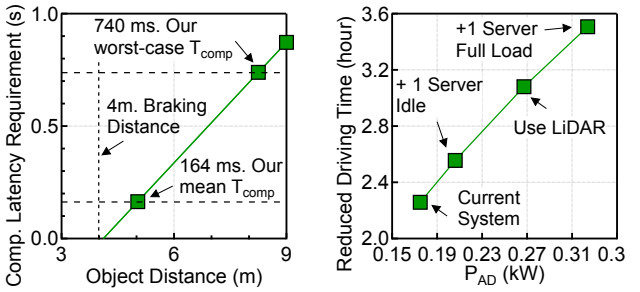
Latency Requirement The end-to-end latency, i.e., the time between when a new event is sensed in the environment (e.g., change of distance, new object) and when the vehicle fully stops, must be short enough to avoid hitting objects.

The latency requirement could be derived using a simple analytical model as shown in Fig. 2. The latency consists of four major components: the time for the computing system to generate control commands from the sensor inputs (T_{comp}), the time to transmit the control commands to the vehicle's actuator through the Controller Area Network (CAN) bus (T_{data}), the time it takes for the mechanical components of the vehicle to start reacting (T_{mech}), and the time for the vehicle to fully stop (T_{stop}). Assuming the brake generates a deceleration of a, the vehicle's speed is v, and an object is of distance D to the vehicle when it is sensed, the following, which is generic to any autonomous vehicle, must hold [21]:

$$(T_{comp} + T_{data} + T_{mech}) \times v + \frac{1}{2} \times a \times T_{stop}^2 \leq D \quad (1a)$$

$$T_{stop} = v/a \quad (1b)$$

We use this model to derive the latency requirement of the computing system T_{comp}, which is the primary target



(a) The computing latency requirement becomes tighter when the object to be avoided is closer.

(b) Driving time reduces as the power of the autonomous driving system increases (P_{AD}).

Fig. 3: Impacts of latency and energy.

of our optimizations. In our vehicles at a typical speed v of 5.6 m/s, the brake generates a deceleration a of about 4 m/s^2. Our measurements show that T_{data} is about 1 ms and T_{mech} is about 19 ms. Given these, Fig. 3a shows that the computing latency requirement T_{comp} (y-axis) tightens as the object distance D (x-axis) becomes closer.

As the computing latency becomes lower, the vehicle could avoid objects that are farther away. Our vehicle has an average computing latency of 164 ms (Sec. V-C), which means the vehicle could avoid any objects that are 5 m away or farther when they are detected. Our worst-case computing latency is 740 ms, under which the vehicle could avoid objects that are detected at least 8.3 m away.

This latency model allows us to understand how much computing latency matters in the end-to-end system, which provides targets/guidances for hardware acceleration. For instance, as shown in Fig. 3a, if the vehicle is to proactively plan the route to avoid an obstacle at 5 m away, the computing latency must be lower than 164 ms.

Note that with an a of 4 m/s^2 and v of 5.6 m/s, the vehicle's braking distance is 4 m, which is the theoretical lower-bound of obstacle avoidance. To improve safety, we deploy a last line of defense, which uses Radar and Sonar to bypass the processing system and directly controls the actuator. This mechanism could effectively avoid objects that are 4.1 m away, approaching the theoretical limit (Sec. V).

Throughput Requirement The throughput requirement, quantified in the number of control commands sent to the actuator per second, dictates how often we can control the vehicle. A higher throughput allows more smooth control without abrupt turns and brakes. We set a 10 Hz throughput requirement for our vehicles, which is much higher than how often a human driver manipulates vehicles. It is worth noting that the throughput requirement is relatively easier to meet than latency due to techniques such as pipelining (Sec. IV).

III-B. Energy and Thermal Constraints

Energy Constraint and Impacts on Hardware The computing systems and sensors that enable autonomous driving consume extra energy, which reduces the driving

Table I: Power breakdown of our vehicles. As a comparison, we also show the power consumptions of typical LiDARs, which are not used by us.

Component(s)		Power (W)	Quantity
Main Computing (CPU+GPU) Server	Dynamic	118	1
	Idle	31	1
Embedded Vision Module (FPGA + Cameras/IMU/GPS)		11	1
Radars		13	6
Sonars		2	8
Total for AD (P_{AD})		175	-
Vehicle without AD (P_V)		600	-
LiDAR (Not used by us)	Long-range [22]	60	1
	Short-range [22]	8	1

range and translates to revenue loss for commercial vehicles. These considerations in turn impact the hardware design.

The impact of energy consumption on driving time reduction ($T_{reduced}$) could be derived using a simple model:

$$T_{reduced} = \frac{E}{P_V} - \frac{E}{P_V + P_{AD}} \quad (2)$$

where P_V denotes the power consumption of the vehicle itself (without the autonomous driving capabilities)[1], P_{AD} denotes the additional power consumed to enable autonomous driving, and E denotes the vehicle's total energy capacity.

Our vehicles are electric cars that are powered by batteries, which have a total energy capacity of 6 kilowatt hour (kW·h). The vehicle itself consumes 0.6 kW on average (P_V)[2], and enabling autonomous driving consumes an additional 0.175 kW (P_{AD}). Effectively, supporting autonomous driving reduces the driving time on a single charge from 10 hours to 7.7 hours.

Table I further breaks down the additional power consumption into four components (see Sec. V-B for a detailed hardware architecture): the main computing server (CPU + GPU), the embedded vision module (an FPGA board with the cameras, Inertial Measurement Unit (IMU), and GPS), the six Radar units, and the eight Sonar units. The computing server contributes the most to the additional power.

Energy consumption significantly impacts hardware system design. Using one of our deployments in a tourist site in Japan, let us consider the implication of adding one additional computing server — presumably to support advanced algorithms or to reduce computing latency. Since computing systems in autonomous vehicles are always-on when the vehicle is active, the idle power of the additional server alone would increase the total power consumption by 31 W, which reduces the driving time by 0.3 hours. Each vehicle in that tourist site operates about 10 hours a day; thus, the reduced operation time would translate to 3% revenue lost per day. Fig. 3b shows how the driving time reduces as P_{AD} increases. If the additional server is operating at full load, the driving time is reduced by about 3.5 hours.

[1]Note that P_V is affected by the weight, which in turn consists of the weight of the vehicle itself and the weight of the passengers. The passenger weight in our 2-seater car is about one-fifth of the vehicle itself. A detailed analytical model of P_V could be found at Kim et. al. [23].

[2]The peak power could be as high as 2 kW.

Table II: Cost breakdown of our vehicle and cost comparison with LiDAR-based vehicles.

Vehicles	Components	Price
Our vehicle (camera-based)	Cameras × 4 + IMU	$1,000
	Radar × 6	$3,000
	Sonar × 8	$1,600
	GPS	$1,000
	Retail Price	$70,000
LiDAR-based vehicle (e.g., Waymo)	Long-range LiDAR	$80,000
	Short-range LiDAR × 4	$16,000
	Estimated Retail Price	>$300,000

Thermal Constraint Since we have managed to optimize the total computing power consumption well under 200 W, thermal constraints do not appear to be a problem in various commercial deployment environments, where temperatures range from -20 °C to + 40 °C. Conventional cooling techniques (e.g., fans) for server systems are used.

III-C. Cost and Safety Considerations

Cost In the long-term, popularizing autonomous vehicles is possible only if they significantly reduce the cost of transportation services, which in turn imposes tight cost constraints on building and supporting the vehicle.

Similar to the concept of total cost of ownership (TCO) in data centers [24], the cost of an autonomous vehicle is a complex function influenced by not only the cost of the vehicle itself, but also indirect costs such as servicing the vehicle and maintaining the back-end cloud services.

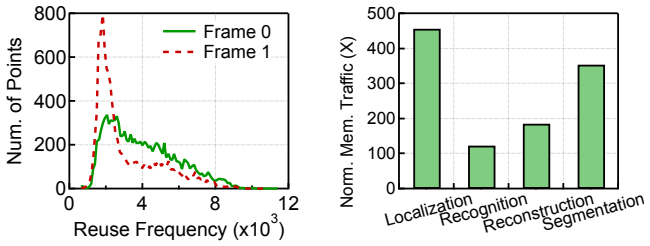
As a reference point, Table II provides a cost breakdown of our vehicles that operate in a tourist site in Japan. Considering all the factors influencing the vehicle's cost, each vehicle is sold at $70,000, which allows the tourist site to charge each passenger only $1 per trip. Any increase of the vehicle cost would directly increase the customer cost.

Safety Safety is undoubtedly the single most important concern of an autonomous vehicle [25], [26]. In our experience, safety issues arise in two scenarios: 1) the computing latency is too long to avoid an object appearing close enough to the vehicle (as modeled in Equ. 1 and Fig. 2); 2) vision algorithms produce wrong results, e.g., missing an object.

Conventional modular redundancy handles neither scenario above. To ensure safety, our system employs a reactive path (Sec. IV), which bypasses the computing system and directly controls/throttles the vehicle. Safety is also a key reason we choose embedded FPGA platforms, which meet the automotive-grade requirements [27], instead of off-the-shelf mobile SoCs to interface with the sensors (Sec. V-A).

III-D. A Case-Study: LiDAR vs. Camera

The choice of sensors dictates the algorithm and hardware design. Almost all the autonomous vehicle companies use LiDAR at the core of their technologies. Examples include



(a) Irregular data reuse. (b) High memory traffic.

Fig. 4: Inefficiencies of LiDAR Processing.

Uber, Waymo, and Baidu. We and Tesla are among the very few that do not use LiDAR and, instead, rely on cameras and vision-based systems. Abandoning LiDAR is a key design decision driven by the various requirements and constraints described above. We elaborate the decision here as a case-study and use-case of the constraint-driven system design.

We do *not* intend to make a general argument about LiDAR vs. cameras. Instead, we show that *for our use-cases* LiDARs processing is slower than camera-based vision processing, but increases the power consumption and cost.

Latency Fundamentally, camera and LiDAR are perception sensors that aid perception tasks such as depth estimation, object detection/tracking, and localization (Sec. IV).

In our measurements, LiDAR processing is general slower than camera-based vision processing. For instance, a LiDAR-based localization algorithm takes 100 *ms* to 1 *s* on a high-end CPU+GPU machine while our vision-based localization algorithm finishes in about 25 *ms* on an embedded FPGA (Sec. V-C). As Fig. 3a shows, even an 100 *ms* latency difference could translate to about 1 *m* in object avoidance range.

The reason that LiDAR processing is inefficient has to do with its irregular algorithms. Vision algorithms consist of regular stencil operations (e.g., convolution), which are well-suited for GPUs, DSPs, and DNN accelerators. In contrast, LiDAR generates irregular point clouds [28], which consist of sparse points arbitrarily spread across the 3D space. Thus, LiDAR processing relies on irregular kernels (e.g., neighbor search) [29], which lead to inefficient memory behaviors, and does not have mature acceleration solutions.

To quantify the irregularity, Fig. 4a shows the histogram of data (point) reuse frequency when running a LiDAR localization algorithm [29], [14], [30]. Each $< x, y >$ point denotes the number of points (y) that is reused certain times (x). We compare the reuse statistics between two different point clouds captured at two different scenes by the same LiDAR. While the data reuse opportunity is abundant, the number of reuses varies significantly both across points within a point cloud and across two points clouds. Therefore, conventional memory optimizations are likely ineffective.

Irregularity leads to inefficiency. We measure four common point cloud algorithms implemented in the well-tuned Point Cloud Library [31] on an Intel Coffee Lake CPU with a 9 MB LLC. Fig. 4b shows the off-chip memory traffic normalized to the optimal communication case, where all the data are reused on-chip. Existing systems require several orders of more off-chip memory accesses than the optimal case. Others have observed a similar trend. For instance, up to 60% of the GPU execution time of point cloud CNNs is spent on irregular memory accesses [32].

Power LiDAR is one order of magnitude more power-hungry than cameras, and thus would leave much less power headroom for the computing system and significantly reduce the driving range. For instance, Waymo's vehicles [33] carry 1 long-range LiDAR (e.g., Velodyne HDL-64E [34]) and 4 short-range LiDARs (e.g., Velodyne Puck [35]), consuming about 92 W of power in total [22]. In contrast, the power of the 4 cameras in our vehicle is under 1 W. Fig. 3b shows that applying Waymo's LiDAR configuration to our vehicle would further reduce the driving time by 0.8 hours on a single charge (8% per day) compared to our current system.

Cost The total cost of operating a vehicle that uses LiDAR is much higher. As shown in Table II, a long-range LiDAR could cost about $80,000 [17], whereas our cameras + IMU setup costs about $1,000. The total cost of a LiDAR-equipped car could cost between $300,000 [36] to $800,000 [17]. In addition, creating and maintaining the High-Definition (HD) maps required for LiDARs can cost millions of dollars per year for a mid-size city [37] due to the need for high resolution.

Depth Quality It is worth noting that LiDARs do outperform cameras in providing more accurate depth information. Using the time-of-flight principle [38], LiDARs directly provide depth information at the precision of 2 *cm* [39]. However, our vehicles require only coarse-grained depth information and could tolerate depth errors at about 0.2 *m*. This is because our vehicles maneuver at a lane-granularity (1 *m* – 3 *m* wide), such as staying in a lane or switching lanes, without maneuvering within a lane.

IV. Software Pipeline

Fig. 5 shows the block diagram of our on-vehicle processing system, which consists of three components: sensing, perception, and planning. Sensor samples are synchronized and processed before being used by the perception module, which performs two fundamental tasks: (1) understanding the vehicle itself by localizing the vehicle in the global map (i.e., ego-motion estimation) and (2) understanding the surroundings through depth estimation and object detection/tracking. The perception results are used by the planning module to plan the path and to generate the control commands. The control commands are transmitted through the CAN bus to the vehicle's Engine Control Unit (ECU), which then activates the vehicle's actuator, from which point on the vehicle's mechanical components take over.

Algorithms We summarize the main algorithms explored in our computing system in Table III. Our localization module is based on the classic Visual Inertial Odometry algorithm [40], [41], which uses camera-captured images and IMU samples to estimate the vehicle's position in the

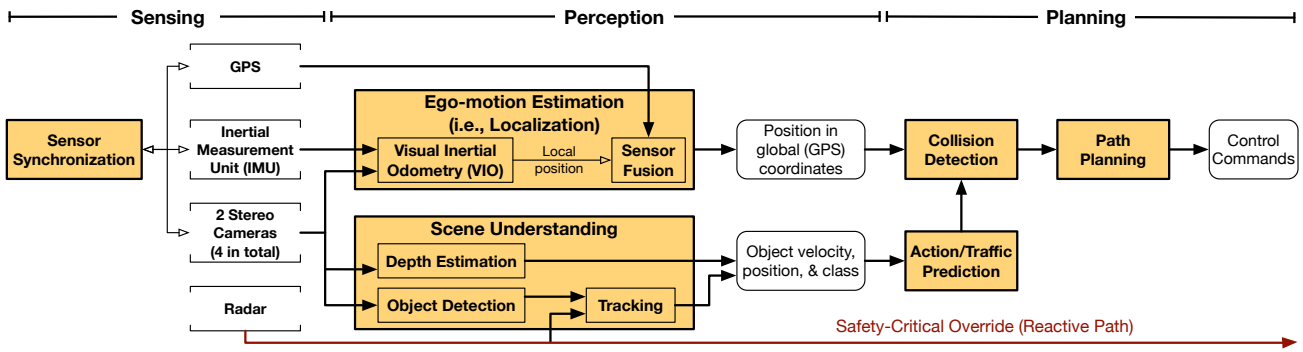


Fig. 5: Block diagram of our on-vehicle processing software system. The normal processing pipeline, a.k.a., the proactive path, processes (continuous) sensor samples to generate commands that control the vehicle (steer, brake, accelerate). Independent of the proactive path, a reactive path acts as the last line of defense to ensure safety.

Table III: Algorithms explored in perception and planning.

Task	Algorithm(s)	Sensors(s)
Depth Estimation	ELAS [44]	Cameras
Object Detection	YOLO [48]/ Mask R-CNN [49]	Camera
Object Tracking	KCF [46]	Camera, Radar
Localization	VIO [41]	Cameras, IMU, GPS
Planning	MPC [47]	–

global map. Depth estimation uses stereo vision algorithms, which calculate object depth by processing the two slightly different images captured from a stereo camera pair [42], [43]. In particular, we use the classic ELAS algorithm, which uses hand-crafted features [44]. While DNN models for depth estimation exist, they are orders of magnitude more compute-intensive than non-DNN algorithms [45] while providing marginal accuracy improvements to our use-cases.

We detect objects using DNN models. Object detection is the only task in our current pipeline where the accuracy provided by deep learning justifies the overhead. An object, once detected, is tracked across time until the next set of detected objects are available. The DNN models are trained regularly using our field data. As the deployment environment can vary significantly, different models are specialized/trained using the deployment environment-specific training data. Tracking is mostly done by a Radar as we will discuss later, but we use the Kernelized Correlation Filter (KCF) [46] as the baseline tracking algorithm when Radar signals are unstable. The planning algorithm is formulated as Model Predictive Control (MPC) [47].

Safety The processing pipeline described so far is what we call the *proactive* path—because the vehicle could carefully plan the most efficient path to the destination while avoiding obstacles. Relying completely on the proactive strategy, however, could be unsafe in two occasions: 1) the computing latency of the proactive path is too long; 2) vision algorithms produce wrong results (Sec. III-C).

To improve safety, we deploy a last line of defense using Radar (and Sonar when available) signals, which show the distance from the nearest object in front of the vehicle's path. These signals directly enter the vehicle's ECU and overrides the current control commands from the proactive path. We call this sub-system the *reactive* path, as its sole goal is to stop the vehicle in reaction to close-by obstacles that the normal processing pipeline could not avoid.

Our reactive path's latency is as low as 30 *ms*, whereas the proactive path's best-case latency is 149 *ms* (Sec. V-C). The reactive path let the vehicle react to objects 4.1 *m* away, reduced from the 5 *m* distance required by the proactive path and approaching the 4 *m* braking distance limit (Fig. 3a).

Task-Level Parallelism Sensing, perception, and planning are serialized; they are all on the critical path of the end-to-end latency. We pipeline the three modules to improve the throughput, which is dictated by the slowest stage.

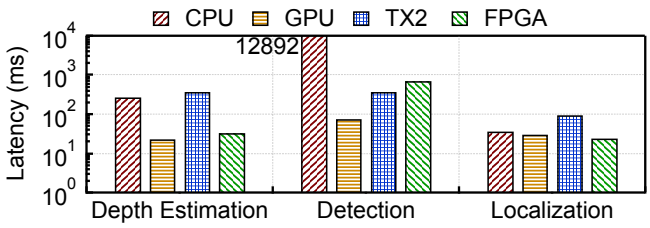
Different sensor processings (e.g., IMU vs. camera) are independent. Within perception, localization and scene understanding are independent and could execute in parallel. While there are multiple tasks within scene understanding, they are mostly independent with the only exception that object tracking must be serialized with object detection. The task-level parallelisms influence how the tasks are mapped to the hardware platform as we will discuss later.

V. Systems-on-a-Vehicle (SoV)

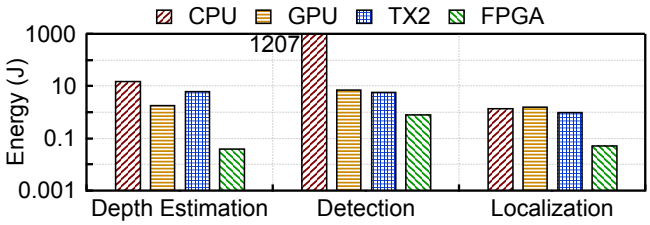
We provide a detailed description of the on-vehicle processing system, which we call "Systems-on-a-Vehicle" (SoV). The SoV is central to an autonomous vehicle. It comprises processes interprets sensory information to generate control commands, which are sent to the actuator to maneuver the vehicle. We start by discussing solutions that we explored but decided not to adopt (Sec. V-A), followed by our current SoV (Sec. V-B). In the end, we provide a detailed characterization of our current SoV (Sec. V-C).

V-A. Hardware Design Space Exploration

Before introducing our hardware architecture for on-vehicle processing, this section provides a retrospective discussion of the two hardware solutions we explored in



(a) Latency comparison.



(b) Energy consumption comparison.

Fig. 6: Performance and energy comparison of four different platforms running three perception tasks. On TX2, we use the Pascal GPU for depth estimation and object detection and use the ARM Cortex-A57 CPU (with SIMD capabilities) for localization, which is ill-suited for GPU due to the lack of massive parallelisms.

the past but decided not to adopt: off-the-shelf mobile SoCs and off-the-shelf (ASIC-based) automotive chips. They are at the two extreme ends of the hardware spectrum; our current design combines the best of both worlds.

Mobile SoCs We initially explored the possibility of utilizing high-end mobile SoCs, such as Qualcomm Snapdragon [50] and Nvidia Tegra [51], to support our autonomous driving workloads. The incentive is two-fold. First, since mobile SoCs have reached economies of scale, it would have been most beneficial for us to build our technology stack on affordable, backward-compatible computing systems. Second, our vehicles target micromobility with limited speed, similar to mobile robots, for which mobile SoCs have been demonstrated before [52], [53], [54], [55].

However, we found that mobile SoCs are ill-suited for autonomous driving for three reasons. First, the compute capability of mobile SoCs is too low for realistic end-to-end autonomous driving workloads. Fig. 6 shows the latencies and energy consumptions of three perception tasks—depth estimation, object detection, and localization—on an Intel Coffee Lake CPU (3.0 GHz, 9 MB LLC), Nvidia GTX 1060 GPU, and Nvidia TX2 [56], which represents today's high-end mobile SoCs. Fig. 6a shows that TX2 is much slower than the GPU, leading to a cumulative latency of 844.2 *ms* for perception alone. Fig. 6b shows that TX2 has only marginal, sometimes even worse, energy reduction compared to the GPU due to the long latency.

Second, mobile SoCs do not optimize data communication between different computing units, but require redundant data copying coordinated by the power-hungry CPU. For

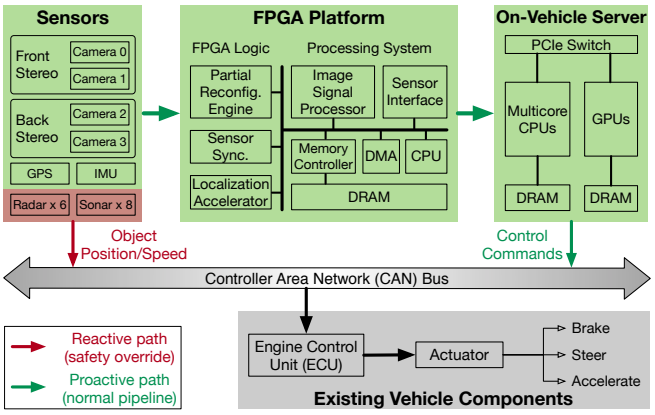


Fig. 7: Overview of our SoV hardware architecture.

instance, when using DSP to accelerate image processing, the CPU has to explicitly copy images from sensor interface to DSP through the entire memory hierarchy [57], [58], [59]. Our measurement shows that this leads to an extra 1 W power overhead and up to 3 *ms* performance overhead.

Finally, traditional mobile SoCs design emphasizes compute optimizations, while we find for autonomous vehicle workloads, sensor processing support in hardware is equally important. For instance, autonomous vehicles require very precise and clean sensor synchronization, which mobile SoCs do not provide. We will discuss our software-hardware collaborative approach to sensor synchronization in Sec. VI.

Automotive ASICs At the other extreme are computing platforms specialized for autonomous driving, such as those from NXP [60], MobileEye [61], and Nvidia [62]. They are mostly ASIC-based chips that provide high performance at a much higher cost. For instance, the first-generation of Nvidia PX2 system costs over $10,000 while a Nvidia TX2 SoC costs only $600.

Besides the cost, we do not use existing automotive platforms for two technological reasons. First, they focus on accelerating a subset of the tasks in *fully autonomous driving*. For instance, Mobileye chips provide *assistance* to drivers through features such as forward collision warning. Thus, they accelerate key perception tasks (e.g., lane/object detection). Our goal is an end-to-end computing pipeline for fully autonomous driving, which must optimize sensing, perception, and planning as a whole. Second, similar to mobile SoCs these automotive solutions do not provide good sensor synchronization support.

V-B. Hardware Architecture

The results from our hardware exploration motivate the current SoV design. Our design 1) uses an on-vehicle server machine to provide sufficient compute capabilities, 2) uses an FPGA platform to directly interface with sensors to reduce redundant data movement while providing hardware acceleration for key tasks, and 3) uses software-hardware collaborative techniques for efficient sensor synchronization.

V-B.1 Overview

Fig. 7 shows an overview of the SoV hardware system. It consists of sensors, a server+FPGA heterogeneous computing platforms, the ECU[3], and the CAN bus, which connects all the components. The ECU and CAN bus are standard components in non-autonomous vehicles, and have open programming interfaces [63], [64], which we directly leverage. We focus on sensors and the computing platforms.

The sensing hardware consists of (stereo) cameras, IMU, GPS, Radars, and Sonars. In particular, our vehicle is equipped with two sets of stereo cameras, one forward facing and the other backward facing, for depth estimation. One of the cameras in each stereo pair is also used to drive monocular vision tasks such as object detection. The cameras along with the IMU drive the VIO-based localization algorithm.

Considering the cost, compute requirements, and power budget, our current computing platform consists of a Xilinx Zynq UltraScale+ FPGA board and an on-vehicle PC machine equipped with an Intel Coffee Lake CPU and an Nvidia GTX 1060 GPU. The PC is the main computing platform, but the FPGA plays a critical role, which bridges sensors and the server and provides an acceleration platform.

V-B.2 Algorithm-Hardware Mapping

There is a large design space in mapping the sensing, perception, and planning tasks to the FPGA + server (CPU + GPU) platform. The optimal mapping depends on a range of constraints and objectives: 1) the performance requirements of different tasks, 2) the task-level parallelism across different tasks, 3) the energy/cost constraints of our vehicle, and 4) the ease of practical development and deployment.

Sensing We map sensing to the Zynq FPGA platform, which essentially acts a sensor hub. It processes sensor data and transfers sensor data to the PC for subsequent processing. The Zynq FPGA hosts an ARM-based SoC and runs a full-fledged Linux OS, on which we develop sensor processing and data transfer pipelines.

The reason sensing is mapped to the FPGA is threefold. First, embedded FPGA platforms today are built with rich/mature sensor interfaces (e.g., standard MIPI Camera Serial Interface [65]) and sensor processing hardware (e.g., ISP [66]) that server machines and high-end FPGAs lack.

Second, by having the FPGA directly process the sensor data in situ, we allow accelerators on the FPGA to directly manipulate sensor data without involving the power-hungry CPU for data movement and task coordination.

Finally, processing sensor data on the FPGA naturally let us design a hardware-assisted sensor synchronization mechanism, which is critical to perception (Sec. VI-A).

Planning We assign the planning tasks to the CPU of the on-vehicle server, for two reasons. First, the planning commands are sent to the vehicle through the CAN bus; the CAN bus interface is simply more mature on high-end servers than embedded FPGAs. Executing the planning module on the server greatly eases deployment. Second, as we will show later (Sec. V-C) planning is very lightweight, contributing to about 1% of the end-to-end latency. Thus, accelerating planning on FPGA/GPU has marginal improvement.

[3]The ECU and actuator are tightly integrated with a *ns*-level delay.

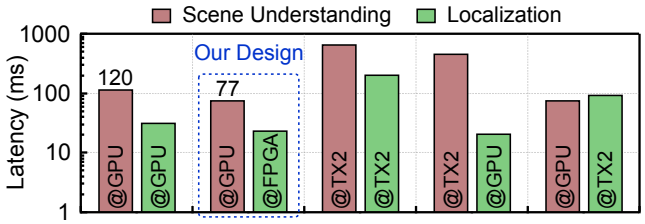


Fig. 8: Latency comparison of different mapping strategies of the perception module. The perception latency is dictated by the slower task between scene understanding (depth estimation and object detection/tracking) and localization.

Perception Perception tasks include scene understanding (depth estimation and object detection/tracking) and localization, which are independent and, therefore, the slower one dictates the overall perception latency.

Fig. 6a compares the latency of the perception tasks on the embedded FPGA with the GPU. Due to the available resources, the embedded FPGA is faster than the GPU only for localization, which is more lightweight than other tasks. Since latency dictates user-experience and safety, our design offloads localization to the FPGA while leaving other perception tasks on the GPU. Overall, the localization accelerator on FPGA consumes about 200K LUTs, 120K registers, 600 BRAMs, 800 DSPs, with less than 6 W power.

This partitioning also frees more GPU resources for depth estimation and object detection, further reducing latency. Fig. 8 compares different mapping strategies. When both scene understanding and localization execute on the GPU, they compete for resources and slow down each other. Scene understanding takes 120 *ms* and dictates the perception latency. When localization is offloaded to the FPGA, the localization latency is reduced from 31 *ms* to 24 *ms*, and scene understanding's latency reduces to 77 *ms*. Overall, the perception latency improves by 1.6 ×, translating to about 23% end-to-end latency reduction on average.

Fig. 8 also shows the performance when using an off-the-shelf Nvidia TX2 module instead of an FPGA. While significantly easier to deploy algorithms, TX2 is always a latency bottleneck, limiting the overall computing latency.

V-B.3 Partial Reconfiguration

We use FPGA not only as a "convenient" ASIC, but also exploit FPGA's reconfigurability to provide a cost-effective *programmable* acceleration platform via Runtime partial reconfiguration (RPR), which time-shares part of the FPGA resources across algorithms *at runtime*.

RPR is useful for on-vehicle processing, because many on-vehicle tasks usually have multiple versions, each used in a particular scenario. For instance, our localization algorithm relies on salient features; features in key frames are *extracted* by a feature extraction algorithm [67], whereas features in non-key frames are *tracked* from previous frames [68];

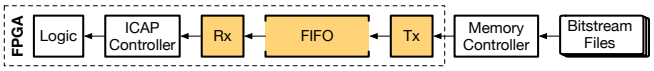


Fig. 9: Runtime partial reconfiguration engine. Augmentations to existing hardware on the Zynq board are shaded.

the latter executes in 10 *ms*, 50% faster than the former. Spatially sharing the FPGA is not only area-inefficient, but also power-inefficient as the unused portion of the FPGA consumes non-trivial static power.

Zynq platforms provide capabilities to partially reconfigure FPGA resources by transferring partial bitstream files [69], [70]. However, this involves the CPU and thus has low reconfiguration speed (300 KB/s) and high power overhead. Instead, we provide hardware support on the FPGA to drastically improve the reconfiguration speed.

Our idea is to remove the CPU's involvement in partial reconfiguration. While a DMA engine is designed for this purpose, it would be inefficient since the Internal Configuration Access Port (ICAP), which accepts the bitstreams to configure the FPGA, is not designed to accept streaming data. Using a conventional DMA introduces new overheads due to frequent interactions with the memory controller. Instead, we use a design that decouples receiving the bitstream data from transmitting the data to the ICAP [71], [72]. Fig. 9 shows an overview of the system. The transmitter (Tx) acts as a lightweight DMA engine, which transfers a bitstream file to a FIFO from the memory, which, critically, is done through one handshake; the receiver (Rx) then transfers data from the FIFO to the ICAP following ICAP's protocol.

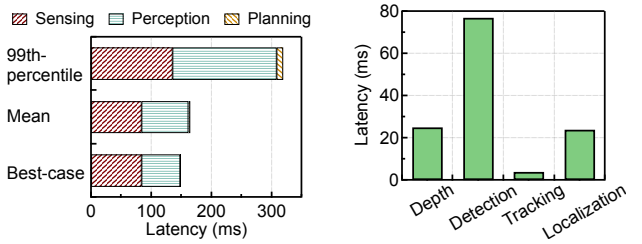
Both the Rx and Tx logic and the FIFO are implemented within the FPGA. The bitstream files for different algorithms are stored in the DRAM on the Zynq board. We empirically fine that an 128-byte FIFO is sufficient. Our RPR engine achieves over 350 MB/s reconfiguration throughput while using only about 400 FFs and 400 LUTs. Since both the feature extraction and tracking bitstreams are less than 10 MB, the reconfiguration delay is less than 3 *ms* and consumes only 2.1 *mJ* of energy each time.

V-C. Performance Characterizations

The significance of a low computing latency is that the vehicle could stay in proactive planning more often and trigger reactive throttling less often, which directly translates to better passenger experience. This section characterizes the end-to-end computing latency on our hardware platform.

Average Latency and Vairation Fig. 10a breaks down the computing latency into sensing, perception, and planning. We show the best- and average-case latency as well as the 99th percentile. The mean latency (164 *ms*) is close to the best-case latency (149 *ms*), but a long tail exists.

To showcase the variation, the median latency of localization is 25 *ms* and the standard deviation is 14 *ms*. The variation is mostly caused by the varying scene complexity. In dynamic scenes (large change in consecutive frames), new features can be extracted in every frame, which slows down the localization algorithm.



(a) Computing latency distribution of on-vehicle processing.

(b) Average-case latencies of different perception tasks.

Fig. 10: Latency characterizations.

Data collected in the field shows that, with this latency profile, our deployed vehicles stay in the proactive paths for over 90% of the time, which is sufficient to our use-cases.

Latency Distribution Unlike the conventional wisdom, sensing contributes significantly to the latency. The sensing latency is bottlenecked by the camera sensing pipeline, which in-turn is dominated by the image processing stack on the FPGA's embedded SoC (e.g., ISP and the kernel/driver), suggesting a critical bottleneck for improvement.

Apart from sensing, perception is the biggest latency contributor. Object detection (DNN) dominates the perception latency. Fig. 10b further shows the latencies of different perception tasks in the average case. Recall that all perception tasks are independent except that object detection and tracking are serialized. Therefore, the cumulative latency of detection and tracking dictates the perception latency.

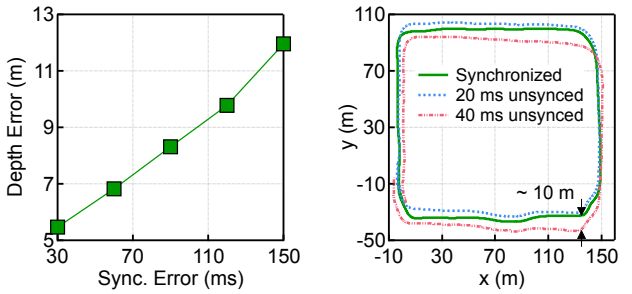
To accelerate object detection, one option we explored is to offload object detection DNN to a high-end FPGA. Apart from the higher cost and power consumption, a key reason we do not pursue that is programmability, as we constantly change the DNN models. While there has been great progress in building DNN accelerators, programming these accelerators is still a challenge. This is a similar observation made to inference in mobile devices at Facebook [73].

Planning is relatively insignificant, contributing to only 3 *ms* in the average case. This is because our vehicles require only coarser-grained motion planning at a *lane granularity* as discussed in Sec. III-D. Fine-grained motion planning and control at centimeter granularities [47], usually found in mobile robots [10], unmanned aerial vehicles [9], and many LiDAR-based vehicles, are much more compute-intensive. One such example is the Baidu Apollo EM Motion Planner [74], whose motion plan is generated through a combination of Quadratic Programming (QP) and Dynamic Programming (DP). On our platform, the EM planner takes 100 *ms*, 33 $\times$ more expensive than our planner.

The sensing, perception, and planning modules operate at 10 Hz – 30 Hz, meeting the throughput requirement.

VI. Sensing-Computing Co-Design

While lots of architectural research lately have focused on perception and planning in robotics and autonomous vehicles [9], [10], [11], [13], [75], [76], sensing has largely



(a) Depth estimation error increases as stereo cameras become more out-of-sync. Even if the two cameras are off by only 30 *ms*, the depth estimation error could be over 5 *m*.

(b) Localized trajectory comparison between using synchronized and unsynchronized (camera and IMU) sensor data. The localization error could be as high as 10 *m*.

Fig. 11: Importance of sensor synchronization.

been overlooked. This section discusses two examples where sensing plays a key role in autonomous vehicles. Critically in each case, we explore opportunities to co-design/co-optimize sensing with computing to improve system efficiency.

VI-A. Sensor Synchronization

Sensor synchronization is crucial to perception algorithms, which fuse multiple sensors. For instance, vehicle localization uses Visual-Inertial Odometry, which uses both camera and IMU sensors. Similarly, object depth estimation uses stereo vision algorithms, which require the two cameras in a stereo system to be precisely synchronized. Overall, the four camera sensors (i.e., two stereo cameras) and the IMU must all be synchronized with each other.

Almost all the academic work assumes that sensors are already synchronized; widely-adopted benchmarks and datasets such as KITTI [77] manually synchronize sensors so that researchers could focus on algorithmic developments. Sensor synchronization in real-world, however, is challenging. No mature solutions exist in the public domain.

VI-A.1 The Problem

An ideal sensor synchronization ensures that two sensor samples that have the same timestamp capture the same event. This in turn translates to two requirements. First, all sensors are *triggered simultaneously*. Second, each sensor sample is associated with *a precise timestamp*.

Out-of-sync sensor data is detrimental to perception. Fig. 11a shows how unsynchronized stereo cameras affects depth estimation in our perception pipeline. As the temporal offset between the two cameras increases (further to the left on *x*-axis), the depth estimation error increases (*y*-axis). Even if the two cameras are off-sync by only 30 *ms*, the depth estimation error could be as much as 5 *m*. Fig. 11b shows how unsynchronized camera and IMU would effect the localization accuracy. When the IMU and camera are off by 40 *ms*, the localization error could be as much as 10 *m*.

Most autonomous vehicles today synchronize sensors at the application layer. Taking our localization pipeline as an example, Fig. 12a shows the idea of software-only synchronization. The application running on the CPU, in this case the localization algorithm, continuously polls the sensors and generates a timestamp for each sensor sample. Sensor samples that have the same timestamp are then treated as capturing the same event by the localization algorithm.

This approach, however, is inaccurate for two reasons. First, the camera and the IMUs sensors are triggered individually, because each sensor operates under their own timer, which might not be synchronized with each other. Therefore, it might be theoretically impossible for different sensors to capture the same event simuntaneously.

More importantly, even if the sensors are triggered simultaneously, there is a long processing pipeline before the sensor data reaches the application. The processing pipeline introduces *variable* latency that is non-deterministic. While known constant latency could be compensated in software, variable latency is hard to capture and compensate.

Fig. 12b illustrates the effect of variable-latency sensor pipeline. Compared with the exact camera sensor triggering time, the moment that a frame reaches the application running on the CPU is delayed by sensor exposure, data transmission to the SoC, as well as the processing at the sensor interface, ISP, and the DRAM. While the sensor exposure time and the transmission delay are fixed, other processing components introduce variable latency. For instance, in our experiments we find that the ISP processing latency may vary by about 10 *ms*. As we move up the software stack on CPU, the temporal variation could be as much as 100 *ms*.

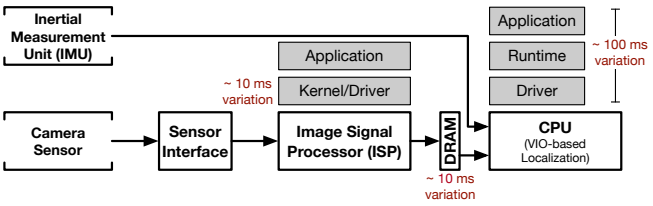
The same issue exists in the IMU processing pipeline too, where the data transmission delay is relatively constant but the CPU code introduces variable latency. As a result, when `C0` reaches the application layer, its timestamp would be the same as the timestamp of `M7`. Thus, the application would regard C0 and M7 as capturing the same event, whereas in fact `C0` and `M0` capture the same event.

VI-A.2 Software-Hardware Collaborative Solution

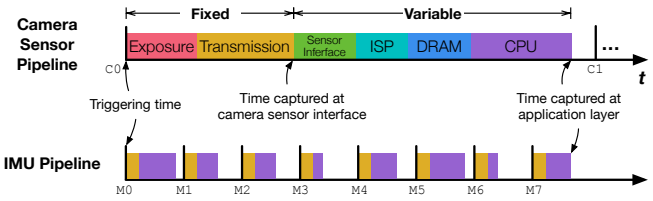
We propose a software-hardware collaborative sensor synchronization design to address the issues in software-only solutions. Fig. 12c shows the high-level diagram of our system, which is based on two principles: (1) trigger sensors simultaneously using a single common timing source, and (2) obtain each sensor sample's timestamp close to the sensor so that the timestamp accurately captures the sensor triggering time while avoiding the variable sensor processing delay; this is what we call "near-sensor" synchronization.

Guided by these two principles, we introduce a hardware synchronizer, which triggers the camera sensors and the IMU using a common timer initialized by the satellite atomic time provided by the GPS device. The camera operates at 30 FPS while the IMU operates at 240 FPS. Thus, the camera triggering signal is downsampled 8 times from the IMU triggering signal, which also guarantees that each camera sample is always associated with an IMU sample.

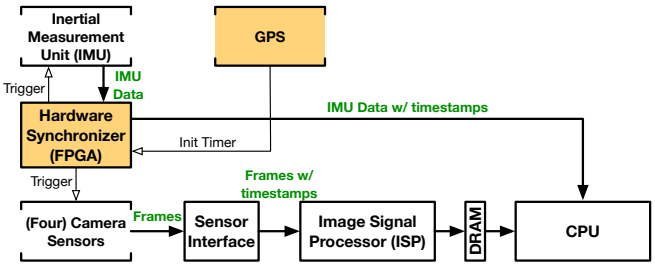
Equally important to triggering sensors simultaneously is to pack a precise timestamp with each sensor sample. This must be done by the synchronizer as sensors are not built to do so. A hardware-only solution would be for the



(a) VIO-based localization (i.e., ego-motion estimation) system, which requires synchronized sensor samples from both the camera and the IMU. The sensor processing pipelines introduce variable latency that challenges the software-based synchronization at the application-level.



(b) Camera and IMU sensor processing pipelines (IMU is triggered 8× faster). If synchronized in software, camera sample `C0` and IMU sample `M7` would be regarded as capturing the same event, whereas in fact `C0` and `M0` capture the same event. Execution times are not drawn to scale.



(c) Our sensor synchronization architecture. GPS provides a single common timing source to trigger cameras and IMU simultaneously. IMU timestamps are obtained in the hardware synchronizer, whereas camera timestamps are obtained in the sensor interface and are later adjusted in software by compensating the constant delay between the camera sensor and the sensor interface.

Fig. 12: Our software-hardware collaborative system for sensor synchronization.

synchronizer to first record the triggering time of each sensor sample, and then pack the timestamp with the raw sensor data before sending the timestamped sample to the CPU. This is indeed what is done for the IMU data as each IMU sample is very small in size (20 Bytes). However, applying the same strategy to cameras would be grossly inefficient, because the each camera frame is large in size (e.g., about 6 MB for an 1080p frame). Transferring each camera frame to the synchronizer just to pack a timestamp is inefficient.

Instead, our design uses a software-hardware collaborative approach shown in Fig. 12c, where camera frames are directly sent to the SoC's sensor interface without the timestamps from the hardware synchronizer. The sensor interface then timestamps each frame. Compared with the exact camera sensor trigger time, the moment that a frame reaches the sensor interface is delayed by the camera exposure time and the image transmission time[4]. Critically, these delays are constant and could be easily derived from the camera sensor specification. The localization application adjusts the timestamp by subtracting the constant delay from the packed timestamp to obtain the triggering time of the camera sensor.

VI-A.3 Performance and Cost Analyses

Our sensor synchronization system is precise. The localization results are indistinguishable from ground truth in our experiments. To adapt to different/more sensors in the future, we implement the hardware synchronizer on the FPGA. The synchronizer is extremely lightweight in design with only 1,443 LUTs and 1,587 registers and consumes 5 mW of power. Our synchronization solution is also fast. It incurs less than 1 *ms* delay to the end-to-end latency.

Our synchronization system can also easily be extended to support more or different types of sensors owing to the software-hardware collaborative strategy. For instance, we are currently exploring next-generation systems that use more than four cameras. Synchronizing more cameras simply requires expanding the number of trigger signals; the rest of synchronization, including timestamp adjustment, is all handled at the application layer. To our best knowledge, this is the first sensor synchronization technology in the public domain that can flexibly adapt to more/different sensors.

VI-B. Augmenting Computing with Sensors

Our perception system is driven by vision algorithms, which are compute-intensive. A key observation we have is that much of the perception information could be derived using additional (inexpensive) sensors while reducing compute cost. In cases where sensing could replace computing, accelerating the computing algorithm has little value. We discuss two concrete instances demonstrated in our vehicles.

Localization VIO is a classic localization algorithm that we use to estimate the position of the vehicle in the global map. VIO, however, fundamentally suffers from cumulative errors [79]. The longer distance the vehicle travels, the more inaccurate the position estimation is. While it is possible to use advanced algorithms to correct the cumulative errors [80], [81], those algorithms tend to be compute-intensive.

To alleviate the VIO cumulative errors with little overhead, we propose a GPS-VIO hybrid approach. A GPS device obtains the Global Navigation Satellite System (GNSS) signal; when the GNSS signal is strong, the GNSS updates are directly used as the vehicle's current position and fed to the planning module. Meanwhile, the GNSS signals are used to correct the VIO errors. In particular, the local position obtained using VIO is fused with the global localization information obtained from the GPS signal using an Extended Kalman Filter (EKF) [47]. If later the GPS reception is unstable (e.g., in underground tunnels) or when GPS multipath problem occurs [82], the corrected VIO results could be

[4]The transmission time consists of the readout time from the sensor's analog buffer [78] and the transmission time on the MIPI/CSI-2 [65] bus.

used to obtain position updates. The EKF fusion algorithm executes in about 1 *ms*, much more lightweight than the VIO localization algorithm (24 *ms*).

Tracking We replace compute-intensive visual tracking algorithms with Radar sensors, which directly measure the relative radial velocity of an object and combine consecutive observations of the same target into a trajectory. Adding Radars increases the vehicle's cost only modestly, as today's automotive Radars cost only about $500 (Table II). Radar is also robust to adverse weather conditions.

While offloading the tracking task to Radars avoids running costly visual tracking algorithms, the challenge is that Radars do not detect objects. Therefore, we must match object detected by vision algorithms with object tracked by Radars. We call this *spatial synchronization*. We propose a spatial synchronization algorithm, which projects object pose from the Radar coordinates to the camera coordinates and then matches the object pose estimations from both the vision algorithms and from the Radar. Our spatial synchronization finishes on the CPU in 1 *ms*, 100 $\times$ more lightweight than KCF, a classic tracking algorithm [46].

VII. Concluding Remarks

This section provides retrospective and prospective views. About 50% of our R&D budget over the past three years has been spent on building and optimizing the computing system. We are not alone, as Tesla also dedicates significant R&D resources towards developing a suitable computing system [83]. We find that autonomous driving incorporates a myriad of different tasks across computing and sensor domains with new design constraints. While accelerating individual algorithms is extremely valuable, what eventually mattered is the systematic understanding and optimization of the end-to-end system.

Holistic SoV Optimizations The SoV of an autonomous vehicle includes many components, such as the sensors, the computing platform, and the vehicle's ECU. We must move beyond optimizing only the computing platform to optimizing the SoV holistically. A critical first step toward holistic SoV optimization is to understand the constraints and trade-offs from an SoV perspective. We present generic latency, throughput, power/energy, and cost models of autonomous vehicles, and demonstrate concrete examples drawn from our own vehicle's design parameters (Sec. III).

Taking an SoV perspective, we show that sensing, a traditionally less optimized component, is a major bottleneck of the end-to-end latency (Sec. V-C). While perception and planning continue to be critical—especially in complex environments and when vehicles are to be precisely maneuvered—optimizing the sensing stack (e.g., sensor processing algorithms, kernel and driver, and hardware architecture) would provide a greater return of investment.

The SoV components are traditionally designed in isolation. However, their tight integration in the SoV requires us to co-optimize them as a whole. We present two examples where sensing and computing could be co-optimized to improve the perception quality (Sec. VI). Broader co-design between optics, sensing, computing, and vehicle engineering could unlock new optimization opportunities in the future.

Horizontal, Cross-Accelerator Optimization Many have studied accelerator designs for individual on-vehicle processing tasks such as localization [84], [14], DNN [85], [86], [87], [88], [89], depth estimation [45], [90], and motion planning [9], [10], but meaningful gains at the system level are possible only if we expand beyond optimizing individual accelerators to exploiting the interactions across accelerators, a.k.a. accelerator-level parallelism (ALP) [91], [92]. We present the data-flow across different on-vehicle algorithms and their inherent task-level parallelisms (Sec. IV), which future work could build on.

While the most common form of ALP today is found on a single chip (e.g., in a smartphone SoC), ALP in autonomous vehicles usually exists across multiple chips. For instance, in our current computing platform localization is accelerated on an FPGA while depth estimation and object detection are accelerated by a GPU (Sec. V-B). Soon on-vehicle processing tasks might be offloaded to edge servers or even the cloud. Efforts that exploit ALP while taking into account constraints arising in different contexts would significantly improve on-vehicle processing.

We also discuss the opportunities and feasibility of time-sharing the FPGA resources across different accelerators via PRP (Sec. V-B3), which presents a new dimension for exploiting ALP in the SoV. We see RPR as a cost-effective solution to support non-essential tasks that used only infrequently. For instance, sensor samples captured in the field could be compressed and upload to the cloud; this task in our deployment happens only once per hour, and thus could be swapped in only when needed.

"TCO" Model for Autonomous Vehicles This paper takes a first step toward presenting the key aspects of an autonomous vehicle's cost, including sensors and computing units in the on-board processing system as well as maintaining and operating the services in the cloud.

We are formulating a comprehensive cost model for autonomous vehicles, which could enable cost-effective optimization opportunities [93] and reveal new design trade-offs such as cost vs. latency, similar in a way that the TCO model [24] drives new optimizations in data centers, e.g., the scale-up vs. scale-out processor designs [94].

References

[1] "Federal Highway Administration, National household travel survey 2017." [Online]. Available: https://nhts.ornl.gov/assets/2017_nhts_summary_travel_trends.pdf

[2] "Mckinsey & company automative & assembly: Micromobility's 15,000-mile checkup." [Online]. Available: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/micromobilitys-15000-mile-checkup

[3] "Denver rtd first and last mile strategic plan." [Online]. Available: https://www.rtd-denver.com/sites/default/files/files/2019-07/FLM-Strategic-Plan_06-10-19.pdf

[4] "Miami-date transportation planning organization: First mile and last mile." [Online]. Available: http://www.miamidadetpo.org/library/studies/first-mile-last-mile-options-with-high-trip-generator-employers-2017-12.pdf

[5] "Rhode island transit master plan: First mile / last mile connections." [Online]. Available: https://transitforwardri.com/pdf/Strategy%20Paper%2015%20First%20Mile%20Last%20Mile.pdf

[6] "In the US, transit deserts are making it hard for people to find jobs and stay healthy." [Online]. Available: https://www.citymetric.com/transport/us-transit-deserts-are-making-it-hard-people-find-jobs-and-stay-healthy-3291

[7] W. Goodall, T. Dovey, J. Bornstein, and B. Bonthron, "The rise of mobility as a service," *Deloitte Rev*, vol. 20, pp. 112–129, 2017.

[8] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, "The architectural implications of autonomous driving: Constraints and acceleration," in *ACM SIGPLAN Notices*, vol. 53, no. 2. ACM, 2018, pp. 751–766.

[9] J. Sacks, D. Mahajan, R. C. Lawson, and H. Esmaeilzadeh, "Robox: an end-to-end solution to accelerate autonomous control in robotics," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 2018, pp. 479–490.

[10] S. Murray, W. Floyd-Jones, Y. Qi, G. Konidaris, and D. J. Sorin, "The microarchitecture of a real-time robot motion planning accelerator," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 45.

[11] S. Murray, W. Floyd-Jones, Y. Qi, D. J. Sorin, and G. Konidaris, "Robot motion planning on a chip." in *Robotics: Science and Systems*, 2016.

[12] B. Boroujerdian, H. Genc, S. Krishnan, W. Cui, A. Faust, and V. Reddi, "Mavbench: Micro aerial vehicle benchmarking," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 894–907.

[13] A. Suleiman, Z. Zhang, L. Carlone, S. Karaman, and V. Sze, "Navion: A 2-mw fully integrated real-time visual-inertial odometry accelerator for autonomous navigation of nano drones," *IEEE Journal of Solid-State Circuits*, vol. 54, no. 4, pp. 1106–1119, 2019.

[14] J. Zhang and S. Singh, "Visual-lidar odometry and mapping: Low-drift, robust, and fast," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2015, pp. 2174–2181.

[15] T. Xu, B. Tian, and Y. Zhu, "Tigris: Architecture and algorithms for 3d perception in point clouds," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 629–642.

[16] "Preliminary statement of policy concerning automated vehicles, national highway traffic safety administration and others," *Washington, DC*, pp. 1–14, 2013.

[17] S. Liu, L. Li, J. Tang, S. Wu, and J.-L. Gaudiot, "Creating autonomous vehicle systems," *Synthesis Lectures on Computer Science*, vol. 6, no. 1, pp. i–186, 2017.

[18] S. Liu, J. Tang, C. Wang, Q. Wang, and J.-L. Gaudiot, "A unified cloud platform for autonomous driving," *Computer*, vol. 50, no. 12, pp. 42–49, 2017.

[19] "Open Street Map." [Online]. Available: https://www.openstreetmap.org/

[20] "Tesla Smart Summon & Custom Maps Will Use Crowdsourced Fleet Data." [Online]. Available: https://cleantechnica.com/2020/04/22/crowdsourced-fleet-data-will-be-used-to-generate-the-next-gen-tesla-maps/

[21] I. Newton, *Mathematical principles of natural philosophy*. A. Strahan, 1802.

[22] "LiDAR Specification Comparison." [Online]. Available: https://autonomoustuff.com/wp-content/uploads/2018/04/LiDAR_Comparison.pdf

[23] E. Kim, J. Lee, and K. G. Shin, "Real-time prediction of battery power requirements for electric vehicles," in *2013 ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2013, pp. 11–20.

[24] L. A. Barroso, U. Hölzle, and P. Ranganathan, "The datacenter as a computer: Designing warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, vol. 13, no. 3, pp. i–189, 2018.

[25] H. Zhao, Y. Zhang, P. Meng, H. Shi, L. E. Li, T. Lou, and J. Zhao, "Towards safety-aware computing system design in autonomous vehicles," *arXiv preprint arXiv:1905.08453*, 2019.

[26] Y. Zhu, V. J. Reddi, R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, "Cognitive computing safety: The new horizon for reliability/the design and evolution of deep learning workloads," *IEEE Micro*, vol. 37, no. 1, pp. 15–21, 2017.

[27] "Automotive Grade Zynq UltraScale+ MPSoCs." [Online]. Available: https://www.xilinx.com/products/silicon-devices/soc/xa-zynq-ultrascale-mpsoc.html

[28] A. Aldoma, Z.-C. Marton, F. Tombari, W. Wohlkinger, C. Potthast, B. Zeisl, R. B. Rusu, S. Gedikli, and M. Vincze, "Tutorial: Point cloud library: Three-dimensional object recognition and 6 dof pose estimation," *IEEE Robotics & Automation Magazine*, vol. 19, no. 3, pp. 80–91, 2012.

[29] K. Xu, Y. Li, T. Ju, S.-M. Hu, and T.-Q. Liu, "Efficient affinity-based edit propagation using kd tree," in *ACM Transactions on Graphics (TOG)*, vol. 28, no. 5. ACM, 2009, p. 118.

[30] J. Park, Q.-Y. Zhou, and V. Koltun, "Colored point cloud registration revisited," in *Proceedings of the IEEE International Conference on Computer Vision*, 2017, pp. 143–152.

[31] R. B. Rusu and S. Cousins, "Point cloud library (pcl)," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 1–4.

[32] Z. Liu, H. Tang, Y. Lin, and S. Han, "Point-voxel cnn for efficient 3d deep learning," *arXiv preprint arXiv:1907.03739*, 2019.

[33] "Waymo Offers a Peek Into the Huge Trove of Data Collected by Its Self-Driving Cars." [Online]. Available: https://spectrum.ieee.org/cars-that-think/transportation/self-driving/waymo-opens-up-part-of-its-humongous-selfdriving-database

[34] "Velodyna HDL-64E." [Online]. Available: https://velodynelidar.com/products/hdl-64e/

[35] "Velodyna Puck." [Online]. Available: https://velodynelidar.com/products/puck/

[36] "What it really costs to turn a car into a self-driving vehicle." [Online]. Available: https://qz.com/924212/what-it-really-costs-to-turn-a-car-into-a-self-driving-vehicle/

[37] J. Jiao, "Machine learning assisted high-definition map creation," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2018, pp. 367–373.

[38] G. J. Iddan and G. Yahav, "Three-dimensional imaging in the studio and elsewhere," in *Three-Dimensional Image Capture and Applications IV*, vol. 4298. International Society for Optics and Photonics, 2001, pp. 48–55.

[39] B. Schwarz, "Lidar: Mapping the world in 3d," *Nature Photonics*, vol. 4, no. 7, p. 429, 2010.

[40] S. A. Mohamed, M.-H. Haghbayan, T. Westerlund, J. Heikkonen, H. Tenhunen, and J. Plosila, "A survey on odometry for autonomous navigation systems," *IEEE Access*, vol. 7, pp. 97 466–97 486, 2019.

[41] M. Bloesch, S. Omari, M. Hutter, and R. Siegwart, "Robust visual inertial odometry using a direct ekf-based approach," in *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*. IEEE, 2015, pp. 298–304.

[42] D. Scharstein, R. Szeliski, and R. Zabih, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," in *Proceedings of 1st IEEE Workshop on Stereo and Multi-Baseline Vision*, 2001.

[43] M. Z. Brown, D. Burschka, and G. D. Hager, "Advances in computational stereo," 2003.

[44] A. Geiger, M. Roser, and R. Urtasun, "Efficient large-scale stereo matching," in *Proceedings of the 10th Asian Conference on Computer Vision*, 2010.

[45] Y. Feng, P. Whatmough, and Y. Zhu, "Asv: Accelerated stereo vision system," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 643–656.

[46] J. F. Henriques, R. Caseiro, P. Martins, and J. Batista, "High-speed tracking with kernelized correlation filters," *IEEE transactions on pattern analysis and machine intelligence*, vol. 37, no. 3, pp. 583–596, 2014.

[47] A. Kelly, *Mobile robotics: mathematics, models, and methods*. Cambridge University Press, 2013.

[48] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[49] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.

[50] "Snapdragon Mobile Platform." [Online]. Available: https://www.qualcomm.com/snapdragon

[51] "NVIDIA Tegra." [Online]. Available: https://www.nvidia.com/object/tegra-features.html

[52] G. Loianno, C. Brunner, G. McGrath, and V. Kumar, "Estimation, control, and planning for aggressive flight with a small quadrotor with a single camera and imu," *IEEE Robotics and Automation Letters*, vol. 2, no. 2, pp. 404–411, 2016.

[53] L. Liu, S. Liu, Z. Zhang, B. Yu, J. Tang, and Y. Xie, "Pirt: A runtime framework to enable energy-efficient real-time robotic applications on heterogeneous architectures," *arXiv preprint arXiv:1802.08359*, 2018.

[54] N. de Palézieux, T. Nägeli, and O. Hilliges, "Duo-vio: Fast, light-weight, stereo inertial odometry," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2016, pp. 2237–2242.

[55] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot, "Computer architectures for autonomous driving," *Computer*, vol. 50, no. 8, pp. 18–25, 2017.

[56] "NVIDIA Jetson TX2 Module." [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/

[57] P. Yedlapalli, N. C. Nachiappan, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Short-circuiting memory traffic in handheld platforms," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 166–177.

[58] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "VIP: Virtualizing IP Chains on Handheld Platforms," in *Proc. of ISCA*, 2015.

[59] J. Tang, B. Yu, S. Liu, Z. Zhang, W. Fang, and Y. Zhang, "π-soc: Heterogeneous soc architecture for visual inertial slam applications," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 8302–8307.

[60] "NXP ADAS and Highly Automated Driving Solutions for Automotive." [Online]. Available: https://www.nxp.com/applications/solutions/automotive/adas-and-highly-automated-driving:ADAS-AND-AUTONOMOUS-DRIVING

[61] "Intel MobilEye Autonomous Driving Solutions." [Online]. Available: https://www.mobileye.com/

[62] "NVIDIA DRIVE." [Online]. Available: https://www.nvidia.com/en-us/self-driving-cars/drive-platform/

[63] "CANopen." [Online]. Available: https://www.can-cia.org/canopen/

[64] "Open Source Car Control (OSCC)." [Online]. Available: https://github.com/PolySync/oscc

[65] "MIPI Camera Serial Interface 2 (MIPI CSI-2)." [Online]. Available: https://www.mipi.org/specifications/csi-2

[66] "Regulus ISP." [Online]. Available: https://www.xilinx.com/products/intellectual-property/1-r6gz2f.html

[67] E. Rublee, V. Rabaud, K. Konolige, and G. R. Bradski, "Orb: An efficient alternative to sift or surf." in *ICCV*, vol. 11, no. 1. Citeseer, 2011, p. 2.

[68] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, 1981.

[69] "Vivado design suite tutorial: Partial reconfiguration." [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug947-vivado-partial-reconfiguration-tutorial.pdf

[70] "Vivado design suite user guide: Partial reconfiguration." [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug909-vivado-partial-reconfiguration.pdf

[71] S. Liu, R. N. Pittman, A. Forin, and J.-L. Gaudiot, "Achieving energy efficiency through runtime partial reconfiguration on reconfigurable systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, p. 72, 2013.

[72] C. Claus, B. Zhang, W. Stechele, L. Braun, M. Hubner, and J. Becker, "A multi-platform controller allowing for maximum dynamic partial reconfiguration throughput," in *2008 International Conference on Field Programmable Logic and Applications*. IEEE, 2008, pp. 535–538.

[73] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[74] H. Fan, F. Zhu, C. Liu, L. Zhang, L. Zhuang, D. Li, W. Zhu, J. Hu, H. Li, and Q. Kong, "Baidu Apollo EM Motion Planner," *arXiv preprint arXiv:1807.08048*, 2018.

[75] Z. Zhang, A. A. Suleiman, L. Carlone, V. Sze, and S. Karaman, "Visual-inertial odometry on chip: An algorithm-and-hardware co-design approach," 2017.

[76] Y. Zhu, A. Samajdar, M. Mattina, and P. Whatmough, "Euphrates: Algorithm-soc co-design for low-power mobile continuous vision," in *Proceedings of the 45th ACM/IEEE Annual International Symposium on Computer Architecture*, 2018.

[77] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2012, pp. 3354–3361.

[78] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Ba, "Energy Characterization and Optimization of Image Sensing Toward Continuous Mobile Vision," in *Proc. of MobiSys*, 2013.

[79] M. A. Skoglund, G. Hendeby, and D. Axehill, "Extended kalman filter modifications based on an optimization view point," in *2015 18th International Conference on Information Fusion (Fusion)*. IEEE, 2015, pp. 1856–1861.

[80] S. Leutenegger, S. Lynen, M. Bosse, R. Siegwart, and P. Furgale, "Keyframe-based visual–inertial odometry using nonlinear optimization," *The International Journal of Robotics Research*, vol. 34, no. 3, pp. 314–334, 2015.

[81] T. Qin, P. Li, and S. Shen, "Vins-mono: A robust and versatile monocular visual-inertial state estimator," *IEEE Transactions on Robotics*, vol. 34, no. 4, pp. 1004–1020, 2018.

[82] T. Kos, I. Markezic, and J. Pokrajcic, "Effects of multipath reception on gps positioning performance," in *Proceedings ELMAR-2010*. IEEE, 2010, pp. 399–402.

[83] "Tesla's new self-driving chip." [Online]. Available: https://www.theverge.com/2019/4/22/18511594/tesla-new-self-driving-chip-is-here-and-this-is-your-best-look-yet

[84] A. Suleiman, Z. Zhang, L. Carlone, S. Karaman, and V. Sze, "Navion: a fully integrated energy-efficient visual-inertial odometry accelerator for autonomous navigation of nano drones," in *2018 IEEE Symposium on VLSI Circuits*. IEEE, 2018, pp. 133–134.

[85] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[86] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[87] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.

[88] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 27–40.

[89] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 27–39.

[90] D. Wofk, F. Ma, T.-J. Yang, S. Karaman, and V. Sze, "Fastdepth: Fast monocular depth estimation on embedded systems," *arXiv preprint arXiv:1903.03273*, 2019.

[91] M. D. Hill and V. J. Reddi, "Accelerator level parallelism," *arXiv preprint arXiv:1907.02064*, 2019.

[92] M. Hill and V. J. Reddi, "Gables: A roofline model for mobile socs," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 317–330.

[93] D. A. Wood and M. D. Hill, "Cost-effective parallel computing," *Computer*, vol. 28, no. 2, pp. 69–72, 1995.

[94] B. Grot, D. Hardy, P. Lotfi-Kamran, B. Falsafi, C. Nicopoulos, and Y. Sazeides, "Optimizing data-center tco with scale-out processors," *IEEE Micro*, vol. 32, no. 5, pp. 52–63, 2012.