Assessing the Adherence of an Industrial Autonomous Driving Framework to ISO 26262 Software Guidelines

Hamid Tabani, Leonidas Kosmidis, Jaume Abella, Francisco J. Cazorla
Barcelona Supercomputing Center

Guillem Bernat
Rapita Systems LTD

ABSTRACT

The complexity and size of Autonomous Driving (AD) software are comparably higher than that of software implementing other (standard) functionalities in the car. To make things worse, a big fraction of AD software is not specifically designed for the automotive (or any other critical) domain, but the mainstream market. This brings uncertainty on to which extent AD software adheres to guidelines in safety standards. In this paper, we present our experience in applying ISO 26262 – the applicable functional safety standard for road vehicles – software safety guidelines to industrial AD software, in particular, Apollo, a heterogeneous Autonomous Driving framework used extensively in industry. We provide quantitative and qualitative metrics of compliance for many ISO 26262 recommendations on software design, implementation, and testing.

KEYWORDS

Critical Systems, Autonomous Driving, ISO 26262

ACM Reference Format:
Hamid Tabani, Leonidas Kosmidis, Jaume Abella, Francisco J. Cazorla and Guillem Bernat. 2019. Assessing the Adherence of an Industrial Autonomous Driving Framework to ISO 26262 Software Guidelines. In *The 56th Annual Design Automation Conference 2019 (DAC '19), June 2–6, 2019, Las Vegas, NV, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3316781.3317779

1 INTRODUCTION

The potential socio-economic benefits of Autonomous Driving (AD) have motivated automotive industry to assess the feasibility of developing autonomous cars. In doing so, intuitively, Advanced-Driver Assistance Systems (ADAS) such as lane keeping can serve the automotive industry as a 'test case' to master the complexity in the specification, design, implementation, verification, and validation of advanced software-controlled functionalites. However, AD brings its own set of challenges. In particular, safety argumentation for hardware and software in AD and ADAS differs. In ADAS, it builds on the ability of the human driver to take the system to a safe state on the event of a software/hardware failure. That is, the human acts as a backup safety mechanism and is responsible for taking the right corrective action. Instead, for AD (especially full AD or Level 5), there can be no driver. This increases the complexity of AD software that must handle safety under driving conditions. To that end, the software implements a complex control building on artificial intelligence (AI) algorithms and manages big amounts of sensor data. This translates into a huge increase in computing performance requirements, 100x from 2016 to 2024 according to ARM prospects [6], and the use of advanced performance-improving processor designs, e.g. GPUs (Graphics Processing Units). Also, software (and hardware) must be designed and validated for a high Automotive Safety Integrity Level (ASIL)[1]. In particular, for high levels of autonomous driving in which AD systems will control safety-related driving aspects, AD systems will reach ASIL-D and must be designed to remain (fail) operational regardless of the presence of a fault. This requires robust mitigation techniques to decrease risks and/or meet specific failure probability bounds.

For road vehicles, ISO 26262 guides the production of software, listing for each process of the life-cycle i) the safety requirements, ii) the activities required to meet those requirements, and iii) the evidence that is required to demonstrate that the requirements are fulfilled, with traceability as a fundamental element to link high-level requirements, low-level requirements, and analyzes. In short, if the designed analyzes and tests are passed (e.g., 100% code coverage is achieved), an assessment of the quality of the implementation and its fit for purpose can be made. For AD software frameworks, the challenge lies on the fact that so far they have been designed with low focus on safety as dictated by safety standards. First of all, during the initial phases of the ISO 26262 safety life-cycle, it is required to define the AD safety goals, with already some efforts trying to formalize them [11]. Then, safety requirements are allocated, via a technical safety concept, to software and other architectural components so that safety goals are not violated. However, the extent to which AD software frameworks adhere to ISO 26262 Part 6, which specifies the product development at the software level, has not been assessed.

In this paper, with focus on Apollo [5], an AD framework currently running on a variety of commercial vehicles, we show AD's adherence to ISO 26262 in terms of: i) software architectural design; ii) software modeling and coding guidelines; iii) software unit design and implementation; and iv) code coverage, the most common form of structural coverage analysis. To our knowledge, this is the first analysis of a complete industrial AD software in the literature regarding adherence to ISO 26262 guidelines for software. Hence, our analysis assesses the gaps between AD software current implementation and the requirements for its certification.

In our analysis, we cover some issues hampering AD heterogeneous software adherence to ISO 26262 that can be solved with

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317779

[1]The automotive functional safety standard ISO 26262, defines 4 ASIL varying from ASIL-A (lowest criticality) to ASIL-D (highest criticality). Besides, the Quality Management (QM) category covers those components that cannot cause safety risks upon a failure.

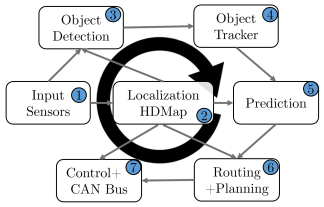


Figure 1: A state-of-the-art AD system pipeline.

limited software engineering effort and those that are much deeper and require research innovations to be successfully addressed. As an illustrative example of the latter, we show that how the GPU part (i.e., the GPU code) of AD applications is programmed is at odds with ISO 26262 requirements. This occurs because programming languages for the mainstream market, e.g., CUDA, make intrinsic use of features not recommended in ISO 26262 (e.g., pointers and dynamic memory). Overall, we provide several key insights and research directions to address the main challenges brought by AD software in terms of safety assurance.

The rest of the paper is organized as follows: Section 2 introduces Apollo and its structure. Section 3 describes the main contribution of our work: the adherence of Apollo to ISO 26262 software guidelines, with explicit references to the (little) related work. Finally, Section 4 presents the main conclusions of this work.

2 APOLLO INDUSTRIAL AD FRAMEWORK

Autonomous cars, also known as driver-less cars and self-driving cars, are vehicles that can guide themselves toward a specified destination without human intervention. AD software combines several input sensors such as video cameras, short-range and long-range radars and laser sensors to detect the surrounding area and track the moving objects around the car.

AD systems implement very precise navigation techniques in order to locate the position of the vehicle. By providing the position of the vehicle and enough information about the surrounding area, the AD system plans the future paths using the specified routing and generates control commands for the vehicle to follow the specified paths. These are the main stages of Apollo and also other state-of-the-art AD frameworks [2, 4, 5]. All of them have similar design and implementation characteristics, so the conclusions we derive for Apollo in this work hold to a large extent for all AD frameworks.

Over 110 industrial partners, encompassing top-tier AI companies and car manufacturers, already contribute to Apollo as a large industrial project [5]. Apollo AD is implemented in a large number of commercial automotive companies, and it is already deployed on several prototype vehicles, including autonomous trucks. This is the primary reason for the selection of Apollo for our study.

AD software comprises several modules [7], see Figure 1, with compute-intensive ones already implemented to use GPUs.

- *Object Detection* identifies objects of interest surrounding the car using the LIDAR, camera, and radar sensors.
- *Object Tracking* is responsible for observing moving objects like other vehicles, bicycles, and pedestrians over time. In some AD frameworks, including Apollo, object detection, and tracking are parts of a bigger module which is called *Perception*.

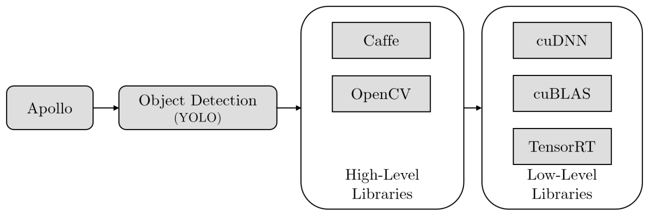


Figure 2: Taxonomy of the Libraries used by Apollo's perception module.

- *Prediction* anticipates the future motion trajectories of perceived obstacles and objects.
- *Localization* uses different sensors to calculate the precise location of the vehicle.
- *Routing* finds the best route and guides the vehicle to reach its destination.
- *Planning* takes the prediction and routing outputs to plan a safe and collision-free trajectory for the vehicle to take.
- *Control*, based on the output of these modules, generates control commands such as accelerating, braking, and steering.
- Finally, the *CAN Bus* module passes all the control commands to the vehicle hardware and also provides some information back to the AD system.

The perception module, which includes object detection and tracking, is known to have very high complexity and computing performance requirements, besides representing a large fraction of the overall Apollo's execution time [5].

For camera-based object detection, Apollo employs YOLO [10], the state-of-the-art object detection algorithm used in a variety of domains. The Convolutional Neural Network inference for object detection is the most computationally intensive function as it requires to perform a huge amount of computations to process at several frames per second. Note that the use of deep learning in the past few years has significantly increased the resulting algorithm accuracy in a variety of areas including computer vision and, therefore, nowadays Deep Neural Networks are considered the reference approach in such domains. The provided functionalities are implemented using low-level GPU accelerated libraries specifically optimized for certain GPUs like the ones from NVIDIA, used in this study. Figure 2 presents a taxonomy of the modules and libraries used in the object detection module of Apollo.

3 SOFTWARE SAFETY GUIDELINES

Part 6 of ISO 26262 specifies the requirements for software development. ISO 26262 covers software's: (a) safety requirements specification, (b) architectural design, (c) unit design, and implementation, (d) unit testing, (e) integration and testing, and (f) safety requirements verification. In this work, we focus on a subset of them. In particular, we cover different aspects of (b), (c), (d), and (e). We present first those aspects for which we can report quantitative (instead of qualitative) results.

3.1 Software Architectural Design

ISO 26262 specifies coding and modeling guidelines for the product development phase at the software level as shown in Table 1. ISO

Table 1: Modeling/coding guidelines (ISO26262_6 Table 1)

Criticality Level	A	B	C	D
1) Enforcement of low complexity	++	++	++	++
2) Use language subsets	++	++	++	++
3) Enforcement of strong typing	++	++	++	++
4) Use defensive implementation techniques	o	+	++	++
5) Use established design principles	+	+	+	++
6) Use unambiguous graphical representation	+	++	++	++
7) Use style guides	+	++	++	++
8) Use naming conventions	++	++	++	++

26262 uses the following notation to capture the extent to which a particular technique is required under a certain ASIL: ++ highly recommended; + recommended; and o 'not required'. As it can be seen, all elements are highly recommended for ASIL D, which is the target ASIL we consider for the entire AD pipeline in Apollo, since all modules affect the car motion. In the following, we provide and discuss our analysis regarding each of the requirements of Table 1.

3.1.1 Enforcement of low complexity. We have analyzed the source code of the different software modules using the Lizard cyclomatic complexity analysis tool [3]. It measures the number of independent paths in a target source. For instance, a program with a single `if` statement has a cyclomatic complexity of two and a program with two nested `if` conditions result in complexity of three.

The crosses and diamonds in Figure 3 respectively show the total number of lines of code (LOC) and the number of functions in each module. Note that the entire Apollo framework is composed of more than 220k LOC, which means that it has a considerable size and this analysis is not trivial. All modules are in the order of tens of thousands of LOC and hundreds or even thousands of functions.

The bars in Figure 3 show the number of functions in different modules of Apollo with a cyclomatic complexity over a given value. While no exact cyclomatic complexity limit fits all domains, in critical systems, it is especially delicate since increasing complexity impacts the already costly verification activities. As reference ranges we use: 1-10 (low); 11-20 (moderate); 21-50 (risky); and >50 (unstable). We can see that, in general, the degree of complexity of the code is high. Modules have in the order of dozens of functions with moderate or higher complexity, amounting 554 for the entire Apollo framework. Such high code complexity challenges the functional verification of the code as well as its timing analysis (e.g., worst-case execution time and response time) estimation.

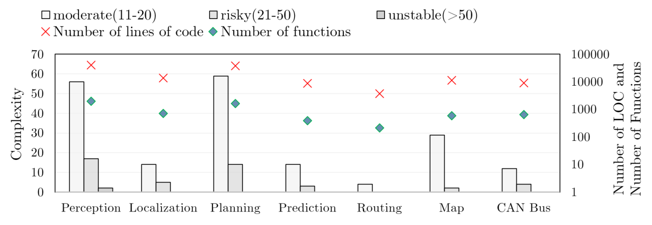


Figure 3: Complexity, number of LOC, and the number of functions in Apollo Modules.

Observation 1. *AD frameworks present a high complexity in terms of cyclomatic complexity.* From Observation 1, it follows that significant redesign and recoding is needed in AD frameworks to reach low complexity levels as expected in high-integrity software functionality. This is mandatory to reduce verification costs and facilitate timing (WCET) estimation.

```
__global__ void scale_bias_kernel(float *output, float *biases, int n, int size)
{
    int offset = blockIdx.x * blockDim.x + threadIdx.x;
    int filter = blockIdx.y;
    int batch = blockIdx.z;

    if(offset < size) output[(batch*n+filter)*size + offset] *= biases[filter];
}

void scale_bias_gpu(float *output, float *biases, int batch, int n, int size)
{
    dim3 dimGrid((size-1)/BLOCK + 1, n, batch);
    dim3 dimBlock(BLOCK, 1, 1);

    scale_bias_kernel<<<dimGrid, dimBlock>>>(output, biases, n, size);
}
```

Figure 4: CUDA code in object detection module.

3.1.2 Use of language subsets. In this work, we focus on MISRA [8], the guideline for the use of the C language in vehicle-based software, which stipulates 143 rules (MISRA C:2012). Since AD applications are not programmed targeting any critical market in particular, they naturally do not adhere to MISRA C.

Observation 2. *The CPU part of AD frameworks is not programmed according to any safety-related guideline.* In our view, it is possible with moderate effort to change the code to adhere to a language subset like MISRA C.

More interestingly, we found that for the GPU code there is no standard (language subset) defined to simplify the safety assessment. It follows that there are no tools to assess whether a particular code sample adheres to it.

Observation 3. *No guideline or language subset exist for GPU code to facilitate code safety assessment in critical systems.*

In this line, we assessed whether some of the features required for CPU code according to MISRA C apply to GPU (CUDA) code. Similar to [14], our analysis shows that CUDA programming heavily builds on the use of pointers and dynamic memory allocation, while ISO 26262 highly recommends not to use dynamic objects or variables. As an illustrative example, Figure 4 shows an excerpt of object detection function `scale_bias_gpu` that has a typical CUDA program structure. As shown, `output` and `biases` pointers are respectively used to access dynamically created arrays of floating-point data containing layer outputs and biases. Note that CUDA memory allocations, `cudaMalloc`, transfer the data to the device memory. Operations to copy the data back from device memory are not shown in this code excerpt. Hence, CUDA programs build on pointers as an indispensable feature to allow the programmer to allocate and maintain two separate sets of pointers explicitly, one for the host memory and one for the device memory.

Observation 4. *CUDA code intrinsically uses features not recommended in ISO 26262 (e.g., use of pointers and dynamic memory).*

Unlike Observation 2 that can be handled with relatively small effort, Observations 3 and 4 carry deeper changes. In particular, the use of GPU programming languages that are more friendly to certification imposes significant code modifications. In this line, alternative initiatives like the Brook Auto GPU programming language [14] help in simplifying certification: in the same way that MISRA C constraints C, Brook Auto defines a subset of the Brook stream-programming language rules that are certification friendly, without limiting the expressiveness of the language. For instance, Brook Auto does not expose pointers to the programmer, and takes care of those tasks automatically, reducing the possibility of human errors. Furthermore, Brook Auto achieves competitive performance results in low-level GPU languages [14].

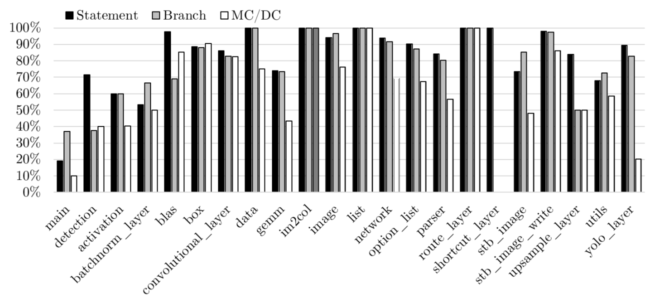


Figure 5: Coverage achieved for object detection (YOLO)

3.1.3 Enforcement of strong typing. While there is no single definition of "strong typing", C and C++ languages are generally agreed as "less strongly typed" than some other languages due to their support for implicit and explicit casting. In Apollo, we have observed more than 1,400 explicit castings, which confronts the requirements of the ISO 26262 standard.

Observation 5. *Most of the AD frameworks and modules are programmed in C or C++, requiring the programmer to identify and resolve any type of castings in the code.*

3.1.4 Use of defensive implementation techniques. Defensive implementation improves software and source code in many aspects. For instance, the software must behave predictably despite unexpected inputs or user inputs. This requires that all the functions should check the validity of their input parameters before using them. Furthermore, all the callers of a function should handle all possible return values from the called function. Our analysis of the source code of Apollo shows that defensive programming techniques are not used.

Observation 6. *AD frameworks do not implement defensive programming techniques.* However, with limited effort, this feature can be added to the code.

3.1.5 Use of established design principles. This category is very broad, but many design principles are related to properties that are statically checkable. For example, design guidelines may impose restrictions on the use of global variables, or exception handling. Although the code properly uses C++ exception handling in most of the cases, we observe the use of global variables frequently. Their use impacts functional validation and testing since it becomes more challenging determining value ranges.

Observation 7. *AD software uses global variables. This requires changing the code to eliminate them or more complex argumentation to support their use and correct behavior.*

3.1.6 Use unambiguous graphical representation. Since all the AD frameworks are written in C/C++, this requirement is not applicable.

3.1.7 Use style guides. Style guides typically cover topics such as code layout, capitalization, comments, and white space. For Apollo source code, we used a style guide tool to process the code, and it verifies that the proper coding style is very well achieved.

Observation 8. *AD software follows style guides.* In particular, Apollo software adopted the Google C++ style guide, and contributors have to validate their code using defined style checkers.

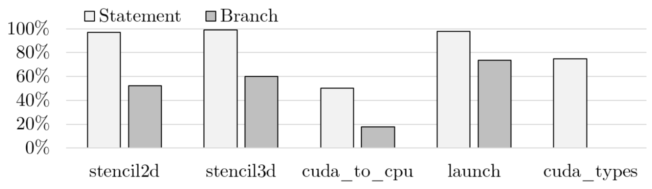


Figure 6: Statement and branch coverage for a CUDA code modified to be run in the CPU.

3.1.8 Use naming conventions. According to several coding guidelines including Google C++ guidelines, the names of all types, classes, structs, type aliases, enums, and type template parameters should have the same naming convention. The Apollo code adheres to all these properties.

Observation 9. *AD software adheres to the properties of coding guidelines.* Apollo software uses a very well-structured implementation, and all coding guidelines are followed.

3.2 Software Unit Testing: CPU Code Coverage

Software unit testing is an important requirement in ISO 26262. In this section, we focus on metrics of code coverage that provide evidence of correct execution by showing that different parts of the program have been sufficiently exercised by the tests.

For the CPU code, we focus on standard statement, branch, and Modified Condition/Decision (MC/DC) coverage metrics. In particular, we run several real-scenario tests and use `Rapita System's` RapiCover commercial tool [12] to measure the object detection code coverage in the Apollo's Object Detection module.

In Figure 5, the X-axis list the files in the different modules of YOLO. Each file covers all the functions implemented in it. In our experiments, we excluded all those functions that were not called. Despite that, statement, branch, and MC/DC coverage are very low. Average coverage is 83%, 75% and 61% for statement, branch and MC/DC respectively, and as low as 19%, 37% and 10% respectively for individual files.

While ISO 26262 does not specify a particular coverage figure, its parent standard, IEC61508 (Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems) recommends 100% coverage for all metrics. In ISO 26262, either branch or code statement are *highly recommended* ('++') for all ASIL.

Observation 10. *Code coverage for AD software is low with available tests. Thus, additional test cases are required to reach much higher coverage (preferably 100% coverage).*

3.3 Software Unit Testing: GPU Code Coverage

For GPU code we make the following observations:

Observation 11. *Tool support in the real-time domain to measure code coverage of GPU code is very limited.*

To our knowledge, no code coverage qualified tool exists to analyze GPU code coverage. Since GPU code of AD software is as critical as CPU code – ASIL-D fail-operational for fully autonomous cars – and hence, requires undergoing the same analysis and testing, automated tools to measure GPU code coverage are needed.

In order to provide some GPU code coverage numbers, we modified the code in such a way that it runs in the CPU or emulates the CUDA API in the CPU. While this approach is not applicable for

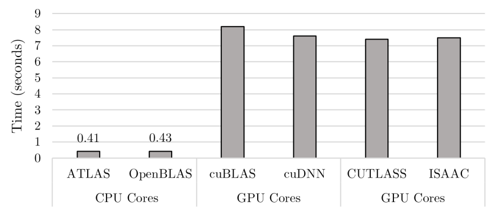


Figure 7: Performance of Apollo's object detection using open-source CUDA libraries in comparison with closed-source libraries implementation.

safety considerations as one the arguments is that code coverage needs to be performed on a representative target and compiler, we use it to get some figures on GPU code coverage. In particular, we used cuda4cpu [1] and applied it to 2D and 3D stencil computation GPU kernels, as a representative of the open-source code. On the resulting code, we applied code coverage tools obtaining the branch and statement coverage figures shown in Figure 6. The reported values show that full code coverage is not achieved either for statements or branches.

3.3.1 Other challenges. Other elements that can hamper carrying out testing in an efficient manner include the use of closed-source libraries.

Observation 12. *Heterogeneous AD software makes extensive use of performance-optimized closed-source CUDA libraries, which hampers assessing compliance against ISO 26262.*

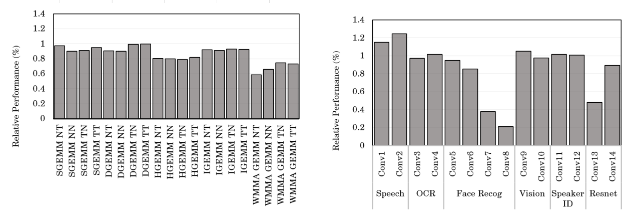
As it can be seen in Figure 2, AD software, or the high-level DNN-related libraries it uses, make use of low-level libraries optimized to run certain functions in the GPU. The latter, for competitiveness reasons, are only offered in closed-source form by the chip vendor. These libraries include:

- *cuDNN.* NVIDIA CUDA Deep Neural Network GPU-accelerated library of primitives for deep neural networks.
- *cuBLAS.* Implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA runtime.
- *TensorRT.* NVIDIA TensorRT high-performance deep learning inference optimizer and runtime that delivers low latency and high-throughput for deep learning inference.

Regarding certification, having closed-source libraries require their owners to go through the certification process and adapt their libraries to fit ISO 26262 requirements. While in theory library users can use black-box testing, this however, has severe implications for ISO 26262 applicability and, in its current form, it is not satisfactory.

Alternatives include the adoption by suppliers of a 'safety culture' based on openness or make that library users deploy state-of-the-art open-source libraries. In this work, we advocate for the latter option. However, in order to make it attractive, used open-source libraries must provide competitive performance with respect to well-known closed-source libraries.

Case Study. In AD software, including Apollo, the perception module is the main module that extensively uses DNN-related libraries. Object detection algorithms can be implemented using either cuBLAS or cuDNN closed-source libraries. The latter library is specifically designed for deep neural networks and can use NVIDIA's tensor cores. Therefore, depending on the capabilities of the target architecture on which Apollo will run, one of these two libraries can be selected by the programmer.



(a) CUTLASS vs cuBLAS

(b) ISAAC vs cuDNN

Figure 8: Relative performance of open-source CUDA libraries compared to closed-source CUDA libraries in variety of widely-used applications and kernels.

For the sake of generality, we take both implementations into account. Then, we implement and run Apollo's object detection module using NVIDIA's CUTLASS [9], an open-source collection of CUDA C++ template abstractions for implementing high-performance matrix-multiplication, and ISAAC [13], an input-aware auto-tuning framework and code-generator for compute-bound HPC kernels.

As Figure 7 shows, the implementations based on CUTLASS and ISAAC provide competitive performance in comparison to cuBLAS and cuDNN, which are the libraries used in the baseline. Also note that the same operations run on the CPU cores using highly optimized libraries (ATLAS and OpenBLAS) with two orders of magnitude higher execution time, which demonstrates the inability of CPUs for such compute-intensive workloads.

As another set of illustrative examples, we compare different general matrix multiplication (GEMM) kernels, widely use in YOLO, implementations using cuBLAS and CUTLASS. In order to construct device-wide GEMM kernels, CUTLASS primitives exhibit performance comparable to cuBLAS for scalar GEMM computations as Figure 8(a) shows.

Similarly, we compare the performance of convolution kernel implementations using cuDNN and ISAAC for a variety of domains. As shown in Figure 8(b), ISAAC provides very competitive performance in comparison with cuDNN for a variety of workloads. We can see that open-source libraries can provide comparable performance.

3.4 Software Modeling and Coding Guidelines

Software architectural design, early in the product design phase, provides a design that realizes the software safety requirements. Table 2 itemizes particular requirements for architectural design.

Table 2: Architectural design (ISO26262_6 Table 3)

Criticality Level	A	B	C	D
1) Hierarchical structure of SW components	++	++	++	++
2) Restricted size of software components	++	++	++	++
3) Restricted size of interfaces	+	+	+	+
4) High cohesion in each software component	+	++	++	++
5) Restricted coupling between SW components	+	++	++	++
6) Appropriate scheduling properties	++	++	++	++
7) Restricted use of interrupts	+	+	+	++

3.4.1 Hierarchical structure of SW components. In ISO 26262, design begins at top level components, which are broken down further to reach the lowest unit of implementation. For instance, the lowest units for software are the functions. To satisfy this requirement, there are several commercial and open-source software tools that provide a hierarchical structure and dependencies across software components.

3.4.2 Other architectural design parameters. The rest of the items in Table 2 can be measured using existing tools and metrics and assessed against specific thresholds. For instance, a limit for the size of software components is not explicitly specified in the standard. Main modules of Apollo have from 5k to 60k lines of code. Provided that a module should be limited to a maximum size, it can be reorganized or redesigned to stay below the maximum size.

Observation 13. *AD frameworks do not comply with many of the principles for software architectural design defined by ISO 26262 such as the restricted size of components and interfaces.* In our view, AD software can be made compliant with these ISO 26262 principles, although with non-negligible effort.

3.5 Software Unit Design and Implementation

Software unit design and implementation build on several guidelines to ensure features such as simplicity, correct order of execution, consistency of interfaces, and data/control flows as presented in Table 3. Our analysis of Apollo's AD software shows that many guidelines are not followed:

Table 3: SW unit design & implement. (ISO26262_6 Table 8)

Criticality Level	A	B	C	D
1) One entry and one exit point in functions	++	++	++	++
2) No dynamic objects or variables, or else online test during their creation	+	++	++	++
3) Initialization of variables	++	++	++	++
4) No multiple use of variable names	+	++	++	++
5) Avoid global variables or justify usage	+	+	++	++
6) Limited use of pointers	o	+	+	++
7) No implicit type conversions	+	++	++	++
8) No hidden data flow or control flow	+	++	++	++
9) No unconditional jumps	++	++	++	++
10) No recursions	+	+	++	++

(1) Functions have several entries and exit points (e.g. 41% of the functions in the object detection module).

(2) Most data structures are allocated dynamically. Since the input data parameters, like the size of the networks and images, are unknown statically, the corresponding data structures are allocated dynamically. As discussed in the previous section, CUDA builds on the use of pointers and dynamic memory.

(3) Using static code analysis tools and compiler options, we have identified several variables as uninitialized.

(4) The use of numerous libraries and namespaces complicates analyzing that all the variables have unique names to minimize programming mistakes. However, commercial custom code checks are developed for this purpose.

(5) We identified the use of global variables (e.g. $\approx$900 in the perception module). Although their use can be eliminated by modifying the application, however, according to the standard, justified usage of such variables may be permitted.

(6) The *limited use of pointers* has been already discussed.

(7) Type conversions have also been discussed in detail.

(8) Hidden data-control flow partially relates to code coverage as already described.

(9) We have observed that AD frameworks use several unconditional jumps. However, by applying minor modifications to the code, they can be eliminated.

(10) We have identified a few uses of recursive functions for well-known purposes such as processing trees. However, in general, the recursive code can be transformed into loop-base iterative code.

Observation 14. *Apollo AD software does not comply with the principles for unit design and implementation.* While code can be modified to cover most of these requirements, some of those requirements, however, require specific techniques as presented in [14] for avoiding the use of pointers in GPU code and/or significant modifications.

4 CONCLUSION

In this paper, we present our experience with the assessment of the safety properties of an advanced, industrial GPU-accelerated AD framework. We have identified complexities and missing features in AD software to adhere to ISO 26262 principles for software (Part 6), and we have proposed different approaches to handle those complexities and missing features. We have highlighted several challenges for certification of GPU code according to the requirements of the ISO 26262 standard. Overall, our analysis contributes to enabling the use of heterogeneous AD software to control high-integrity vehicle functionalities by assessing the gaps between its current implementation and the requirements for its certification.

ACKNOWLEDGMENT

This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773). This work has also been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P and the HiPEAC Network of Excellence. MINECO partially supported Jaume Abella under Ramon y Cajal postdoctoral fellowship (RYC-2013-14717), and Leonidas Kosmidis under Juan de la Cierva-Formación postdoctoral fellowship (FJCI-2017-34095).

REFERENCES

[1] 2014. cuda4cpu. Library and headers to make CUDA codes run seamlessly on CPUs. https://github.com/javier-cabezas/cuda4cpu. (2014).
[2] 2016. Autoware. An open autonomous driving platform. https://github.com/CPFL/Autoware/. (2016).
[3] 2017. Lizard. An extensible cyclomatic complexity analyzer for many imperative programming languages including C/C++. https://github.com/terryyin/lizard. (2017).
[4] 2017. Udacity. An Open Source Self-Driving Car. htps://www.udacity.com/self-driving-car/. (2017).
[5] 2018. Apollo, an open autonomous driving platform. http://apollo.auto/. (2018).
[6] ARM. 2015. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade. https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade.php. (2015).
[7] S.-C. Lin et al. 2018. The Architectural Implications of Autonomous Driving: Constraints and Acceleration. In *ASPLOS*.
[8] MISRA. 2013. *Guidelines for the Use of the C Language in Critical Systems.*
[9] NVIDIA. 2018. CUTLASS 1.1. https://github.com/NVIDIA/cutlass. (2018).
[10] J. Redmon et al. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *CVPR*.
[11] Amnon Shashua. 2018. The Responsibility Sensitive Safety (RSS) Formal Model toward Safety Guarantees for Autonomous Vehicles. https://www.date-conference.com/date18/keynotes. (2018).
[12] Rapita Systems. 2008. *RapiCover. Low-overhead coverage analysis for critical software.* https://www.rapitasystems.com/products/rapicover.
[13] P. Tillet and D. Cox. 2017. Input-aware auto-tuning of compute-bound HPC kernels. In *SC*.
[14] M. M. Trompouki and L. Kosmidis. 2018. Brook Auto: High-Level Certification-Friendly Programming for GPU-powered Automotive Systems. In *DAC 2018*.