Requisitos del software de automoción

Designing RISC-V-based Accelerators for next generation Computers (DRAC)

Código de Proyecto: 001-P-001723

Número de entregable:	E4.1
Nombre de entregable:	Requisitos del software de automoción
Periodo cubierto:	Month 01 to Month 013
Revisión:	01
Fecha límite del entregable:	31 Julio, 2020
Fecha de entrega:	31 Julio, 2020
Fecha de inicio del proyecto:	1 June 2019
Duración:	36 Months
Partner responsible del entregable:	Barcelona Supercomputing Center
Autor (es) del entregable:	Hamid Tabani, Martí Caro, Jaume Abella (BSC)

Este proyecto está financiado por el Fondo Europeo de Desarrollo Regional de la Unión Europea en el marco del programa Operativo FEDER de Cataluña 2014-2020 con una financiación de 2.000.000€ y con el soporte de la Secretaria de Universidades e Investigación

Grado de divulgación		
PU	Público	**X**
CO	Confidencial, solo para miembros del consorcio	

Unió Europea
Fons Europeu
de Desenvolupament Regional

Generalitat de Catalunya
Departament d'Empresa i Coneixement
Secretaria d'Universitats i Recerca

Historia del documento

Versión	Fecha	Descripción/Ccambios	Razones
01	31 Julio, 2020	Entregable enviado	

Index

Executive summary

This report presents the selected automotive software framework intended to be optimized by implementing an approximate hardware accelerator. In particular, our target is developing a hardware accelerator particularly efficient to perform object detection from camera images by trading off accuracy and power consumption. Therefore, here we introduce the particular software framework, the requirements that such a software framework poses on the hardware design, and a quantitative analysis of the sensitivity of such framework to the potential approximate results that the low-power approximate accelerator to be developed may cause.

1 Introduction

The provision of increasingly advanced (and complex) software functionalities, e.g. Autonomous Driving (AD), is a key competitive advantage in every new product in the critical embedded market attracting significant interest from industry [1, 2, 3]. Supporting those advanced software functionalities requires complex software frameworks capable of performing real-time processing of a massive amount of diverse data, in the order of gigabytes of data per second, consistently coming from a score of on-board sensors, like cameras and Light Detection and Ranging devices (LiDARs), just to mention a few. Moreover, those data are inherently involved in the critical AD decision-making process, from perception to motion planning, for which advanced AI algorithms are sought [4, 5].

While those software frameworks are needed for AD, they build upon a variety of existing frameworks and libraries, since many of the processes involved are already developed and used in other domains. Therefore, it is of paramount importance understanding the basis upon which AD software frameworks are built to understand their characteristics and, ultimately, their requirements. This report aims at reviewing some basic concepts on machine learning and neural networks, and existing deep learning frameworks and libraries in Section 2. Once settled the ground on machine learning in general, and deep learning in particular, Section 3 presents the Apollo AD framework, with particular emphasis on the YOLO camera-based object detection framework, which ultimately builds upon the same type of libraries and frameworks described in Section 2. Section 4 presents the automotive requirements emanating from YOLO for the DRAC accelerator. In order to understand the sensitivity of YOLO to input data, operations precision and approximations, Section 5 presents a quantitative study for YOLO where we explore the design space of different parameters of relevance for the subsequent design of a suitable approximate and low power accelerator. Finally, Section 6 draws the main conclusions of this report.

2 From Deep Learning frameworks to automotive requirements

AI and deep learning approaches are at the heart of a variety of domains. Computer vision algorithms such as image classification and object detection are just among many algorithms that are built relying on deep learning approaches since, in many cases, deep learning approaches provide much higher accuracy than any other alternative solution and, in some domains, they are the only existing solution. Given the widespread use of these approaches, the need for an accelerated AI development with a seamless path from prototyping to deployment is inevitable. In this context, deep learning frameworks offer access to building blocks for designing, training, validating, and testing deep learning models, through a high-level and simplified programming interface.

Widely-used deep learning frameworks such as MXNet [6], PyTorch [7], TensorFlow [8], Keras [9], Caffe [10] and others provide implementations for CPUs by using libraries such as openBLAS [11] and ATLAS [12], Intel MKL [13] and implementations for GPUs with libraries such as cuBLAS [14] and cuDNN [15] to deliver high-performance multi-GPU accelerated training. Developers, researchers and data scientists can get easy access to optimized deep learning framework containers from vendors such as NVIDIA. This eliminates the need to manage packages and dependencies or build deep learning frameworks from scratch.

While there are abundant frameworks, mostly building upon Neural Networks (NNs), as well as target platforms and platform-dependent libraries, each domain tends to focus on specific

frameworks, platforms and libraries. However, deep learning frameworks mainly use common and well-known low-level libraries to perform the matrix operations.

Next, we provide a brief background on deep learning frameworks. Then, we analyze the deep learning component ecosystem and provide a top-down structured view. We also present our analysis of the different deep learning frameworks and the libraries that they build upon. From there, we move to the Apollo AD framework (Section 3), which ultimately builds upon those frameworks and libraries. We review the particular characteristics of YOLO approach, on which Apollo's object detection framework for camera images builds. Finally, we detail specific requirements for YOLO. However, as those requirements are common for a larger number of frameworks and libraries, as discussed next, any solution built for YOLO can be easily reused for a variety of domains and frameworks, despite such analysis is beyond the scope of this report.

2.1 Background on Machine Learning

Machine learning algorithms can be categorized into Supervised learning, Unsupervised learning and Reinforcement learning as Figure 2-1 shows. In supervised learning, the process of learning builds upon extracting information from examples (i.e. training dataset), as it is the case for classification and regression to name a few. This is the usual case for many relevant problems such as computer vision, speech recognition, trajectory prediction, etc. For the sake of focusing the discussion, we consider, therefore, supervised learning in the rest of the report. Unsupervised and reinforcement learning, which may lack sufficient training data (unsupervised) or build on a feedback loop with the end user (reinforcement), are not further discussed in this report, although a similar analysis reaching analogous conclusions could be elaborated.

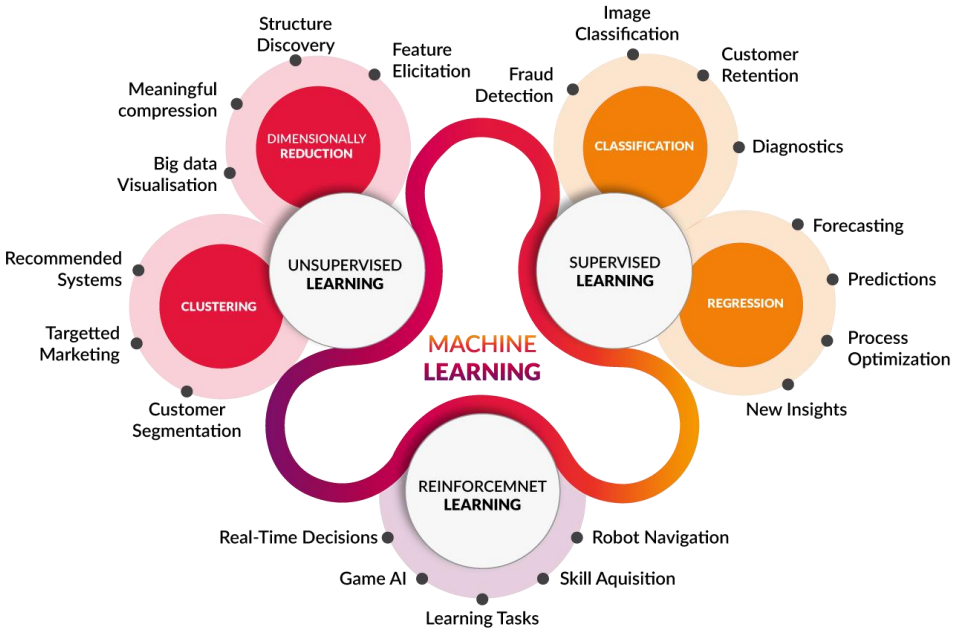

Figure 2-1 Machine learning categories (image source: [23]).

Most of the approaches under supervised learning go through a similar process to be developed and then employed. The machine learning experts decide which approach better suites a specific problem. Accordingly, they design the architecture of the model. For instance, in case of a Deep Neural Network (DNN), the number of layers, type of the layers (e.g., convolution, pooling, etc.), their order and dimensions are specified by the experts. This process may be performed after a design space exploration and trying numerous configurations to find the best, and most appropriate and accurate architecture.

After choosing the most suitable category and designing the high-level architecture, the models have to be trained by using huge amounts of data. Once the model is trained and tuned to provide acceptable accuracy, it can process the so called “unseen data”, which did not exist in the training dataset. In the literature, the first process is called *Training* whereas the latter is referred to as *Inference*. In this paper, we focus on deep learning algorithms that are trained with a large dataset and then used to infer the unseen data.

The development of deep learning models is usually done using a Deep Learning Framework. A deep learning framework is an interface, library or tool that allows developers to build their deep learning models more easily and quickly, without the need to get into the details of underlying algorithms. Such frameworks provide a clear and concise way for defining models using a collection of pre-built and optimized components and libraries. There are several deep learning frameworks each with different features and advantages. For instance, Tensorflow, Caffe, Pytorch, Theano and Keras are common, widely-used and well-known frameworks.

Usually any standard model architecture can be implemented using these frameworks. The frameworks also provide implementation for different hardware platforms such as CPU, GPU and recently, specialized accelerators. However, how these frameworks relate to specific low-level libraries and platforms is unknown to most users, which typically have some partial knowledge of the ecosystem (e.g. one or two frameworks and a subset of the libraries).

2.2 Deep Learning ecosystem structured view

In this section, we provide a structured top-down view of the components for a deep learning-based solution, spanning from the top-level NNs regarded as appropriate for the target application by end users, and reaching down to linear algebra operations that ultimately implement those applications. Note that our analysis aims at illustrating the different existing layers, but not being exhaustive in any of those since the number and type of frameworks, platforms and libraries is too large to be exhaustively covered in this report.

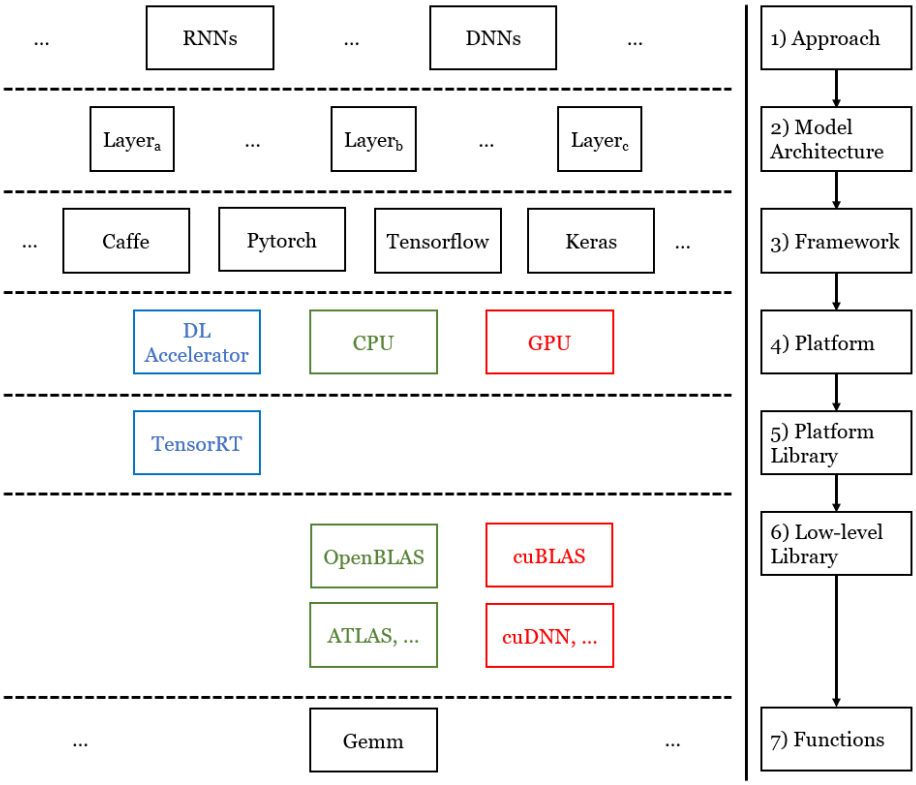

Figure 2-2 – Top-down view of the deep learning algorithms.

As Figure 2-2 shows, at the highest level, we have different Machine learning or deep learning models that are composed of a series of units. For instance, a convolutional neural network (CNN) is composed of several layers such as convolution, pooling, etc. The architecture of the model and the approach itself, levels 1 and 2 in Figure 2-2, are designed and specified by the deep learning experts. In this Figure, we correlate low-level libraries that are developed for a specific hardware platform (e.g., CPU or GPU) using the same colors.

2.2.1 The framework

The choice of the deep learning framework to use is usually independent from the deep learning approach itself. Different deep learning frameworks provide different advantages such as flexibility, or simpler interfaces for instance. The users decide what framework fits better their needs for the development of their model. As an illustrative example, Keras [9] and TensorFlow [8] are both very well-known frameworks. Keras was developed to be more user-friendly with high-level APIs and a modular structure. However, sometimes the user needs to define something new such as a cost function, a metric, a layer, etc. Although Keras has been designed to be flexible, low-level libraries of TensorFlow provide further flexibility.

Most of the deep learning frameworks use similar libraries to perform the low-level operations, and they mainly differ at higher levels. For instance, unlike TensorFlow, the PyTorch framework uses a dynamically updated graph, which allows the user to make changes to the architecture in the process. Still, TensorFlow has a level of abstraction similar to that of MXNet, Theano, and PyTorch. In this level, mathematical operations such as Generalized Matrix-Matrix multiplication and Neural Network primitives are implemented. Keras, instead, works at a

higher abstraction level where the lower level primitives are used to implement neural network abstractions such as various layers and models. In general, other useful APIs such as model saving and model training are implemented at this level.

2.2.2 The hardware platform

Different frameworks offer implementations for different hardware platforms, for both training and inference. For instance, TensorFlow provides implementations for both CPU and GPU, and the user can choose the target hardware platform. The framework is then installed and run on the chosen platform accordingly. In the case of some specialized hardware platforms such as the NVIDIA Deep Learning Accelerator (NVDLA), additional libraries are required to launch and manage the operations on the platform. In the case of NVDLA, NVIDIA's TensorRT [16] is used to launch and manage the workloads on the NVDLA.

2.2.3 Low-level libraries

Once the hardware platform is selected, a platform-dependent optimized library is used, which includes the majority of the required low-level functions implemented and optimized for that specific target platform. For instance, ATLAS [12] and openBLAS [11] are two well-known libraries for CPUs, as well as cuBLAS [14] and cuDNN [15] that are specific for NVIDIA GPUs.

At this low level, different operations are performed by calling the corresponding functions of a library providing their required parameters. Such compute-intensive operations are linear algebra operations (mostly matrix operations) whose platform-dependent implementations are provided by the corresponding libraries. Normally, these implementations have been optimized with average performance in mind.

2.3 A deeper view of the low-level libraries

We have analyzed some of the most popular deep learning libraries such as TensorFlow [8], Keras [9], MXNet [6], Caffe [10] and PyTorch [7], as illustrative examples of this type of frameworks, considering both CPU and GPU target platforms. The result of this analysis shows that their computations are translated into operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix operations with different dimensions. Deep learning frameworks do not implement those operations directly but, instead, build upon low-level libraries providing efficient implementations (mostly targeting optimized average performance) for different CPU and GPU target platforms. In particular, the frameworks considered in our analysis build upon one or several of the following low-level libraries (e.g., TensorFlow uses cuBLAS, TensorRT and cuDNN libraries):

- BLAS [17] (Basic Linear Algebra Subprograms) is a specification that prescribes a set of low-level routines for performing common linear algebra operations such as vector addition, scalar multiplication, dot products, linear combinations, and matrix multiplication.
- OpenBLAS [11] is an open-source implementation of the BLAS API with many hand-crafted optimizations for specific processor types.
- ATLAS [12] (Automatically Tuned Linear Algebra Software) is a library for algebra targeting high-performance computing platforms.

- Intel MKL [13] (Intel Math Kernel) Library is a library of optimized math routines for science, engineering, and financial applications. Core math functions include BLAS, LAPACK, ScaLAPACK, sparse solvers, fast Fourier transforms, and vector math.
- NVIDIA's cuBLAS [14] library is a fast GPU-accelerated implementation of the standard basic linear algebra subroutines (BLAS).
- cuDNN [15] (NVIDIA CUDA Deep Neural Network) library is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers, which ultimately build upon matrix operations.
- TensorRT [16] is a library for high-performance deep learning inference which includes a deep learning inference optimizer and runtime that delivers low-latency and high-throughput for deep learning inference applications, also building mostly on matrix operations.

2.3.1 Discussion

So far, deep learning approaches have been mostly used for applications where the main concern is performance and energy, with both objectives being achieved with almost identical optimizations. This relates to the fact that, performing the same computations in shorter time, as needed for average performance reasons, often leads to lower energy consumption due to the lower active time of the platform.

However, the increasing number of applications of deep learning approaches in different domains brings additional non-functional metrics to be optimized. For instance, autonomous navigation in cars, planes, satellites and drones has some degree of criticality since malfunctioning may cause fatalities or large economical losses. Moreover, the performance and power requirements of those systems may not need to be optimized but kept within specific budgets. As an example, we may need to keep object detection working within limited time and power, but maximizing observability and controllability for verification and validation purposes.

It is expected that those new goals together with new platforms needed in those domains, pose the need for new low-level libraries targeting, for instance, security or fault-tolerance for specific embedded platforms with limited power envelopes and specific deadlines. Thus, deep learning frameworks will have to further consider additional low-level libraries and the overall ecosystem is expected to grow.

In this context, it is particularly important reusing efforts across domains and across platforms whenever possible. Low-level libraries build on linear algebra operations such as matrix multiplication, and hence, frameworks and applications above also end up inheriting the features of the linear algebra operations used in the lowest levels of the stack. Thus, the key to meet the requirements of applications ultimately lies on the ability to optimize specific matrix operations and other linear algebra operations for the different needs of applications, and let frameworks in-between act as interfaces to ease programmability while using highly optimized operations from the low-level libraries. It is, therefore, of paramount importance devoting efforts to the development and optimization of linear algebra operations for different goals such as average performance, worst-case execution time, energy consumption, security, fault-tolerance, testability, and maintainability among others.

In the context of DRAC, this translates into the development of the appropriate hardware support, in the form of an accelerator, able to deliver the high-level functionalities demanded by applications building on those frameworks, but leveraging efficiency at the implementation level. In particular, the goal is leveraging approximate computing, low-power operation and fault tolerance at circuit design to deliver semantically correct predictions at application level, thus with lower power and area cost than traditional solutions such as NVDLA, GPUs and CPUs.

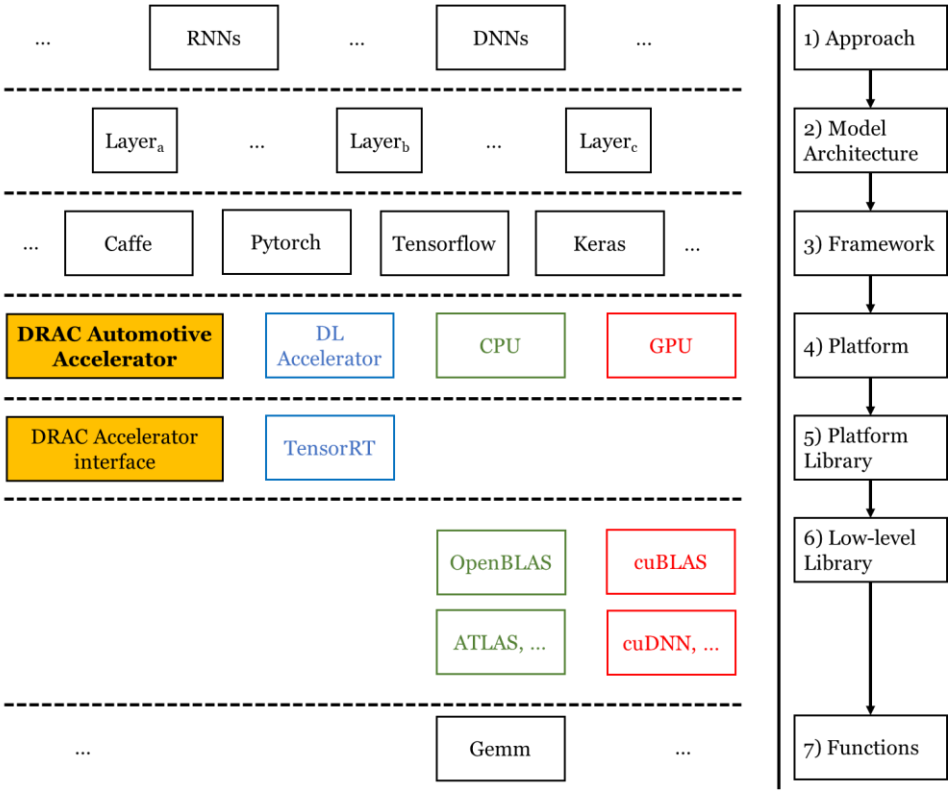

Figure 2-3 – Top-down view of the deep learning algorithms including DRAC.

Figure 2-3 contextualizes DRAC automotive accelerator in the scope of the different deep learning frameworks. As shown, it provides a new platform, the DRAC automotive accelerator, which will come along a simple software accelerator interface to transfer input/output data back and forth to the accelerator and to use it by programming it as needed. Therefore, YOLO object detection, part of Apollo, would build on the DRAC accelerator interface to use DRAC accelerator instead of using other low-level libraries such as cuBLAS, ATLAS, etc. However, while the scope of this work is the automotive domain, the way deep learning algorithms are implemented will allow DRAC automotive accelerator be used for other applications and domains.

3 Apollo AD framework

Apollo [18] is an industrial-quality AD software framework with over 120 industrial partners, most of them top-tier AI companies and car manufacturers. Apollo has been already deployed on a variety of prototype vehicles (including autonomous trucks and robotaxies) and supports

state-of-the-art hardware such as the latest LiDARs and cameras, from Velodyne and other vendors, as well as GPU acceleration.

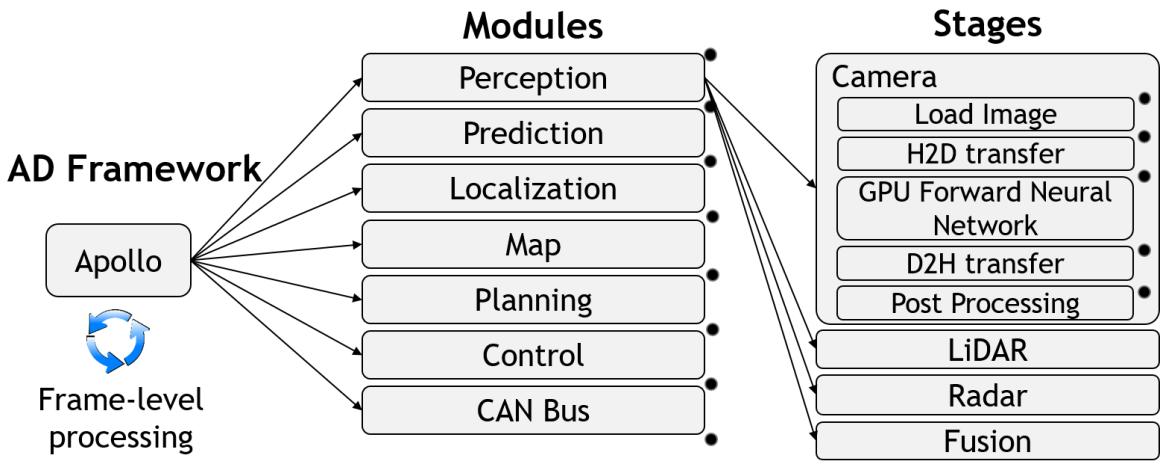

Figure 3-1 – Apollo AD system pipeline (source image: [24]).

Apollo is structured as a set of processes that are meant to execute on a recurring basis. The execution model is organized into stages where each stage is allocated to a specific functional step in each module. Figure 3-1 presents the main software modules in Apollo:

- **Perception** identifies the area surrounding the autonomous vehicle by detecting objects, obstacles, and, traffic signs. It is considered as the most critical and complex module of an AD system. It fuses the output of several types of sensors such as LiDAR, radar, and camera to improve its accuracy.
- **Localization** estimates where the autonomous vehicle is located using various information sources such as GPS, LiDAR and IMU. State-of-the-art localization algorithms, including the one in Apollo, are capable of localizing the position of the vehicle at centimeter-level accuracy.
- **Prediction** anticipates the future motion trajectories of the perceived obstacles.
- **Planning** plans the spatio-temporal trajectory for the autonomous vehicle to take.
- **Control** executes the planned spatio-temporal trajectory by generating control commands such as acceleration, braking, and steering.
- **CANBus** is the interface that passes control commands to the vehicle hardware. It also passes chassis information to the software system.
- **Map** acts like a library. Instead of publishing/subscribing messages, it works as a query engine support that provides ad-hoc structured information regarding the roads.

One of the key components of the AD framework is the perception module, given (i) its complexity, instrumental to deal with inputs from various components (e.g., video cameras, short- and long-range radars and laser sensors), and (ii) its long execution time that represents a large fraction of the overall execution time of the AD framework. For this reason, we focus on this module to perform our requirements analysis, and this will be the focus of the DRAC accelerator, which aims at accelerating this compute-intensive module.

3.1 You Only Look Once (YOLO) object detection as part of Apollo

Apollo uses a variant of YOLO [19] for camera-based object detection, as the one of the main parts of the perception module. YOLO (You Only Look Once) is an award-winning, widely-used object detection system. Its most computationally-intensive function is a Convolutional Neural Network inference algorithm. While YOLO can be implemented for multiple hardware targets (e.g. CPU, GPU, accelerators), Apollo relies on GPUs to accelerate and execute YOLO kernels. The main stages of the YOLO object detection module (when running on a GPU) are shown in Figure 3-1 as camera macro-stages. Every second, each camera captures multiple frames, and the object detector processes them on a frame-by-frame basis:

- For every frame the detector first loads the frame (in an appropriate format) into the main memory.
- Then, all the data is moved to the GPU memory (host-to-device transfer).
- Once the data is stored in the GPU memory, GPU kernels are launched to perform the neural network evaluation.
- The result of the operations is transferred back to the main memory (device-to-host transfer).
- As the last step, some post-processing operations are performed to finalize and publish the result of the detection.

Note that camera-based object detection is part of the perception module, which is fused with the LiDAR-based object detection and Radar processes.

Since the goal of DRAC is designing an accelerator, rather than using the GPU implementation of YOLO, we will use instead a CPU-based version, which will allow us to profile and modify the code as needed to collect information on the sensitivity of the object detection process to the precision of the input data and data manipulation operations (e.g., additions and multiplications).

The internal structure of the computation intensive part of YOLO is depicted in Figure 3-2. YOLO performs object detection building on a Deep Neural Network (DNN) consisting of 100+ layers, out of which, the most compute-intensive ones are the 53 convolutional layers. The DNN model is trained for the inference of the input data, which are in form of images captured by the autonomous car's camera(s). While the actual result of the calculation depends on the particular image processed and the trained model, computation behavior, in terms of performance and power, is almost independent of the actual image being processed (other than the image size) and the size of the model. Thus, one could replace the input image by any other image of the same dimensions, and also replace the weights of each layer, and the computations and memory accesses performed would remain unaltered. However, the results in terms of objects detected and the confidence level in terms of probability for each detection would obviously change.

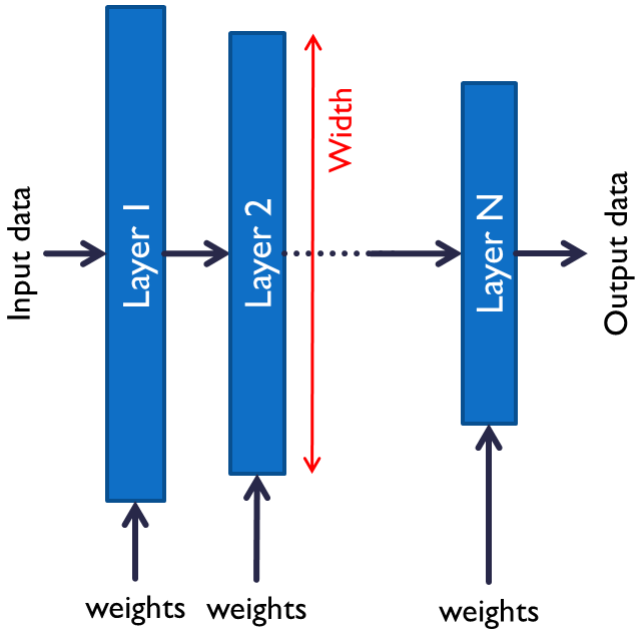

Figure 3-2 – Structure of YOLO DNN.

4 DRAC accelerator automotive requirements

The requirements that object detection (and in our particular case, YOLO) pose on the DRAC automotive accelerator can be summarized into a single high-level requirement:

- REQ1: Detect and classify objects in an image with sufficient accuracy (correct classification) and precision (location of the objects in the image) w.r.t. the baseline algorithm to enable autonomous driving operation building on top of this detection and classification process.

In order to provide a precise set of requirements that can be quantified and used as success criteria for the design of the accelerator, we decompose REQ1 into multiple sub requirements linked to an actual high-precision implementation of YOLO (e.g. on a CPU or GPU building on 32-bit IEEE754 floating point precision). For convenience of the discussion, we refer to such reference implementation as REF, whereas we refer to the DRAC automotive accelerator as DRACacc. New, and measurable, requirements are as follows:

- REQ1.1: DRACacc detects the same objects (and of the same type) as REF with some high degree of accuracy (e.g. >90%).

  This requirement can be quantitatively assessed as follows:

$$ObjectMatching = \frac{TotalObjectsREF - MissClassifiedObjectsDRACacc}{TotalObjectsREF}$$

  Where *TotalObjectsREF* stands for the total number of objects detected by REF, and *MissClassifiedObjectsDRACacc* stands for those objects that the DRACacc classifies differently to REF, e.g. due to lack of detection whereas REF detects them, due to detecting them whereas REF does not detect them, or due to classifying them differently to REF. An important decision that will need being taken will be identifying a

relevant set of images against which evaluate DRACacc and REF to obtain meaningful results.

Note that, with this metric, it is possible that correct DRACacc detections that were wrong for REF are counted as mistakes. However, due to the approach followed to design the DRACacc, its ability to detect objects is expected to be lower than that of REF, and hence, any such improvement w.r.t REF in terms of detections occurs solely by chance, not because of the specific features of the accelerator.

- REQ1.2: For objects detected by REF and DRACacc, and classified identically, the detection confidence discrepancy across both implementations is below a given bound (e.g., 5% or 10%).

  This requirement can be quantitatively assessed as follows for a given object:

$$DetectConfidDifference = |DetectConfidREF - DetectConfidDRACacc|$$

  Where *DetectConfREF* and *DetectConfDRACacc* stand for the detection confidence of REF and DRACacc respectively.

  Combining such metric for different objects can be done with different central behavior or error estimation metrics such as average, median, mean square error or the like.

Another high-level requirement relates to the ability to detect objects in images at a sufficiently high rate and within affordable power envelopes:

- REQ2: Object detection must be fast and operate at low power.

As before, we need to break down this requirement into finer-grain and measurable requirements with appropriate success criteria for their assessment:

- REQ2.1: DRACacc must detect objects in an image at a rate sufficient to process all images received from a single camera. Therefore, if the system requires to process FPS frames every second, then the maximum processing time for each single frame should not exceed $\frac{1}{FPS}$. For instance, if the camera works at 25 frames per second, each frame (image) should be processed in up to 40ms (preferably less).

  Note that image resolution should be high enough to allow meaningful object detection. In particular, we will stick to the resolution of the default input sets for YOLO for reference [19].

  For the sake of concreteness, we set the requirement at being able to process a 608x608 pixels image every 40ms.

- REQ2.2: DRACacc energy consumption per image must be lower than that of comparable counterparts (e.g., CPUs, GPUs, and other accelerators) when implemented in analogous technology. The specific values to use as reference will depend on the actual hardware in the market, which will likely use different technology than that used by DRACacc. Hence, we will set reasonable extrapolation methodologies for a fair comparison.

  Note that this is the main optimization parameter that must be pushed to the limit while meeting all the other requirements.

In summary, DRACacc requirements relate to the accuracy and precision in the detection process, as well as to the performance and power efficiency of the accelerator.

5 Quantitative study

In order to evaluate the sensitivity of YOLO to varying precision or potential faults affecting the quality of the input data in the different layers of the neural network, we have conducted a set of experiments. Since the DRACacc is still to be devised and designed, experiments have aimed at evaluating REQ1.1 and REQ1.2 under a set of setups where the default implementation of YOLO is modified to use 16-bit floating point numbers (FP16 for short) instead of the default FP32, and then replacing fully-precise mantissa operations by approximate ones, thus further reducing the accuracy, but facilitating the decrease of the hardware cost, with direct impact in performance and power, thus contributing to REQ2.1 and REQ2.2, although such contribution is not explicitly quantified in this report.

5.1 Experimental framework

The reference configuration of YOLO uses FP32 fully precise logic. We refer to such configuration as YOLO32full. We have generated a new version using FP16 fully precise, YOLO16full, by replacing FP32 input data, input parameters, and arithmetic operations by FP16 counterparts. In particular, we use the FP16 library "Branch-free implementation of half-precision (16 bit) floating point" [20]. This library implements the FP16 operations: ADD, SUB and MUL in C. We have extended this library to support DIV, SQRT and EXP operations which are used in YOLO.

We have also further evaluated the impact of approximate computing units by replacing mantissa operations in the FP16 library by approximate multiplications [21]. This paper provides a method to build 2x2 approximate multipliers and also provides a method to build larger multipliers using blocks of 2x2. Using this method, given 2 operands (A and X), split into their half highest bits (AH and XH respectively) and their half lowest bits (AL and XL respectively), we have implemented an approximate multiplier where the blocks $ALxXL$, $AHxXL$ and $ALxXH$ are approximate, and the block $AHxXH$ is accurate since this block has a higher impact on the result of the multiplication. We refer to such approximate FP16 implementation to as YOLO16approx.

Note that YOLO32full corresponds to REF setup, whereas YOLO16full and YOLO16approx could be different incarnations of DRACacc.

Our evaluation is conducted on a set of reference images of the COCO dataset [22]. This dataset contains 5000 images, and our evaluation is done with a subset of 25 images due to the large computation time of implementing FP operations and bit-level approximate operations in software. We plan to extend this study to a larger number of images in the future.

In order to provide results, we use the metrics as defined in REQ1.1 and REQ1.2. For the detection confidence difference, we use the average to combine the values obtained across multiple images. Moreover, we use boxplots in some cases to show the actual distributions rather than only a single number.

5.2 Results

5.2.1 First comparison

We first evaluate and compare YOLO32full vs YOLO16full. Figure 5-1 shows the results in terms of object detection confidence for a dataset with 25 images when using YOLO32full. In this evaluation, we consider a detection threshold of 50% (0.5 probability), meaning that objects detected with lower confidence are simply dismissed and only those above 50% will be considered. For the sake of comparison, we also include those objects with a probability below 50% that YOLO16full detects instead with a probability above 50% in Figure 5-2, so that both plots include the same objects. Figure 5-1 shows results with boxplots, thus indicating maximum and minimum values with the whiskers, and quartile 3 (Q3, 75%) and quartile 1 (Q1, 25%) with the upper and lower ends of the bar. For some objects, there is a single detection, so maximum, minimum, Q3 and Q1 collapse into a single value, as shown. We use different colors for each object type (see x-axis) to ease the reading.

Analogous boxplots are shown for YOLO16full in Figure 5-2, also including those objects with probability below 50% for YOLO16full but, instead, above 50% for YOLO32full. Visual inspection reveals that, in general, decreasing precision from FP32 to FP16 causes low impact in the general trends for object detection. Relatively large variations for some object types often relate to differences in the detection of a single object, which may cause large variations in the boxplot when the number of objects of that category is tiny (e.g. 2 or 3).

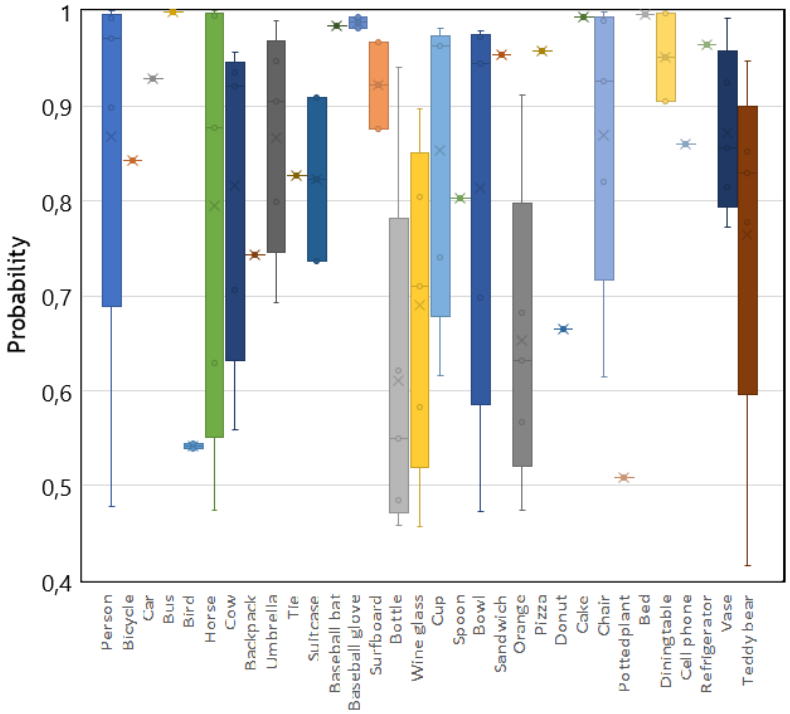

Figure 5-1 – Confidence for object detection in the form of probability for different object types when using YOLO32full.

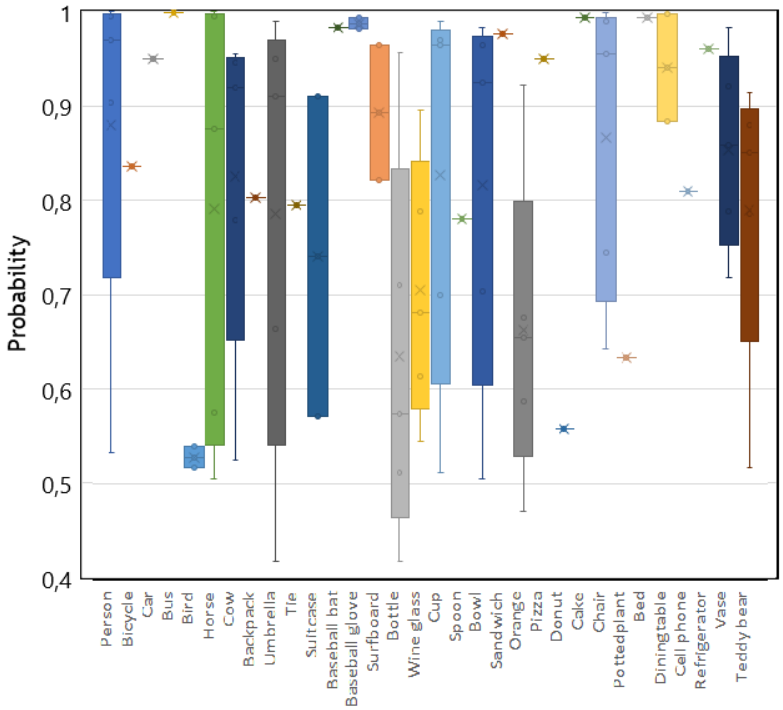

Figure 5-2 – Confidence for object detection in the form of probability for different object types when using YOLO16full.

Note that in these experiments, there is the possibility of retraining the DNN model to recover most of the accuracy loss.

We have quantified differences between YOLO32full and YOLO16full with the formulas described in REQ1.1 and REQ1.2. In particular, in terms of object detections, YOLO32full detects 142 objects with probability above 50%. YOLO16full, instead, detects 148 objects that include 140 of those detected by YOLO32full and 8 new objects. Thus, the total discrepancy between both implementations is of 10 objects.

$$ObjectMatching_{32full-16full} = \frac{142 - 10}{142} = 0.930$$

As shown, this corresponds to a 93.0% object matching probability. However, if we dig further into the results, we realize that all misclassifications still are detected with a probability above 40% - so close to the 50% threshold – by the other implementation. This is the case for the 8 new objects detected by YOLO16full, whose confidence (probability) for YOLO32full is above 40% in all cases, and above 45% for all but 1 case (see Figure 5-1). This is also the case for the 2 objects detected by YOLO32full but not by YOLO16full, whose confidence for YOLO16full is also in the range 40%-50%. Therefore, in fact, it is quite common that classification probabilities are within a relatively narrow interval for misclassified objects (e.g. within a 10% interval).

We have further quantified the impact of decreasing prediction with the detection confidence difference, as described in REQ1.2. By averaging those absolute differences, we obtain that prediction confidence differs by 2.7% on average, thus reflecting that the impact of decreasing precision is very low.

5.2.2 Second comparison

We have also analyzed the detection degradation when moving from YOLO16full to YOLO16approx. In this particular evaluation, only approximate multiplications are used, while the rest of operations (mostly additions) use full 16-bit precision for YOLO16approx. Boxplots are shown in Figure 5-3 for YOLO16approx for the same data set with 25 images. Visual inspection reveals that object detections change negligibly w.r.t. YOLO32full and YOLO16full, thus suggesting that the use of approximate logic has not a meaningful impact in the process of object detection, which is inherently approximate in nature.

If we further dig into the results, we observe that YOLO16approx detects 139 objects, whereas YOLO32full detects 142. In particular, YOLO16approx detects 136 out of those 142 objects, and 3 new objects. Thus, the discrepancy is of only 9 objects. This leads to the following object matching values:

$$ObjectMatching_{32full-16approx} = \frac{142 - 9}{142} = 0.937$$

As shown, matching probability is better when using approximate operations (YOLO16approx) than fully precise ones (YOLO16full). Of course, this is an unexpected result that can only be attributed to chance. However, it already indicates that approximate logic does not degrade object detection accuracy meaningfully. Of course, a deeper evaluation is convenient using even larger data sets and further approximate operations.

Note also that, in the case of misclassified objects, they are always detected with a probability above 40% by the other implementations, thus indicating that actual confidence variations are low, even if they are around the 50% threshold.

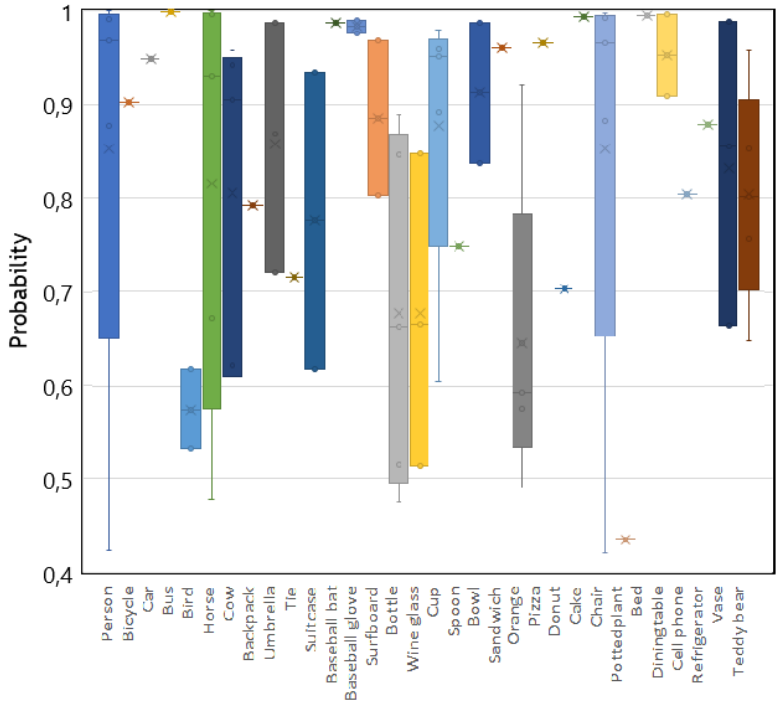

Figure 5-3 – Confidence for object detection in the form of probability for different object types when using YOLO16approx.

Finally, we have also quantified the detection confidence difference of YOLO16approx w.r.t YOLO32full, which is 3.1% on average respectively, thus slightly worse than that of YOLO16full (2.7%). Again, results indicate that decreasing precision and introducing approximation has negligible impact on predictions since YOLO16approx is slightly worse than YOLO16full.

For the sake of completeness, we provide a table with the complete object count detected for each category for the 3 configurations. Discrepancies w.r.t. YOLO32full are marked with yellow background.

OBJECT TYPE	YOLO32full	YOLO16full	YOLO16approx
Person	48	49	48
Bicycle	1	1	1
Car	1	1	1
Bus	1	1	1
Bird	2	2	2
Horse	7	8	6
Cow	11	11	11
Backpack	1	1	1
Umbrella	3	2	3
Tie	1	1	1
Suitcase	2	2	2
Baseball bat	1	1	1
Baseball glove	2	2	2
Surfboard	2	2	2
Bottle	8	9	7
Wine glass	2	3	3
Cup	6	6	6
Spoon	1	1	1
Bowl	3	4	3
Sandwich	1	1	1
Orange	9	10	9
Pizza	1	1	1
Donut	1	1	1
Cake	1	1	1
Chair	11	11	10
Pottedplant	1	1	0
Bed	1	1	1
Dining table	2	2	2
Cell phone	1	1	1
Refrigerator	1	1	1
Vase	3	3	3
Teddy bear	6	7	6
TOTAL	**142**	**148**	**139**

Table 5-1 – Summary of object counts per configuration.

6 Summary

Deep learning is at the heart of many applications across multiple domains, thus making that new designs, either hardware or software, can have a broad application. In our particular case, our analysis of the existing frameworks and libraries reveals that, despite the target of our work

is building an automotive accelerator, such accelerator should be usable in a plethora of other domains.

Focusing on the automotive domain, we have identified that camera-based object detection is a relevant target to design an accelerator due to compute-intensive nature of the problem. We have shown that such object detection process, implemented as part of the popular (and industrial) YOLO tool, builds upon neural networks, which ultimately build upon matrix multiplications, being floating point multiplications and additions the main operations used.

We have also identified the particular requirements that the automotive domain poses on an accelerator intended to be used for object detection. Those requirements can be summarized into the efficacy of the accelerator to properly detect and classify objects, and its performance and power characteristics, that must adhere to some specific constraints.

Finally, we have assessed quantitatively to what extent predictions can remain accurate if lower precision numbers and approximate operations are used, thus replacing slower and more power-hungry higher precision and fully accurate operations. Our preliminary set of results shows that the intrinsic approximate nature of object detection is highly tolerant to small variations caused by lower precision (e.g. FP16 instead of FP32) and approximate operations (e.g. approximate multiplications). Therefore, these observations open the door to devising an automotive accelerator in DRAC building on those key observations.

7 References

[1] Ford. Media Center Release. https://media.ford.com/content/fordmedia/fna/us/en/news/2017/02/10/ford-invests-in-argo-ai-new-artificial-intelligence-company.html, 2017.

[2] Detlev Mohr et al. The road to 2020 and beyond: What's driving the global automotive industry, 2013.

[3] Toyota Motor Corporation. Toyota News Release. https://corporatenews.pressroom.toyota.com/releases/toyota+establish+artificial+intelligence+research+development+company.htm, 2015

[4] Brian Paden, Michal Cap, Sze Zheng Yong, Dmitry S. Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. IEEE Trans. Intelligent Vehicles, 1(1):33–55, 2016.

[5] Cuong Cao Pham and Jae Wook Jeon. Robust object proposals re-ranking for object detection in autonomous driving using convolutional neural networks. Signal Process. Image Commun., 53:110–122, 2017.

[6] T. Chen et al., "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," arXiv preprint arXiv:1512.01274, 2015.

[7] A. Paszke et al., "Automatic differentiation in pytorch," 2017.

[8] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[9] F. Chollet et al., "Keras," https://keras.io, 2015

[10] Y. Jia et al., "Caffe: Convolutional architecture for fast feature embedding," in Proceedings of the 22nd ACM international conference on Multimedia, 2014.

[11] "An optimized BLAS library (OpenBLAS)," http://www.openblas.net/

[12] "Automatically Tuned Linear Algebra Software (ATLAS)," http://math-atlas.sourceforge.net/

[13] "Intel, Math Kernel Library," https://software.intel.com/en-us/intel-mkl.

[14] "cuBLAS," http://docs.nvidia.com/cuda/cublas/.

[15] S. Chetlur et al., "cudnn: Efficient primitives for deep learning," arXiv preprint arXiv:1410.0759, 2014.

[16] "TensorRT: A platform for high-performance deep learning inference." [Online]. Available: https://developer.nvidia.com/tensorrt

[17] L. Blackford et al., "An updated set of basic linear algebra subprograms (blas)," ACM Transactions on Mathematical Software, vol. 28, no. 2, pp. 135–151, 2002.

[18] Apollo, an open autonomous driving platform. http://apollo.auto/, 2019

[19] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, 2017, pp. 6517-6525, doi: 10.1109/CVPR.2017.690.

[20]https://cellperformance.beyond3d.com/articles/2006/07/branchfree_implementation_of_h_1.html (accessed July 2020)

[21] P. Kulkarni, P. Gupta and M. Ercegovac, "Trading Accuracy for Power with an Underdesigned Multiplier Architecture," 2011 24th Internatioal Conference on VLSI Design, Chennai, 2011, pp. 346-351, doi: 10.1109/VLSID.2011.51.

[22] https://cocodataset.org/ (accessed July 2020)

[23] http://www.cognub.com/index.php/cognitive-platform/ (accessed July 2020)

[24] M. Alcon, H. Tabani, L. Kosmidis, E. Mezzetti, J. Abella and F. J. Cazorla, "Timing of Autonomous Driving Software: Problem Analysis and Prospects for Future Solutions," 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Sydney, Australia, 2020, pp. 267-280, doi: 10.1109/RTAS48715.2020.000-1.