Hardware Acceleration of Deep Neural Networks for Autonomous Driving on FPGA-based SoC

Gerlando Sciangula[*], Francesco Restuccia[†], Alessandro Biondi[*], and Giorgio Buttazzo[*]
[*]TeCIP Institute, Scuola Superiore Sant'Anna, Pisa, Italy
[†]University of California, San Diego, United States

***Abstract*—In the last decade, enormous and renewed attention to Artificial Intelligence has emerged thanks to Deep Neural Networks (DNNs), which can achieve high performance in performing specific tasks at the cost of a high computational complexity. GPUs are commonly used to accelerate DNNs, but generally determine a very high power consumption and poor time predictability. For this reason, GPUs are becoming less attractive for resource-constrained, real-time systems, while there is a growing demand for specialized hardware accelerators that can better fit the requirements of embedded systems. Following this trend, this paper focuses on hardware acceleration for the DNNs used by Baidu Apollo, an open-source autonomous driving framework. As an experience report of performing R&D with industrial technologies, we discuss challenges faced in shifting from GPU-based to FPGA-based DNN acceleration when performed using the DPU core by Xilinx deployed on an Ultrascale+ SoC FPGA platform. Furthermore, it shows pros and cons of today's hardware acceleration tools. Experimental evaluations were conducted to evaluate the performance of FPGA-accelerated DNNs in terms of accuracy, throughput, and power consumption, in comparison with those achieved on embedded GPUs.**

I. Introduction

During the last decade, renewed attention to Artificial Intelligence has consistently emerged thanks to the impressive performance achieved by *Deep Neural Networks (DNNs)* in several applications, including computer vision, robotics, and autonomous driving, reaching and overcoming human accuracy in some specific tasks, [1], [2]. This is mainly due to the capability of DNNs of extracting high-level features from raw data using learning algorithms applied to a large amount of data, which allow achieving an effective representation of the input space. This comes at the cost of high computation complexity and memory requirement, which challenge computing platforms to achieve real-time performance combined with energy efficiency [3]. To obtain high throughput, *Graphics Processing Units (GPUs)* are typically used as hardware accelerators (HAs) to accelerate DNNs during training and inference phases. Nevertheless, their power consumption (in the order of hundreds of watts) often results to be prohibitive [4] for resource-constrained embedded systems. Moreover, GPUs exhibit poor time execution predictability, which is instead a critical requirement in safety-critical systems, such as autonomous vehicles.

A promising alternative to GPUs are FPGA SoCs. Such platforms enable the deployment of problem-specific HAs meeting the computational and memory requirements of complex DNNs while maintaining a contained power consumption [5]. FPGA-based platforms proved their capabilities in power efficiency, combined with low latency, and timing predictability in different deep learning (DL) applications [6], [7]. Although great performance and power efficiency can be achieved by customizing the FPGA hardware of a heterogeneous embedded platform for DNN inference [8], [9], significant efforts and expertise are required to enable efficient acceleration, often leading to long development times. DL algorithms are having a big impact in the autonomous driving field and currently represent its core technology for implementing perception tasks. Autonomous vehicle technologies are progressively migrating from research laboratories to public roads, promising to decrease accidents and traffic congestion, as well as taking automated mobility to the next level.

Contribution. This paper focuses on Baidu Apollo, a popular open-source autonomous driving framework [10], which includes a set of Caffe-based [11] DNNs running on Nvidia GPUs. We discuss all the challenges that we addressed to accelerate the Apollo DNNs inference on a Xilinx SoC FPGA embedded platform (*Zynq Ultrascale+ MPSoC ZCU102*), hence offering a valid alternative to GPU acceleration. This required analyzing in details the design of the Apollo DNNs and their role in the Apollo Perception System. The *Vitis AI* framework by Xilinx was adopted to accelerate the Apollo DNNs. The paper individually addresses the acceleration of each DNN, highlithing the issues that were encountered and discussing the adopted solutions against other alternative ones. It then provides an experimental evaluation of the accelerated DNN performance that focuses on DNN accuracy, throughput, and power consumption comparing FPGA-based against GPU-based acceleration. This work represents a complex engineering effort to accelerate a complete and mature perception system on an FPGA-enabled embedded device, also showing the pros and cons of today's acceleration tools.

II. Related work

DNN optimization for acceleration. The deployment of DNNs on edge devices requires dealing with compression techniques to address the typically limited resources available on embedded devices. Several methods have been proposed to compress DNNs. The *compact model* [12] technique aims at designing smaller base models still achieving acceptable application accuracy. *Data quantization* [13] aims at reducing the

number of bits with which weights and activations are represented. *Network sparsification* [14] reduces the complexity of the DNN by compressing the amount of connections/neurons. Regarding data quantization techniques, the numerical precision with which weights are stored and computed strongly impacts the accuracy and efficiency of the network. Typically, the training step is performed leveraging high numerical precision representations: 32-bit floating-point (FP32). Nevertheless, quantization to integer is crucial for obtaining high-performance and power efficiency in the inference phase: FP operations are computationally demanding and require plenty of energy [15]. Lower numerical precision representations are reasonably effective during inference, e.g., 8-bit integer (INT8) [16]. Various quantization techniques have emerged. They can be classified as belonging to **1)** post-training quantization *(PTQ)* [17] or **2)** quantization-aware training *(QAT)* [18] techniques. PTQ is performed after a high-precision model has been trained. Firstly, a floating-point model has to be evaluated using a small dataset representative of the task's real input data. Statistics about the interlayer activation distributions are collected. As a final step, the quantization scales of the model's activation tensors are determined using optimization objectives. This process is well known as *calibration*, and the representative dataset used is the so-called *calibration dataset*. Sometimes PTQ is not able to achieve acceptable task accuracy. This is when you might consider using QAT. QAT can improve the accuracy of quantized models including the quantization error in the training phase. It enables the network to adapt to the quantized weights and activations.

FPGA acceleration of DNNs. Multiply-and-accumulate are DNNs fundamental operations, easily parallelized. To achieve high performance, highly parallel computing paradigms are used, including both *temporal* and *spatial* architectures [5]. The temporal one appears mostly in CPUs or GPUs, and employ a variety of techniques to improve parallelism such as vectors (SIMD) or parallel threads (SIMT). Whereas, spatial architectures are employed for DNNs acceleration in ASIC and FPGA-based designs. This paradigm is based on *dataflow processing*, i.e., ALUs form a processing chain so that they can pass data from one to another. The architecture increases data reuse from low-cost memories in the memory hierarchy in such a way to reduce energy consumption. Dataflow processing and compression techniques have given researchers and industries the possibility to propose multiple frameworks for porting floating-point DNNs to FPGA-based platforms: Xilinx *Vitis AI* [19], *CHaiDNN* [20], and *FINN* [21]. In this work, we take Vitis AI as our reference framework that, to the best of our records, is the most mature solution of this kind for Xilinx platforms. Unfortunately, dealing with Vitis AI is not always straightforward: using the framework with modern and complex DNNs often requires dealing with the limitations of the tools and the architecture of the accelerators.

DNN for autonomous systems. Shaheen et al. [22] discussed the limits of DNN models in adapting to changing environments (to make an example, in autonomous systems), and showing methods and techniques for continuous learning in autonomous systems. Putra et al [23] proposed a method for unsupervised continual learning applicable to autonomous systems based on Spiking Neural Network (SNN). Viale et al. [24] proposed *CarSNN*, an 8-bit-weight SNN model for autonomous driving. To the best of our knowledge, this work represents the first comprehensive attempt to accelerate DNN models of a real-world perception system on an FPGA SoC embedded platform.

III. Motivation and Background

A. FPGA SoC platforms

FPGA SoCs are heterogeneous computing platforms typically composed of two subsystems: a *Processing System* (PS), incorporating multiple ARM-based processors combined with an FPGA *Programmable Logic*. These devices provide

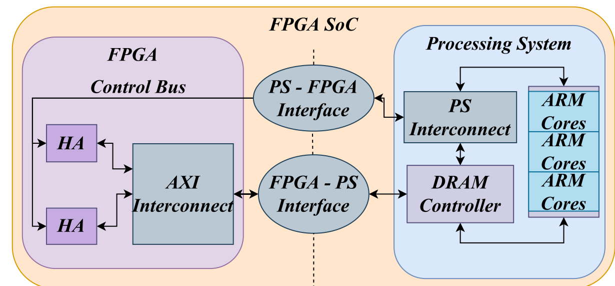

Fig. 1: Illustration of a typical FPGA SoC architecture.

higher integration, lower power consumption, and higher time predictability compared to GPU-based ones. Moreover, they can leverage high bandwidth communication bus between the processors and the FPGA. This is achieved through the standard (multi-manager and multi-subordinate) interface for interconnections, i.e., the ARM *Advanced Microcontroller Bus Architecture Advanced eXtensible Interface (AMBA AXI)* [25], known simply by the name of AXI. Software tasks run on the processors in PS. The FPGA logic can host custom hardware devices or peripherals, such as HAs, i.e., hardware components designed to perform specific functionality more efficiently than standard software. HAs are activated by software tasks, issuing an AXI request, whenever a hardware acceleration is needed. The PS and the FPGA subsystems communicate through two interfaces: the *PS-FPGA* interface and the *FPGA-PS* interface. Communications between HAs and processors can also occur through a shared DRAM memory controller located in PS and directly accessible by the HAs. For our purposes, DNN inference execution is a combination of executions on both HAs and on processors. Note that software task execution is typically required whenever the execution of a layer is not supported by HAs (see Section IV).

B. Vitis AI

Vitis AI is a framework for Xilinx platforms that aims at providing a set of tools for running complex DNN models on FPGA SoC platforms. The framework comprises a quantizer tool and a compiler tool, as shown in Figure 2. The quantizer converts FP32 weights and activations to INT8 fixed-point

format representation, using a PTQ algorithm. The conversion is performed at the cost of minimal accuracy loss [19](Vitis AI Quantizer). The quantized model is then parsed by the compiler, which builds a control-data flow representation of the operations. The compiler optimizes instructions scheduling and data reuse and produces an executable file containing specialized instructions for executing the model. Quantization and compilation steps are executed on powerful host machines. Then, the executable file is loaded by a software application, running on the target platform, developed through the *Vitis AI RunTime (VART)* API. The accelerated DNN execution is performed on the *Deep Learning Processor Unit core* (DPU core), a specialized HA to be deployed on FPGA. The compiler reports issues whenever it finds operations not supported for the DPU core. Following the official guidelines provided by Xilinx, unsupported layers must be executed by deploying software implementations running in the PS. This step can require to split the network into multiple subnets. Under some

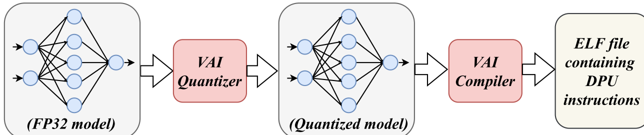

Fig. 2: The Vitis AI workflow and its two main tools (Quantizer and Compiler).

circumstances, we were able to execute unsupported DNN layers on the DPU core by converting them to equivalent layers compatible with the DPU core (see Section IV-B for more details).

C. The Apollo Perception Module

Autonomous driving is a challenging task that requires accurately sensing the environment to enable safe navigation. The Baidu Apollo platform, designed for the deployment of fully autonomous driving systems, has been implemented as a modularized architecture. Modules are described by their input/output (I/O) relationship to other modules. The *perception module* is responsible for perceiving the surrounding environment by analyzing sensor data (i.e., cameras, LiDAR, etc.). The Apollo perception system relies on six DNNs: **1)** *Lane Mark detector*, to detect lanes in camera scenes; **2)** *Denseline lane tracker*, to track lanes, taking as input the output of the lane detector; **3)** *Traffic light detector*, to identify traffic lights in camera scenes; **4)** *Traffic light recognition*, to classify lights status, starting from the output of the traffic light detector; **5)** *Obstacle detector*, to detect 3D objects in camera scenes; **6)** *LiDAR-based detector*, to complement the information of the obstacle detector using LiDAR data.

Figure 3 reports a graphical representation of the perception module. Models **(1-5)** input images coming from cameras, while **(6)** inputs cloud points coming from the LiDAR [26]. Following the Caffe standard [11], each DNN is represented through two files: a `.prototxt` describing the model structure and a `.caffemodel` containing weights and biases.

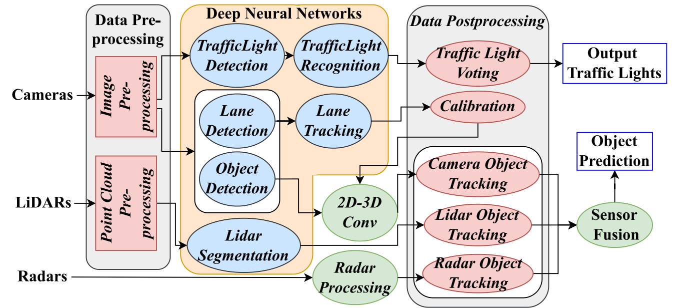

Fig. 3: Block diagram of the Apollo Perception Module.

IV. Accelerating Apollo's DNNs on FPGA SoC

In the following we illustrate the Apollo DNNs and discuss the challenges that were faced in moving them from GPU-based to FPGA-based acceleration. The Vitis AI tool quantizes DNNs using an *8-bit* precision for both weights and activations. The quantization process requires a calibration dataset. Unfortunately, the complete original Apollo dataset used for training its DNNs is not publicly available. Nevertheless, we found a valid alternative on the Apollo Scape website [27], which contains a reasonable amount of unlabeled camera-based images and LiDAR point-cloud data. An unlabeled dataset is anyway enough to perform the calibration (as also acknowledged by the Vitis AI documentation [19](Calibration process)) because the calibration process is based on analyzing the layer activations distribution. Next, we describe the steps required for accelerating each of the DNN models on FPGA-based platforms.

A. Denseline-lane tracker (DT)

Description. DT model is leveraged by the perception module to detect and predict roadway lines. The model includes more than 70 layers. It takes as input an RGB image acquired by the camera and it outputs a tensor representing 9 feature maps, with shape $(80 \text{ x } 192)$ each. The feature maps are then post-processed by other components of the Apollo framework (not analyzed in this paper).

Challenges. This DNN shows a simple architecture composed of standard operation layers, such as *convolutions*, *deconvolutions*, *pooling*, and simple *element wise operations*. No particular challenges emerged in dealing with this model as these layers are particularly suitable to the Vitis AI framework – the functionalities of the standard Xilinx Vitis AI quantizer and compiler were enough to accelerate the DNN.

B. Lane Mark detector (DarkSCNN or LMD)

Description. DarkSCNN is a YOLOv3-based model aiming at visual localization. It exploits spatial relationships among pixels to identify straight-shaped objects, even if partially obstructed (such as lanes). This model has more than 700 layers, including *convolutions*, *concatenations*, *fully connected (FC)*, *slices*, and *softmax*. It inputs RGB images and outputs detected objects shapes and the corresponding classification.

Challenges. The quantization process was successfully completed using the Vitis AI quantizer. Unfortunately, the compilation process was not straightforward and reported four major issues: **(1)** a compiler bug for a *slice* layer, **(2)** unsupported axis concatenation for a *concat* layer, **(3)** limitations related to the size of an *FC* layer, and **(4)** unsupported *softmax* operations. To solve these issues, we proposed novel algorithmic solutions that aim at converting DNNs to make them compatible with Vitis AI. They are described next.

Slice2Conv algorithm. A slice layer is intended for splitting an input tensor into multiple outputs along a given axis with certain section indices (i.e., points where input tensor must be divided), each named *slice point*. This first compilation issue is caused by a bug discovered in the compiler. The compiler checks that the number of slice points is equal to the output tensors number. If not, the compilation process is aborted. However, the official Caffe documentation [11](Slice Layer) indicates that the number of slice points must be equal to the output tensors number minus one. This bug has been reported and confirmed by Xilinx in [28]. Thus, we were not able to directly implement slice layers for the DPU. Analyzing the operations available on the DPU, we realized that the slice layer could be equivalently implemented through a set of 60 convolution operations. Therefore, we conceived the Slice2conv algorithm. To make the operations equivalent, we should find the correct dimension for the convolution kernels and the value of their weights. Note that the output tensor dimensions of a out_W x out_H convolution layer are:

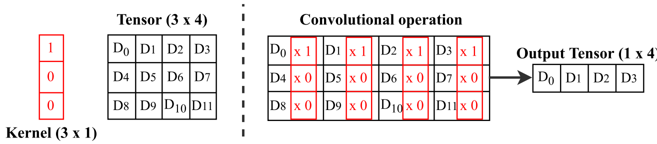

Fig. 4: Example of the Slice2conv algorithm in action.

$$out_w = \frac{in_w - k_w + 2p}{s} + 1, \quad (1)$$

$$out_h = \frac{in_h - k_h + 2p}{s} + 1, \quad (2)$$

where in_w and in_h are the input tensor dimensions, k_w and k_h are the kernel dimensions, and p and s are the padding and stride of the convolutional kernel, respectively. The slice input dimensions of the LMD are (60 x 80), while the required output dimensions for the slice are (1 x 80). To implement the slicing as a convolution we can then consider a convolutional kernel with stride $s = 1$ and zero padding ($p = 0$), hence obtaining the equivalent kernel size (k_w x k_h) by just rearranging the above formulas as follows:

$$k_w = in_w + 1 - out_w = 80 + 1 - 80 = 1, \quad (3)$$

$$k_h = in_h + 1 - out_h = 60 + 1 - 1 = 60. \quad (4)$$

Besides kernel dimensions, kernel weights must be properly selected. A simple example can be used to explain this concept. Figure 4 shows a convolution operation between a tensor and a kernel to realize slicing. The output vector represents the tensor row extrapolated by setting to 1 the kernel weight in the same row position of that which must be extracted by slicing (in this case the first one, i.e., D_0, D_1, D_2, D_3), leaving the others set to 0. Unfortunately, this first attempt was not successful due to a limitation of the DPU core, which is capable to deal with kernels with a maximum dimension equal to 16, [19](DPU operations), while we required (1 x 60) kernels.

Slice2MulConv algorithm. To overcome this limitation, we replaced each of the 60 Slice2Conv convolutions with 5 convolution layers, having reduced kernel dimensions, thus respecting the DPU constraints. We rearranged the slice operation by means of 300 convolutional layers, organized in 60 sets of convolutions. Each set takes as input the same slice input tensor with dimensions (60 x 80) and follows a 5-convolution hierarchical fashion. The hierarchical organization means that the convolutional layers execute one after the other. Each set is responsible to output one of the 60 original slice layer output tensor. Unlike the Slice2Conv algorithm, here, in each set, the first four convolutional layers share the same kernel and attempt at dividing the height dimension (60) of the input tensor into 15 groups of $60/15 = 4$ slice output tensors. The shared kernel has dimension (15 x 1) and serves the purpose of reducing the size of the input tensor, leading to a partial output tensor with dimensions (4 x 80). The shared kernel has all the weights set to 0, except for the weight corresponding to the row labeled by a parameter named $GroupID$, which is set to 1. The $GroupID$ value is calculated as the integer part of the division between the number of the output tensor slice to be produced (a number in the range [0, 59]) and 4. Whereas, the fifth and last convolutional layer has kernel dimensions (4 x 1) and all its weight values are set to 0, except for the weight corresponding to the kernel row with index $slice_index$, which is set to 1. The $slice_index$ parameter is calculated as the remainder of the previously-mentioned division. To better clarify this behavior, an example is reported in Figure 5, where the slice number 26 is obtained as output. After computing the $GroupID$ and $slice_index$ parameters, the shared and last kernels weights are set. The input tensor goes through the first convolution. The output of this phase is an intermediate tensor with dimensions (46 x 80), which is also the input of the second convolution. The same happens for the third and fourth convolution, until we have an intermediate tensor with dimension (4 x 80), which contains the slice output number 26. Afterwards, the fifth convolution is performed to extrapolate the right slice output tensor.

Regarding challenge **(4)**, according to the Vitis AI manual [19], the DPU incorporates a softmax core able to accelerate softmax operations. Unfortunately, the softmax core is separated from the DPU core and is not managed directly by the Vitis compiler. This means that it should be explicitly managed through the VART APIs. The softmax core is designed to take as input a tensor represented as INT8 values. This generated a format representation mismatch since in this case the outputs are in FP32 format. We were forced to address challenge **(4)** by

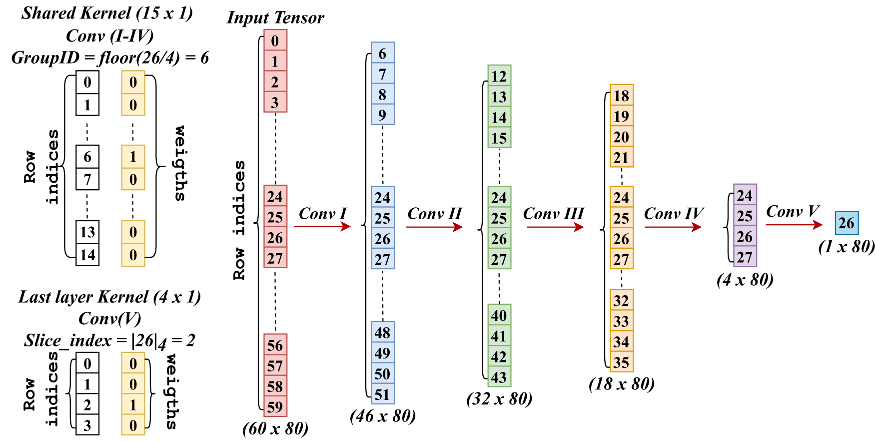

Fig. 5: Example of the Slice2MulConv algorithm in action. Dashed lines denote parts of the tensors that are not illustrated.

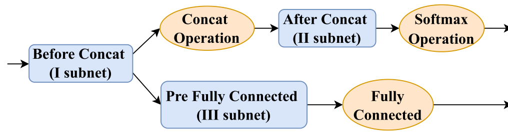

Fig. 6: In blue subnets running on DPU, in orange software operations.

implementing the softmax operation in software. For efficiency reasons, this has been done by employing a LookUp Table (LUT). This was feasible because the Quantizer provides the range of fixed-point values processed by the DPU. For the case of the LMD model, the range of 8-bit-width fixed-point values is in the interval [-8.00, 7.9375], with a step precision of 0.0625. The LUT data structure contains precomputed values of the $2^8 = 256$ possible softmax values for inputs in the range [-8.00, 7.9375]. It is implemented as a regular floating point array (of dimension 256) and it is accessed by simply taking into account the unsigned value of the input in two's complement representation. The choice of adopting a LUT proved to be effective in drastically improving the performance with respect to a standard software solution computing the multiple exponential functions required by the softmax at runtime. Finally, we were forced to implement equivalent software layers also to solve challenges **(2)**-**(3)**. Both concat and FC layers required about 50 lines of C++ code. Overall, this required to split the network into three subnets. The final solution is graphically depicted in Figure 6.

C. Traffic Lights Detection (TLD)

Description. TLD model executes a detection task of traffic lights through a YOLOv3-based network counting more than 80 layers. The network inputs images from the camera and a *Region of Interest (ROI)* and outputs bounding boxes for the detected traffic lights. As there might be more lights in the ROI, the DNN leverages three custom layers to select the proper ones according to their position and shape. If no lights are detected, the status is marked as unknown.

Challenges. This DNN relies on custom layers whose implementation is unfortunately not publicly disclosed by Baidu. Also, these networks were deployed using a modified version of the Caffe framework to handle such custom layers, which is not publicly available. Our attempts in retrieving any information about these custom layers failed. The custom layers are located in the final part of the model. Thus, we decided to exclude them as a temporary solution and focus on the rest of the network. The quantization phase passed with no errors. Conversely, the compilation raised two issues related to **(1)** *a reshape layer* and **(2)** *a softmax layer* – both of them unsupported by the DPU (see Section IV-B for the softmax layer). From the official Caffe documentation [11](Reshape Layer), a reshape layer is meant for changing dimensions of a tensor. Unfortunately, both (1) and (2) cannot be accelerated on the DPU core. Thus, we were forced to deploy them in software. Both layers are in the final part of the model. Thus, no network split was required. The softmax was implemented using the same LUT-based data structure discussed in the section IV-B. Whereas, the reshape layer required a very few lines of code to adjust the tensor dimensions.

D. Traffic Lights Recognition (TLR)

Description. TLR model aims at recognizing the color of traffic lights. It includes around 20 layers, among convolution, pooling, scale, and softmax ones. It inputs camera images, an ROI, and the bounding boxes coming from the Traffic Lights Detection model. The final softmax function outputs a vector of size $4n$ (with n being the number of bounding boxes), representing four scores for each bounding box related to the classes 'unknown', 'red', 'yellow', and 'green'. The highest scores, if large enough, determines the traffic lights' status. Otherwise, the status is set to 'unknown'.

Challenges. After solving a minor issue related to the lack of a bias parameter, which we found to be possible to be safely set to zero after checking the Caffe manual [11](Scale Layer), two major issues were reported by the compiler. They were related to **(1)** a *global average pool (GAP)* layer and **(2)** the unsupported *softmax* layer. Again, we were forced to implement such layers in software. Since the GAP layer is placed in the middle of the model, we split the network

into two sub-networks that can run on the DPU core. The

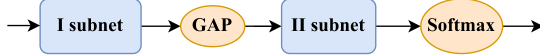

Fig. 7: TLR solution: in blue subnets on DPU, in orange software operations.

two sub-networks are connected by the software GAP layer. Figure 7 reports a graphical representation of the proposed solution. Finally, the software softmax inputs the output of the second sub-network and was implemented using another LUT, as discussed in Section IV-B. Whereas, the GAP layer required a simple 15-line C++ function.

E. LiDAR-based Segmentation (LS)

Description. LS model is a CNN and performs obstacle detection by analyzing the point-cloud data provided by LiDARs. It consists of around 30 layers. It inputs a feature map obtained by a pre-processing conversion of the LiDAR point-cloud. The network outputs 6-edge bounding boxes, where each bounding box completely wraps an obstacle.

Challenges. The network takes as input a (6 x 672 x 672) tensor. Unfortunately, the Vitis AI quantizer supports only gray scale (1-channel) or RGB (3-channel) images—6-channels tensors (as the one under analysis) are not supported by the current implementation [19]. To overcome this limitation, we

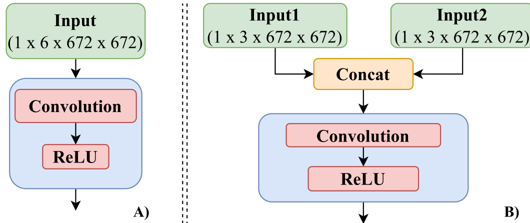

Fig. 8: LS model input layers: A) original network, B) proposed solution.

replaced the 6-channel input layer of the network with two 3-channel input layers, each having shape (3 x 672 x 672). Then, we deployed a concatenation layer on the channels to restore the original input size of the network (6 x 672 x 672). A graphical representation of the solution is reported in Figure 8). Thanks to these modifications we were able to successfully quantize the network.

Proceeding with the compilation, two issues arose due to **(1)** a *slice* unsupported layer and to **(2)** *sigmoid* unsupported operations. In this case, to solve issue **(1)** we opted for not employing the Slice2Conv or Slice2MulConv algorithms presented in Section IV-B. This is because such a slice layer has a simple structure that is not particularly computationally intensive, while the implementation through equivalent DPU-accelerated convolutions would have required the usage of considerable logical resources on FPGA. Concerning issue **(2)**, no equivalent operations supported by the DPU were available to accelerate sigmoid layers. Hence, they were implemented in software using LUTs, following the same procedure described for handling the softmax layer in the section IV-B.

F. Obstacle detection (OD)

Description. OD model aims at detecting 3D obstacles. It is a YOLOv3-based DNN consisting of more than 70 layers. It inputs images acquired from cameras and outputs bounding boxes and the corresponding classifications for stationary and dynamic object classes (cars, pedestrians, traffic cones, etc.)

Challenges. The quantization phase was completed successfully without issues. The compiler then reported two issues: **(1)** unsupported parameter *group* for six convolution layers **(2)** unsupported *power*, *sigmoid*, and *reshape* layers. Concerning issue **(1)**, the DPU supports only nominal convolution (group=1) or depthwise-like convolution (group=number of input channels). No approaches for converting these convolutional layers to make them compatible with the DPU were available to us. The same holds for issue **(2)**. We were hence forced to solve these issues by excluding the unsupported layers from the compilation process and deploying them in software, similarly as done for other DNNs.

In this case, to solve issue **(1)** we implemented the grouped convolutions through a C++ function consisting of about 50 lines. To do so, we retrieved and employed the quantized weights from the quantized model: this allowed us to perform the convolution operations among fixed-point integers only, avoiding expensive floating-point operations. The power and sigmoid layers of the issue **(2)** were implemented in software using LUTs, following the same procedure already described for handling the softmax and sigmoid layers in Sections IV-B and IV-E. Finally, the reshape layer was implemented as a C++ function according to its description from the Caffe manual [11](Reshape Layer), as already discussed in section IV-C.

V. Experimental evaluation

This section reports the experimental evaluation we conducted to assess the performance of the accelerated DNN models. Our target platform is the Xilinx Zynq Ultrascale+ on a ZCU102 board. The performance of the quantized networks running on the FPGA SoC were also compared with the same DNNs running on the Nvidia Xavier AGX SoC platform. Our evaluation is based on three performance metrics: network accuracy (Section V-B), throughput (Section V-C), and power consumption (Section V-D). While, the DPU configuration we adopted is reported in Section V-A. We deployed floating-point DNNs on GPU platforms when evaluating accuracy to assess the best achievable performance implied by our network modifications. To obtain a fair comparison, we evaluated floating-point and quantized (INT8) DNNs running on GPU when evaluating throughput and power. Models quantization for GPU has been carried out using the TensorRT framework [29]. Just like the Vitis AI quantizer, the TernsorRT quantizer leverages a PTQ quantization algorithm. However, note that the algorithms used by the two quantizers may differ – no public information is avaiable about their internal behavior.

A. DPU core configuration

In our experiment, the DPU was configured with the parameters recommended by Xilinx – the DPU operating frequency

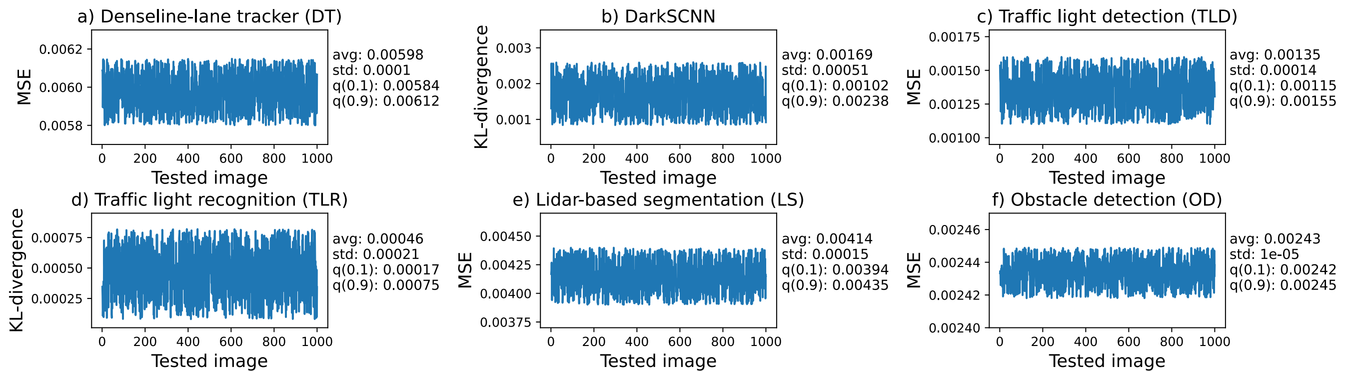

Fig. 9: DNNs MSE and KL-Divergence evaluations for statistical mathematical results

is set to 325MHz, corresponding to the maximum frequency guaranteed to meet the timing constraints [30]. The DPU has two cores of the B4096 architecture, low RAM usage, channel augmentation, and depthwise convolution enabled. Since DNN models can have different independent processing flows, the 2-core architecture has been chosen in order to exploit processing parallelism. The report generated by the Xilinx Vivado tool provides the following resource utilization for the selected DPU configuration:

- **87.56%** of Look-up-Tables (LUT): used to buffer data;
- **96.41%** of LUT distributed RAM (LUTRAM): used as small data buffers;
- **93.76%** of flip-flops (FF): used to describe logic circuits;
- **98.68%** of built-in RAM (BRAM): to store data;
- **76.43%** of Digital Signal Processing (DSP): used to process signals inside the FPGA;

Note that about more than 93% of the FPGA logic fabric is used for deploying the DPU core. Due to this high resource utilization, it is difficult to deploy other HAs on the FPGA fabric. This is the reason for which we were forced to implement DPU-unsupported layers using software approaches.

B. Accuracy

As mentioned above, the Apollo datasets are not publicly available. Therefore, the accuracy of the original DNN models cannot be directly compared with the accuracy of the quantized ones. Thus, we opted for evaluating the accuracy of the networks using the unlabeled Apollo Scape dataset. Our accuracy comparison is based on retrieving outputs from the quantized networks running on FPGA SoC and comparing them with the results obtained with the non-quantized networks running on GPUs, using statistical metrics for evaluating accuracy in the absence of the original labeled dataset [31], [32]. For comparison, we leveraged two evaluation metrics: *Mean Squared Error* [33] (MSE) for comparing bounding box coordinates and *Kullback–Leibler Divergence* [34] (KLD) for comparing probability distributions of classifiers – the lower their values, the better. The comparison results are reported in Figure 9. The plots report MSE or KLD (y axis) for each of the 1000 test images (x axis). We report only the most meaningful metrics for each DNN model. Note that the fluctuation of the metrics is very limited, and the same behavior was observed by testing more than 1000 images per setting/metric. For each DNN, we also report average, standard deviation and quantile (10% and 90%) values of the 1000 MSE or KLD samples.

Figure 9(a) regards the DT model. In this first case, the results show that the MSE ranges between $[0.00580, 0.00615]$. Figure 9(b) reports the results for the LMD model. This network has two outputs: one coming from an FC layer (detection) and one from a softmax layer (classification). The KLD metric was employed for evaluating the output of both layers. The results for FC layer output show KLD values in the range $[0.00084, 0.0026]$. Regarding the softmax output, KLD values are in the range $[0.061, 0.0625]$ (not reported in the charts). Figure 9(c) shows the results for the TLD network. The figure reports the metrics comparison for the bounding box coordinates prediction. MSE values are in the range $[0.0011, 0.0016]$. Figure 9(d) regards the TLR model. For this network, the output vector represents a probability distribution. Thus, here we used the KLD metric. We measured them in the range $[0.00008,\ 0.00082]$. Figure 9(e) regards the LS network. In this case, the comparison was carried out through MSE evaluation. The results show that MSE values are quite stable around 0.004343. Figure 9(f) regards the camera-based OD network. Also, this comparison was carried out through MSE calculation. The results range between $[0.002418; 0.002449]$.

Take-away message. For all the tested scenarios, the evaluated metrics showed how the quantized networks on FPGA SoC well approximate the original floating-point networks.

C. Throughput

Table I, II and III report the average, peak and standard deviation throughput performance in terms of Frames Per Second (FPS), respectively. Three scenarios are reported for each DNN under analysis: **(1)** INT8-quantized models running on FPGA SoC (ZCU102 platform), **(2)** INT8-quantized models running on GPU SoC (Xavier AGX platform), and **(3)** floating-point (32-bit) models running on GPU SoC (Xavier AGX platform). Each model was tested on 1000 images or 1000 LiDAR point-clouds.

The results show that the DNNs that rely most on DPU acceleration, such as DT, TLD, and TLR models, provide considerable performance improvements with respect to GPU-based

TABLE I: Average FPS on FPGA and GPU platforms.

	DT	LMD	TLD	TLR	LS	OD
INT8 FPGA	9.43	1.35	92.36	680.4	0.3	3.35
INT8 GPU	7.57	13.24	51.45	401.92	13.41	6.17
FP32 GPU	3.09	4.96	19.79	125.6	4.5	2.66

TABLE II: Peak FPS on FPGA and GPU platforms.

	DT	LMD	TLD	TLR	LS	OD
INT8 FPGA	9.43	1.43	95.22	713.31	0.34	3.37
INT8 GPU	8.43	13.96	52.17	404.32	14.04	7.46
FP32 GPU	3.26	5.62	21.67	127.21	4.97	2.67

TABLE III: Std deviation for FPS on FPGA and GPU platforms.

	DT	LMD	TLD	TLR	LS	OD
INT8 FPGA	0	0.04	1.47	13.11	0.02	0.01
INT8 GPU	1.44	2.52	5.49	18.22	0.58	0.9
FP32 GPU	0.27	0.47	2.12	2.38	0.25	0.031

acceleration (with and without quantization). Conversely, the other networks that rely less on DPU acceleration exhibit decreased performance on FPGA SoC with respect to GPU SoC. This is mainly connected with the presence of DPU-unsupported layers, which we were forced to run in software. For instance, this is the case of the LS model.

Take-away message. As long as the majority of the layers are able to execute on the DPU accelerator, FPGA SoCs are a valid alternative to GPU SoCs for the execution of complex DNNs for autonomous driving, and they can even achieve better throughput performance.

D. Energy consumption

We evaluated the energy consumption of quantized and non-quantized models. The measurements were conducted by leveraging the on-board sensors available on the platforms (both platforms share the very same energy sensor [35] [36]). Figure 10 compares: (a) average and (b) peak power consumption of each model for the same scenarios considered in the previous experiment. The results show how the FPGA SoC

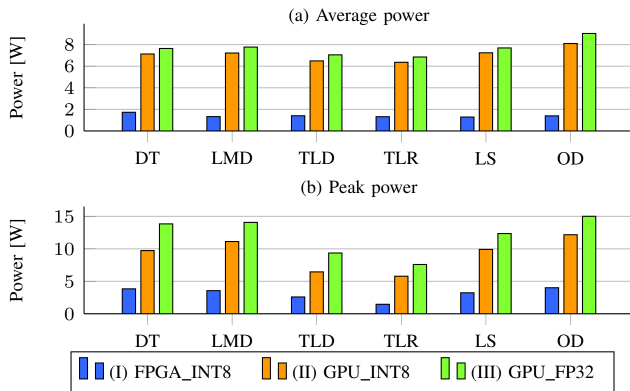

Fig. 10: Average and peak power consumption for FPGA and GPU SoCs

platform is more power efficient compared to the GPU SoC for both INT8-quantized and FP32 models. By comparing the power consumption of INT8-quantized DNNs on GPU and FPGA SoCs, it is possible to observe how the latter is capable of providing an improvement of at least 75% and 59% for average power consumption and peak power consumption, respectively. As one may expect, the results are even better when comparing against FP32 models running on GPUs. In this case, the FPGA SoC platform can provide an improvement of at least 77% and 72% for average power consumption and peak power consumption, respectively.

Take-away message. FPGA SoC platforms are definitively much more power efficient than GPU SoC ones for accelerating the tested DNN models.

VI. Conclusion

This work addressed the problem of accelerating modern DNN models with FPGA technology. Specifically, we accelerated the DNNs of Baidu Apollo, a popular open-source framework for autonomous driving, using an FPGA SoC heterogeneous platform. This required to address a series of challenges and propose new algorithmic solutions to deal with the limitations of commercial tools and accelerators. The performance of the FPGA-accelerated DNNs were compared with the one obtained by leveraging GPU-based acceleration, which is the standard approach used by Apollo and many other systems. FPGA SoCs have shown improvements in terms of power consumption and, in most cases, also throughput, while providing comparable results for the DNN accuracy. Future work will focus on improving the performance of the accelerated DNNs by implementing the unsupported software layers in FPGA logic. In conclusion, according to the authors' experience, it must be noted that FPGA-based hardware acceleration tools still seem to be too immature to handle the acceleration of complex DNNs without excessive efforts, as it is instead the case for GPU-based embedded platforms. In fact, there are still many shortcomings in terms of supported layers and operations. Nonetheless, given the considerable benefits that FPGA acceleration can provide, it is advisable to continue pushing on improving these DNN acceleration tools as their limitations do not pertain to the FPGA technology itself but rather to the maturity level of the surrounding ecosystem of software and programmable logic modules. Overcoming these issues would most likely drastically reduce the energy footprint of embedded DNN-enabled systems, which are not limited to autonomous vehicles and are expected to be ubiquitous in the near future.

References

[1] S. Nagpal, M. Kumar, Maruthi, M. R. Ayyagari, and Kumar, "A survey of deep learning and its applications: A new paradigm to machine learning," *Archives of Computational Methods in Engineering*, 07 2019.

[2] S. Dong, P. Wang, and K. Abbas, "A survey on deep learning and its applications," *Computer Science Review*, vol. 40, p. 100379, 2021.

[3] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.

[4] D. Li, X. Chen, M. Becchi, and Z. Zong, "Evaluating energy efficiency of deep convolutional neural networks on cpus and gpus," 2016.

[5] M. Capra, B. Bussolino, A. Marchisio, M. Shafique, G. Masera, and M. Martina, "An updated survey of efficient hardware architectures for accelerating deep convolutional neural networks," *Future Internet*, 2020.

[6] F. Restuccia and A. Biondi, "Time-predictable acceleration of deep neural networks on fpga soc platforms," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, pp. 441–454, IEEE, 2021.
[7] E. Ohbuchi, "Low power ai hardware platform for deep learning in edge computing," in *2018 IEEE CPMT Symposium Japan (ICSJ)*, 2018.
[8] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, and G. Buttazzo, "Axi hyperconnect: A predictable, hypervisor-level interconnect for hardware accelerators in fpga soc," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.
[9] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Modeling and analysis of bus contention for hardware accelerators in fpga socs," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
[10] Fan, Zhu, Liu, Zhang, Zhuang, Li, Zhu, Hu, Li, and Kong, "Baidu apollo EM motion planner," *CoRR*, 2018.
[11] Jia, Shelhamer, Donahue, Karayev, Long, Girshicks, Guadarrama, and Darrell, "Caffe: Convolutional architecture for fast feature embedding," https://caffe.berkeleyvision.org.
[12] A. Howard, A. Zhmoginov, L.-C. Chen, M. Sandler, and M. Zhu, "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation," in *CVPR*, 2018.
[13] M. Courbariaux and Y. Bengio, "Binarynet: training deep neural networks with weights and activations constrained to +1 or -1," 2016.
[14] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, 2015.
[15] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, 2014.
[16] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *International Conference on Learning Representations*, 2018.
[17] Y. Cai, Z. Yao, D. Zhen, A. Gholami, M. Mahoney, and K. Keutzer, "Zeroq: A novel zero shot quantization framework," *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, p. 13169–13178, 01 2020.
[18] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, p. 1737–1746, JMLR.org, 2015.
[19] Xilinx, "Vitis ai user guide ug1414 (v1.3)," 2021.
[20] Xilinx, "Chaidnn," 2017. https://github.com/Xilinx/chaidnn.
[21] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, p. 65–74, 2017.
[22] K. Shaheen, M. A. Hanif, O. Hasan, and M. Shafique, "Continual learning for real-world autonomous systems: Algorithms, challenges and frameworks," *J. Intell. Robotic Syst.*, vol. 105, p. 9, 2022.
[23] R. V. W. Putra and M. Shafique, "lpspikecon: Enabling low-precision spiking neural network processing for efficient unsupervised continual learning on autonomous agents," 2022.
[24] A. Viale, A. Marchisio, M. Martina, G. Masera, and M. Shafique, "Carsnn: An efficient spiking neural network for event-based autonomous cars on the loihi neuromorphic research processor," pp. 1–10, 07 2021.
[25] ARM, "Amba axi and ace, protocol specification," 2021. https://documentation-service.arm.com.
[26] Toschi, Sanic, Leng, Chen, Wang, and Guo, "Characterizing perception module performance and robustness in production-scale autonomous driving system," in *Network and Parallel Computing*, 2019.
[27] "Apollo scape dataset," 2017. http://apolloscape.auto/index.html.
[28] Xilinx, "Slice bug," 2020. https://github.com/Xilinx/Vitis-AI/issues/180.
[29] NVIDIA, "Tensor rt," 2017. https://developer.nvidia.com/Tensorrt.
[30] Xilinx, "Dpu product guide," 2020. https://www.xilinx.com/support/documentation/ip_ documentation/dpu/v3_2/pg338-dpu.pdf.
[31] Zhen, Zeng, Wang, and Han, "A global evaluation criterion for feature selection in text categorization using kullback-leibler divergence," in *2011 International Conference of Soft Computing and Pattern Recognition (SoCPaR)*, 2011.
[32] Jierula, Wang, and Oh, "Study on accuracy metrics for evaluating the predictions of damage locations in deep piles using artificial neural networks with acoustic emission data," *Applied Sciences*, 2021.
[33] C. Sammut and G. I. Webb, eds., *Mean Squared Error*. 2010.
[34] J. M. Joyce, *Kullback-Leibler Divergence*, pp. 720–722. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.
[35] Xilinx, "Zcu102 evaluation board ug1182," 2019.
[36] Nvidia, "Trm nvidia agx xavier system-on-chip," 2020.