Implementation of the PointPillars Network for 3D Object Detection in Reprogrammable Heterogeneous Devices Using FINN

Joanna Stanis¹ · Konrad Lis¹ · Marek Gorgon¹

Received: 25 April 2021 / Revised: 8 December 2021 / Accepted: 13 December 2021 / Published online: 27 December 2021
© The Author(s) 2021

Abstract

In this paper, we present a hardware-software implementation of a deep neural network for object detection based on a point cloud obtained by a LiDAR sensor. The PointPillars network was used in the research, as it is a reasonable compromise between detection accuracy and calculation complexity. The Brevitas / PyTorch tools were used for network quantisation (described in our previous paper) and the FINN tool for hardware implementation in the reprogrammable Zynq UltraScale+ MPSoC device. The obtained results show that quite a significant computation precision limitation along with a few network architecture simplifications allows the solution to be implemented on a heterogeneous embedded platform with maximum 19% AP loss in 3D, maximum 8% AP loss in BEV and execution time 375ms (the FPGA part takes 262ms). We have also compared our solution in terms of inference speed with a Vitis AI implementation proposed by Xilinx (19 Hz frame rate). Especially, we have thoroughly investigated the fundamental causes of differences in the frame rate of both solutions. The code is available at <https://github.com/vision-agh/pp-finn>.

1 Introduction

Object detection, e.g. vehicles, pedestrians, cyclists, animals, is crucial in advanced driver assistance systems (ADAS) and autonomous vehicles (AV, self-driving cars). It can be based on data from several sensors like radars, cameras and LiDARs (Light Detection and Ranging). Currently, the most popular solutions are based on cameras and/or radars. However, the advantages of LiDAR, which is an active sensor, are low sensitivity to lighting conditions (including correct operation at nighttime) and a fairly accurate 3D mapping

of the environment, especially at a short distance from the sensor. The disadvantages include improper operation in the case of heavy rainfall or snowfall and fog (when laser beam scattering occurs), deterioration of the image quality along with the increasing distance from the sensor (sparsity of the point cloud) and a very high cost. The last issue is the major reason that prevents this technology from being used more widely in commercially available vehicles (e.g., to improve ADAS solutions). However, it should be noted that there is constant progress in this area, including the so-called solid-state solutions (devices without moving parts). Therefore, one should expect that the LiDAR cost will be much lower soon. The output from a LiDAR sensor is a point cloud, usually in the polar coordinate system. A reflection intensity coefficient is assigned to each point. Its value depends on the properties of the material from which the beam was reflected. An example point cloud from a LiDAR sensor with a corresponding camera image is presented in Fig. 1.

Because of the rather specific data format, object detection and recognition based on a LiDAR point cloud significantly differs from methods known from “standard” vision systems. Generally, two approaches can be distinguished: “classical” and based on deep neural networks. In the first case, the input point cloud is subjected to preprocessing (e.g. ground removal), grouping (using clustering or fixed three-dimensional cells), handcrafted feature vector

The work presented in this paper was supported by the AGH University of Science and Technology project no. 16.16.120.773. We would like to especially thank Tomasz Kryjak for his help and insight while conducting research and preparing this paper.

✉ Joanna Stanis
stanisz@agh.edu.pl

Konrad Lis
kolis@agh.edu.pl

Marek Gorgon
mago@agh.edu.pl

¹ Embedded Vision Systems Group, Computer Vision Laboratory, Department of Automatic Control and Robotics, AGH University of Science and Technology, Al. Mickiewicza 30, 30-059 Krakow, Poland

Figure 1 A sample point cloud from the KITTI data set [9] alongside with the corresponding camera image.

calculation and classification (e.g. using SVM (Support Vector Machine)). These methods achieve only moderate accuracy on widely recognised test data sets – i.e. KITTI [9].

In the second case, deep convolutional neural networks are used. They provide excellent results (cf. the KITTI ranking [9]). However, the price for the high accuracy is the computational and memory complexity, and the need for high-performance graphics cards (Graphics Processing Units – GPUs) – for training and, what is even more important, inference. This stands in contrast to the requirements for systems in autonomous vehicles, where the aim is to reduce the energy consumption while maintaining the real-time operation and high detection accuracy.

Recently, a very promising research direction in embedded deep neural networks is the calculation precision reduction (quantisation). As reported in many publications, the transition from a 32-bit or 64-bit floating point representation to a fixed point and in extreme cases even to a binary one, results in a relatively small loss of precision, and a very significant reduction in computational and memory complexity.

The computing platform used in such a system is also relevant. The processing of a large number of points from the LiDAR sensor heavily uses the CPU (Central Processing Unit) and memory resources of a classical computer system (sequential computations). Especially algorithms based on deep neural networks need high-performance graphics cards (parallel computations) for learning and inference.

In this work, we evaluate the possibility of applying a DCNN based solution for object detection in LiDAR point clouds on a more energy efficient platform than a GPU. We propose the use of the ZCU 104 board equipped with a Zynq UltraScale+ MPSoC (MultiProcessor System on Chip)

device. The tools used in this work are PyTorch, Xilinx’s Brevitas, Xilinx’s FINN and Xilinx’s Vitis AI. PyTorch is a popular framework for training and inference of “ordinary” DCNNs on both CPUs and GPUs. It also includes pruning support. Brevitas [14] is a PyTorch based library used for defining and training quantised DCNNs. Likewise PyTorch, Brevitas enables computations on CPUs and GPUs as well. FINN [21] is a tool based on Python and C++ (code synthesisable with Vivado HLS). It allows to process quantised DCNNs trained with Brevitas and deploy them on a Zynq SoC or Zynq UltraScale+ MPSoC.

FINN implements consecutive layers as separate accelerators connected by AXI Stream interfaces into a pipeline. Users can choose the number of PEs (Processing Elements) applied for each layer. One PE computes one output channel at a time, so the maximum number of PEs is equal to the number of output channels. Each PE contains a user-defined number of SIMD lanes – each SIMD lane performs one multiply-add operation for one channel at a time, so the maximum number of SIMD lanes is equal to the kernel size times the number of input channels. If each layer would contain the maximum number of PEs with the maximum number of SIMD lanes, one output pixel could be computed in one clock cycle. However, with a large number of PEs and SIMD lanes, resource utilisation grows significantly – for the majority of architectures it can reach the target platform resource capacity. The solution for this is to increase the “folding” of the network (i.e. the number of cycles per layer) at the cost of lower frame rate. Folding can be expressed as: $\frac{k_{size} \times C_{in} \times C_{out} \times H_{out} \times W_{out}}{PE \times SIMD}$, where:

- k_{size} – kernel size,
- C_{in} – number of input channels,
- C_{out} – number of output channels,
- H_{out} – layer output height,
- W_{out} – layer output width.

It is recommended [19] to keep the same folding for each layer. The same number of cycles per layer ensures no bottlenecks in the pipeline and the absence of “too fast” layers waiting for input data. Therefore, one can achieve a given frame rate with minimum resource utilisation.

The last tool, often referenced in this paper, is Xilinx’s Vitis AI. It is a framework for quantising, optimising, compiling, and running neural networks on a Xilinx’s Zynq DPU (Deep Processing Unit) accelerator. It supports multiple DNN frameworks, including PyTorch and Tensorflow. In turn, DPU is a configurable iterative neural network accelerator supplied as IP block for programmable logic implementation.

Based on the initial analysis, we have selected the PointPillars [10] network for hardware-software implementation, mainly due to the favourable ratio of detection precision to

the computational complexity. Then, using the Brevitas and PyTorch libraries, we conducted a series of experiments to determine how limiting the precision and pruning affects the PointPillars performance – this part is described in our previous paper [16]. Afterwards, we processed the quantised network in the FINN tool to obtain its hardware implementation. The “unsynthesisable” parts of PointPillars were implemented on the processor system (PS) of the target platform.

This article is an extension of the conference paper [17] presented at DASIP’21 workshop in January 2021. We conducted more experiments with the PointPillars implementation in FINN – using the newest version 0.5. Finally, we achieved PL part execution time equal to 262ms and we proved there is no possibility for its further reduction. The PS part was rewritten to C++, thus reducing the whole point cloud processing time to 375ms. Moreover, a comparison between the Brevitas/FINN and Vitis AI frameworks was described. Besides, the difference of PointPillars frame rate in Vitis AI and FINN implementation was explained.

The main contribution of this paper:

- a hardware-software implementation of the PointPillars network on a reprogrammable heterogeneous computing platform,
- a case study on how applying optimisation to a quite complicated deep network architecture can result in an energy-efficient embedded LiDAR object detection system with a moderate loss on accuracy,
- an analysis of inference acceleration options in the FINN tool and a proof that our PointPillars implementation cannot be more accelerated using FINN alone,
- a comparison of the FINN and Vitis AI tools in terms of DNN inference speed in reprogrammable logic,
- analysis of PointPillars frame rate difference between FINN and Vitis AI implementations.

The rest of the paper is organised as follows. In Sect. 2 a general overview of DCNN (Deep Convolutional Neural Network) based methods for object detection in LiDAR point clouds, as well as the commonly used datasets are briefly discussed. Then in Sect. 3 the PointPillars network architecture is described and in Sect. 4 its optimisation is presented. In Sect. 5 the process of running the PointPillars through the FINN framework is discussed. Afterwards, in Sect. 6 the proposed hardware-software car detection system is presented. In Sect. 7 the performance experiments conducted with FINN tool are described. Finally, in Sect. 8 a comparison of pipeline and iterative neural network accelerators is performed regarding the inference speed. Based on the analysis results, a PointPillars inference speed difference between FINN based and Vitis AI based implementation is justified. The paper ends with conclusions and future research directions indication.

2 DCNN Methods of 3D Object Detection from a LiDAR Point Cloud

Recently, LiDAR data processing has been often realised with the use of deep convolutional neural networks. Likewise, in image processing, DCNNs combine the whole processing workflow (end-to-end), including both feature extraction and classification. They provide high recognition performance at a cost of high computational and memory complexity. Neural networks for LiDAR data processing can be categorised in the following way:

- 2D methods – the point cloud is projected onto one or more planes, which are then processed by classical convolutional networks - e.g. the MV3D method [5],
- 3D methods – no dimension is removed, the following subdivision can be made:
 - point based methods – they perform semantic segmentation or classify the entire point cloud as an object – e.g. PointNet [4],
 - cell based methods – they divide the 3D space into cells of fixed size, extract a feature vector for each of them and process the tensor of cells with 2D or 3D convolutional networks – examples are VoxelNet [23] and PointPillars [10] (described in more detail in Sect. 3),
 - hybrid methods – they use elements of both aforementioned approaches – an example is PV-RCNN [15].

The detection results for selected methods from the KITTI ranking are presented in Table 1. The KITTI ranking evaluation rules are explained below in the paragraph about the KITTI dataset. The list was limited only to car detection, because for these data the experiments described in Sect. 4 were carried out. The AP (Average Precision) measure is used to compare the results: $AP = \int_0^1 p(r)dr$ where: $p(r)$ is the precision in the function of recall r .

The detection results obtained using the PointPillars network in comparison with selected methods from the KITTI ranking are presented in Table 1. When analysing the results, it is worth paying attention to the following issues. First, progress in the field is rather significant and rapid – the PointPillars method was published at the CVPR conference in 2019, the PV-RCNN at CVPR in 2020 and SE-SSD was presented at CVPR in 2021. Second, for the BEV (Bird’s Eye View) case, the difference between PointPillars and SE-SSD method is about 7.5%, and for the 3D case about 12.5% – this shows that the PointPillars algorithm does not very well regress the height of the objects. Third, both SE-SSD and PV-RCNN networks are

Table 1 Comparison of the AP results for the **BEV** (Bird's Eye View) and **3D** KITTI ranking (the **Place** column indicates the algorithm's place in the ranking). The best result is in bold. In April 2021 one of the top methods is SE-SSD. PointPillars, with an up to 7.5% lower AP in BEV and 12.5% lower in 3D was ranked as 158th and 161th.

	Place	Method	Car		
			Easy	Mod.	Hard
BEV	-	VoxelNet	89.35	79.26	77.39
	160	PointPillars	90.07	86.56	82.81
	88	Patches	92.72	88.39	83.19
	53	STD	94.74	89.19	86.42
	8	PV-RCNN	94.98	90.65	86.14
	65	Voxel RCNN	94.85	88.83	86.13
	69	SIENet	92.38	88.65	86.03
	1	SE-SSD	95.68	91.84	86.72
3D	-	VoxelNet	77.47	65.11	57.73
	163	PointPillars	82.58	74.31	68.99
	136	Patches	88.67	77.20	71.82
	83	STD	87.95	79.71	75.09
	34	PV-RCNN	90.25	81.43	76.82
	20	Voxel RCNN	90.90	81.62	77.06
	15	SIENet	88.22	81.71	77.22
	3	SE-SSD	91.49	82.54	77.15

much more complex than PointPillars. Unfortunately, in neither [15] nor [22] the authors present data that would allow to describe this important parameter of the network unambiguously.

The main advantage of PointPillars over other cell-based methods is the use of 2D instead of 3D convolutions (like in VoxelNet [23]). 3D convolutions have a 3D kernel, which is moved in three dimensions. For a square/cuboid kernel of size K and a square/cuboid input tensor of size N , a 3D convolution takes $K \times N$ times more computations than the two-dimensional version of this operation. Therefore, the use of 2D convolutions substantially reduces the computational complexity of the PointPillars network, while maintaining the detection accuracy (PointPillars could be considered as VoxelNet without 3D convolutions, for average precision see Table 1) Thus, we decided to start research on its acceleration.

According to the authors knowledge, only three FPGA implementations of deep networks for LiDAR data processing were described in recent scientific papers (two 2019, one 2020), but none of them considered PointPillars. In the first one [12] a convolutional neural network implemented in FPGA was used to segment drivable regions in LiDAR point clouds. The authors used a series of optimised convolutional layers with quantised inputs, outputs, and parameters. The input tensor had a size of 180×64 with 14 channels. This size was constant throughout the whole network.

The maximum number of intermediate tensor channels was 64. The system was launched on an FPGA with clock 350 MHz in real time.

The second article [6] presents a real-time FPGA adaptation of the aforementioned VoxelNet model. The authors have based their convolutional layer implementation on the approach from [12]. Besides this information, they provide no details about the hardware implementation.

Authors of the third article [1] present an FPGA-based deep learning application for real-time point cloud processing. They used PointNet and a custom implementation on the Zynq Ultrascale+ MPSoC platform to segment and classify LiDAR data. The authors provide only resource utilisation and timing results without any detection accuracy.

Recently (December 2020), Xilinx has released a real-time implementation of PointPillars using the Vitis AI framework [20]. In Sect. 8 we compare this implementation with ours in terms of inference speed and give fundamental reasons why a significant frame rate difference occurs.

In the object detection systems task for autonomous vehicles, the most commonly used databases are KITTI, Wyamo Open Dataset, and NuScenes. The most popular of them is the KITTI Vision Benchmark Suite [7], which was created in 2012. It provides point clouds from the LiDAR sensor, images from four cameras (two monochrome and two colour), and information from the GPS/IMU navigation system. The training set in the object detection category consists of 7481 images along with the corresponding point clouds and annotated objects. In addition, KITTI maintains a ranking of object detection methods in two perspectives: BEV (Birds Eye View) and 3D. In the former, the output of the algorithm is compared to a rectangle describing the object in the top view (3D data is projected into 2D along the height dimension). In the latter, the output is compared with a cuboid describing the object in 3D. Furthermore, the annotated objects are split into three levels of difficulty (Easy, Moderate, Hard) corresponding to different occlusion levels, truncation, and bounding box height.

The Waymo Open Dataset [18] contains sensor data, collected by the Waymo autonomous vehicles operating in different geographical and weather conditions and at distinct times of the day. It was published in 2019. It includes 1950 sequences, 20s each, collected at 10Hz, which corresponds to 200000 frames. The following sensors were used: one medium range LiDAR (up to 75m), four short range LiDARs (up to 20m) and five cameras located at the front and sides of the vehicle. Of the entire data set, only 1200 sequences are annotated. However, they contain as much as 12.6 million objects.

Nuscenes [3] is a data set available publicly for noncommercial use. It was developed by Aptiv Autonomous Mobility (nuTonomy). It contains 1000 sequences, 20s each from Boston and Singapore, which are cities with heavy traffic.

Figure 2 An overview of the PointPillars network structure [10]. The Pillar Feature Network converts the point cloud into a “pseudo-image”, then using a 2D CNN (with transposed convolutions) this image is transformed into a feature map used in the final detection (Single Shot Detector).

The data was recorded in different weather conditions and at distinct times of the day. NuScenes provides data from the following sensors: 6 cameras, 1 LiDAR with 32 lasers, 5 radars, GPS and IMU. The data set includes approximately 1.4 million images, 390 thousand LiDAR scans, 1.4 million radars scans and 1.4 million annotated objects.

In this work, we decided to use the KITTI dataset due to the following reasons. Although 9 years old, KITTI still holds the position of the most widely used LiDAR database. The reason is its highly recognisable ranking which contains results for many methods. Thanks to this, new solutions can be easily compared with those proposed so far.

3 The PointPillars Network

The input to the PointPillars [10] algorithm is a point cloud from a LiDAR sensor (in Cartesian coordinates) limited to the area located in front of the vehicle. At the very beginning, the network divides the input along the XY plane into a “pillar” mesh. A “pillar” is a three-dimensional cell (cuboid) containing some number of points. An overview of the network structure is shown in Fig. 2.

The first part – *Pillar Feature Net* (PFN) – converts the point cloud into a sparse “pseudo-image”. Initially, the input data is divided into pillars. Each point, represented by four parameters (x, y, z Cartesian coordinates and reflection intensity), is extended to a nine-dimensional space ($D = 9$). The five new coordinates are x, y, z offsets from centre of mass of the points forming the pillar (denoted as x_c, y_c, z_c respectively) and x, y offsets from geometric centre of the pillar (denoted as x_p, y_p respectively). Because of the sparsity of the LiDAR data, most of the pillars contain no points. As a consequence, only a limited number of nonempty pillars (P) form the input of the network (instead of a tensor contacting all pillars). Such an approach reduces the memory

complexity. Furthermore, the number of points in a pillar (N) is also limited – this minimises the differences between very dense and sparse pillars.

The pillars are therefore fed to the network in the form of a dense tensor with dimensions (D, P, N) . Afterwards, each D dimensional point is processed by a linear layer with *batch normalisation* and *ReLU* activation function resulting in a tensor with dimensions (C, P, N) . Subsequently, for each cell, all of its points are processed by a *max-pooling* layer creating a (C, P) output tensor. Then it is mapped to a (C, H, W) tensor by moving the pillars to their original locations in the point cloud and filling the rest with zeros – it is called a “scatter” operation. H and W are dimensions of the pillar grid and simultaneously the dimensions of the “pseudoimage”.

The second part of the network – *Backbone (2D CNN)* – processes the “pseudo-image” and extracts high-level features. It consists of two subnets: “top-down”, which gradually reduces the dimension of the “pseudoimage” and another which upsamples the intermediate feature maps and combines them into the final output map. The “top-down” network can be described as a series of blocks: *Block* (S, L, F). A block has L convolution layers with a 3×3 kernel and F output channels. Each convolution is followed by a *batch normalisation* and a *ReLU* activation function. The first layer in the block has a $\frac{S}{S_{in}}$ step, while the next ones have a step equal to 1. At the end of each block, the feature maps are upsampled from input stride S_{in} to output stride S_{out} , using a transposed convolution with F output channels denoted as $Up(S_{in}, S_{out}, F)$. Then, after upsampling, a *batch normalisation* and a *ReLU* activation are used. The final feature map is derived from the concatenation of all upsampled output pillar feature maps.

The last part of the network is the *Detection Head (SSD)*, whose task is to detect and regress the 3D cuboids surrounding the objects. The objects are detected on a 2D grid using

Table 2 Results of 3D and BEV car detection for various network variants after training for 160 epochs. Easy, Moderate (Mod.) and Hard are the KITTI object detection difficulty levels. The most signif-

icant precision drop is visible in the 3D case (up-to 19%). In the BEV case, the drop is slightly above 8%. The size reduction is about 55x.

	3D			BEV			Size [kiB]
	Easy	Mod.	Hard	Easy	Mod.	Hard	
Original: PP(F32, F32, F32, F32)	84.17	76.12	70.51	90.15	87.42	84.46	18784.3
Quantised: PP(I8, I2, I8, I4)	76.64	67.33	65.56	90.15	81.08	80.53	1180.1
Final: PP(F32, B, I8, I2)	70.84	57.12	55.67	89.03	79.34	77.87	340.25

the Single-Shot Detector (SSD) network [11]. The position of the object along the Z axis is derived from the regression map. After inference, overlapping objects are merged using the Non-Maximum-Suppression (NMS) algorithm.

4 Optimisation of the PointPillars Network

There are three common methods of optimising a deep neural network: reducing the number of layers, quantisation and pruning (i.e. removing connections with insignificant weights).

To check how the PointPillars network optimisation affects the detection precision (AP value) and the network size, we carried out several experiments, described in our previous paper [16]. We focused on data from the KITTI database, especially car detection in the 3D category for three levels of difficulty: Easy, Moderate, and Hard. We split the quantisation of the PointPillars network into four parts: Pillar Feature Net (PFN), Backbone, Detection Head (SSD) and activation function. It turned out that the most important is the quantisation of the Backbone part, which is responsible for 99% of the size of the input network. Ultimately, we obtained the best result for the PointPillars network in the variant *PP(INT8, INT2, INT8, INT4)*, where the particular elements represent the type of the PFN, the Backbone, the SSD and the activation functions quantisation, respectively. The AP value of this variant of quantisation is presented in the second row in Table 2. Compared to the reference PointPillars version (GPU implementation by *nuTonomy* [13]), it provides almost 16x lower memory consumption for weights, while regarding all three categories the 3D AP value drops by max. 9% and BEV AP value drops by max. 7%.

5 Running PointPillars Network through the FINN Framework

After quantisation with Brevitas, the next step is passing the network through the FINN workflow. Finally, the user gets a bitstream (programmable logic configuration) and a Python driver to run on a PYNQ supported platform – ZCU 104 in

the considered case. At this stage of our project, we came across several constraints that FINN puts on the network and some limitations that FINN has itself.¹ It should be emphasised that FINN is still in the development stage, so the limitations described below may already be rectified.

Firstly, FINN supports only typical operations used by neural networks, i.e. convolutions and linear layers, batch normalisation, activation functions, and a few more. The “scatter” operation between the PFN and Backbone modules is certainly uncommon and atypical – it consists of feature vector relocation operations based on the coordinates of their corresponding pillars in the original point cloud. This makes the acceleration of the whole PointPillars network with the FINN framework impossible. PFN consists of a linear layer, batch normalisation, activation function and a max operation, which is currently not synthesisable with FINN. However, it is also a small part of the network, so it has a relatively low acceleration potential. The Backbone and Detection Head consist only of convolutional layers, batch normalisation, and activation functions. Therefore, only these two parts were chosen for hardware implementation.

Secondly, FINN puts two main constraints on the input tensor shape: it should be constant and symmetric. The former finally excludes the PFN part from the hardware implementation, as its input shape depends on the number of voxels, which varies among the point clouds returned by the LiDAR. The latter constraint was fulfilled by changing the point cloud range, so that the pillar mesh is square.

The next FINN constraint, that had a relatively big impact on our network architecture, was the lack of support for transposed convolutions. They are used for upsampling in the Backbone part and play a relatively important role, as they provide multiple detection scales. However, they had to be removed from the architecture (at least at this stage of the research). To preserve the same output map resolution, as now there are no upsampling layers, the convolution blocks strides were changed. They were reduced from 2 to 1 in the first layers of the 2nd and 3rd

¹ In [17] we used FINN in version 0.3. Now we proceed with FINN 0.5 in which some issues were solved.

convolution blocks. At this stage, after training such a modified network for 20 epochs, it turned out that these changes did not cause a huge loss of detection accuracy – c.a. 5%.

We also encountered issues with overutilisation of FPGA resources. Having overcome all aforementioned constraints, it turns out that the FINN implementation of the network utilises too much resources, slightly above the ZCU 104 capacity. The first idea was to increase the network folding (described in [2]). However, the reduction of resources usage was too little for the model to fit into the target platform. In the second attempt, all convolution blocks in the Backbone were reduced to three layers. This modification reduced the network to an extent that enabled it to fit onto the ZCU 104 platform.

Finally, the FINN framework usually implements a majority of the network in hardware, but it also keeps some unsynthesisable operations in the ONNX graph (Open Neural Network Exchange), next to the FPGA implementation. Currently, the default way of running the implemented network inference in FINN is conducting non-hardware ONNX operations on the PC, remote execution of the FPGA part, and a few more non-hardware operations on the PC. However, in this project it was assumed that the detection system on the ZCU 104 board should be as standalone as possible (we aim at a fully embedded solution). Therefore, it was decided to move nonhardware operations from the PC to the Zynq processor system. In this case, five operations were not implemented in hardware: MultiThreshold (activation function), 2 Transpositions, Add (adding a constant) and Mul (multiplying by a constant). Transpositions are responsible for changing the tensor dimensions order from NCHW to NHWC and the other way around. Consecutive tensor dimensions stand for:

- N – dimension related to point cloud number in a batch,
- C – feature dimension,
- H – height dimension,
- W – width dimension.

Apart from all aforementioned constraints, we also had to:

- add an additional “dummy” activation function between the PFN and Backbone – FINN has to ensure that a quantised tensor will be passed to the hardware implemented part of the network,
- reduce the input point cloud range, as it directly affects the PointPillars tensors heights and widths – the FPGA inference time was lowered 2x.

FINN allows to choose “folding”, i.e. the degree of parallelism in neural network hardware implementation. We have used the following configuration:

- Backbone – for each layer, one pixel is computed at a time, every pixel is computed in 288 iterations,
- Detection Head – one pixel is computed at a time, every pixel is computed in 64 iterations.

Besides this, every layer has an input FIFO queue with a capacity of 256 pixels.

Quantisation type of the final PointPillars version is $PP(FP32, B, INT8, INT2)$ – for detection accuracy see Table 2, row “Final”. In comparison to the PointPillars version after quantisation experiments (c.f. Sect. 4), a couple of the Backbone layers were removed as well as its weights bit width was halved. What is more, the activations bit width was also reduced. The PFN weight type was changed to floating point as it was not going to be implemented in hardware. All these changes (and a few minor ones) resulted in:

- compared to the version after quantisation experiments:
 - 3D average precision drop of maximum 10%,
 - BEV average precision drop of maximum 3%,
 - 3x network size reduction – from 1180.1 kiB to 340.25 kiB,
- compared to the original floating point version:
 - 3D average precision drop of maximum 19%,
 - BEV average precision drop of maximum 8%,
 - 55x network size reduction – from 18784.3 kiB to 340.25 kiB.

This PointPillars version was ready to implement in hardware.

6 Hardware-software Implementation of the Car Detection System

In this section, each step of the hardware implementation is presented. The whole LiDAR data processing system was divided between programmable logic (PL) and processing system (PS) (Fig. 3). The PC is used only for visualisation purposes. The PL is responsible for running the Backbone and Detection Head parts of the PointPillars network. The rest of LiDAR data processing is computed on the PS. It includes: reading input point cloud from a SD card, data preparation, the PFN network part, pre/post processing and output maps interpretation. The Zynq processing system runs a Linux environment.

In the current version of the system, the LiDAR point clouds are stored on the SD card of the ZCU 104 board. In the future, we plan to send point clouds to the processing system as UDP (User Datagram Protocol) frames with

Figure 3 Scheme of the proposed HW/SW detection system. The main computing platform is the ZCU 104 board equipped with a Zynq UltraScale+ MPSoC device. The processing system (PS) runs Linux, as it is responsible for the PFN module of PointPillars and some pre and post-processing. In the programmable logic the Backbone and the SSD modules are implemented. The PC is responsible only for the visualisation.

the same payload structure as LiDAR sensors use. This would allow to use PC emulated LiDAR or a real sensor interchangeably.

The workflow of the system is described below. The PS reads the point cloud from the SD card, voxelises it, and extends the feature vector for each point to nine dimensions (as it was described in Sect. 3). For every pillar, its points are processed by PFN, which outputs a feature vector for the whole pillar. Then, all pillar feature vectors are put into the tensor corresponding to the point cloud pillars mesh (“scatter” operation). Afterwards, the tensor is preprocessed – all unsynthesisable FINN operations prior to FPGA part are conducted. The next step is packing the tensor to some known format and sending it to the PL with DMA (Direct

Memory Access). The PL computes the Backbone + Detection Head network output and sends it back to the PS, also with DMA. The PS runs postprocessing on the received tensor – it consists of non-hardware FINN operations after the FPGA part. Then the PS splits the tensor into a classification and regression map and interprets both maps. As a final result, the PS computes the output 3D object bounding boxes. Results are visualised on the screen by the PC.

Figure 4 shows the output of the detection system applied to a sample point cloud in the bird eye view, whereas Fig. 5 presents the camera image corresponding to the considered point cloud with bounding boxes around the detected cars. The camera image is presented here only for visualisation purposes – the bounding boxes which are plotted on it are based on 3D LiDAR data processed by the network and projected on the image. Detections on the aforementioned images have at least a 50% score. It means that the probability of the detected object being a car is more than 50%.

Having quantised the final version of PointPillars network, after 160 epochs of training, we achieved the AP shown in Table 2. This table contains also the reference AP value for the network with FP32, and AP for the network chosen after quantisation experiments (c.f. Sect. 4). The final, reduced version of the PointPillars network has a 3D AP drop of 14% in Easy, 19% in Moderate, 15% in Hard and a BEV AP drop of 1% in Easy, 8% in Moderate, 7% in Hard regarding the original network version without quantisation.

Figure 4 LiDAR point cloud in BEV with detected cars marked with bounding boxes (bird eye view) [16].

Figure 5 Detected cars marked with bounding boxes projected on image – the same scene as in Fig. 4 [16].

Table 3 Resource utilisation and power consumption for FPGA part of the system. The numbers indicate, that for the considered device there is no room for accelerating other parts of the algorithm in PL.

Resource	Consumption
LUT	189074 (82.06%)
FF	150187 (32.59%)
BRAM	159 (50.96%)
DSP	88 (5.09%)
CLB	28684 (99.60%)
Power	6.515 W

However, the network size was simultaneously reduced 55 times. The memory footprint of the PS network part is just 7.03 kiB, as only one fully connected layer with 900 double precision floating point weights is implemented in the PS.

The FPGA resources usage for the FINN network is given in Table 3. The current design utilises more PL resources than in the original conference version [16] because of folding decreasing and clock rate change. In the current implementation there is no room for improvement for the hardware part of the network as CLB utilisation has almost reached the limit. The PL clock is currently running at 150 MHz. The power consumption reported by Vivado is equal to 6.515W. The peak power consumption on the ZCU 104 board (including both Zynq PS and PL, as well as additional devices), measured with the PMBus (Power Management Bus), is equal to 14.02W. It should be noted that if a PC with a high performance GPU is used, at least a 500W power supply is required. The power consumption on our target platform is therefore at least 35 times smaller.

The timing results of the PS and PL parts (averaged over 100 point clouds from the KITTI validation dataset) are listed below:²

- Voxelisation takes 7.13 milliseconds,
- Extending feature vector takes 1.65 milliseconds,
- PFN takes 62.43 milliseconds,
- Scatter operation takes 0.72 milliseconds,
- FPGA preprocessing (on ARM) takes 3.1 milliseconds,
- FPGA part takes 261.74 milliseconds³,
- FPGA postprocessing (on ARM) takes 2.93 milliseconds,
- Classification and regression maps interpretation takes 32.13 milliseconds,
- Total average inference time: 374.66 milliseconds (2.67 FPS).

² Timing results do not include reading LiDAR data from a SD card.

³ This can be further reduced to c.a. 180 milliseconds by increasing the clock frequency to 200 MHz. However, currently the FINN flow targeting C++ applications does not support to freely choose the clock rate. The 180 ms value for the FPGA part was achieved in the previous Python version of the application, however it had much higher overall time of execution.

In the case of LiDAR data, real-time processing can be defined as performing all computation tasks on the point cloud in time equal or lower than a single LiDAR scan period. The reached total average inference time is 4x too long to fulfil real-time requirements, as a LiDAR sensor sends a new point cloud every 0.1 seconds. The FPGA part takes 262 milliseconds to finish, but other operations on PS part are expensive too. In [17] the FPGA part execution time was 1.99 seconds. It was reduced by:

- changing FINN from version 0.3 to 0.5,
- increasing the clock rate from 100MHz to 150MHz,
- setting lower folding.

The processor part of the network is implemented in C++. Initially, the total inference took c.a. 70 seconds. We have made several optimisations like:

- rewriting the application from Python to C++,
- taking advantage of multithreading (using the standard C++ threading library) – the following components of the processor part of the network were parallelised:
 - Voxelisation – split into 4 threads, each thread handles a portion of input points,
 - Extending feature vector – split into 4 threads, each thread handles a portion of voxels,
 - PFN – split into 4 threads, each thread handles a portion of voxels,
 - Scatter operation – split into 2 threads, each thread handles a portion of voxels,
- implementing matrix multiplications in PFN with the *Eigen* library[8] instead of a naive nested loop approach.

7 Experiments with FINN

After running the PointPillars through FINN, we did some more experiments with this framework to evaluate its possibilities and dependencies between the following parameters: clock frequency, folding, queue size, frame rate and resource utilisation.

We used a simple convolutional network to conduct experiments. It consisted of 5 layers with kernel size (3, 3), stride 1 and padding 1. The network input shape was equal to (1, 1, 32, 32) in the NCHW format. The last layer output was a tensor (1, 128, 32, 32). The layers had an input channel number of 1, 32, 32, 64, 64 and output channel number of 32, 32, 64, 64, 128 consecutively.

Results of the experiments are presented in Figs. 6, 7, 8, 9, 10, and 11. On the first two of them (Fig. 6, Fig. 7) we show the clock frequency vs frame rate and vs utilisation.

Figure 6 Frame rate in function of clock frequency. The relation is linear.

As it should be expected, the frame rate is a linear function of the clock frequency. Utilisation of LUTs and FFs slightly increases with the rising clock rate, the BRAM consumption remains constant. The FINN uses C++ with Vivado HLS to synthesise the code. As the target clock rate increases,

the HLS synthesis tool has to increase the maximum logic throughput. This is probably done by increasing the latency at the cost of additional LUTs and FFs. CLB utilisation varies strongly without an apparent pattern – probably due to the synthesis tool (Vivado) specificity.

Figure 7 Resources utilisation in function of clock rate. LUT and FF consumption slightly increases whereas CLB varies strongly and BRAM remains constant.

Figure 8 Frame rate in function of folding. The relation is almost hyperbolic.

Folding in Figs. 8 and 9 is counted relatively to the network implementation with 1, 32, 32, 64, 64 SIMD lanes and 32, 32, 64, 64, 128 PEs for consecutive layers. It is worth emphasising that the folding should be kept the same for each layer. According to [19] the slowest layer determines the

throughput of the whole network, so it is a waste of resources to keep faster layers along with slower ones. One should expect that a two times folding decrease implies a two times frame rate increase. This relationship is preserved except for the folding change from 4 to 2 – the frame rate increase is

Figure 9 Resources utilisation in function of folding. Consumption of all resources, except for BRAM, strongly increases when folding approaches 1.

Figure 10 Frame rate in function of queue size. The function is monotonically rising with decreasing slope.

significant but not as big as expected. This anomaly may be caused by communication overheads which are increasingly substantial when the frame rate increases. The resource utilisation is almost like expected – the less folding is, the more

resources are consumed for additional PEs and SIMD lanes. In Fig. 9 there is one anomaly – significant BRAM utilisation decrease for folding equal 2 in comparison to folding equal 4. It is probably FINN specific behaviour, but the precise

Figure 11 Resources utilisation in function of queue size. CLB and LUT utilisation slightly increases.

reason is not known. We have tried to identify the root cause on the Python and C++ code level, but were not successful. The issue is difficult to trace back, as FINN modules are synthesised from C++ code to HDL (Hardware Description Language) via Vivado HLS. The generated HDL code is complicated and difficult to analyse.

The queue size presented in Figs. 10 and 11 refers to the FIFO pixel queue before each layer. We checked four queue sizes: 32, 64, 128 and 256 (the maximum supported value in FINN). The chart in Fig. 10 is a raising function. We suppose that the frame rate significantly increases with the queue size because it smooths the communication between layers. The price of speedup is a slight LUT utilisation increase as one can see in Fig. 11. Additional LUTs are consumed probably by the increased FIFOs memory.

To sum up, in FINN, there are three general ways to speed up the implementation of a given network architecture. The first one is setting a higher clock frequency. It results in a slight resource utilisation increase, as the place & route tools have to meet the timing constraints. The second possibility is increasing the input queue size. It consumes a relatively small amount of resources and the performance gain can be significant. Moreover, the folding decrease requires a lot of resources, but the speedup is potentially the highest. With our PointPillars FINN implementation, we have already set the maximum queue size. There is no room to either decrease folding or increase the clock frequency, as we are at the edge of CLB utilisation for the considered platform. The only option left for a significant increase of implementation speed is further architecture reduction. However, this would lead to further detection accuracy reduction, which is definitely not a good option taking into consideration the target applications: self-driving cars and ADAS solutions, as well as the state-of-the-art performance in this field.

8 PointPillars in FINN and Vitis AI – a Comparison

Recently (December 2020), Xilinx released a real-time PointPillars implementation using the Vitis AI framework [20]. It runs at 19 Hz, the Average Precision for cars is as follows:

- BEV: 90.06 for Easy, 84.24 for Moderate and 79.76 for Hard KITTI object detection difficulty level,
- 3D: 79.99 for Easy, 69.07 for Moderate and 66.73 for Hard KITTI object detection difficulty level.

Our Backbone and Detection Head FINN implementation runs at 3.82 Hz and has smaller AP (compare with Table 2). Thus, an obvious question arises: why the

implementation of PointPillars on Vitis AI is faster than the FINN one? In the following analysis, we will try to provide an answer. It is worth noting that the Vitis AI implementation of PointPillars network is not described in any report or scientific paper. Xilinx has not provided code or description in what exact manner the PointPillars is compiled for the DPU execution. Thus, even though C++ code is released, some implementation details are unknown.

The Vitis AI tool is based on the DPU accelerator. It is an IP block, on which calculations are performed iteratively. On the other hand, FINN is based on a pipeline of computing elements (accelerators), each responsible for a different layer of the neural network.

In neural networks, the computing mainly consists of performing multiply-add operations. The DPU accelerator and individual accelerators in FINN can perform a certain number of these operations per clock cycle. Suppose there are no delays in data transfer and accelerator control. Additionally, assume that the considered neural network has L layers. Let:

- N_k – number of multiply-add operations for the k th layer,
- a_k – number of multiply-add operations per clock cycle, for the k th layer FINN accelerator,
- b – number of multiply-add operations per clock cycle, for the DPU accelerator,
- C_F – number of cycles needed for calculating network output in FINN,
- C_D – number of cycles needed for calculating network output in DPU.

Therefore, the number of cycles for k th layer in FINN is equal to $\frac{N_k}{a_k}$ and in DPU to $\frac{N_k}{b}$.

In the case of DPU, layers are iteratively computed on the accelerator, so $C_D = \sum_{k=1}^L \frac{N_k}{b}$. In FINN, the calculations are performed in a pipeline, so the layer with the highest number of cycles is the bottleneck and determines the speed of the entire system. Therefore $C_F = \max_k \frac{N_k}{a_k}$.

Consider the cases:

- if $\forall k \in \{1, \dots, L\}, a_k > b$ then $\max_k \frac{N_k}{a_k} < \max_k \frac{N_k}{b}$ and as sum of positive elements is always greater or equal than one of its elements we have $\max_k \frac{N_k}{b} \leq \sum_k \frac{N_k}{b}$, so $C_F < C_D$
- if $\forall k \in \{1, \dots, L\}, L \times a_k < b$ then:
 - from the assumption, we have $\sum_k \frac{N_k}{a_k \times L} > \sum_k \frac{N_k}{b}$,
 - by summing inequalities $\forall l \in \{1, \dots, L\}, \max_k \frac{N_k}{a_k} \geq \frac{N_l}{a_l}$ we get $L \times \max_k \frac{N_k}{a_k} \geq \sum_k \frac{N_k}{a_k}$
 - therefore $\max_k \frac{N_k}{a_k} \geq \sum_k \frac{N_k}{a_k \times L} > \sum_k \frac{N_k}{b}$ so $C_F > C_D$

This is a proof of concept that either FINN or DPU, for certain network architectures, can perform better than the other tool. Of course, the time of data transfer or any other overhead is not taken into account here, but choosing a sufficiently large difference between b and individual a_k one can come to similar conclusions. An intuitive rule can be drawn that if $\forall k \in \{1, \dots, L\}, a_k \gg b$ then better results can be obtained using FINN, if $\forall k \in \{1, \dots, L\}, a_k \ll b$ DPU should be faster.

In the DPU version that was used to run PointPillars on the ZCU 104 platform, the accelerator can perform 2048 multiply-add operations per cycle and operates at a frequency of 325 MHz (650 MHz is applied for DSP). According to [20], Vitis AI PointPillars implementation has 10.8×10^9 operations, counting multiplications and additions separately. Effectively, it has 5.4×10^9 multiply-add operations. Thus, the frame rate equals $\frac{2048 \times 325 \text{ MHz}}{5.4 \times 10^9} \approx 123.26 \text{ Hz}$.

Taking into account the configuration used in the FINN tool ($\forall k, a_k \leq 2048$) $C_F = \max_k \frac{N_k}{a_k} = 7372800$ and the clock frequency is 150 MHz. Hence, the FINN frame rate is around 20.35 Hz. Therefore, the DPU should perform better.

The Vitis AI implementation of PointPillars includes the entire original PointPillars model. The workflow is as follows: preprocessing performed in the PS, Pillar Feature Net on the DPU, scatter operation in the PS, Backbone with Detection Head on the DPU and postprocessing in the PS. The operations in the PS are implemented using the C++ language. The frame rate equals to 19 Hz and is measured taking into account both components performed on the DPU and components computed in the PS. Even so, the Vitis AI implementation runs faster than the Backbone with Detection Head alone in FINN (3.82 Hz).

It is worth noting that despite removing a couple of layers in our PointPillars version (compared to the original version), it has 70×10^9 multiply-add operations – c.a. 13x more than the Vitis AI PointPillars version. It may be surprising, but the explanation is as follows. As we have changed the convolution stride in the 2nd and 3rd convolution blocks, these Backbone parts process, respectively, 4x and 8x larger tensors than in the original PointPillars version. A sufficiently larger number of features (128 for layers in the 2nd block and 256 for layers in the 3rd block) escalate the effect. In the result, our PointPillars version is reduced and simultaneously more computationally complex. If our PointPillars version was executed on the DPU the theoretical frame rate would be equal to 9.51 Hz. It is less than the theoretical FINN framerate (20.35 Hz). Resource consumption for the Vitis AI PointPillars implementation is as follows:

- LUT – 18.84%,
- FF – 20.10%,
- BRAM – 82.37%,
- DSP – 39.93%.

There is a significant difference in LUT (63% less), BRAM (31% more) and DSP (35% more) utilisation in comparison to our implementation. With DPU, there is still a considerable amount of free programmable logic resources. Therefore, one can implement some other algorithm in the PL next to DPU. In the case of our FINN implementation, it is not possible as almost all CLBs are consumed by PointPillars.

In summary, the Vitis AI implementation is faster than the implementation in FINN due to the larger computational complexity of our PointPillars version. Hypothetically, if the FINN PointPillars version was run on the DPU, it would perform worse than FINN. In the current FINN version, there is no good alternative but to perform architecture changes, as FINN has no support for transposed convolutions. With no upsampling and with the original stride values, the output map would have a 4x smaller resolution compared to the original PointPillars, what requires further changes in the object detection head and output map post-processing, as well as reduces the detection accuracy. Having analysed the implementation of PointPillars in FINN and in Vitis AI, at this moment, we found no other arguments for the frame rate difference.

9 Conclusions

In this article, we have presented a hardware-software implementation of a car detection system based on LiDAR point clouds. We have used the Brevitas tool for quantisation of the PointPillars neural network and the FINN framework for synthesis and FPGA implementation of the quantised network. The solution was verified in hardware on the ZCU 104 evaluation board with Xilinx Zynq UltraScale+ MPSoC device. The system is characterised by a relatively small power consumption along with high object detection accuracy. The low power consumption of reprogrammable SoC devices is particularly attractive for the automotive industry, as the energy budget of new vehicles is rather limited.

The achieved AP of the final, reduced and quantised version of PointPillars has a maximum 3D AP drop of 19% and a maximum BEV AP drop of 8% regarding the original network version (floating point). At the same time, modifying PointPillars allowed for implementing its majority in programmable logic as well as to reduce its size 55 times. The inference time of one sample in the implemented system is around 375 milliseconds. The PL

part processing lasts for 262 milliseconds. The inference time should be reduced 4x to reach real-time, as a LiDAR sensor sends a new point cloud every 0.1 seconds. At present, no operations can be moved to the PL as almost whole CLB resources are consumed. Currently, in FINN there is no possibility to optimise our PointPillars implementation without loss of detection accuracy (e.g. further architecture reduction). It implies that the considered FINN based implementation would never reach real-time performance, as the PL part takes more than 0.1 seconds, mainly due to the lack of transposed convolution support. However, the Vitis AI framework shows promising results as Xilinx's implementation of PointPillars using Vitis AI runs at 19Hz.

As a future work, we would like to analyse the newest networks architectures, and with the knowledge about FINN and Vitis AI frameworks, implement object detection in real-time – possibly using a more accurate and recent algorithm than PointPillars. Implementing the transposed convolution in FINN is also worth considering. It would allow us to create a demonstrator cooperating with a LiDAR sensor. Then, we plan to conduct experiments on the network for the other categories from the KITTI set, i.e. pedestrians and cyclists, as well as the Waymo and NuScenes sets. Working with simulation environments like CARLA is also an interesting option. Furthermore, we would like to use data fusion for LiDAR, video, and radar sensors.

According to the authors knowledge, only three FPGA implementations of LiDAR networks were described in very recent scientific papers [1, 6, 12]. However, they do not consider PointPillars network and they do not use the FINN framework. They work in real-time, but ChipNet [12] use smaller tensors with a much smaller number of features – what greatly reduces the computational complexity. Our version of PointPillars has more than 2.7M and the ChipNet 760k parameters what is another premise of the higher computational complexity of our implementation. Finally, the system runs with a 350 MHz clock – ours with 150 MHz. The authors of [6] did not provide enough details to compare that network implementation with ours. It is hard to compare with the implementation described in [1] as PointNet network architecture significantly differs from ours. Regarding Xilinx's Vitis AI implementation of the PointPillars, we made an analysis why it is faster than our approach. Our PointPillars version has larger computational complexity and in Vitis AI a higher clock rate is applied – 325MHz instead of 150MHz. In the current FINN version, we had to apply network architecture changes as FINN does not support transposed convolutions. Summing up, we believe that our project made a contribution to this rather unexplored research area.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Bai, L., Lyu, Y., Xu, X., & Huang, X. (2020). Pointnet on FPGA for real-time lidar point cloud processing.
2. Blott, M., Preußner, T. B., Fraser, N. J., Gambardella, G., O'Brien, K., Umuroglu, Y., Leeser, M., & Vissers, K. (2018). Finn-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11(3), 1–23.
3. Caesar, H., Bankiti, V., Lang, A. H., Vora, S., Liong, V. E., Xu, Q., Krishnan, A., Pan, Y., Baldan, G., & Beijbom, O. (2019). nuscenes: A multimodal dataset for autonomous driving. arXiv preprint [arXiv:1903.11027](https://arxiv.org/abs/1903.11027)
4. Charles, R. Q., Su, H., Kaichun, M., & Guibas, L. J. (2017). Pointnet: Deep learning on point sets for 3d classification and segmentation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 77–85). <https://doi.org/10.1109/CVPR.2017.16>
5. Chen, X., Ma, H., Wan, J., Li, B., & Xia, T. (2016). Multi-view 3d object detection network for autonomous driving.
6. García López, J., Agudo, A., & Moreno-Noguer, F. (2019). 3D vehicle detection on an FPGA from lidar point clouds. 21–26. <https://doi.org/10.1145/3369973.3369984>
7. Geiger, A., Lenz, P., Stiller, C., & Urtasun, R. (2013). Vision meets robotics: The KITTI dataset. *International Journal of Robotics Research (IJRR)*.
8. Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. <http://eigen.tuxfamily.org>. Last access: 27 December 2021.
9. KITTI. (2020). KITTI database website. <http://www.cvlibs.net/datasets/kitti/>. Last access: 15 May 2020.
10. Lang, A. H., Vora, S., Caesar, H., Zhou, L., Yang, J., & Beijbom, O. (2019). Pointpillars: Fast encoders for object detection from point clouds. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 12689–12697). <https://doi.org/10.1109/CVPR.2019.01298>
11. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.-Y., & Berg, A. C. (2016). SSD: Single shot multibox detector. In B. Leibe, J. Matas, N. Sebe, & M. Welling (Eds.), *Computer Vision – ECCV 2016* (Cham, pp. 21–37). Springer International Publishing.
12. Lyu, Y., Bai, L., & Huang, X. (2019). Chipnet: Real-time lidar processing for drivable region segmentation on an FPGA. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66, 1769–1779.
13. nuTonomy. Pointpillars repository. <https://github.com/nuTonomy/second.pytorch>. Last access 17 June 2020.
14. Pappalardo, A., & Team, X. R. L. Brevitas repository. <https://github.com/Xilinx/brevitas/>. Last access 17 June 2020.
15. Shi, S., Guo, C., Jiang, L., Wang, Z., Shi, J., Wang, X., & Li, H. (2019). PV-RCNN: Point-voxel feature set abstraction for 3d object detection. ArXiv [abs/1912.13192](https://arxiv.org/abs/1912.13192)

16. Stanisz, J., Lis, K., Kryjak, T., & Gorgon, M. (2020). Optimisation of the pointpillars network for 3D object detection in point clouds. In *2020 Signal Processing: Algorithms, Architectures, Arrangements and Applications (SPA)* (pp. 112–127).
17. Stanisz, J., Lis, K., Kryjak, T., & Gorgon, M. (2021). Hardware-software implementation of the pointpillars network for 3D object detection in point clouds. In *Workshop on Design and Architectures for Signal and Image Processing* (14th edn.).
18. Sun, P., Kretzschmar, H., Dotiwalla, X., Chouard, A., Patnaik, V., Tsui, P., Guo, J., Zhou, Y., Chai, Y., Caine, B., Vasudevan, V., Han, W., Ngiam, J., Zhao, H., Timofeev, A., Ettinger, S., Krivokon, M., Gao, A., Joshi, A., Zhang, Y., Shlens, J., Chen, Z., & Anguelov, D. (2019). Scalability in perception for autonomous driving: Waymo open dataset.
19. Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M., & Vissers, K. (2017). Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (pp. 65–74). FPGA '17, ACM.
20. VitisAI. Pointpillars implementation in vitisai. https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo/model-list/pt_pointpillars_kitti_12000_100_10.8G_1.3. Last access 17 April 2021.
21. Yaman, U., Jakoba, P.-K., Andrea, R., & Hendrik, B. Finn repository. <https://github.com/Xilinx/finn>. Last access 13 April 2021.
22. Zheng, W., Tang, W., Jiang, L., & Fu, C.-W. (2021). SE-SSD: Self-ensembling single-stage object detector from point cloud.
23. Zhou, Y., & Tuzel, O. (2018) Voxelnet: End-to-end learning for point cloud based 3D object detection. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 4490–4499). <https://doi.org/10.1109/CVPR.2018.00472>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.