Deployment of Deep Neural Networks for Object Detection on Edge AI Devices with Runtime Optimization

Lukas Stäcker[1,2,*], Juncong Fei[1,3,*], Philipp Heidenreich[1], Frank Bonarens[1],
Jason Rambach[2], Didier Stricker[2], and Christoph Stiller[3]

[1]Stellantis, Opel Automobile GmbH, Germany
[2]German Research Center for Artificial Intelligence, Germany
[3]Institute of Measurement and Control Systems, Karlsruhe Institute of Technology, Germany
[*]equal contribution
lukas.staecker@external.stellantis.com
juncong.fei@external.stellantis.com

Abstract

Deep neural networks have proven increasingly important for automotive scene understanding with new algorithms offering constant improvements of the detection performance. However, there is little emphasis on experiences and needs for deployment in embedded environments. We therefore perform a case study of the deployment of two representative object detection networks on an edge AI platform. In particular, we consider RetinaNet for image-based 2D object detection and PointPillars for LiDAR-based 3D object detection. We describe the modifications necessary to convert the algorithms from a PyTorch training environment to the deployment environment taking into account the available tools. We evaluate the runtime of the deployed DNN using two different libraries, TensorRT and TorchScript. In our experiments, we observe slight advantages of TensorRT for convolutional layers and TorchScript for fully connected layers. We also study the trade-off between runtime and performance, when selecting an optimized setup for deployment, and observe that quantization significantly reduces the runtime while having only little impact on the detection performance.

1. Introduction

Nowadays, we witness a great success of AI-based object detection algorithms with deep neural network (DNN) models. These find applications in automotive scene understanding for advanced driver assistance systems and automated driving. Many object detection algorithms are well studied and their performance in development conditions is known in the literature. However, deployment aspects of these DNN models on edge AI devices, and embedded systems in general, are often not addressed in scientific papers but only in blog posts in the form of partly very helpful or incomplete notes and hints. In this paper, we intent to present our major findings when deploying two representative DNN models on a widely used edge AI device, the NVIDIA Jetson AGX Xavier [1]. As representative DNN models, we chose 2D object detection using RetinaNet [2] and 3D object detection using PointPillars [3]. Our findings are presented in a concentrated form and include necessary manipulations of the architecture, that take into account the supported functions of TensorRT or TorchScript.

When deploying DNN models, there are several techniques to optimize the runtime for deployment [4], including (1) the design and manipulation of the DNN model architecture in terms of the model size, depth and width, (2) pruning techniques to remove neurons, groups of neurons, or filters, which have little impact on the output, and (3) quantization to change the numerical representation of data and network weights [5]. These techniques generally result in a trade-off between runtime and performance. In this paper, we consider techniques (1) and (3). On the one hand, we perform experiments with varying resolution of the input image for RetinaNet [2], and with varying the number of pillars and points per pillar for PointPillars [3]. On the other hand, we study the effect of quantization to half-precision floating-point format and to fixed-point arithmetic.

Our contribution is a case study covering the joint treatment of DNN deployment aspects on a widely used edge AI device and an in-depth experimental evaluation of runtime optimization techniques. We are not aware of comparable work. In the following, we provide a description of the used concepts and tools for DNN deployment on the NVIDIA Jetson AGX Xavier, and present the modified architectures of our selected DNNs.

2. Concepts and tools

As it is common in the DNN research community, we develop, train and evaluate our algorithms in a Python environment, where we have chosen the PyTorch library [6]. However, when deploying the network on an embedded system for real-time inference, we move to a C++ environment, in which other tools are needed. In this section, we give a short overview on the concepts and tools that have proven useful to us.

2.1. Network conversion

When deploying a trained DNN on an embedded system in a C++ environment, a conversion becomes necessary. A standardized format to exchange networks within different tools, is the Open Neural Network Exchange (ONNX) [7] format. It can be created using network tracing, which converts the network into a static computational graph based on exemplary input data and allows for efficient hardware acceleration with the NVIDIA TensorRT [8] SDK for runtime optimized inference. However, tracing does not allow data-dependent control-flow operations, so that the original network architecture has to be divided into network graphs and logical operations. These logical operations do not contain network weights and have to be realized with C++ functions. PyTorch offers an alternative built-in framework TorchScript, with the options tracing and scripting. For reasons of comparability and a limited usability of scripting, we focus on tracing with TorchScript as well.

2.2. Quantization

Quantization has the goal to efficiently represent numerical values by a finite number of bits. Widely used formats include single- and half-precision floating-point formats [9], referred to as Float32 and Float16, respectively, and fixed-point arithmetic with 8 bit integers, referred to as Int8. A reduction of the used quantization format can be beneficial in a deployment setup with limited computational, memory or energy resources. Moving from the default Float32 to Float16 is a straightforward and often considered option. When further moving to Int8 quantization, several tuning and calibration steps have to be taken into account. A recent survey of quantization techniques for DNN inference is given in [4]. In [5], a corresponding practical workflow for Int8 quantization is recommended.

For the sake of simplicity, in this paper, we focus on post-training quantization and do not consider quantization-aware training. In particular, for Int8 quantization, we consider the TensorRT MinMax and entropy calibrator functions [10]. The MinMax calibration measures the maximum absolute activation of each layer and provides an equidistant and symmetric mapping. Likewise, the entropy calibration determines a mapping which minimizes the information loss by saturating the activations above a certain threshold. Since TorchScript currently only supports Int8 quantization for CPU usage, we only consider the corresponding TensorRT variant.

2.3. Further tools

We use the Robot operating system (ROS) [11] as our real-time environment in C++. ROS contains helpful tools for sensor data streaming, communication of different nodes and visualization of sensor data and detection results. With the tools nuscenes2bag [12] and kitti2bag [13], we can convert data from automotive datasets into a ROS compatible format that allows us to simulate a real driving scenario while having access to labeled ground truth. For the pre- and post-processing of the data in C++, efficient implementations from libraries like OpenCV [14] for images and OpenPCDet [15] for pointclouds can be used. For many operations, there are also CUDA-based [16] implementations for hardware acceleration in these stages.

3. Deployment architectures

For our case study, we choose to work with two DNN architectures: RetinaNet [2] and PointPillars [3]. Note that we select them as representative and well-known algorithms, that provide a reasonable trade-off between runtime and performance, and are therefore suitable for automotive applications. Further, note that our selection also covers diversity in terms of the object detection task and and the sensor modality.

3.1. RetinaNet

RetinaNet [2] is an image-based 2D object detection algorithm. It is one of the pioneering one-stage object detectors, which surpassed the preceding two-stage networks in terms of runtime while offering similar detection performance. When deploying RetinaNet, we divide the pipeline shown in Figure 1 into the following processing steps:

Pre-processing. In the pre-processing stage, the image is resized to a lower resolution of choice and normalized based on the mean and standard deviation of the RGB images in the ImageNet dataset [17], on which the backbone network is pre-trained. Both of these steps can efficiently be done using the OpenCV CUDA library.

Inference. The architecture of RetinaNet [2] is made out of a ResNet [18] backbone network, a subsequent Feature Pyramid Network (FPN) [19] as well as regression and classification heads. It uses the concept of anchor boxes as pre-defined regions in the image, eliminating the need for

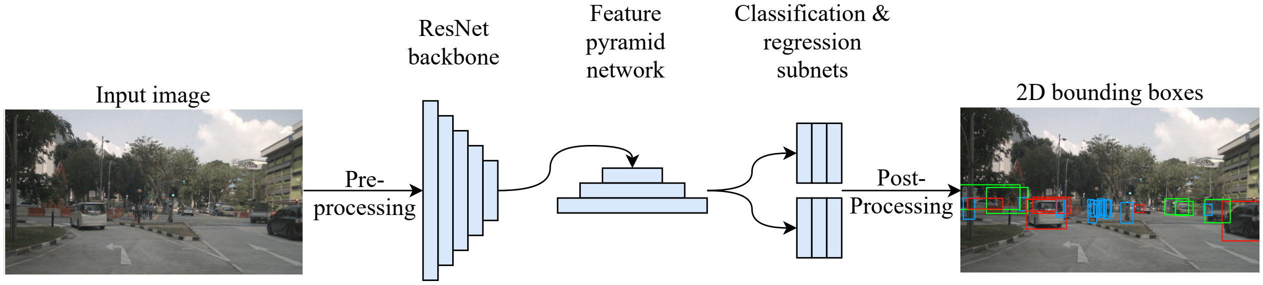

Figure 1. RetinaNet processing pipeline.

a Region Proposal Network. All of these steps can be realized with a single neural network graph, that can be created by tracing the computations of an exemplary network input. Assuming that all pre-processed images have the same resolution, the anchor boxes can be included in the graph as a constant tensor. The graph can then be efficiently deployed using either TensorRT or TorchScript.

Post-processing. In the post-processing stage, the neural network classification and regression outputs are filtered by a detection threshold and decoded to generate the 2D bounding boxes. Finally, non-maximum suppression (NMS) is applied to avoid multiple boxes per object. Again, this can be efficiently computed using the OpenCV library. Note that these steps can not be included in the network graph, as they require logical operations that depend on the network input but would be treated as constant by the tracer.

3.2. PointPillars

PointPillars [3] is a LiDAR-based 3D object detection network that achieves state-of-the-art performance on public benchmarks with real-time inference speed. As depicted in Figure 2, PointPillars mainly consists of the following four blocks:

Pre-processing. In this stage, the input point cloud is first discretized into a set of pillars in the xy plane. Then, each point in a pillar is augmented with its offsets from the arithmetic mean of all points in this pillar and its offsets from the pillar center, resulting in a D-dimensional point encoding. For each pillar, we random sample N points if it has more than N points or apply zero-padding to fix the size of the input tensor. Similarly, the number of non-empty pillars per point cloud is kept to P using the aforementioned policy. Consequently, the augmented input feature is of size (P, N, D). The pre-processing block is deployed efficiently with parallel processing on GPU via the CUDA library.

Feature Extraction. This block aims to extract high-level features from the input feature obtained in pre-processing, and further form them in a 2D top-view representation. To this end, each point in pillars is first consumed by a simplified PointNet [20], outputing a tensor of size (P, N, C). Then, a max operation along the N axis is applied to generate pillar-wise feature, resulting in a (P, C) sized tensor. Finally, these pillar-wise features are scattered back to the pillar locations on the xy plane to create a top-view representation. In this work, we name the combination of the simplified PointNet and the max operation as Pillar Feature Network (PFN), and deploy it using TensorRT or TorchScript network graphs. The scatter operation is realized by C++ functions using the CUDA library.

Backbone and Detection Head. The backbone used in PointPillars [3] is common 2D CNN and consists of a top-down module that gradually captures higher semantic information and a second module that performs upsampling and feature concatenation. The final feature map is processed by the detection head for anchor classification, box offsets regression and direction regression. During deployment, the backbone and detection head are jointly realized by TensorRT or TorchScript network graphs.

Post-processing. To get the final 3D bounding boxes, the predicted box offsets and directions are decoded together with the anchor information. Then, NMS is applied to select the best predictions out of overlapping boxes. We realize these two logical operations with CUDA-based C++ functions to leverage parallel processing on GPU.

4. Experiments

We train and evaluate our algorithms in a Python environment using the PyTorch library. As training and validation data, we use the nuScenes dataset [21] for RetinaNet [2] and the KITTI dataset [22] for PointPillars [3]. We deploy the networks in a C++ environment with ROS for data streaming and visualization. We measure the algorithm runtime as the average runtime across the validation data. We experiment with different tools for hardware acceleration and quantization on the target platform, TensorRT and TorchScript. For the inference with TensorRT, we first export the trained PyTorch model to ONNX and parse it to an optimized TensorRT runtime engine in a C++ environment on the target system. TensorRT allows us to select the desired quantization when building the engine, with the options of Float32, Float16, and Int8. The latter

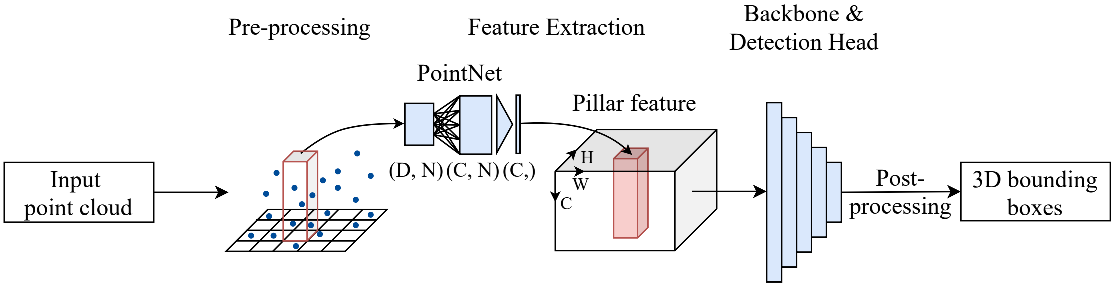

Figure 2. PointPillars processing pipeline.

requires a use-case specific calibration, which can be done with the MinMax or entropy technique. With TorchScript, we need to select the quantization before exporting the PyTorch model. Currently, it is only possible to run Float32 and Float16 calculations with hardware acceleration. In addition, we experiment with varying the input image resolution for RetinaNet and the number of pillars and points per pillar for PointPillars [3]. Finally, we study the impact of the available power supply on the runtime of our entire detection pipeline.

4.1. RetinaNet

Experimental Setup. For our experiments, we choose to work with RetinaNet-18 [2], which is based on a small Resnet-18 [18] backbone to account for our real-time requirements. We train and evaluate the network with the nuScenes [21] *train* and *val* dataset, respectively. To adapt the original dataset for the task of 2D object detection, we project the original 3D bounding boxes onto the image plane and map the 27 classes to a reduced set of *Car*, *Pedestrian*, *Truck*, *Motorcycle* and *Bicycle*. We list both the mean average precision (mAP), with IoU threshold of 0.5, and the weighted mAP, which takes the number of objects per class as weights and is more robust to changes in rare classes.

Framework Analysis. In the first experiment, we study the runtime of RetinaNet [2] when deploying it with TensorRT and TorchScript, using the available quantization techniques. We consider two different inference batch sizes, corresponding to a single image and a full surround view of six images, as present in autonomous vehicle prototypes or the used nuScenes dataset. Table 1 shows the results of this experiment. When running the model with the default Float32 values, we can observe that the inference with TensorRT is substantially faster than with TorchScript, which underlines its optimization capabilities for convolutional layers. When using Float16 precision, we observe a significant reduction of the inference time across both tools, while TensorRT still allows for faster inference. With Int8 quantization in TensorRT, we again significantly reduce the runtime, resulting in over four times faster inference relative to Float32. Some studies have shown that the runtime reduction can have even greater effects when working with larger batch sizes, offering more room for optimization [10]. In our experiment, we can confirm this trend when running inference on images from all six cameras in the nuScenes dataset in a single batch, observing the reduction factors slightly rising for Int8 quantization with TensorRT.

Runtime-Performance Analysis. Of course, we are not only interested in the runtime reduction but rather in the trade-off between runtime and detection performance. In the next experiment, we therefore study the trade-off achievable with quantization versus changing the resolution of the input image. The results of this experiment are shown in Table 2. We compare low, mid and high resolution, which are chosen to have about twice as many pixels as the preceding resolution. Note that the number of anchor boxes increases proportional to the number of pixels. All models have been trained with input data down-scaled to the desired resolution. The mid resolution is the same that we used in the first experiment and we chose to work with TensorRT for this experiment due to the lower runtime and availability of Int8 quantization.

The runtime reduction factor offered by quantization is similar in all dimensions with almost three times faster inference with Float16 and about four times faster inference with Int8. Interestingly, the impact of Float16 quantization on the performance is minimal, both in terms of mAP and weighted mAP. This also holds for Int8 quantization, where the performance is slightly reduced for the MinMax calibrator. We assume that this is due to the 8-bit RGB images as input data, eliminating the need for higher-precision calculations. The impact of the image resolution is high on both runtime and detection performance, where the runtime decreases proportional to the number of pixels. We therefore argue that the mid resolution model with Int8 quantization offers the best trade-off between runtime and performance.

Power Supply Analysis. In a final experiment for RetinaNet [2], we study the impact of the available power on the runtime of the model. We select the mid resolution model and Int8 quantization from the previous experiment due to its good runtime-performance trade-off. All experiments so far have been conducted with the MAXN power mode of the NVIDIA Jetson AGX, which corresponds to roughly 50 W. However, in a practical deployment setup, the power resources are usually limited. As shown in Table 3, reducing the power mode has a high impact on the resulting runtime of the entire detection pipeline. We therefore list detailed results of pre-processing, inference and post-processing run-times. Depending on the use-case, a trade-off between power supply and runtime has to be made.

Summary. In conclusion, the experiments show that TensorRT achieves lower inference times for RetinaNet [2] that TorchScript. While quantization should always be considered for deployment, lowering the resolution can help to further reduce the runtime, but at the cost of a decreased detection performance. It is therefore advisable to prefer a mid to high resolution and Int8 quantization over a low resolution and Float32 precision. The runtime is also heavily influenced by the available power supply. A video showing qualitative results on a nuScenes sequence obtained by this detection pipeline is available at:
https://youtu.be/iPoC0YTsiqg.

Table 1. Runtime evaluation of RetinaNet with TensorRT and TorchScript using various quantization techniques and batch sizes.

Batch size	Quant.	TensorRT	TorchScript
1	Float32	104 ms	210 ms
	Float16	38 ms	67 ms
	Int8	25 ms	-
6	Float32	619 ms	1041 ms
	Float16	201 ms	271 ms
	Int8	136 ms	-

Table 2. Performance and runtime evaluation of RetinaNet using different image resolutions and quantization techniques on the nuScenes val set.

Resolution	Quant.	Runtime	mAP	weighted mAP
Low 416×736	Float32	60 ms	0.298	0.391
	Float16	22 ms	0.299	0.391
	Int8, Entropy	16 ms	0.298	0.391
	Int8, MinMax		0.288	0.381
Mid 576×1024	Float32	104 ms	0.361	0.455
	Float16	38 ms	0.356	0.454
	Int8, Entropy	25 ms	0.355	0.455
	Int8, MinMax		0.355	0.454
High 832×1472	Float32	224 ms	0.395	0.488
	Float16	74 ms	0.395	0.487
	Int8, Entropy	50 ms	0.393	0.485
	Int8, MinMax		0.388	0.483

4.2. PointPillars

Experimental Setup. We train PointPillars [3] and then evaluate the performance of the deployed network on the KITTI [23] 3D object detection dataset, which contains 7481 image and point cloud pairs with available 3D bounding box annotations for three categories: *Car*, *Pedestrian* and *Cyclist*. For the experimental evaluation, we follow [24] and split the samples into a *train* and *val* set with 3712 and 3769 samples, respectively. The performance is evaluated using Average Precision (AP) in moderate level for the 3D object detection task with Intersection over Union (IoU) thresholds of 0.7 for *Car* and 0.5 for *Pedestrian* and *Cyclist*. To construct and train the model, we use the same KITTI-PointPillars configuration as in the OpenPCDet codebase [15].

Table 3. Detailed runtime of each block in the optimal RetinaNet detection pipeline using different power modes.

Block	MAXN	30W	15W	10W
Pre-process	3 ms	5 ms	6 ms	7 ms
Inference	24 ms	35 ms	45 ms	94 ms
Post-process	4 ms	4 ms	5 ms	6 ms
Total	31 ms	44 ms	56 ms	107 ms

For the deployment on the target platform, we implement the pipeline based on Autoware [25] by developing additional features. The runtime measurement is conducted on the target platform, if not specified otherwise, with the MAXN power mode using the sequence 0095 with 236 frames in the KITTI raw data [22]. We average the runtime starting from the tenth frame to consider the warm-up of GPU on the target platform and thus eliminate occasionality.

Runtime Analysis. The runtime results for both Pillar Feature Network (PFN) and Backbone & Detection Head (named as 2D CNN for short) when deployed with TensorRT and TorchScript using various quantization techniques are presented in Table 4. When comparing the runtime of the model deployed using TensorRT and TorchScript, PFN and 2D CNN show opposite trends. For the 2D CNN, TensorRT achieves significantly better runtime optimization, which is consistent with our observations when deploying RetinaNet. In contrast, for the PFN, TorchScript provides an improved runtime optimization. One possible explanation could be slight advantages of TensorRT for convolutional layers and TorchScript for fully connected layers. When comparing the quantization techniques, Float16 offers a significant speed improvement for the PFN and 2D CNN across both tools, whereas Int8 only provides additional minor improvements for the 2D CNN. Note that we do not consider to use Int8 quantization for the PFN, since the input are 3D coordinates, with approximate range $\pm 10^2\,\mathrm{m}$ and accuracy $10^{-2}\,\mathrm{m}$, so that Int8 quantization would result in a significant loss of information.

Performance Analysis. From the runtime analysis, we can conclude that the combination of PFN with TorchScript and 2D CNN with TensorRT achieves the lowest runtime. We therefore consider this setup and further study the object detection performance of the model deployed using different quantization techniques. The results of the performance evaluation for three classes as well as the runtime for the entire pipeline are shown in Table 5. When going down from Float32 to Float16 precision, we observe no significant performance change in terms of AP for all class categories for both PFN and 2D CNN. When using Int8 quantization for the 2D CNN, the performance drops considerably for both calibration methods, where the reduction is even more severe for the entropy calibrator. We assume that the used pillar feature map representation and subsequent activations require a larger arithmetic range to preserve essential geometric information. When further considering the runtime of the complete PointPillars [3] detection pipeline, it is clear that deploying PFN and 2D CNN using Float16 precision with TorchScript and TensorRT, respectively, offers the best trade-off between detection performance and runtime. We name it the optimal variant in the following study.

Further Study on PFN. From the previous analysis we observe that the optimal variant takes in total 41 ms runtime, where the single PFN costs 19 ms, being the bottleneck in the deployment. We thus conduct this experiment to investigate the potential of further reducing the runtime of PFN by modifying the network input. This is achieved by adjusting two parameters of PFN, namely the number of non-empty pillars P, and the number of points per pillar N. While we use $P = 16000$ and $N = 32$ in the previous experiments, we additionally choose $P = 12000$ and $N \in \{24, 16\}$ and measure the runtime of PFN and the overall performance. As shown in Table 6, keeping the number of points while lowering the number of pillars from 16000 to 12000 does not introduce notable change on the performance metrics, while it improves the runtime considerably from 19 ms to 14 ms for the case with 32 points per pillar. Additional runtime boost can be achieved by decreasing the number of points per pillar without remarkable performance loss. In this study, we figure out that the setup with 12000 pillars and 24 points per pillar is the most appropriate for deploying PFN with a runtime of 9 ms.

Summary. From our extensive studies, we summarize that deploying PointPillars [3] using separate PFN and 2D CNN network graphs on Float16 precision with Torchscript and TensorRT, respectively, is the optimal solution for the deployment on the target platform. We further observe that the input parameters of PFN have a significant impact on the runtime, where we find $P = 12000$ and $N = 24$ as most suitable in our study. We finally report the detailed runtime of the optimal PointPillars [3] pipeline using various power modes on the target platform in Table 7. A video showing qualitative results on the KITTI sequence obtained by this detection pipeline is available at:
https://youtu.be/paYXkkXDKGs.

Table 4. Runtime of PFN and 2D CNN deployed with TensorRT and TorchScript using various quantization techniques.

Network	Quant.	TensorRT	TorchScript
PFN	Float32	30 ms	21 ms
	Float16	26 ms	19 ms
2D CNN	Float32	46 ms	82 ms
	Float16	16 ms	31 ms
	Int8	14 ms	-

Table 5. Performance evaluation of PFN and 2D CNN combinations using different quantization techniques on the KITTI val set.

PFN quant.	2D CNN quant.	AP_{3D}			Pipeline runtime
		Car	Ped.	Cyc.	
Float32	Float32	78.40	51.41	62.81	72 ms
Float32	Float16	78.30	51.31	62.89	43 ms
Float32	Int8, minmax	71.04	48.00	55.94	40 ms
	Int8, entropy	68.91	22.03	29.66	
Float16	Float32	78.40	51.46	62.92	71 ms
Float16	Float16	78.31	51.39	62.97	41 ms
Float16	Int8, minmax	70.99	47.65	56.42	38 ms
	Int8, entropy	69.37	21.86	29.53	

Table 6. Performance and runtime evaluation of PFN with different parameters.

Pillar num.	Point num.	AP_{3D}			PFN runtime
		Car	Ped.	Cyc.	
16000	32	78.31	51.39	62.97	19 ms
16000	24	78.28	50.68	62.20	12 ms
16000	16	78.14	50.81	61.33	10 ms
12000	32	78.29	51.37	62.98	14 ms
12000	24	78.25	51.92	62.19	9 ms
12000	16	78.17	49.02	61.38	8 ms

Table 7. Detailed runtime of each block in the optimal PointPillars detection pipeline using different power modes.

Block	MAXN	30W	15W	10W
Pre-process	4 ms	6 ms	8 ms	14 ms
PFN	9 ms	14 ms	21 ms	52 ms
Scatter	1 ms	2 ms	2 ms	4 ms
2D CNN	16 ms	25 ms	32 ms	71 ms
Post-process	1 ms	1ms	2 ms	2 ms
Total	31 ms	48 ms	65 ms	143 ms

5. Conclusion

In this paper, we have presented our major insights when deploying two representative algorithms on the NVIDIA Jetson AGX Xavier for automotive scene understanding: RetinaNet [2] for image-based 2D object detection and PointPillars [3] for LiDAR-based 3D object detection. We have discussed the necessary modifications and tools that we found most helpful. We studied the runtime of TensorRT and TorchScript and found that TensorRT should be preferred for RetinaNet [2] and the convolutional part of PointPillars [3], whereas TorchScript should be preferred for the fully connected part of PointPillars [3]. The runtime of both algorithms can be further reduced by utilizing quantization, which is available up to Int8 with TensorRT and up to Float16 with TorchScript. While the impact on the detection performance for RetinaNet [2] is low even with Int8, PointPillars [3] should only be quantized to Float16, which is possibly due to the difference in input data, with 8-bit RGB images and 3D coordinates of LiDAR point clouds, respectively. We also studied the influence of some design parameters of the algorithms and found that a good runtime-performance trade-off can be achieved with an input resolution of 576×1024 for RetinaNet [2], as well as 12000 pillars and 24 points per pillar for PointPillars [3]. The available power supply in the embedded environment also has a significant impact on the runtime, which additionally has to be considered when choosing a setup for deployment. In future work, we plan to extend our findings on fusion methods between image and LiDAR or radar point cloud data. Additionally, we consider to address pruning techniques for further runtime optimization.

Acknowledgment

The research leading to these results is partly funded by the German Federal Ministry for Economic Affairs and Energy within the project “Methoden und Maßnahmen zur Absicherung von KI basierten Wahrnehmungsfunktionen für das automatisierte Fahren (KI-Absicherung)”. The authors would like to thank the consortium for the successful cooperation.

References

[1] NVIDIA Corporation, “Jetson AGX Xavier.” https://www.nvidia.com/de-de/autonomous-machines/embedded-systems/jetson-agx-xavier/, 2018. [Hardware; accessed 09-July-2021]. 1

[2] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, “Focal Loss for Dense Object Detection,” in *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2999–3007, 2017. 1, 2, 3, 4, 5, 6, 7

[3] A. H. Lang, S. Vora, H. Caesar, L. Zhou, J. Yang, and O. Beijbom, “PointPillars: Fast encoders for object detection from point clouds,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 12689–12697, 2019. 1, 2, 3, 4, 5, 6, 7

[4] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, “A Survey of Quantization Methods for Efficient Neural Network Inference,” 2021. arXiv:2103.13630. 1, 2

[5] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, “Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation,” 2020. arXiv:2004.09602. 1, 2

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, “PyTorch: An imperative style, high-performance deep learning library,” *arXiv preprint arXiv:1912.01703*, 2019. 2

[7] J. Bai, F. Lu, K. Zhang, *et al.*, “ONNX: Open Neural Network Exchange.” https://github.com/onnx/onnx, 2019. [GitHub repository; accessed 09-July-2021]. 2

[8] H. Vanholder, “Efficient Inference with TensorRT,” in *GPU Technology Conference*, vol. 1, 2016. 2

[9] “IEEE Standard for Binary Floating-Point Arithmetic,” *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985. 2

[10] S. Migacz, “8-bit Inference with TensorRT,” in *GPU Technology Conference*, vol. 2, p. 5, 2017. 2, 4

[11] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, “ROS: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009. 2

[12] V. Comito, “Nuscenes2bag.” https://github.com/clynamen/nuscenes2bag, 2019. [GitHub repository; accessed 09-July-2021]. 2

[13] T. Krejci, “Kitti2bag.” https://github.com/tomas789/kitti2bag, 2019. [GitHub repository; accessed 09-July-2021]. 2

[14] G. Bradski, “The OpenCV Library,” *Dr. Dobb’s Journal of Software Tools*, 2000. 2

[15] OpenPCDet Development Team, “OpenPCDet: An Open-source Toolbox for 3D Object Detection from Point Clouds.” https://github.com/open-mmlab/OpenPCDet, 2020. 2, 5

[16] D. Luebke, “Cuda: Scalable parallel programming for high-performance scientific computing,” in *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pp. 836–838, 2008. 2

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” *Advances in Neural Information Processing Systems*, vol. 25, pp. 1097–1105, 2012. 2

[18] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016. 2, 4

[19] T.-Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan, and S. Belongie, “Feature Pyramid Networks for Object Detection,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2117–2125, 2017. 2

[20] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 652–660, 2017. 3

[21] H. Caesar, V. Bankiti, A. H. Lang, S. Vora, V. E. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom, “nuScenes: A multimodal dataset for autonomous driving,” 2020. arXiv:1903.11027. 3, 4

[22] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The KITTI dataset," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1231–1237, 2013. 3, 5

[23] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite," in *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3354–3361, IEEE, 2012. 5

[24] J. Fei, W. Chen, P. Heidenreich, S. Wirges, and C. Stiller, "SemanticVoxels: Sequential Fusion for 3D Pedestrian Detection using LiDAR Point Cloud and Semantic Segmentation," in *IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pp. 185–190, 2020. 5

[25] S. Kato, S. Tokunaga, Y. Maruyama, S. Maeda, M. Hirabayashi, Y. Kitsukawa, A. Monrroy, T. Ando, Y. Fujii, and T. Azumi, "Autoware on board: Enabling autonomous vehicles with embedded systems," in *2018 ACM/IEEE 9th International Conference on Cyber-Physical Systems (ICCPS)*, pp. 287–296, IEEE, 2018. 5