Repercussions of Using DNN Compilers on Edge GPUs for Real Time and Safety Critical Systems: A Quantitative Audit

OMAIS SHAFI, Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi, India
MOHAMMAD KHALID PANDIT, Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi, India
AMARJEET SAINI, Department of Computer Science and Engineering
Indian Institute of Technology, Dharwad, Karnataka, India
GAYATHRI ANANTHANARAYANAN, Department of Computer Science and Engineering
Indian Institute of Technology, Dharwad, Karnataka, India
RIJUREKHA SEN, Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi, India

Rapid advancements in edge devices has led to large deployment of deep neural network (DNN) based workloads. To utilize the resources at the edge effectively, many DNN compilers are proposed that efficiently map the high level DNN models developed in frameworks like PyTorch, Tensorflow, Caffe etc into minimum deployable lightweight execution engines. For real time applications like ADAS, these compiler optimized engines should give precise, reproducible and predictable inferences, both in-terms of runtime and output consistency. This paper is the first effort in empirically auditing state of the art DNN compilers viz TensorRT, AutoTVM and AutoScheduler. We characterize the NN compilers based on their performance predictability w.r.t inference latency, output reproducibility, hardware utilization. etc and based on that provide various recommendations. Our methodology and findings can potentially help the application developers, in making informed decision about the choice of DNN compiler, in a real time safety critical setting.

CCS Concepts: • **Computer systems organization → Embedded systems**; • **Neural Networks → Optimizing inference**.

Additional Key Words and Phrases: Deep neural network compilers, Performance predictability, Output reproducibility

Authors' addresses: Omais Shafi, Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi, India, omais.shafi@cse.iitd.ac.in; Mohammad Khalid Pandit, Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi, India, khalid@cse.iitd.ac.in; Amarjeet Saini, Department of Computer Science and Engineering
Indian Institute of Technology, Dharwad, Karnataka, India, amarjeets167@gmail.com; Gayathri Ananthanarayanan, Department of Computer Science and Engineering
Indian Institute of Technology, Dharwad, Karnataka, India, gayathri@iitdh.ac.in; Rijurekha Sen, Department of Computer Science and Engineering
Indian Institute of Technology, New Delhi, India, riju@cse.iitd.ac.in.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© Association for Computing Machinery.
XXXX-XXXX/2/-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Omais Shafi, Mohammad Khalid Pandit, Amarjeet Saini, Gayathri Ananthanarayanan, and Rijurekha Sen. 2. Repercussions of Using DNN Compilers on Edge GPUs for Real Time and Safety Critical Systems: A Quantitative Audit. , (2), 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Deep neural networks (DNNs) are increasingly becoming core components of safety critical applications such as advanced driver assistance systems (ADAS) [14, 24, 35], industrial control [8] etc. Data streams from sensors like LiDAR, RADAR and multiple cameras are fed as inputs to various DNNs that process the sensor inputs to orchestrate critical functionalities. For example, traffic sign detection, lane departure warning etc. are important tasks performed by DNNs using camera inputs, in an ADAS enabled car.

DNN models are trained offline, on powerful servers using large datasets, to execute certain tasks. Typical DNN tasks comprise detecting multiple objects and creating bounding boxes around them in an image (e.g. for locating pedestrians in front of a self driving car), or classifying the object present in an image (e.g. for reading whether a road sign allows U turn in a self driving car). Such a trained DNN model, after deployment in the application scenario, keeps performing the task it is trained for, on unseen test data. This post-deployment continuous working of a DNN model is called inference. We perform a quantitative audit of DNN inferences in this paper, as inferences need to be fast, accurate and predictable, for appropriate functioning of the underlying real time safety critical system.

DNN adoption in different cyber-physical systems has resulted in a tremendous spurt of hardware accelerator production, for fast execution of these DNNs in constrained deployment scenarios (within a self driving car, at an intelligent traffic intersection controller etc.). Emerging processors and hardware accelerators, that can run DNN inferences, include CPU (Central Processing Unit), GPU (Graphics Processing Unit), TPU (Tensor Processing Unit), NPU (Neural Processing Unit), DPU (Data Processing Unit), FPGA (Field Programmable Gate Array), DSP (Digital Signal Processor) etc. We specifically examine DNN inferences on embedded GPU platforms from Nvidia, which due to their excellent performance per watt characteristic, are the dominant hardware platforms adopted in automotive and other safety critical industries [7, 8].

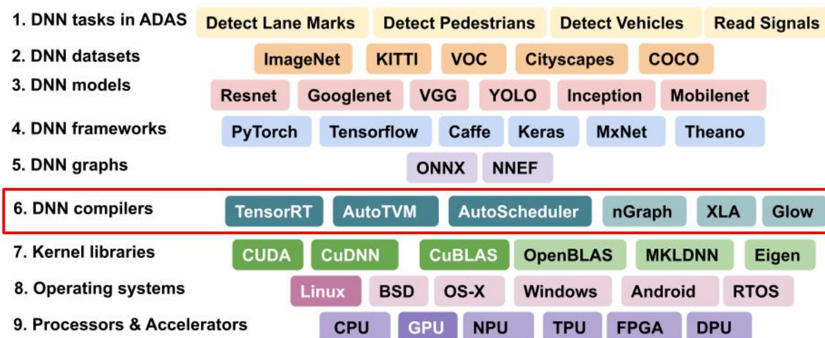


Fig. 1. Hardware and software eco-system for DNN execution in real time applications like ADAS [14, 24, 35] and industrial control [8].

Figure 1 visually depicts the hardware and software eco-system for DNN inferences in real time applications like ADAS. The first row shows some sample tasks that DNNs are typically responsible for. We use such tasks in this paper, as sample DNN tasks with real time requirements. The next three rows show the components relevant for DNN training: row 3 shows many possible DNN

model architectures, that can be trained using one or more datasets listed in row 2, using one of the DNN frameworks for model training listed in row 4. Row 5 shows intermediate compute graph representations of a trained DNN model. The last row shows emerging hardware processors and accelerators, for DNN inferences in constrained environments. GPU is highlighted, as the experimental platform used in this paper. We run the Linux operating system on our Nvidia embedded GPUs, and use highly optimized kernel functions from Nvidia's extensive libraries (CUDA, CuDNN, CuBLAS highlighted in row 7). These library functions perform linear algebra and other mathematical operations like FFT, matrix multiplications and convolutions, for efficient DNN inference execution using GPU cores.

The novelty of this paper lies in its unique characterization of row 6 in Figure 1, which lists a variety of DNN compilers. This is a fast growing set of software tools, which ❶ takes as input pre-trained DNN models in different formats like ONNX, ❷ optimizes the DNN model to maintain accuracy at much lower execution time, and ❸ maps the operations of the optimized model to hardware processor functions (with or without using existing libraries listed in row 7). This layer of the software stack is becoming increasingly important with emergence of new hardware platforms, and fast growth of new DNN models for an expanding set of real time applications. DNN compilers seek to reduce the manual overhead of optimizing *emerging DNN models* for *evolving hardware platforms*, by automating some of the optimization and hardware mapping steps.

Hardware vendors are aggressively working on commercializing DNN compilers for their specific hardware platforms, so that adopting their processor or accelerator offering becomes easier for application developers. Intel's OpenVino™[4] and nGraph [18], Nvidia's TensorRT™[9], ARM's Compute Library[1] and Qualcomm's SNPE[10], are such hardware specific DNN compilers. Software giants like Google and Facebook are also coming up with DNN compilers such as XLA [12], TfLite [11] and Glow [3], that can optimize pre-trained DNN models for a variety of hardware platforms. There are also open-source offerings in the field of DNN compilers, namely AutoTVM [17] and AutoScheduler [38], which produce optimized DNN models for multiple hardware processors and accelerators. While DNN compilers are increasingly being adopted by application developers, there are surprisingly no prior quantitative audits of these software tools, in the context of safety critical real time applications. We seek to bridge this important research gap in real time systems literature, through careful experimentation and empirical analysis.

The optimized model created by the DNN compiler, is henceforth termed as DNN execution engine. We frame a set of important metrics to characterize the DNN compilers and their compiled engines — ❶ do the DNN engines perform accurate inferences? ❷ Given the same input, do the DNN engines give same inference output, i.e. are the inference results reproducible? ❸ Do inferences have low latency? ❹ Are inference latencies predictable? ❺ Do inferences optimally use hardware resources, as promised by all DNN compiler vendors? ❻ Is the DNN engine building time using the DNN compiler too long? We believe characterizing the DNN compilers based on these metrics, and analyzing the trade-offs across these metrics, is vitally important, before inclusion of the compiled engines in a safety critical system.

We make the following three important contributions in this paper.

❶ We identify the research gap in quantitative audit of DNN compilers, a fast growing software tool adopted increasingly by application designers in safety critical domains. We define new metrics to characterize and compare DNN compilers, before including their compiled engines in any cyber-physical system.

❷ Using real hardware platforms with Nvidia edge GPU and server GPUs, a wide variety of DNN models, and three state-of-the-art DNN compilers, we perform an extensive empirical analysis of the compilers based on our defined metrics. To the best of our knowledge, this is the first

work that does a in-depth analysis of all these DNN compilers with a lot of interesting insights based on the various metrics. We make several novel observations such as: (a) DNN compiler from hardware vendor manufacturing GPU, gives slower engines with worse resource utilization than open source compilers, refuting the intuition that hardware vendors know their hardware best and therefore can squeeze maximum performance from them. (b) None of the DNN compilers are deterministic, i.e. the same DNN model compiled and optimized multiple times by the same compiler, will generate different engines with varying execution times. (c) Not only does inference latency vary across engines, the inference output can vary too i.e. two DNN engines compiled using the same pre-trained model, can give different results on the same input image. We also analyze the root causes behind these non-intuitive observations.
❸ We use the observations and results from the above experiments to quantify the impact of using various DNN compilers on two real time safety critical applications, namely intelligent traffic intersection control and ADAS, and also examine dependence of WCET for these applications on DNN compiler choice. Our work will potentially help the application designers in the right choice of DNN compilers that can be used in a safety critical scenario.

Organization of the paper: The rest of the article is organized as follows. Section 2 describes the NN compilers examined by us. Section 3 discusses the hardware setup and NN models used in our experiments. Section 4 examines DNN inference output correctness and reproducibility. Section 5 analyzes the inference and compilation latencies along with the predictability of inference latencies across different engines of NN models. Section 6 provides the impact of our findings on real time applications, with WCET analysis. Section 7 illustrates the related work. We give final recommendations and conclude the paper in Section 8.

2 Neural Network Compilers

In this Section, we describe the three Neural Network (NN) compilers which we have used for our evaluation - namely TensorRT [9], AutoTVM [17] and Autoscheduler [38]. These take pre-trained DNN models as input and generate highly optimized executables for the inference. Figure 2 shows the overall steps of the three NN compilers used. We next discuss each one of them in detail.

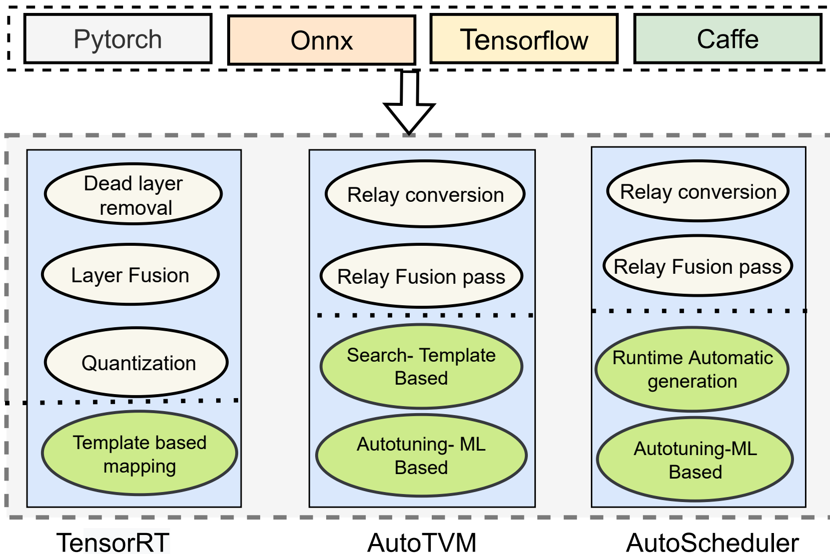


Fig. 2. TensorRT, AutoTVM and AutoScheduler optimizations

TensorRT: TensorRT [9] is an Nvidia tool used for DNN inference optimizations, specialized for Nvidia GPUs. TensorRT performs the following steps on a given pre-trained model: ❶ Removing unused layers ❷ Layer fusion, so that sequential operations are performed in a single kernel launch thereby reducing the overhead of reading and writing tensor for individual kernel launch. TensorRT identifies layers in the graph with common input data and filter size and merge them as single layer such that it is launched as one kernel. ❸ Quantization where the `32 bit` floats are quantized to `16 bit` half floats or `8 bit` integers, to make engines smaller and faster. ❹ Kernel mapping where optimized NN layers are mapped to manually written highly optimized CUDA kernels.
AutoTVM: AutoTVM [17] optimizations can be leveraged on any hardware back-end like GPU (Nvidia as well as others), CPU. FPGA etc. AutoTVM optimizations include ❶ *Relay conversion* where the unoptimized model is converted to TVM's internal high level graph format known as *Relay* ❷ *Relay Fusion Pass* where TVM applies some high-level optimizations to the graph at the Relay level, then lowers it into a low-level IR called Tensor Expressions (TE). At the TE level, the computational graph is split into a set of subgraphs that the TVM engine determines are good optimization targets ❸ *Template based searching* where TVM finds the order of operations for computational tasks in the graph for fastest inference time ❹ *ML based cost model* guides the search across pre-defined templates to be optimized for the chosen hardware platform. TVM constructs a search space over candidate programs based on manual templates, and runs one of the meta-heuristic based search algorithms (such as XGBoost, GA etc) with ML based cost model guiding the search for a particular hardware back-end, to find the best schedule.
AutoScheduler: AutoTVM's search approach improves optimization of the executable by a significant amount. However developing templates requires substantial manual effort, and therefore AutoTVM covers limited program structures as manually enumerating all optimization choices for all operators as templates, is prohibitive. AutoScheduler based on Ansor [38], aims at fully automating the scheduler for generating the high performance code for tensor computations without manual templates. It is an improved version of AutoTVM which automatically generates the search space for small sub graphs, eliminating the need to manually develop templates. It then samples complete programs from the search space and performs fine-tuning on complete programs. AutoScheduler avoids the significant manual effort required in adding the code for the templates. For instance, the code repository of TVM already contains more than 15K lines of code for the given set of templates. This number continues to grow as new operators and new hardware platforms emerge.
NN compiler paramaters: We use `FP16` quantization across all the compilers for our evaluation. We set the number of iteration(N) as 200 and the searching algorithm as *Genetic Algorithm* for both AutoTVM and AutoScheduler. The number of iterations set is less than the recommended value as these embedded boards are resource constrained devices and take a significant amount of time if the number of iterations are set to a larger value. We show a representative execution by increasing the number of iterations to 20,000 on V100 DGX, which is a server GPU. Increasing the number of iterations improves the performance and will not impact our other observations based on smaller values. Since TensorRT does not provide us knobs for the searching, we keep it with the default setting.

There are some other NN compilers such as XLA [12], Glow [3], TC [31], nGraph [18]. However, as examined by us, they have limited support either on the software side or the hardware which they support. For instance, TC does not provide any quantization support and nGrpah support only `INT8` quantization. Further, nGraph supports only Intel GPU/CPU. XLA is integrated into the Tensorflow framework that has a more intricate runtime compared to TVM. Therefore, in our evaluation, we choose well known, widely used and supported NN compilers - TensorRT, AutoTVM and AutoScheduler.

3 Experimental Setup

In this Section, we enumerate the NN models and hardware platforms used in the experiments.
NN Models: We use multiple popular DNN models from the literature, regularly used in edge inferences, for two classes of computer vision tasks - *image classification models*, *object detection models*. We list these models[1], along with their number of learned parameters (in millions) and frameworks associated with training each network in Table 1. These wide variety of NN models are needed to better characterize the optimized engines, as layer fusion, quantization and other optimizations might affect the different NN model architectures in different ways.

NN Model	Vision Task	# Learned Parameters
Resnet-18 [37]	Classification	11M
Resnet-50 [23]	Classification	25.6M
Squeezenet [21]	Classification	1.2M
Vgg-16 [29]	Classification	138.4M
Inception_v3 [34]	Classification	23.9M
Mobilenet_v2 [19]	Classification	3.5M
Googlenet [30]	Classification	7M
Densenet [20]	Classification	8.1M
Yolov2 [36]	Detection	8.85M
Retinanet [15]	Detection	23.6M

Table 1. Neural network models used in the study

Hardware Setup: In a typical ADAS setup there are a number of cameras installed in a car capturing input feeds from different angles. These data generators (typical edge devices) forward these frames to edge GPUs installed in the car for inference. We evaluate the performance of NN models on two such edge GPUs based on the Volta architecture - Jetson Xavier NX^{TM} [6] and Jetson Xavier AGX^{TM} [5]. Such an architecture is common in any resource constrained cyber physical system setup. We use a 384 core Xavier NX and 512 core Xavier AGX, with correspondingly larger numbers of SMs and tensor cores, and RAM size in AGX (refer to Table 2) obtained using *deviceQuery* [2] utility available on both the boards. It is worth noting that both of these embedded boards have DLAs (Deep Learning Accelerators) onboard, which can be used to offload some jobs from the GPU. However, because GPUs are more commonly available in edge devices than DLAs, we characterize GPUs (excluding DLAs) for the execution of neural networks in this paper. On the systems software part, we use an Ubuntu 18.04 Operating System, TVM v0.9, LLVM v13.0 along with cuda 10.1 for our experimentation.

4 Are Inferences Accurate and Reproducible?

Real time applications are often safety critical. Using NN models for inferencing the object class in an image or detecting bounding boxes for important objects in an image, should therefore be as accurate as possible, for ensuring the reliability of the safety critical system. Accuracy depends on the NN model architecture, the richness of the training dataset, the number of iterations or epochs for which NN model training is conducted, and the similarity between the training data and the unseen test data distributions. These factors affecting NN model accuracy during training are beyond the scope of this paper, as we work with pre-trained models. Our focus instead is on

[1]https://pytorch.org/vision/stable/models.html

	Xavier NX	Xavier AGX
CPU	6-core NVIDIA Carmel ARM®v8.2 64-bit CPU 6MB L2 + 4MB L3	8-core ARM® v8.2 64-bit CPU 8MB L2 + 4MB L3
# GPU cores	384 (64 per SM)	512 (64 per SM)
# SMs	6	8
# Tensor cores	48 (8 per SM)	64 (8 per SM)
L1 cache	128KB per SM	128KB per SM
L2 cache	512KB	512KB
Memory	8GB 128-bit LPDDR4x 51.2GB/s	32GB 256-bit LPDDR4x 137GB/s
GPU Clock	1.1 GHz	1.137 GHz

Table 2. Evaluation platforms with NVIDIA GPU

different NN compilers that optimize these pre-trained NN models. We examine whether the NN compilers maintain the accuracy of the pre-trained models after optimization, and also whether each inference result for a given input image is reproducible or deterministic for different optimized engines.

4.1 Do optimizations retain the pre-trained model accuracy?

As discussed in Section 2, the NN compilers perform optimizations like quantization, pruning, layer-fusion etc., which might lead to accuracy drop of the pre-trained unoptimized models. We compare the accuracy of different optimized models obtained using the NN compilers, vs. the accuracy of the unoptimized models, in Table 3. We use the *Imagenet* dataset [26] consisting of 1000 classes, for this inference accuracy and reproducibility analysis. To ensure fair comparison, we use the same `16-bit` quantization for all three compilers.

NN model	Unoptimized	TensorRT	AutoTVM	AutoScheduler
Resnet-18	71.96	71.96	70.46	70.46
Resnet-50	78.82	78.86	77.20	76.72
Squeezenet	61.32	61.24	58.26	58.26
Mobilenet_v2	73.60	73.56	71.68	71.68
Vgg-16	73.88	73.92	72.16	72.10
Googlenet	72.42	72.42	70.0	70.0
Densenet	79.70	79.74	77.90	77.90

Table 3. Top-1 Accuracy Comparison (in %) across different NN compiled models and unoptimized NN models.

We observe from the table that the unoptimized models have the highest accuracies, as expected. However, the TensorRT optimized models are able to almost maintain the same accuracy as unoptimized model. The other NN compilers, AutoTVM and AutoScheduler, have only a slight accuracy drop of 1-3%, compared to the unoptimized models.

4.2 Do optimized engines give consistent output on same input?

We next discuss the consistency of the output labels across the different engines of the NN compilers. We create three engines of the same model for each NN compiler. We then check the output

mismatches across the three engines. While comparing mismatches across engine1-engine2, we say there is a mismatch when engine 1 results in one predicted output for an image and engine 2 results in other predicted output for the same input image. We similarly count mismatches between engine2-engine3 and engine3-engine1.

Top-1 accuracy reproducibility in TensorRT: We tabulate the results of output mismatches across engines for TensorRT in Table 4. Though the pairwise mismatch values comprise only 0.1-0.8% of the total number of predictions, these results show that the TensorRT compiler does not guarantee the same output compared to the original model, or across different TensorRT engines for the same input image, even if all engines are built from the exact same pre-trained NN model on the same hardware platform. Similar observations have been reported in our prior work [27].

NN Model	Engines1-2	Engines2-3	Engines1-3
NX-Resnet-18	105	105	0
AGX-Vgg-16	269	0	269
AGX-Inception_v3	461	296	497
AGX-Resnet-18	243	224	183

Table 4. Top-1 mismatches across TensorRT engines

Top-5 accuracy reproducibility in TensorRT: Table 5 shows that there are no output mismatches when we take top-5 accuracy across the engines. From this we infer, that an output prediction done by engine 1 lies in top-5 output predictions done by engine 2.

NN Model	Engines1-2	Engines2-3	Engines1-3
Resnet-18	0	0	0
Vgg-16	0	0	0
Inception_v3	0	0	0

Table 5. Top-5 consistency across TensorRT engines

To understand why top-1 accuracy is not perfectly reproducible between engine 1 and engine 2, while top-5 accuracy is, we next check the probability values of output predictions generated by engine 1 and engine 2. We compute the difference of probability values of engine 1 and engine 2 for all the output mismatches for a representative execution of Resnet-18 and plot a histogram as shown in Figure 3. We observe that the difference in the probability values of output predictions for engine 1 and engine 2 lies in the ballpark of only 0.001-0.007. This slight change in class probability values, however, cause the engines to give different output labels for the same input image. The change in output, with same input and same pre-trained model, might critically hamper the predictability requirements of any safety critical real time system.

Is quantization responsible for output probability change?

The above results of top-1 output label mismatches in Table 4 use FP16 quantized TensorRT engines. Table 6 additionally shows the same results for FP32 quantized TensorRT engines. We observe that even with no quantization and the same FP32 precision as that of the pre-trained model original weights, there are still significant output mismatches. Thus removing quantization and using FP32 precision will not make TensorRT engines give reproducible outputs on a given input. It will only significantly increase the runtime.

Caching Technique for maintaining output consistency in TensorRT: We present a technique for reproducible TensorRT engines called caching, wherein exact same layer to kernel mapping

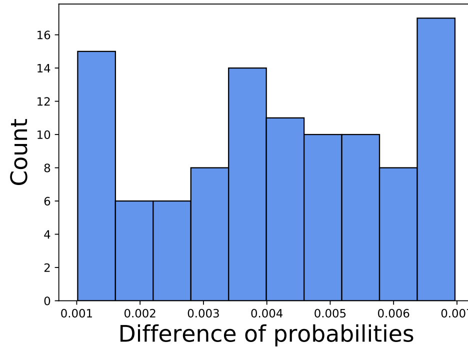


Fig. 3. Small probability differences for output predictions for engine1 and engine2 in Resnet-18.

NN Model	Engines1-2	Engines2-3	Engines1-3
NX-Resnet-18	87	85	89
AGX-Vgg-16	223	223	0
AGX-Inception_v3	275	161	347
AGX-Resnet-18	151	119	95

Table 6. Output mismatches for FP32 TensorRT engines

is maintained across engines. It helps mitigate the output inconsistency problem in TensorRT. We dumped the layer to kernel mapping information into the file `AlgorithmCache.txt`. Upon rebuilding the engine, instead of using the dynamic layer to kernel mapping (default in TensorRT), we use mapping information from `AlgorithmCache.txt` file. This results in generation of reproducible inference engines in TensorRT. Table 7 shows that the output mismatches do not arise if we generate engines using the above method making the output determinism realizable in TensorRT engines.

NN Model	Engines1-2	Engines2-3	Engines1-3
NX-ResNet-18	0	0	0
AGX-Vgg-16	0	0	0
AGX-Inception_v3	0	0	0
AGX-ResNet-18	0	0	0

Table 7. Output mismatches across inference engines in TensorRT using caching technique

How AutoTVM/AutoScheduler ensures output consistency: We observed the output consistency across the three engines for each model in AutoTVM/AutoScheduler and we found that there are no output mismatches across the engines in either of them.

To illustrate why there are no output inconsistencies in AutoTVM/AutoScheduler, we present a code snippet in Listing 1. The first line of the listing produces the output from the un-optimized model and the second line is the output generated by the AutoTVM optimised model. The last line of the listing actually is used as the correctness criteria by AutoTVM/AutoScheduler at each step. It compares the output produced by unoptimised model with the optimised model and checks whether the two outputs match within some precision (given by parameter `rtol` in the Listing-1).

Thus unlike TensorRT, AutoTVM and AutoScheduler explicitly handles the possibility of errors introduced during kernel optimization, and rejects an optimized kernel if its computed value is not within some acceptable threshold of the unoptimized kernel output. This keeps check on the possible error accumulating in each optimization step, and keeps the end-to-end class probability calculations within thresholds, so that output remains same.

Listing 1 TVM correctness check at each optimization step

```
/** Unoptimized output */
out_np = np.maximum(conv_np + bias_np, 0.0)
/** TVM output*/
out_tvm=tvm.nd.empty(out_np.shape,device=dev)
/** Check correctness */
np.testing.assert_allclose(out_np,
out_tvm.numpy(), rtol=1e-3)
```

> **Observation ❶ Inference accuracy:** TensorRT maintains the same accuracy as unoptimized NN engines. AutoTVM and AutoScheduler have a slight decrease of around 1-3%. Overall, the accuracies of all three NN compilers are reasonable, given the runtime and model size improvements.

> **Observation ❷ Inference reproducibility:** Given the same input image, different TensorRT engines of the same pre-trained NN model, can output different class labels. This is the default behaviour of TensorRT. However with caching, the same compilation steps are maintained giving reproducible output. AutoTVM and AutoScheduler do not cache the compilation steps like in TensorRT caching, instead they maintain output reproducibility using precision checks on intermediate outputs during compilation.

5 Are Inference Latencies Low and Predictable?

Safety critical real time systems need low execution times, and more importantly predictable execution times. If DNN model based inferences form a core part of such systems, then examining whether the associated latencies are low and predictable, is vitally important.

5.1 Comparison of Inference Latencies across NN compilers

In this Section, we compare the inference latencies of different NN compilers for different NN models tabulated in Table 1. We present the inference latencies in Table 8 and observe that AutoTVM and AutoScheduler latencies are significantly lower than TensorRT. TensorRT picks from a set of fixed pre-written kernels, while AutoTVM and AutoScheduler generate optimal kernels, with and without code templates respectively. In addition to increased search space compared to TensorRT, AutoTVM/AutoScheduler also have a knob, namely the number of iterations to search for an optimal kernel. TensorRT does not have any such tunable hyper-parameter, and runs a set of fixed pre-written kernels to measure runtimes, and picks the fastest. We use 200 search iterations for AutoTVM and AutoScheduler in Table 8. AutoScheduler inference times are the lowest among the three NN compilers, as it has a wider search space than manually created templates in AutoTVM, and can find the most optimal kernel implementation. These results in Table 8 are from the AGX platform, but we observe similar trends in NX.

Since embedded boards take a significant amount of time for building AutoTVM/Autoscheduler engines with large number of search iterations, we additionally provide the representative execution

NN Model	TensorRT	AutoTVM	AutoScheduler
Resnet-18	3.93	0.49	0.36
Resnet-50	8.96	0.66	0.60
Squeezenet	4.03	0.56	0.45
Mobilenet_v2	5.65	0.70	0.5
Inception_v3	15.45	1.20	1.10
Vgg-16	8.82	0.49	0.228
Googlenet	16.97	1.18	0.80
Densenet-161	14.65	3.32	2.54
Yolov2	4.97	0.89	0.51
Retinanet	9.13	2.76	2.23

Table 8. Inference Latency (ms) across NN compilers

for few NN models on Nvidia DGX (Tesla V100 GPU). This GPU server is much powerful than embedded boards, where we can vary the number of iterations N from 200 (used on the embedded boards) to 20000 (impractical to run on embedded boards) for AutoScheduler.

	Iteration count				
NN Model	**200**	**1000**	**5000**	**10000**	**20000**
Resnet-50	0.54	0.51	0.49	0.39	0.28
Resnet-18	0.22	0.35	0.19	0.17	0.13
Squeezenet	0.19	0.17	0.18	0.17	0.15
Vgg-16	0.16	0.15	0.14	0.13	0.12

Table 9. Inference times (ms) for Auto-Scheduler engines on V100, with increasing search iterations.

Table 9 shows that as we increase the number of iterations, the inference latency keeps decreasing till it saturates. Thus the comparison between TensorRT and AutoTVM/AutoScheduler is not fair in Table 8, as the inference latencies can decrease further for AutoTVM and AutoScheduler, with increased search iterations. However, we show that even for the less number of search iterations in the order of few hundreds, which are practical on embedded platforms, AutoTVM and AutoScheduler significantly outperform TensorRT in terms of inference latencies, for all NN models.

5.2 Effect of Compression for different NN Compilers

This sub-section deals with one of the most important aspects of DNN inference in edge GPUs, i.e. network compression. We here show the ability of different NN compilers to take full precision pre-trained model and compress that to half precision or INT8 with minimal loss of accuracy. Table 10 shows the effect of compression to different quantization levels for NN compilers on the model size and the inference latency. In all these experiments we used FP32 pre-trained model and compressed those to FP16 and INT8. We observe that on going down in the quantization levels, the model size reduces by half. In addition, we also observe from the table that the inference latencies reduce significantly with negligible loss of accuracy (around 2% maximum dip) when we go from FP32 to INT8 quantization primarily because the operations have to be performed on integers with less bits. This trend is consistent for all the compilers used.

5.3 Inference Latency Predictability Across Compiled Engines

Table 11 shows the comparison of inference latency (in ms), along with standard deviations, for three different engines built for the same NN model using AutoTVM on AGX. We observe that the

	Model Size(MB)/Inference time(ms)								
	Float32			Float16			Int8		
NN Models	TensorRT	AutoTVM	AutoSched	TensorRT	AutoTVM	AutoSched	TensorRT	AutoTVM	AutoSched
Resnet-18	69.0/3.90	72.3/0.49	71.9/0.36	24.3/1.7	24.3/0.42	21.8/0.33	12.1/1.34	14.2/0.35	12.7/0.29
Resnet-50	113.4/8.82	120.8/0.66	119.9/0.6	52.1/3.47	49.8/0.61	47.4/0.56	26.5/2.32	25.3/0.41	23.5/0.35
Inception_v3	108.1/13.8	125.6/1.2	124.4/1.1	49.1/5.16	46.6/0.92	44.3/0.85	25.3/3.42	24.2/0.59	21.6/0.52

Table 10. Model Size (in MBs) and Inference latencies (in ms) for different NN compilers with different levels of quantization

Platform-NN Model	Engine1	Engine2	Engine3
AGX-Resnet-18	0.49±0.38	0.49±0.25	0.47±0.26
AGX-Resnet-50	0.66±0.36	0.65±0.24	0.66±0.33
AGX-Squeezenet	0.56±0.27	0.52±0.33	0.83±0.27
AGX-Googlenet	1.18±0.60	1.12±0.55	1.46±0.58
AGX-Yolov2	0.89±0.20	1.23±0.28	1.38±0.42
AGX-Retinanet	2.76±0.72	3.22±0.62	3.45±0.86
NX-Resnet-18	3.65±0.46	4.98±0.73	5.78±1.36
NX-Resnet-50	4.56±0.23	5.67±0.66	5.20±0.34
NX-Squeezenet	3.61±0.53	4.32±0.40	4.97±0.22

Table 11. Inference latency (ms) across AutoTVM engines

NN Model	Engine1	Engine2	Engine3
Resnet-18	3.93±0.87	5.34±1.56	6.21±1.48
Resnet-50	8.96±1.78	10.28±2.07	12.65±2.22
Squeezenet	4.03±0.62	4.09±0.61	4.16±0.68
Mobilenet_v2	5.65±0.98	7.34±1.13	8.54±1.54
Inception_v3	15.45±1.75	13.04±0.18	18.22±2.87
Vgg-16	8.82±1.23	11.12±2.21	13.87±2.98
Googlenet	16.97±2.29	18.79±2.67	19.36±2.96
Densenet-161	14.65±1.73	17.23±2.21	19.25±2.97
Yolov2	4.97±0.67	6.45±0.96	7.87±1.13
Retinanet	9.13±1.28	12.22±1.65	14.65±1.93

Table 12. Inference time (ms) across TensorRT engines on AGX

standard deviations in inference latencies for a given engine is low, which is a positive result for real time systems requiring predictable execution times with low variance. However, the inference latency varies significantly across the three engines. We observe similar behaviour for TensorRT and AutoScheduler, Tables 12 and 13 show the inference latencies across engines for TensorRT and AutoScheduler respectively.

During engine building process, each NN compiler picks different CUDA kernels based on their runtime measurement. Table 14 shows the CUDA kernels picked by two TensorRT engines for a particular NN layer, namely `res2a_branch1 + res2a + res2a_relu`, in a given NN model. A complete list of layer to kernel mapping for resnet-18 is given in Appendix-1. We observe that the candidate CUDA kernels (listed in Table 14) TensorRT considered for this particular NN layer, are very close in runtime to each other. Therefore their ordering in terms of increasing runtimes can

NN model	Engine 1	Engine 2	Engine 3
Resnet-18	0.36±0.13	0.39±0.2	0.32±0.33
Resnet-50	0.6±0.27	0.66±0.336	0.61±0.391
MobileNet_v2	0.50±0.45	0.55±0.376	0.61±0.0.35
Vgg-16	0.228±0.19	0.197±0.24	0.32±0.32
Inception_v3	1.1±0.1	1.27±0.11	1.19±0.14
Yolov2	0.51±0.1	0.50±0.17	0.51±0.16
Retinanet	2.23±0.47	2.3±0.42	2.41±0.39
Densenet-161	2.54±0.29	2.4±0.366	2.55±0.317
Googlenet	0.8±0.1	0.89±0.18	0.95±0.12

Table 13. Inference time (ms) across AutoScheduler engines on AGX

Engine1 kernels	Time	Engine2 kernels	Time
trt_volta_h884cudnn_256x64_ldg8_relu_exp_interior_nhwc_tn_v1	0.0432	trt_volta_h884cudnn_256x64_ldg8_relu_exp_interior_nhwc_tn_v1	0.0427
trt_volta_h884cudnn_128x128_ldg8_relu_exp_small_nhwc_tn_v1	0.0468	trt_volta_h884cudnn_128x128_ldg8_relu_exp_small_nhwc_tn_v1	0.0481
trt_volta_h884cudnn_128x128_ldg8_relu_exp_interior_nhwc_tn_v1	0.0482	trt_volta_h884cudnn_128x128_ldg8_relu_exp_interior_nhwc_tn_v1	0.0473
trt_volta_h884cudnn_256x128_ldg8_relu_exp_medium_nhwc_tn_v1	0.0521	trt_volta_h884cudnn_256x128_ldg8_relu_exp_medium_nhwc_tn_v1	0.0529
trt_volta_h884cudnn_256x64_sliced1x2_ldg8_relu_exp_medium_ nhwc_tn_v1	0.0498	trt_volta_h884cudnn_256x64_sliced1x2_ldg8_relu_exp_medium_ nhwc_tn_v1	0.0498
trt_volta_h884cudnn_256x64_ldg8_relu_exp_medium_nhwc_tn_v1	0.0480	trt_volta_h884cudnn_256x64_ldg8_relu_exp_medium_nhwc_tn_v1	0.0426
trt_volta_h884cudnn_256x64_sliced1x2_ldg8_relu_exp_small_ nhwc_tn_v1	0.0479	trt_volta_h884cudnn_256x64_sliced1x2_ldg8_relu_exp_small_ nhwc_tn_v1	0.0514

Table 14. Layer to Kernel mapping along with measurement time (μs) for each kernel during engine building for TensorRT. Engine1 (highlighted in blue) and engine2 (highlighted in red) pick two different kernels for the same layer.

change during each compilation. In this particular example, we observe the blue kernel is chosen in engine-1, while the red kernel is chosen in engine-2, for the same NN layer. These two kernels are very close in runtime and therefore have a toggled order when engine-1 is built (blue comes at the top in terms of minimum runtime), vs. when engine-2 is built (red comes at the top in terms of minimum runtime). If a different kernel is chosen in an engine, the way computation is mapped to the hardware for a particular operation will change, leading to overall change in inference latencies across engines. Note that in Section 4, we show a technique called caching in TensorRT that helps to mitigate the non-determinism in outputs across different engines of TensorRT. We study the effect of caching in TensorRT on the inference time also and we observe that across the multiple engines, the inference time remains deterministic as shown in Table 15.

A similar example of same NN layer getting mapped to different kernels is shown for AutoScheduler in Table 16. The full list of kernel mapping for Resnet-18 is given in Appendix-1. The execution time of the two kernels are different in the two engines. The kernels also take different percentages of the total inference times, in the two engines. Such non-determinism of which kernel gets included in the engine is even more intuitive for AutoTVM and AutoScheduler, than in TensorRT. Unlike the fixed set of kernels TensorRT chooses from based on minimum runtime, AutoTVM and AutoScheduler generates optimal kernels with and without manually defined code templates respectively. Thus deterministic builds, which will give the exact same kernels every time an engine is built, is not part of these compiler specifications. Every time an engine is built for a given NN model, a different set of kernels are chosen, leading to differences in end-to-end inference latencies across engines. Implications of this unpredictability in inference times on real time applications,

NN Model	Engine1	Engine2	Engine3
Resnet-18	3.93	3.93	3.94
Resnet-50	8.96	8.97	8.96
Squeezenet	4.03	4.029	4.03
Mobilenet_v2	5.65	5.65	5.65
Inception_v3	15.45	15.45	15.44
Vgg-16	8.82	8.82	8.8
Googlenet	16.97	16.97	16.98
Densenet-161	14.65	14.64	14.65
Yolov2	4.97	4.97	4.98
Retinanet	9.13	9.13	9.13

Table 15. Inference time (ms) across TensorRT engines on AGX using caching

Kernel name	Time for execution (μs)		% Time of total execution	
	Engine-1	Engine-2	Engine-1	Engine-2
`tvmgen_default_fused_nn_conv2d_add_multiply_add_nn_relu_kernel0`	511.0	164.26	20.16	6.54
`tvmgen_default_fused_nn_conv2d_add_add_nn_relu_2_kernel0`	442.4	163.94	17.45	6.53

Table 16. Sample kernels with execution times and % of total execution time spent in them, for two AutoScheduler engines.

whenever a new engine is built from the same pre-trained model, will be discussed in more detail in Section 6.

5.4 Effect of Dynamic Voltage and Frequency Scaling (DVFS) on Inference Latency

NN Model	TensorRT	AutoTVM	AutoScheduler
Resnet-18	2.75	0.62	0.45
Resnet-50	6.87	0.96	0.8
Squeezenet	3.65	0.72	0.64
Mobilenet_v2	4.30	0.84	0.69
Inception_v3	12.82	1.53	1.36
Vgg-16	7.43	0.64	0.39
Googlenet	9.96	1.48	1.1
Densenet-161	13.24	3.62	2.78
Yolov2	4.67	0.97	0.59
Retinanet	8.05	2.98	2.54

Table 17. Inference Latency (ms) across NN compilers when DVFS is disabled on AGX

In all the forementioned analysis, we keep the DVFS (Dynamic Voltage Frequency Scaling) enabled and we observe that on the same hardware, for a given neural network model, AutoScheduler runs at a higher frequency compared to TensorRT. However To nullify the effect of varying frequencies, we set the minimum and maximum frequency to be same(1033 Mhz) and compare the inference latencies among different compilers . We tabulate the inference latency numbers in Table 17. We observe that even with the fixed frequency, AutoScheduler has the lowest inference time among all the compilers while as TensorRT has higher inference numbers. This is because of the following

reasons as mentioned in the forementioned section also: ❶ AutoScheduler and AutoTVM generates kernels corresponding to layers at run time based on the tuning hyperparameter called as *number of iterations* ❷ AutoScheduler has a wider search space compared to AutoTVM that generates kernels based on template based matching. Thus, due to various NN optimizations possible in AutoScheduler, we observe it has the lowest inference times even if we mitigate the effect of varying frequencies.

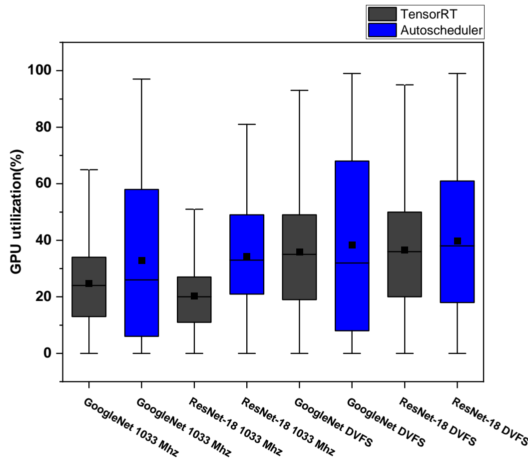


Fig. 4. GPU utilization for ResNet-18 and GoogleNet for DVFS enabled and fixed frequency on AGX

Upon further profiling, we calculated the GPU utilization for AutoScheduler and TensorRT in both settings. Figure 4 shows the GPU utilization distribution on two widely used neural network models (GoogleNet and ResNet-18) for DVFS enabled and fixed frequency settings. We observed from the profiling results that AutoScheduler has similar or slightly better GPU utilization than TensorRT in both the settings suggesting that AutoScheduler utilizes the underlying hardware much better than the TensorRT. We want to mention here that at the similar or slightly better utilization of hardware resources in AutoScheduler than TensorRT, the inference latency in AutoScheduler is much lower for both DVFS enabled and disabled settings indicating that the overall efficiency of AutoScheduler is higher.

We further calculate the *energy per inference* for TensorRT, AutoTVM and AutoScheduler in DVFS enabled and fixed frequency settings. As can be seen from figure 5, for GoogleNet, the AutoScheduler, AutoTVM and TensorRT uses 0.025J/image, 0.031J/image and 0.034J/image respectively in DVFS enabled setting while as for fixed frequency they utilize 0.033J/image, 0.034J/image and 0.04J/image respectively. Similar trend can be seen for ResNet-18 (figure 5) wherein TensorRT uses more energy per inference than AutoTVM and AutoScheduler in both settings. Further enabling DVFS results in lower energy utilization than fixing GPU frequency. Note that we show the observation for the representative execution, we see the similar trends for other NN models.

5.5 Comparison of Compilation Latency across NN compilers

While inference times obtained with AutoTVM and AutoScheduler engines are lower, the time to build an optimized NN engine for AutoTVM and AutoScheduler is significantly more compared to TensorRT. Table 18 shows the execution engine building time across NN compilers for different models on AGX. We observe that AutoTVM takes around 6.7× time for engine building than TensorRT. AutoScheduler takes around 10.3× and 1.5× times more than TensorRT and AutoTVM

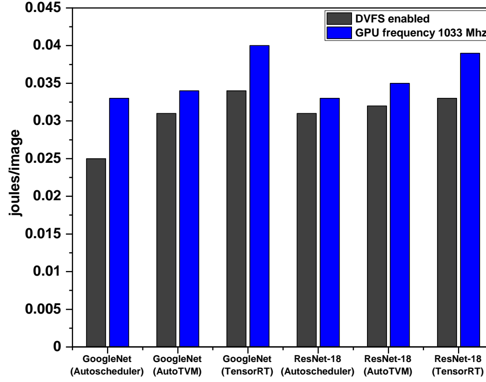


Fig. 5. Energy utilization (joules/image) for ResNet-18 and GoogleNet for DVFS enabled and fixed frequency on AGX

respectively. While the absolute compilation times vary, the same trends hold across the three NN compilers on the embedded NX platform and DGX server.

NN Model	TensorRT	AutoTVM	AutoScheduler
Resnet-18	50.87	846.19	1702.55
Resnet-50	118.58	1680.53	2482.70
Squeezenet	107.34	1457.70	2148.60
Mobilenet_v2	154.50	1414.66	2379.0
Inception_v3	218.96	2548.65	5032.0
Vgg-16	336.76	1460.70	1722.0
Googlenet	209.01	3565.05	5173.02
Densenet-161	1422.69	4753.43	6273.04

Table 18. Engine building time (secs) on AGX

There are more CUDA kernels to generate and choose from in AutoTVM, using the manually defined code templates, compared to the fixed set of pre-written CUDA kernels that TensorRT chooses from. AutoScheduler generates an even greater number of CUDA kernels, instead of taking it from built-in templates as in AutoTVM. In addition to the larger search space to choose the optimal kernel for each operation, the search times for AutoTVM and AutoScheduler can also be increased, based on the tunable hyper-parameter of search iterations during the compilation process. Table 18 gives the engine building times, when search iterations are set to 200 for AutoTVM and AutoScheduler. Since there is no such hyper-parameter in TensorRT, it finishes engine building in one go, picking from a fixed set of CUDA kernels for each operation based on minimum runtime. TensorRT engines therefore have significantly less compilation time, compared to AutoTVM and AutoScheduler.

While some real time application designers might need quick compilation and deployment of the NN engines, we expect most safety critical applications to allow for higher compilation and engine building times. Engine building would be done much less frequently than the real time inferences by the deployed engine. Thus ensuring a very low inference time, as given by AutoTVM and AutoScheduler, with a trade-off of higher compilation time than TensorRT, will

possibly be appealing to most real time application designers. We discuss this choice of NN compiler for minimizing inference times in more detail in Section 6, in the context of a concrete real time application like ADAS.

5.6 Is Resource Utilization Optimal for NN Compilers?

	TensorRT		AutoTVM		AutoScheduler	
NN Model	**NX**	**AGX**	**NX**	**AGX**	**NX**	**AGX**
Resnet-18	2.56	2.75	1.12	0.62	0.89	0.45
Resnet-50	7.2	6.87	1.21	0.96	1.1	0.8
Squeezenet	3.47	3.65	0.96	0.72	0.83	0.64
Vgg-16	7.28	7.43	0.82	0.64	0.74	0.39

Table 19. Inference latencies (in ms) for NX vs AGX

One of the main selling points of the various NN compilers for edge GPUs has been: NN model builders need not worry about mapping their computations to hardware, as it is difficult for an ML practitioner to keep track of fast evolving hardware accelerators for edge DNN. The NN compilers would solve this particular problem of mapping the NN computations effectively to a given hardware platform. Our final experiment for inference times is to examine this claim of NN compilers, whether the hardware specific engine generated by them optimally utilizes hardware resources on a given platform. This evaluation is important for real time application designers, for planning hardware budget. Will the worst case execution times for a safety critical real time application improve, if he invests higher budget in an embedded platform with more hardware resources? We examine our three NN compilers on the NX and AGX platforms, to explore this important question of effective hardware utilization.

From Table 2, it is clear that the Xavier AGX has more resources compared to Xavier NX. We tabulate the inference latencies for NX and AGX, for some of the NN models in Table 19, using our three NN compilers. For AutoTVM and AutoScheduler, the performance in AGX is relatively better compared to NX. The ML based cost model used by AutoTVM and AutoScheduler to predict runtime of candidate CUDA kernels, seem to effectively capture characteristics of the underlying hardware platform. These two compilers, therefore, utilize the hardware resources on AGX effectively for faster inference than NX.

TensorRT engines for some NN models (Resnet-18, Squeezenet, Vgg-16), however, perform better in NX compared to AGX. TensorRT does not use any ML based cost model to predict runtimes of candidate CUDA kernels. Instead it directly measures a set of pre-written kernel runtimes. Thus if manually written CUDA kernels are not available to TensorRT, that themselves utilize hardware resources effectively, TensorRT will have poor candidate kernels to choose from, leading to poor resource utilization. AutoTVM and AutoScheduler generates hardware specific kernels, with and without manual templates, and is not limited by manually written and optimized kernels. AutoTVM/AutoScheduler are therefore better NN compilers in terms of optimal hardware resource utilization, compared to TensorRT.

Observation ❹ Inference latency mean: AutoScheduler, has minimum inference latency, closely followed by AutoTVM. TensorRT engines have significantly higher inference latencies for all NN models, compared to the the other two NN compilers.

Observation ❺ Effect of compression to different quantization levels: For all the NN compilers, we see the reduction in the model size as we go from FP32 to INT8 quantization. In addition, the inference latencies also reduce significantly from FP32 to INT8 (with minimal loss of accuracy) as the operations have to be performed on less bits.

Observation ❻ Inference latency variance: None of the three compilers are deterministic, i.e. given the same input NN model, the execution engine output by the compiler will vary in terms of the constituent kernels. This causes inference latencies to vary across engines for the same NN model. For a particular engine, however, standard deviation in inference latency is very low.

Observation ❼ DVFS vs Fixed frequency: The inference time and energy utilization is lower in AutoScheduler than TensorRT for both DVFS enabled and fixed frequency settings. In addition, we also observed from our experiments, for models optimized with AutoScheduler (DVFS enabled) on 200 vs 1000 iterations, the inference latency and energy requirements get lowered with frequency remaining the same which implies AutoScheduler performs more optimizations only on increasing the number of iterations without changing the frequency.

Observation ❽ Engine building times: Increasing search iterations increases engine building time for AutoTVM/AutoScheduler. TensorRT engine building time is constant and small.

Observation ❾ Resource utilization: AutoTVM/AutoScheduler give better hardware utilization, giving lower inference latencies with more hardware resources on AGX than NX.

6 Impact Analysis on Real Time Applications

Embedded GPU platforms such as Xavier NX or AGX, as well as NN inference engines, are widely used in the automotive, medical, agricultural, mining, industrial automation, last mile delivery, construction, retail, and other application domains. We discuss two specific automotive applications in this section, that extensively use these edge GPU devices for DNN inferences, namely intelligent traffic intersection control [13] and advanced driving assistance systems (ADAS) [14, 24, 35]. We examine the end-to-end response time components for these applications, analyze NN compiler dependencies for those response time components, and also present a worst case response time or execution time (WCET) analysis.

6.1 Safety Critical Real Time Application Instances

Traffic intersection control: These systems evaluate the length or density of traffic queues in the incoming lanes at each intersection and adjust the green and red lights to maximise transportation metrics such as intersection throughput or average/worst case vehicle waiting times. They can additionally detect rule infractions such as over-speeding or jumping of red lights by the vehicles. Detecting and reading number plates of violating vehicles into a vehicle number, enable these systems to issue rule violation penalty fees. NN model inferences are extensively used for object detection and tracking to detect rule violations and identify number plates. Image classification DNN models are also needed to read the number plates.

Advanced Driver Assistance Systems (ADAS): These systems offer technological support to the drivers. DNNs are the most important component of the analytical engine of any ADAS system, performing critical tasks like ❶ lane detection ❷ obstacle detection ❸ pedestrian detection ❹ traffic signal, and road sign detection and recognition etc. Both object detection and image classification DNN models, as evaluated in this paper (Table 1), are important in ADAS, in order to gather

the actionable information such that the actuators (steering, braking system etc.) are controlled properly.

Both traffic intersection control and ADAS applications can feed multiple camera feeds to a single edge GPU device for NN inferences. The intersection controller requires multiple cameras to monitor the various incoming roads, and the ADAS instrumented car requires multiple cameras to monitor its surroundings in all directions.

6.2 Response Time Components for Deployed DNN Models

A typical DNN execution engine has the following components in response time:
❶ τ_{read}: the image read time, where the image or video frame incoming from the camera is copied into RAM, either from the disk where the camera saves the input frame, or directly from the camera over PCI bus or ethernet.
❷ τ_{pre}: the pre-processing time for image conversion to the required format needed by NN compiler (NHWC for AutoTVM/Autoscheduler and NCHW for TensorRT), image resizing 224×224 for model processing, normalization etc., typically executed on the CPU cores of the edge platform.
❸ τ_{inf}: the inference time or time to run the DNN execution engine on the GPU cores of the edge platform.
❹ τ_{post}: the post-processing time which includes time to search for maximum probability from output distribution, matching corresponding class labels, computing co-ordinates for generating object detection bounding boxes etc.
The total response time for a DNN execution engine can therefore be calculated as:

$$\Re_T = \tau_{read} + \tau_{pre} + \tau_{inf} + \tau_{post} \tag{1}$$

6.3 Response Times vs. NN Compilers For Single Camera

We show a representative execution of Resnet-18 in Figure 7 and Figure 6, for one camera.

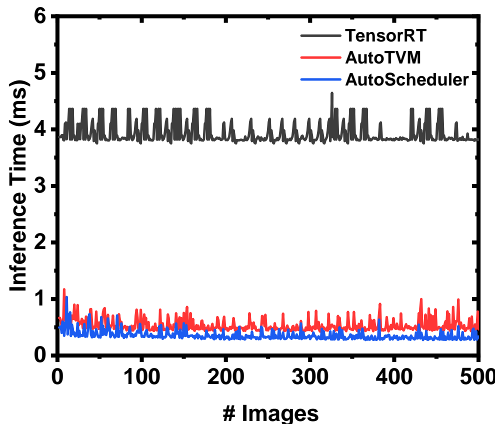
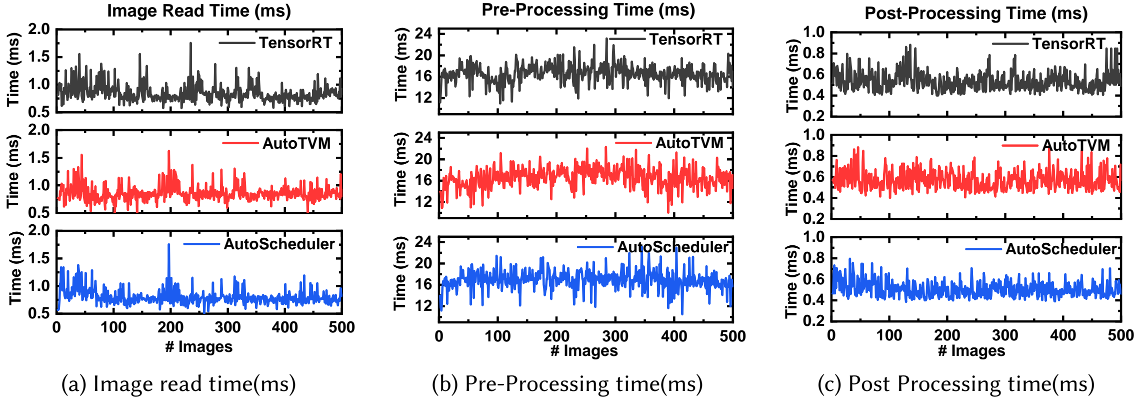


Fig. 6. Inference time(ms) for ResNet-18 on AGX

Table 20 shows the response times for 10 NN models on a single camera input, across the three NN compilers. As seen from Figure 7, image read and post-processing times are insignificant (in the order of 1 msec and 0.5 msecs respectively. Pre-processing time in the order of 15 msecs is significant, but nonetheless it is constant across the NN compilers. The main difference in response times across the three compilers come from the inference times, where TensorRT engines having the



(a) Image read time(ms)

(b) Pre-Processing time(ms)

(c) Post Processing time(ms)

Fig. 7. Response time components across NN compilers for 500 ImageNet samples (Resnet-18 on AGX)

slowest inference, gives the maximum response times. We present this dependency more formally in Section 6.5.

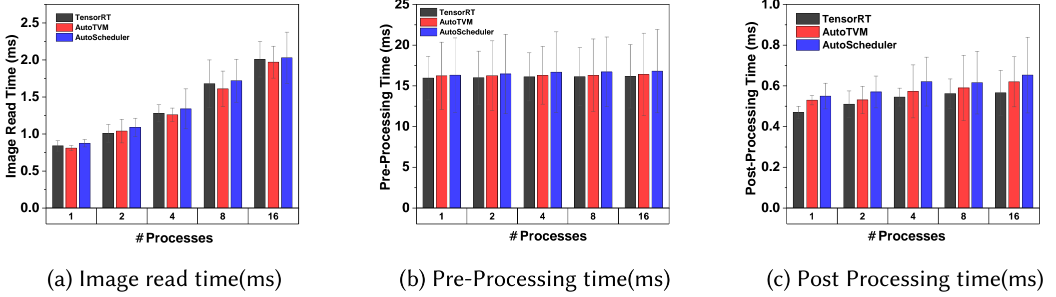
NN Model	TensorRT	AutoTVM	AutoScheduler
Resnet-18	21.91	18.95	18.78
Resnet-50	26.94	19.13	19.02
Squeezenet	22.01	18.96	18.87
Mobilenet_v2	23.63	19.11	18.97
Inception_v3	33.43	19.61	19.52
Vgg-16	26.80	18.90	18.71
Googlenet	34.95	19.56	19.22
Densenet-161	32.63	21.73	20.96
Yolov2	22.95	19.37	18.93
Retinanet	27.11	20.89	20.65

Table 20. Response latencies (ms) across NN compilers

6.4 Response Times vs. NN Compilers For Multiple Cameras

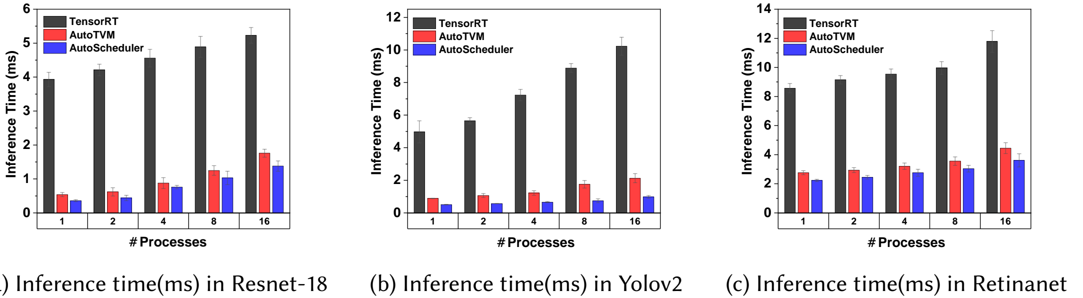
Multiple camera feeds cover different directions for ADAS and traffic intersection controller, and DNN inferences need to run in parallel. We show a representative execution of Resnet-18 image classification network in Figure 8 and Figure 9(a), for 1-16 concurrent CNNs on multiple cameras. We additionally show the inference times across compilers for two object detection networks Yolo and RetinaNet in Figure 9(b) and Figure 9(c) respectively. We start with a single input feed for our analysis, and keep on increasing the number of input feeds from 1 to 16, using multiple processes.

We observe the following trends for the four response time components, in the multi camera setting. ❶ τ_{read} in Figure 8a increases as we increase the number of CNN instances from 1 to 16 *(more than 2×)*. Multiple CNN instances send bus requests to fetch the images, increasing bus contention and image read time. ❷ τ_{pre} in Figure 8b remains almost same even when we increase the number of input camera feeds, while standard deviation slightly increases. This is due the presence of non-trivial computational capabilities of modern edge CPUs, on which pre-processing runs. ❸ τ_{inf} in Figure 9 goes up, as CNN concurrency increases, while the standard deviations stay low. Inference time increase is maximum in TensorRT followed by AutoTVM and then AutoScheduler,



(a) Image read time(ms) (b) Pre-Processing time(ms) (c) Post Processing time(ms)

Fig. 8. Image read, pre-processing and post-processing times are similar for different NN compilers for 500 different images of ImageNet dataset, with increasing camera feeds. This is representative execution of Resnet-18 on AGX platform.



(a) Inference time(ms) in Resnet-18 (b) Inference time(ms) in Yolov2 (c) Inference time(ms) in Retinanet

Fig. 9. Inference time(in ms) in multiple camera feeds for image classification and object detection networks used in ADAS

for classification as well as object detection. Also as in single camera setting, the inter-compiler differences are highest for inference time, and thus will be important in WCET analysis with respect to the choice of an appropriate NN compiler. ❹ τ_{post} in Figure 8c remains almost similar with increasing CNNs, with slight increase in standard deviation. CPU cores on which post-processing is run, is powerful enough to handle all the concurrent processes.

6.5 Worst Case Response or Execution Time (WCET) Analysis

We finally present the worst case response or execution time (WCET) analysis of the DNN models running on edge GPUs. We consider the four time components of end-to-end response time (equation 1), i.e. τ_{read}, τ_{pre}, τ_{inf} and τ_{post}. and examine how they influence WCET.

Lemma-1: *The overall response time $\mathfrak{R}_T$ is bounded by the neural network model inference time τ_{inf}*

Case-1: For single camera input feed As observed from Figure 6 and Figure 9, τ_{inf} is dependent on the choice of NN compiler. TensorRT engines are the slowest and the AutoScheduler engines are the fastest for inference. Secondly, increasing the number of search iterations during model optimization for AutoTVM and AutoScheduler, further reduces inference time (Table 9). Thirdly, τ_{inf} also depends on which particular inference engine compiled by a given NN compiler, is used, as every time an engine is compiled by the same NN compiler from the same pre-trained NN model, the constituent kernels and the inference times change (Table 11). τ_{read}, τ_{pre} and τ_{post} are independent of the NN compiler as well as inference engine choice (Figure 7 and Figure 8). The

DNN compiler desired behavior	Implication of desired behavior for real time applications	TensorRT	AutoTVM	AutoScheduler
Maintains accuracy	Almost same or better accuracy can lead to better classification outputs and accurate bounding box co-ordinates.	✓	✓	✓
Response time does not increase with concurrent inputs	Increasing the number of simultaneous input camera feeds should not significantly increase the response times, for timely decision making	✗	✓	✓
Deterministic outputs across different inference engines	If the optimized inference engine is rebuilt for the same NN model, it should always generate same output for a given input sample, to ensure reliable decision making in safety critical systems.	✗	✓	✓
Deterministic inference times across inference engines	When the NN engines are rebuilt, the same NN model on the same edge GPU should have same inference latencies, else WCET analysis in real-time applications will be difficult. The braking system may not get the detection inference in ADAS in time.	✗	✗	✗
Optimal resource utilization	NN compilers should optimize the DNN models in a way that takes underlying hardware platform into account. Inference engines running on better platforms (e.g Nvidia AGX) should have less inference time than ones running on lower end boards (Nvidia NX)	✗	✓	✓
Energy utilization	NN compilers should optimize the DNN models in a way that takes lower energy per inference such that total energy requirement is minimized. This is particularly important for battery operated devices.	✗	✓	✓

Table 21. Desirable behaviors of NN compilers and their impact on intersection control and ADAS applications. ✓represents that the NN compiler shows the desired behavior, ✗indicates it doesn't. As TensorRT has more ✗, AutoTVM/AutoScheduler can be recommended for real time application settings.

operations performed in image read, pre and post processing are hardware platform dependent and image size dependent, hence cannot be optimized by the NN compiler or inference engine choice. Therefore for a single camera setting, we can re-write equation 1 as: $\mathfrak{R}_T = C1 + C2 + \tau_{inf} + C3$ where C1, C2 and C3 are the image read, pre-process and post-process constant times, independent of NN compiler. Proper choice of NN compiler, appropriate choice of search iteration hyper-parameter if present for the chosen compiler, and choice of the exact NN engine built by the chosen compiler, affect τ_{inf} and therefore are important in reducing WCET.

Case-2: For multiple camera input feed In case of taking feeds from multiple cameras, the time required for image read increases as concurrent DNNs increase, due to memory contention on the bus (Figure 8a). Pre-process and post-process times do not show any notable change with increasing number of concurrent CNNs, due to significant processing capability of edge CPU on which these steps run (Figure 8b and Figure 8c). As in single camera setting, none of these time components depend on the NN compiler choice (Figure 8). Inference time varies across NN compilers in similar way, as discussed for the single camera setting, with choice of NN compiler, choice of search iteration hyper-parameter for a chosen NN compiler, and the particular execution engine built by the chosen compiler. The growth of inference times with increasing concurrency, also varies across the NN compilers. TensorRT shows a faster increase in inference time with increased concurrency (Figure 9). Therefore for a multi-camera concurrent DNN setting, we can re-write equation 1 as: $\mathfrak{R}_T = (\sum_{f=1}^{F} C1_f) + (C2 + \varepsilon) + (\tau_{inf} + \Delta_{inf}) + (C3 + \varepsilon)$ where F is the number of simultaneous frames processed, ε signify the insignificant change & Δ_{inf} represents the increase in the inference time. In case-1, F=1, therefore the term $\sum_{f=1}^{F} C1_f$ equals to $C1$, is not subject to any optimization given different NN compilers or different inference engines generated. The $\sum_{f=1}^{F} C1_f$ also remains constant at a given concurrency for a given NN compiler and computational platform. Thus the inference time τ_{inf} is the only variable, to bound the overall response time.

Observation ⑩ DNN concurrency increases τ_{read}, while τ_{pre} and τ_{post} remain constant. τ_{read}, τ_{pre} and τ_{post} are all independent of the NN compiler, in both single and multi camera settings.

Observation Inference time τ_{inf} changes across NN compilers and also across engines built using the same NN compiler. WCET in a real time system, is therefore dependent on τ_{inf}, decided by NN compiler, compiler hyper-parameter and built engine choices.

7 Related Work

We evaluate DNN inference engines relevant for various ADAS application tasks, in this paper. Restuccia et al. [25] examined the latency characteristics of DNN models employed in ADAS, using FPGA accelerators. Other studies, such as Wurst et al [33], employed NVIDIA edge GPUs (TX2) to examine DNN execution on heterogeneous platforms, again for ADAS performance analysis. However, none of these prior works used DNN compilers to optimize and deploy their models on FPGA or GPU. As DNN compilers are becoming increasingly predominant in an application designer's software stack, our examination of three state of the art DNN compilers, for the first time to the best of our knowledge, bridges an important research gap in real time systems.

Mingzhen et al. [22] is an excellent qualitative survey on DNN compilers, describing both the compiler frontend i.e. DNN model optimizations and backend i.e. mapping the optimized model to hardware processor cores. It describes the inner workings of a comprehensive set of DNN compilers. Verma et.al [32] does performance evaluation of Tensorflow-Lite and Tensorflow-TensorRT on two NVIDIA GPUs - GeForce RTX 2080 and Tesla T4 using commonly employed DL models for edge devices. This paper looks at the inference latencies and power drawn by the GPU to perform an inference. However, they do not consider other metrics such as output accuracy and predictability, compilation time or hardware resource utilization, characterized by us in this paper. These works do not examine the causes of a compiler specific behavior and their performance implications, that is studied in-depth in this paper. We focus on edge GPU platforms typically used in real time applications, evaluating wide range of important compiler metrics. We also present a comprehensive *quantitative audit* of different NN compilers, significantly extending the qualitative survey presented in [22] with many more observations and recommendations that will help the researchers for developing the solutions based on these software tools. In another line of work based on NN compilers, [28] performs a comprehensive study of NN compiler bugs, focusing on three compilers - Apache TVM, Facebook Glow and Intel nGraph. They examine the underlying causes of the defects and asserts that a significant number of DL compiler bugs are connected to Tensor types. On the similar lines, Cao.et.al [16] presents the first comprehensive study to characterize performance problems in Deep learning systems written in TensorFLow and Keras. However, as opposed to compiler testing and debugging, our work places a greater emphasis on the characterization and analysis of various real-world metrics that are crucial for ADAS application scenarios.

Omais et. al. [27] is a recent paper closest to our work. They examined Nvidia's TensorRT engines for a similar set of safety critical applications as us. However, the paper posed more questions than answers, as some surprising behaviors were demonstrated for TensorRT engines, without any root cause analysis of why such behaviors occurred. Our head-on comparison across multiple DNN compilers, TensorRT and additionally AutoTVM and AutoScheduler, help us explain their quirks and advantages. We also additionally do an end-to-end response time analysis for typical DNN tasks, to comprehensively connect our empirical observations about DNN compilers to WCET analysis in real time application settings.

8 Recommendations and Conclusion

Table 21 summarizes the possible implications of our findings in real time settings. We further list some recommendations, based on our observations: ❶ In order to maintain determinism in the

output generated, we suggest that AutoTVM or Autoscheduler should be preferred over TensorRT. ❷ For time critical tasks, Autoscheduler engines should be chosen as they have minimum inference times. ❸ When considering an upgrade in hardware platforms, TensorRT does not guarantee that better hardware results in better runtimes, while AutoTVM/ Autoscheduler does. ❹ When considering migration to different vendor based hardware platform, TensorRT engines run only on Nvidia GPUs. AutoTVM and Autoscheduler support optimizations on variety of hardware back-ends like GPUs, CPUs, TPUs, FPGA etc. ❺ All compilers suffer from inconsistency in inference times across different inference engines. We suggest using caching technique for reproducible engines in TensorRT or generating a single inference engine and re-using the exact same binary to be deployed on different hardware beckends. ❻ Advances in NN architecture and operators need manual engineering effort in TensorRT and AutoTVM for optimized code translation. Autoscheduler automatically generates machine code, and is therefore well suited to such upgrades.

This paper aims to provide the in-depth analysis of performance predictability of various state-of-the-art DNN compilers and their optimization performance on Nvidia's embedded GPUs Xavier AGX and NX. All the experiments are conducted on modern DNN architectures typically used in ADAS. We showed how a proper choice of DNN compiler can help lower the analytical bound on overall response time in a real time setting. Further, we present a compendium of implications and recommendations based on our findings, that will help the researchers and application developers for developing solutions based on these software tools.

9 Acknowledgements

We thank our reviewers for their insightful feedback that helped us to improve our paper. This work was partially supported by the Science and Engineering Research Board under grant SERB-POWER (SPG/2021/000731).

References

[1] [n. d.]. ARM Compute Library. https://github.com/ARM-software/ComputeLibrary.
[2] [n. d.]. Device Query Utility for GPU platforms. https://docs.nvidia.com/cuda/cuda-samples/index.html.
[3] [n. d.]. Facebook Glow. https://ai.facebook.com/tools/glow/.
[4] [n. d.]. Intel® Distribution of OpenVINO™ Toolkit. https://github.com/openvinotoolkit/openvino.
[5] [n. d.]. Jetson Xavier AGX. https://www.nvidia.com/en-in/autonomous-machines/embedded-systems/jetson-agx-xavier/.
[6] [n. d.]. Jetson Xavier NX. https://www.nvidia.com/en-in/autonomous-machines/embedded-systems/jetson-xavier-nx/.
[7] [n. d.]. Nvidia Drive Platform. https://developer.nvidia.com/drive.
[8] [n. d.]. Nvidia Industry Inspection. https://www.nvidia.com/en-us/industries/industrial/.
[9] [n. d.]. NVIDIA. TensorRT. https://developer.nvidia.com/tensorrt.
[10] [n. d.]. Qualcomm Neural Processing SDK for AI. https://developer.qualcomm.com/software/qualcomm-neural-processing-sdk.
[11] [n. d.]. Tensorflow Lite. https://www.tensorflow.org/lite.
[12] [n. d.]. Tensorflow XLA. https://www.tensorflow.org/xla..
[13] Advantech. [n. d.]. Traffic Intersection Monitoring. https://www.advantech.com.co/resources/case-study/intelligent-video-traffic-monitoring-for-self-adaptive-traffic-signal-control-system.
[14] Mohamed Aladem and Samir A Rawashdeh. 2020. A single-stream segmentation and depth prediction CNN for autonomous driving. *IEEE Intelligent Systems* (2020).
[15] Laha Ale, Ning Zhang, and Longzhuang Li. 2018. Road damage detection using RetinaNet. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 5197–5200.
[16] Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding Performance Problems in Deep Learning Systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 357–369. https://doi.org/10.1145/3540250.3549123
[17] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing

Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[18] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. 2018. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058* (2018).

[19] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[20] Forrest Iandola, Matt Moskewicz, Sergey Karayev, Ross Girshick, Trevor Darrell, and Kurt Keutzer. 2014. Densenet: Implementing efficient convnet descriptor pyramids. *arXiv preprint arXiv:1404.1869* (2014).

[21] Hyo Jong Lee, Ihsan Ullah, Weiguo Wan, Yongbin Gao, and Zhijun Fang. 2019. Real-time vehicle make and model recognition with the residual SqueezeNet architecture. *Sensors* 19, 5 (2019), 982.

[22] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. 2020. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems* 32, 3 (2020), 708–727.

[23] Ishrat Zahan Mukti and Dipayan Biswas. 2019. Transfer learning based plant diseases detection using ResNet50. In *2019 4th International Conference on Electrical Information and Communication Technology (EICT)*. IEEE, 1–6.

[24] Nvidia. [n. d.]. Self Driving Cars. https://www.nvidia.com/en-us/self-driving-cars/partners/.

[25] Francesco Restuccia and Alessandro Biondi. 2021. Time-Predictable Acceleration of Deep Neural Networks on FPGA SoC Platforms. In *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 441–454.

[26] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[27] Omais Shafi, Chinmay Rai, Rijurekha Sen, and Gayathri Ananthanarayanan. 2021. Demystifying TensorRT: Characterizing Neural Network Inference Engine on Nvidia Edge Devices. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 226–237.

[28] Qingchao Shen, Haoyang Ma, Junjie Chen, Yongqiang Tian, Shing-Chi Cheung, and Xiang Chen. 2021. A Comprehensive Study of Deep Learning Compiler Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 968–980. https://doi.org/10.1145/3468264.3468591

[29] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[30] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[31] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[32] Gaurav Verma, Yashi Gupta, Abid M Malik, and Barbara Chapman. 2021. Performance evaluation of deep learning compilers for edge inference. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 858–865.

[33] Falk Wurst, Dakshina Dasari, Arne Hamann, Dirk Ziegenbein, Ignacio Sanudo, Nicola Capodieci, Marko Bertogna, and Paolo Burgio. 2019. System performance modelling of heterogeneous hw platforms: An automated driving case study. In *2019 22nd Euromicro Conference on Digital System Design (DSD)*. IEEE, 365–372.

[34] Xiaoling Xia, Cui Xu, and Bing Nan. 2017. Inception-v3 for flower classification. In *2017 2nd International Conference on Image, Vision and Computing (ICIVC)*. IEEE, 783–787.

[35] Robail Yasrab, Naijie Gu, and Xiaoci Zhang. 2017. An encoder-decoder based convolution neural network (CNN) for future advanced driver assistance system (ADAS). *Applied Sciences* 7, 4 (2017), 312.

[36] Zhang Yi, Shen Yongliang, and Zhang Jun. 2019. An improved tiny-yolov3 pedestrian detection algorithm. *Optik* 183 (2019), 17–23.

[37] Xiang Yu and Shui-Hua Wang. 2019. Abnormality diagnosis in mammograms by transfer learning based on ResNet18. *Fundamenta Informaticae* 168, 2-4 (2019), 219–230.

[38] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: Generating {High-Performance} Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 863–879.