Demystifying TensorRT: Characterizing Neural Network Inference Engine on Nvidia Edge Devices

Omais Shafi*, Chinmay Rai*, Gayathri Ananthanarayanan§, Rijurekha Sen*

*Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
New Delhi, India
*{omais.shafi@cse.iitd.ac.in,chinmayrai01@gmail.com,riju@cse.iitd.ac.in}

§Department of Computer Science and Engineering
Indian Institute of Technology, Dharwad
Karnataka, India
§ {gayathri@iitdh.ac.in}

***Abstract*—Edge devices are seeing tremendous growth in sensing and computational capabilities. Running state-of-the-art deep neural network (NN) based data processing on multi-core CPU processors, embedded Graphics Processing Units (GPU), Tensor Processing Units (TPU), Neural Processing Units (NPU), Deep Learning Accelerators (DLA) etc., edge devices are now able to handle heavy data computations with limited or without cloud connectivity. In addition to hardware resources, software frameworks that optimize a trained neural network (NN) model through weight clustering and pruning, weight and input-output quantization to fewer bits, fusing NN layers etc., for more efficient execution of NN inferences on edge platforms, play an important role in making machine learning at the edge (namely EdgeML) a reality.**

This paper is a first effort in characterizing these software frameworks for DNN inference optimizations on edge devices, especially edge GPUs which are now ubiquitously used in all embedded deep learning systems. The interactions between software optimizations and the underlying GPU hardware is carefully examined. As most NN optimization engines are proprietary softwares with undocumented internal details in the public domain, our empirical analysis on real embedded GPU platforms using a variety of widely used DNNs, provide various interesting findings. We observe tremendous performance gain and non-negligible accuracy gain from the software optimizations, but also find highly unexpected non-deterministic behaviors such as different outputs on same inputs or increased execution latency for same NN model on more powerful hardware platforms. Application developers using these proprietary software optimization engines, would benefit from our analysis and the discussed implications of our findings, with examples from real applications like intelligent traffic intersection control and Advanced Driving Assistance Systems (ADAS). There are important implications of our findings on performance modeling and prediction research too, that focus on micro-architecture modeling based application performance prediction, but should now additionally consider optimization engines that this paper examines.

I. Introduction

Deep Neural Networks (DNNs) have made significant progress in the recent years and are being widely used in various application domains such as image recognition, speech recognition, natural language processing and various other computer vision related tasks. Many safety critical applications such as autonomous driving and advanced driver assistance systems (ADAS) also heavily use neural networks (NN) to perform tasks like pedestrian and obstacle detection [1]–[3]. Using NN models in a wide variety of application domains have resulted in a concomitant increase in the interest and efforts from chip designers and various hardware manufacturers towards building optimal hardware for NN execution [4]–[6]. Embedded edge devices are especially becoming equipped with multi-core processors and custom accelerators, which can now process incoming sensor data with limited or without cloud support. In recent years, there has been tremendous progress in the research and development of both the hardware platforms as well as software stacks aimed towards improving the efficiency of embedded NN inference. Figure 1 gives a glimpse of the popular hardware and software stacks available in the NN market segment for edge platforms.

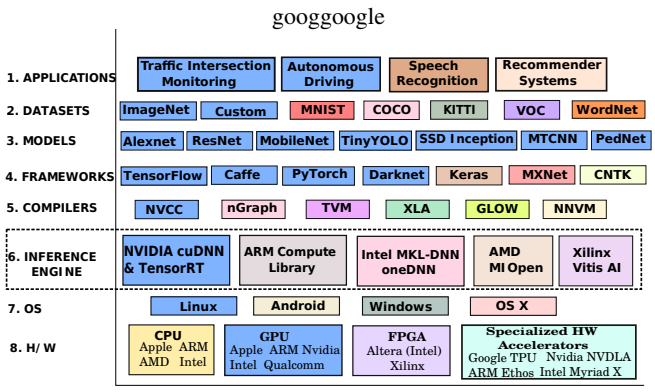


Fig. 1: Popular DNN hardware and software stacks adapted from [7]. Blue boxes indicate components used here to examine effect of inference engines (level 6), on inference accuracy and performance.

As can be seen from Figure 1, applications at level 1 use labeled datasets in level 2 to train the necessary NN models in level 3, using different software frameworks in level 4. These frameworks like TensorFlow, Caffe, Pytorch etc. make training NN models with given datasets convenient, with a set of function APIs to optimize the desired loss function in an iterative process. Training typically happens in the cloud and not on an embedded edge device. Once an NN model is trained, inferences using the trained model can be run directly on embedded hardware platforms, with different processor cores as shown in level 8 of Figure 1. The platforms run the operating systems mentioned in level 7, that include the compilers in level 5 to translate the NN model based inference code to hardware instructions. The focus of this paper is in the recently introduced software layer in level 6, namely *Inference Engine*, that takes the trained NN models and first compresses them for more efficient inferences (referred to as *Model Compression* in subsequent sections), and then maps the compressed NN model to the most optimal hardware functions (referred to as *Hardware Mapping* in subsequent sections). There are various inference engines provided by different vendors (refer to level 6 of Figure 1), however, the focus of the present work is in particular to characterize the TensorRT inference framework provided by Nvidia in embedded GPUs. To the best of our knowledge, this is the first work that characterizes such software frameworks on edge devices.

This paper examines the impacts of the inference engine software (level 6) on inference accuracy and performance, using real edge applications (level 1), datasets (level 2), and a wide variety of NN models (level 3) trained on various frameworks (level 4). Blue boxes in first four levels of Figure 1 mark what we use in our

analyses. To reiterate, inference engines comprise two functional steps of *Model Compression* and *Hardware Mapping*, each step being a result of frenzied research in this domain over the last few years. *Model Compression* includes one or more methods from various recently proposed techniques [8]–[17], such as (a) fusing NN layers, (b) clustering of NN parameters or weights to have fewer unique weights [18], [19], (c) pruning or removal of weights less than a threshold [20], [21], (d) quantization of weights represented as 32 bit floating point numbers in the trained model as 16 bit half floats, 8 bit integers etc. [22], [23]. *Hardware Mapping* similarly needs to be aware of the growing set of NN processors and customized accelerators (CPU, GPU, TPU, NPU, DLA, FPGA etc.), and choose the appropriate set of library functions that are specially tuned to extract maximum performance from these heterogeneous processor cores.

As shown in the level 6 boxes of Figure 1, these inference engines are proprietary softwares from the hardware vendors namely NVIDIA, ARM, INTEL, AMD and XILINX, as these vendors only know the hardware architectures and instruction sets of their platforms in detail. Thus, the logic these platform vendors use for *Model Compression* or *Hardware Mapping* are not available in the public domain. To understand these black-box inference engines better, this paper uses state-of-the-art embedded GPU based platforms from NVIDIA (namely Jetson Xavier NX^{TM} and Jetson Xavier AGX^{TM}), that run Linux OS and the TensorRT inference engine comprising NVCC compiler. Our choice of hardware and inference engines are guided by the following factors - ❶ NVIDIA GPUs and TensorRT inference engines are extensively adopted by embedded industries, in automotive, medical, agricultural, mining, industrial automation, last mile delivery, construction, retail and other application domains [1], [2], [24]–[26] ❷ Among all inference engines, TensorRT supports the maximum number of input NN frameworks (level 4 in Figure 1) and NN models (level 3 in Figure 1), so that our examination of inference accuracy and performance can use a variety of NN models and frameworks ❸ NVIDIA's TensorRT engine includes all possible NN optimizations unlike other inference engines, as discussed in detail in [27] and shown in Figure 2.

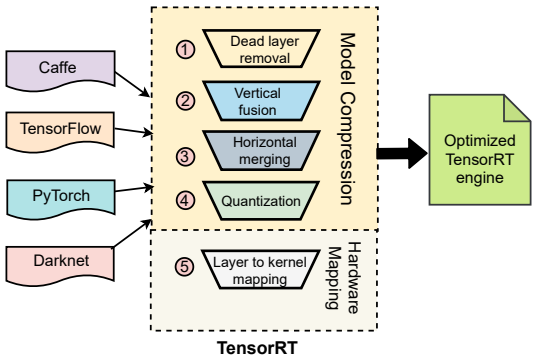


Fig. 2: TensorRT optimization steps: ❶ unused NN layers are removed, ❷ consecutive NN layers fused into one operation, ❸ multiple branches among NN layers merged, ❹ 32 bit floats quantized to 16 bit half floats or 8 bit integers, ❺ optimized NN layers are mapped to an extensive library of pre-implemented CUDA kernels.

Through rigorous experiments on real platforms with a variety of NN models, we find TensorRT (a) maintains or even slightly improves accuracy on benign and adversarial images in image classification tasks, (b) gives 23-26x gain in classification throughput and (c) can pack upto 36 concurrent NN inferences for object detection tasks. However, these positive impacts are coupled with certain unpredictable or non-deterministic behaviors, namely (a) the output for a given input image is not deterministic across different TensorRT engines, both on the same platform and across the platforms (b) inference latencies for the same NN model on the same platform change, every time a new TensorRT engine is compiled and (c) TensorRT engines compiled and run on more expensive embedded platforms with more hardware resources, can have longer inference latencies than engines compiled and run on platforms with less hardware specifications. We discuss important implications of our findings on real life applications, that embedded deployments should carefully consider. We additionally discuss interesting implications of our findings in computer architecture research that model and predict application performance.

II. Measurement Setup

In this section, we describe the edge devices comprising of NVIDIA embedded GPU cores and the neural network (NN) models used for examining TensorRT optimization engines. Additionally, we discuss the profiling tools used in our measurements and describe the image datasets and evaluation metrics used in our analyses.

A. Evaluation Platforms with Embedded NVIDIA GPU Cores

We evaluate our neural network (NN) models on two edge devices widely used in a number of embedded applications such as traffic intersection control and autonomous driving systems [25], [26]. Both devices use the GPU Volta architecture - Jetson Xavier NX^{TM} [28] and Jetson Xavier AGX^{TM} [29]. We keep the underlying GPU architecture same, so that the low level hardware instruction set remains constant and comparable across the devices, and all measured evaluation metric differences (if any) result only from the TensorRT optimizations. 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 I) obtained using *deviceQuery* [30] utility available on both the boards. Therefore, the expectation is that the same NN model should give better performance on the AGX platform, as it is more powerful in terms of hardware resources and is also an order of magnitude more expensive than NX. It should be noted here that both these embedded boards contain DLAs (Deep Learning Accelerators) that can be used for offloading some tasks from GPU. In this paper, we characterize the GPUs (excluding the DLAs) for the execution of neural networks, as GPUs are more predominantly available in edge devices as compared to DLAs. On the systems software part, we use an Ubuntu 18.04 Operating System along with cuda 10.1 for our experimentation.

	Xavier NX (GV10B)	**Xavier AGX (GV10B)**
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
Technology	12nm	12nm

TABLE I: Evaluation platforms with Embedded NVIDIA GPU

NN Model	Vision Task	Framework	# Layers	Un-optimized Model Size (MB)	TensorRT Engine for NX (MB)	TensorRT Engine for AGX (MB)
Alexnet [31]	Classification	Caffe	5 conv, 3 max pool	232.56	120.11	120.11
ResNet-18 [32]	Classification	Caffe	21 conv, 2 max pool	44.65	22.5	52.49
vgg-16 [33]	Classification	Caffe	13 conv, 5 max pool	527.8	264.7	264.7
inception-v4 [34]	Classification	Caffe	149 conv, 19 max pool	163.12	82.68	82.68
Googlenet [35]	Classification	Caffe	57 conv, 14 max pool	51.05	13.62	21.08
ssd-inception-v2 [36]	Detection	Tensorflow	90 conv, 12 max pool	95.58	48.9	48.9
Detectnet-Coco-Dog [37]	Detection	Caffe	59 conv, 12 max pool	22.82	12.45	12.45
pednet [38]	Detection	Caffe	59 conv, 12 max pool	22.82	12.72	12.73
Tiny-Yolov3 [39]	Detection	Darknet	13 conv, 6 max pool	33.1	17.83	17.83
facenet [40]	Detection	Caffe	59 conv, 12 max pool	22.82	12.03	12.05
Mobilenetv1 [41]	Detection	Tensorflow	28 conv, 1 max pool	26.07	13.50	13.53
MTCNN [42]	Detection	Caffe	12 conv, 6 max pool	1.9	3.8	4.78
fcn-resnet18-cityscapes [43]	Segmentation	PyTorch	22 conv, 1 max pool	44.95	24.7	48.78

TABLE II: Neural network (NN) models used for our evaluation, with model sizes (MB) with and without TensorRT optimizations.

B. Evaluated Neural Network (NN) Models

We use TensorRT optimized NN models from model zoo [44] as our EdgeML workloads. We classify the models into three categories based on the computer vision task they perform - *image classification models*, *object detection models* and *the image segmentation models*. We list these models, along with their architectures and frameworks associated with training each network in Table II. As can be seen, the models differ in number and type of layers. Consequently the un-optimized model sizes, as well as the TensorRT optimized engine sizes for the two embedded platforms, vary across the models. These wide variety of NN models are needed to better characterize the TensorRT optimization engines, as layer fusion, quantization and other optimizations in TensorRT might affect the different model architectures in different ways.

C. Application Profiling Softwares

We use two profiling tools for measurements: (a) **Nvprof** [45] allows profiling CUDA-related activities on CPU and GPU, including kernel execution, memory transfers, etc. It comes in various modes such as summary mode (provides the overview of GPU kernels and memory copies in the application), GPU trace mode (provides the list of all kernel launches), etc. The detailed description of *nvprof* can be read from [45]. (b) **Tegrastats** [46] is a command line based utility that provides the detailed statistics of the processor and the memory usage for Jetson based devices. It provides the statistics such as RAM usage, GPU utilisation, CPU utilisation, thermal and power statistics etc.

D. Image Datasets and NN Model Outputs

Since our evaluated NN models in Table II are predominantly related to image classification and object detection computer vision tasks, we use both well-known public image datasets, as well as a custom image dataset, to evaluate the classification and object detection accuracies. We use a subset of the *Imagenet* dataset (henceforth referred to as benign data) [47] consisting of 1000 classes with each class containing 50 images for our analysis. We additionally use an adversarially perturbed image dataset (henceforth referred to as adversarial data) [48] consisting of images with 15 different types of noises and five different severity levels that indicate the amplitude of the noise added. Increase in the severity levels (1 to 5) implies increase in the amplitude of the noise. The adversarial data comprises the same 1000 classes with 50 images for each class, as the benign data. Each image classification model takes an image from the benign or adversarial dataset as input, and outputs a class label for that image. In addition to this, we use a developing region labeled traffic image dataset [49] to evaluate object detection accuracies. We use 3896 images from this dataset for training vehicle detection CNN models and 1670 images for testing. Each object detection model takes an image comprising a traffic scene, and outputs rectangular bounding boxes for different vehicle classes (bus, car, truck etc.).

E. Evaluation Metrics

Accuracy Metrics: We use *top-1 error* as a metric to compare the classification models for both the benign and the adversarial datasets. Top-1 error is defined as the percentage of the test images on which the classification model fails to output the correct class label. For object detection task, we consider the standard Intersection Over Union (IOU) of the predicted bounding box output, with respect to the bounding box manually marked as ground truth in the test images. IOU of 0.5 is traditionally considered a true positive, with precision increasing as IOU tends towards 1. We report precision and recall values corresponding to IOU 0.75, in our analysis.

Performance Metrics: To measure performance, we use the standard metric for throughput used in computer vision domain, namely *Frames Per Second (FPS)* which is defined as the number of frames or images that are inferenced per second by the NN model. Note that we measure this time only for the inferencing part, excluding the time to load the image from the disk or camera to the main memory. We measure GPU utilization using the tegrastats utility. We use latency of each inference task as a second performance metric, in addition to throughput. We measure whether TensorRT optimizations improve or degrade the metric values on *average*, so that the application's requirements are met. We also check for performance *predictability*, so that these engines can be reliably deployed in the field for safety-critical applications.

F. Experimental Methodology

We build a TensorRT engine for each NN model using either the *Caffemodel*, *TensorFlow model*, and *PyTorch or Darknet* (refer to Table II) containing model weights obtained from the model zoo [44]. TensorRT engine building consists of multiple steps (refer to Figure 2), and needs to be executed on the edge device itself. *How these optimizations and final CUDA kernel mapping affect the accuracy and performance metrics of the NN models across the edge platforms, is the topic of interest in this paper*. In our experiments for performance metrics, each TensorRT engine obtained is executed for 10 runs on an image and the average of the 10 runs along with standard deviation across these 10 runs is reported in the analysis. The GPU frequency is set at 599 MHz for NX and 624 MHz for AGX. Note that the goal is to have the same GPU frequency for

both the platforms for a fair comparison, however, there is no GPU frequency value that is common in both the platforms. Therefore, we chose the values that are nearest to each other. AGX GPU frequency is slightly more, so along with more hardware resources, this slightly higher GPU frequency setting should also ideally give better NN performance on AGX than NX.

III. Accuracy Metric Analysis

In this section, we analyze the accuracy of the NN models listed in Table II, on our two evaluation platforms listed in Table I. We first discuss the effect on the accuracy of image classification networks for benign and adversarial datasets. Subsequently, we discuss whether the model outputs are same for a given image when TensorRT engines are built multiple times on the same platform or across different platforms.

A. Classification Accuracy on Benign and Adversarial Datasets

In this Section, we study the accuracy of un-optimized and optimized (TensorRT) image classification networks on benign (clean images) and adversarial (corrupted images) datasets. For our evaluation, we use 100 classes with each class containing 50 images, thus total of 5000 images for benign dataset. Similarly, for the adversarial dataset, we used all the 15 noises with 2 severity levels (severity 1 and severity 5) along with 100 classes with 20 images in each class, thus a total of 60,000 images (15 X 2 X 100 X 20). Table III shows the average error in percentage for three image classification networks on benign dataset using TensorRT and un-optimized engines. We can observe from the table, un-optimized networks have significantly more error by around 2-9% on average for benign image dataset compared to the TensorRT networks (on both NX and AGX).

NN Model	AGX Error(%) TensorRT	NX Error(%) TensorRT	Error(%) Unoptimized
Alexnet	45.16	45.1	47.72
ResNet-18	35.9	35.76	55.18
vgg-16	33.76	33.78	38.46

TABLE III: Top-1 Error(%) for image classification networks on benign dataset using TensorRT optimized and un-optimized engines.

NN Model	Noise Severity Level	AGX Error(%) TensorRT	NX Error(%) TensorRT	Error(%) Unoptimized
Alexnet	1	64.36	64.33	74.90
	5	90.28	90.28	94.12
ResNet-18	1	46.7	46.7	75.31
	5	87.1	87.14	97.9
vgg-16	1	40.65	40.67	51.36
	5	86.01	86.02	90.82

TABLE IV: Top-1 Error(%) for image classification networks on adversarial dataset using TensorRT optimized and un-optimized engines.

Similarly for the adversarial images, we show the error of un-optimized networks and the corresponding TensorRT engines in Table IV. We observe that for severity level 1, the average error is less by around 33.86% compared to the severity level 5 (highlighted in blue). This shows that if we increase the noise in the images, the overall accuracy of the network goes down as expected. Like the benign dataset, we observe that the TensorRT optimizations reduce the top-1 error by around 4-12% on average compared to the un-optimized networks for adversarial dataset too. The original ML models are big, possibly over-fitting the training data, so a little adversarial change affects them more. TensorRT optimizations, for example weight quantizations, can reduce this over-fitting, thereby improving classification accuracy.

Finding 1: TensorRT maintains accuracy compared to the un-optimized NN models for image classification. Un-optimized networks possibly over-fit the training data, increasing error on test data. TensorRT optimizations, like weight quantizations, can reduce over-fitting maintaining or even slightly improving accuracy.

B. Consistency of Output Labels

Once both the NN model architecture and weights/parameters have been frozen through the model training process, TensorRT optimizations should give the same output for that NN model on a given test image. Whether this output consistency holds when TensorRT engines are built across the NX and AGX platforms, or when multiple TensorRT engines are built on the same platform, is verified next. We create 3 engines of each network for each platform, thus a total of 6 engines for each network.

We tabulate the difference in the number of predictions across these engines in Table V. NX1-AGX1 shows the difference in the number of predictions for engine 1 of NX and engine 1 of AGX. Similarly other possible combinations of the engines are tabulated. The values indicate number of different predictions done by pair of TensorRT engines for the same input image, out of the 60,000 total output predictions. For the engines on the same platform, we report all the output label mismatches in Table VI for both NX and AGX. For example, 1-2 indicates the number of different predictions done by engine 1 and engine 2. Though the pairwise mismatch values comprise only 0.1-0.8% of the total number of predictions, these results show that TensorRT engines do not guarantee the same output compared to the original model, every time an engine is built on the same platform or across different platforms.

Though TensorRT is meant to optimize performance with minimal effects on NN model accuracy, if we create multiple instances of the engines, it can lead to different predictions for the same input images across these engines. Similar observations have been reported by Chou.et.al in [50]. The paper discusses the GPU non-determinism obtained when the network is trained repeatedly, leading to the difference in the accuracy of the network. This non-determinism is attributed to different ordering of threads in each execution along with the non-associative nature of the floating point arithmetic. However, our non-determinism is attributed to the unpredictable nature of TensorRT framework where each engine is a different binary executable performing different computations on the same input pixel, leading to output inconsistencies. This inconsistency in TensorRT engines' outputs for the same input image can impact real life applications, such as fining vehicles in automated rule violation detection for intersection cameras. Different vehicles may be fined based on which NX or AGX platform the TensorRT number classification model engine is built on, leading to total ridicule for such penalties in a legal setting. We discuss many such practical implications of our findings in Section VI.

Finding 2: Output of TensorRT engines might vary for the same input image across different engines built on an edge platform, and across many engines built on different platforms, even when the same un-optimized NN model (identical architecture and weights as trained on training images) is used for the TensorRT builds.

NN Model	NX1-AGX1	NX1-AGX2	NX1-AGX3	NX2-AGX1	NX2-AGX2	NX2-AGX3	NX3-AGX1	NX3-AGX2	NX3-AGX3
ResNet-18	380	362	365	379	363	359	380	362	365
vgg-16	438	451	451	438	451	451	438	451	451
inceptionv4	485	288	300	485	288	300	485	288	300
Alexnet	422	422	422	422	422	422	422	422	422

TABLE V: Number of different prediction output across engines for different platforms (cross platform engines).

Platform	NN Model	Engines 1 - 2	Engines 2 - 3	Engines 1 - 3
NX	ResNet-18	105	105	0
AGX	vgg-16	269	0	269
AGX	inceptionv4	461	296	497
AGX	ResNet-18	243	224	183

TABLE VI: Number of different prediction output across engines (platform specific engines).

IV. Performance Metric Analysis

This section examines the throughput in terms of Frames Per Second (FPS) and inference latencies per frame or image, with and without TensorRT optimizations.

A. Throughput for Image Classification Networks

Table VII shows FPS values of some image classification networks for NX and AGX TensorRT optimized and un-optimized engines. We observe that TensorRT optimizations increase FPS by around 27x on NX, compared to the un-optimized NN models. Similarly, FPS increases by around 23x on the AGX platform.

NN Model	NX-Unoptimsed	NX-TensorRT	AGX-Unoptimsed	AGX-TensorRT
Alexnet	12.1	190.4	14.2	192.5
ResNet-18	4.6	227.01	5.63	232.4
vgg-16	0.66	49.1	0.8	43.6

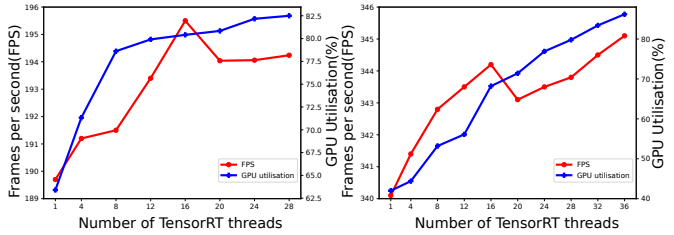
TABLE VII: FPS for TensorRT optimized and un-optimized engines

B. Throughput for Object Detection Models and Concurrency

We next examine the throughput for object detection models with TensorRT optimizations. We additionally check how much concurrency can be achieved by the TensorRT engines in running multiple instances of a CNN application, at higher GPU utilization.

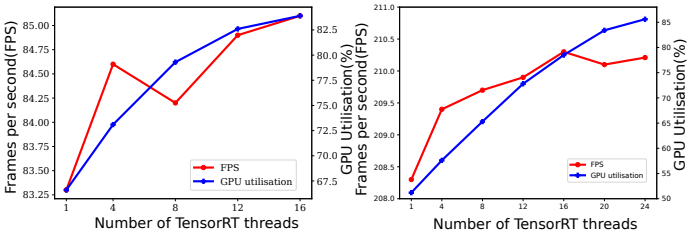
To support concurrency on the embedded GPU platforms, we use CUDA *Contexts* and *Streams*. A CUDA *Context* represents a virtual address space on a GPU and holds management data (e.g. memory allocation list, CPU-GPU mapping etc.). A CUDA *Stream* masks the memory copy latency and enables concurrent execution of multiple GPU kernels to increase GPU utilization. We use a single context and bind multiple streams within that context. Each inference task is connected to a specific stream and all the streams are executed in parallel. In this mode, all threads in the same stream use the same NN model in a shared memory address space. Therefore, if the edge applications can be designed to reuse the same NN model across different threads, then this method is a viable concurrency option. We will discuss such an application in Section VI, namely traffic intersection control, where many traffic cameras pointing in different directions, can all use the same fine-tuned CNN model such as Tiny-Yolov3 for vehicle and number plate detection tasks. This application, and similarly many others where the same NN model needs to applied to many input images, can use CUDA stream and multi-threading based concurrency.

We show the GPU utilization and FPS characteristics with increasing number of threads for a small CNN *Tiny-Yolov3* (Figure 3) and a larger CNN *Googlenet* (Figure 4), both of which are widely used CNNs for object detection applications [35], [39]. For Tiny-Yolov3, the GPU utilization for NX saturate at 82% (refer to Figure 3a). The maximum number of threads that are supported for NX are 28. For Xavier AGX, the GPU utilization increases to 86.2% for



(a) NX-Tiny-Yolov3 (b) AGX-Tiny-Yolov3

Fig. 3: FPS and GPU utilizations on NX and AGX when Tiny-Yolov3 CNN is run. NX saturates at 28 GPU threads and AGX at 36 GPU threads, with GPU utilization slightly above 80% in both cases. RAM bandwidth bottleneck marks this thread saturation points. FPS is around 196 and 346 per thread for the saturation thread count, at 1109.25 MHz GPU frequency on NX and 1377 MHz on AGX.



(a) NX-Googlenet (b) AGX-Googlenet

Fig. 4: FPS and GPU utilizations on NX and AGX when Googlenet CNN is run. NX at 16 GPU threads and AGX at 24 GPU threads, with GPU utilization slightly between 82-85% for the two boards. RAM bandwidth bottleneck marks this thread saturation points. FPS is around 85 and 210 per thread for the saturation thread count, at 1109.25 MHz GPU frequency on NX and 1377 MHz on AGX.

36 threads (refer to Figure 3b) as more resources are available in Xavier AGX, compared to NX. However, for the heavier model Googlenet, the number of threads supported are 16 and 24 (less compared to Tiny-Yolov3) for NX and AGX respectively (refer to Figure 4a and Figure 4b). The GPU utilization increases on increasing the number of threads and saturates at 82.1% and 85.6% for NX and AGX respectively. CNN threads supported are less for a heavier model compared to a lighter model, as the GPU utilization reaches saturation more quickly. Note that we obtain these statistics on the maximum GPU frequency to check for the maximum performance and the concurrency supported by these edge platforms. Additionally, the RAM bandwidth can saturate earlier as more model weights are read by the increasing number of threads. The number of threads, N, are bounded by:

$$N = \mathrm{O}\left(\frac{F_{mem} \times B_{wid}}{B_{th}}\right) \tag{1}$$

where N is the number of threads, B_{th} is the bandwidth used by one thread, F_{mem} is the memory frequency and B_{wid} is the memory bus width. However, even in presence of this memory bandwidth bottleneck, the edge devices with embedded GPUs support good number of CNN threads with a significant throughput as seen for both a light and a heavy CNN model above. This high level of NN concurrency using TensorRT engines, with excellent throughput,

NN Model	cNX rNX	cNX rAGX	cAGX rAGX	cAGX rNX	Detected Anomalies
Alexnet	44.79 (0.58)	44.34(0.73)	44.47(0.63)	46.93(1.01)	none
ResNet-18	**12.65** (0.05)	12.15(0.51)	**13.95** (0.1)	27.66(1.24)	case ❶
vgg-16	**111.77** (1.11)	112.59(2.89)	**113.45**(3.67)	117.22(1.06)	case ❶, case ❷
inception-v4	**59.89**(2.86)	63.01(2.38)	**62.84** (1.96)	60.89(1.04)	case ❶, case ❷, case ❸
Googlenet	798.12 (57.71)	740.75(45.13)	542.33(60.63)	669.45(83.69)	none
ssd-inception-v2	**34.32** (0.39)	34.24(1.34)	**41.19**(1.90)	36.49(0.55)	case ❶, case ❸
Detectnet-Coco-Dog	28.48 (0.14)	29.35(3.09)	27.77(2.49)	29.30(0.08)	case ❷
pednet	**33.43** (0.11)	38.15(1.19)	**37.28**(1.27)	35.71(1.28)	case ❶, case ❷, case ❸
facenet	**18.28** (0.07)	22.92(1.32)	**22.67** (2.54)	19.58(1.85)	case ❶, case ❷, case ❸
Tiny-Yolov3	634.89 (2.21)	470.23(4.13)	484.11(13.07)	634.93(1.69)	none
Mobilenetv1	11.97 (0.07)	13.99(0.10)	10.98(0.17)	12.2(0.09)	case ❷
MTCNN	911.43 (1.22)	804.83(13.83)	861.2(41.2)	922.12(3.38)	none
fcn-resnet18-cityscapes	**18.76** (0.87)	19.73(0.54)	**29.87**(1.36)	29.13(0.49)	case ❶, case ❷

TABLE VIII: Average run time or inference latency (in ms) of different networks along with standard deviation, when run with TensorRT optimizations. Platform AGX with more hardware resources and slightly higher GPU clock frequency during experiments, should have lower inference latencies than platform NX. When this expectation does not hold, an anomaly occurs. There are 3 possible anomalous cases of higher AGX runtime (rAGX) than NX runtime (rNX) - case ❶ (highlighted in bold font) when two different engines are compiled on NX and AGX and each platform runs its own compiled engine, case ❷ (highlighted in blue color) when both platforms run the same engine compiled on NX and case ❸ (highlighted in red color) when both platforms run the same engine compiled on AGX.

therefore has tremendous promise in terms of application support.

Finding 3: TensorRT optimizations which include layer fusion, quantizations etc. obtain a significant (around 23-27x) throughput gain over un-optimized models. The optimizations also help in packing more concurrent ML model processing threads in the embedded GPU, further increasing edgeML application performance.

C. Inference Latency for Classification and Detection Networks

We next examine the runtime or inference latency of TensorRT engines using *nvprof* for four cases - *cNX rNX* (compiled on NX and run on NX), *cNX rAGX* (compiled on NX and run on AGX), *cAGX rAGX* (compiled on AGX and run on AGX) and *cAGX rNX* (compiled on AGX and run on NX). We tabulate the run time numbers for various NN models listed in Table VIII. Each cell contains the average run time across 10 runs with standard deviations in brackets.

As AGX has more hardware resources than NX (Table I) and is run at slightly higher GPU frequency than NX, AGX should have lower inference times. We verify this expectation of better AGX performance and mark the anomalies to these expectations grouped into three categories in Table VIII - case ❶ compares cNX rNX with cAGX rAGX (anomalies marked in bold), case ❷ compares cNX rNX with cNX rAGX (anomalies marked in blue) and case ❸ compares cAGX rNX with cAGX rAGX (anomalies marked in red). Case ❶ refers to the platform specific engines, which means engine is used for inferencing on the platform on which it was built. This is what NVIDIA recommends [51] for most optimal performance, to run engines after compiling on a particular platform and not use engines compiled on other platforms. Generally, more are the hardware resources, better should be the performance in terms of execution time. However, we observe from the bold entries in Table VIII that there are seven networks (*Resnet-18, vgg-16, inception-v4, ssd-inception-v2, pednet, facenet, fcn-resnet18-cityscapes*) which surprisingly behave badly (execution time more) in AGX compared to NX. Next we check if AGX performs better when an engine is compiled on one platform, and that exact same engine is run on both NX and AGX. Case ❷ refers to the engines that are built on NX and the same engine is then run on NX and AGX. We find seven networks (*vgg-16,inception-v4,Detectnet-coco-dog,pednet, facenet, mobilenet,fcn-resnet18-cityscapes*) highlighted in blue have higher runtimes in AGX compared to NX. Case ❸ is the converse of case ❷. The engine is built on AGX and the same engine in then run on both AGX, and NX. In this case also, we observe there are four anomalous cases shown in red in Table VIII (*inception-v4,ssd-inceptionv2,pednet,facenet*) that are performing better on NX.

NN Model	cNX rNX	cNX rAGX	cAGX rAGX	cAGX rNX
Inception-v4	**31.64**(0.33)	46.06(3.03)	**42.60**(2.59)	34.56(1.89)
Pednet	**29.75**(0.3)	34.55(0.9)	**33.12**(0.37)	31.49(0.56)

TABLE IX: Average inference time (in ms) along with standard deviation, when TensorRT engines are run without nvprof tool.

All our analysis above are based on the run times obtained for NN model inferences, while the nvprof profiling tool runs in the background. However, the trends in the run time across the platforms are observed without nvprof tool as well. In Table IX, we tabulate the run time of NN inference only, without nvprof. We show only two representative examples of NN models, inception-v4 and pednet, due to space limitations. The three cases of anomalies of slower AGX run times than NX hold without nvprof for both these NN models, while the absolute run times are lower in Table IX without nvprof than in Table VIII with nvprof. Thus these anomalies of higher inference latencies on AGX with better hardware resources and slightly higher GPU clock frequency, are not a manifestation of the profiling overhead. Summarizing the insights of the three cases of inference time anomalies, we infer the following:

Finding 4: Even if an edge platform has more hardware resources such as number of cores, L1/L2 cache and RAM sizes, it does not imply that TensorRT engine will run faster on that platform. This can happen if engines are compiled on specific platforms for running on that platform (case ❶), or are compiled on the same platform and run across different platforms (case ❷ and case ❸).

V. Inference Latency Anomaly Analysis

In this section we examine possible sources of inference latency anomalies observed in the previous section. We explore these anomalies across platforms, when AGX is slower than NX. We further examine whether different engines built on the same platform have exact same NN inference times, when run on that platform.

	cNX rNX		cNX rAGX	
NN Model	**CUDA memcpy Included**	**CUDA memcpy Excluded**	**CUDA memcpy Included**	**CUDA memcpy Excluded**
ResNet-18	12.651(0.050)	3.683(0.009)	12.147(0.514)	3.088489(0.008)
Inception-v4	59.893(2.86)	23.071(2.738)	63.015(2.38)	20.908(1.027)
Pednet	33.428(0.106)	27.39(0.074)	38.147(1.186)	31.83(1.12)
Facenet	18.285(0.071)	12.618(0.047)	22.92(1.32)	17.100(1.239)
Mobilenetv1	11.97(0.070)	5.956(0.011)	13.987(0.1026)	9.342(0.016)

TABLE X: Average run time (in ms) with CUDA mempcy time included and excluded along with standard deviation

NN Model	Kernels	cNX rNX	cNX rAGX
Pednet	trt_volta_h884cudnn_256x64_ldg8_relu_exp_small_nhwc_tn_v1	8.96	11.76
	trt_volta_h884cudnn_128x128_ldg8_relu_exp_medium_nhwc_tn_v1	1.8	2.3
Facenet	trt_volta_h884cudnn_256x64_ldg8_relu_exp_small_nhwc_tn_v1	2.17	4.4
	trt_volta_h884cudnn_256x128_ldg8_relu_exp_medium_nhwc_tn_v1	2.1	4.8
	lrn::lrnForward_NChWH2	0.45	1.08
Mobilenetv1	cub::DeviceSegmentedRadixSortKernel1	0.97	1.38
	cub::DeviceSegmentedRadixSortKernel2	0.88	1.32
	trt_volta_h884cudnn_128x128_ldg8_relu_exp_medium_nhwc_tn_v1	0.73	1.23
	cuDepthwise::depthwiseConvHMMAPrefetchKernel	0.6	0.9

TABLE XI: Average run time (in ms) of similar set of kernels on NX and AGX along with standard deviation

A. Latency Anomalies of same TensorRT Engines across Platforms

Table X shows the run time of some anomalous cases of Table VIII without the CUDA memcpy time. CUDA memcpy time is *CUDA memcpyHostToDevice* transfer time i.e. the time taken to copy the TensorRT engine from CPU to GPU. We observe that the two networks (Resnet-18 and inception-v4) without the CUDA memcpy time run faster on AGX compared to NX (highlighted in blue). Thus for these two NN models, the engine copying to GPU memory is slower on AGX, though in this case we use the exact same engine built on NX on both boards. Slower CUDA memcpy leads to overall higher inference time on AGX for these two models. However, for the other three networks (pednet, facenet, and mobilenet), CUDA memcpy time does not make much of a difference to the anomaly. AGX is slower than NX even without CUDA memcpy. To examine this, we record all the CUDA kernels that are invoked by these NN models, and further obtain the running times of the individual CUDA kernels. We find that some CUDA kernels run slower on AGX compared to NX, leading to the overall increase of run time in AGX. Table XI shows the kernels that incur more run time on AGX compared to NX for pednet, facenet and mobilenet. To summarize, we find that there are two major reasons that lead to increase in run time on AGX compared to NX ❶ CUDA memcpy time is higher on AGX than NX even if the exact same TensorRT engine is copied to GPU memory on the two platforms, and ❷ some CUDA kernels run slower on the bigger platform.

Finding 5: Some NN models take longer CUDA memcpy time on AGX than NX for copying the engine to GPU memory. Some CUDA kernels take longer to execute on AGX than NX. These two factors contribute to the overall increase in the run time on AGX.

B. Latency Differences Across TensorRT engines on Same Platform

We build multiple instances of TensorRT engines for each NN model on the same platform. Table XII shows the average run time along with the standard deviation for the instances of TensorRT engines, built and run on the AGX platform. We observe that run times vary across the engines for a particular network (highlighted in blue), even when all engines are built and run on the same AGX platform. Similar inference latency differences are seen for different engines built on NX, which we exclude due to space constraints.

NN Model	Engine1	Engine2	Engine3
Alexnet	44.47(0.62)	43.72(0.7)	44.86(0.34)
ResNet-18	13.94(1.73)	9.15(1.17)	9.02(0.39)
vgg-16	113.45(4.9)	129.5(5.09)	124.7(3.86)
inception-v4	62.83(4.65)	73.5(9.08)	68.15(2.53)
Googlenet	542.33(60.62)	541.14(57.9)	540.86(58.6)
ssd-inception-v2	41.18(1.90)	40.23(1.65)	42.2(2.2)
Detectnet-Coco-Dog	27.77(2.49)	25.56(2.74)	26.3(2.86)
pednet	37.96(3.69)	37.66(2.27)	37.78(4.17)
facenet	22.67(2.54)	22.54(2.45)	21.96(2.1)
Tiny-Yolov3	484.11(13.07)	484.29(12.96)	486.54(13.43)
Mobilenetv1	10.98(0.16)	13.25(0.35)	12.13(0.67)
MTCNN	861.2(41.2)	862.5(43.5)	861.67(41.9)
fcn-resnet18-cityscapes	29.87(1.36)	34.56(1.19)	35.23(0.85)

TABLE XII: Average run time (in ms) using different TensorRT engines of the same NN models for AGX platform. Blue rows indicate run time differences across the three engines.

Inception-v4 NN model shows run time differences across the three engines in Table XII. Therefore we next check its CUDA kernel invocations across the engines. Table XIII shows how many times a representative CUDA kernel *trt_volta_h884cudnn_128x128_ldg8_relu_exp_interior_nhwc_tn_v1* is invoked in the *Inception-v4* NN model on AGX, for the three TensorRT engine instances. We observe that for engine1, engine2 and engine3, the number of calls are 9, 8 and 6 respectively for this particular kernel. To the best of our knowledge, nvprof [45] profiling tool does not output the specific arguments in a particular CUDA kernel invocation. Therefore, this is the maximum information about this kernel that we can infer from the available profiling tools. For this paucity of profiling information, though we see each kernel invocation in Table XIII takes different times in μs, it is difficult to conclude what difference in invocation arguments creates these run time differences. However, it is clear that every time a TensorRT engine is built for the same NN model on the same platform, the mapping to CUDA kernels changes – (a) a given CUDA kernel is invoked varying number of times across engines and (b) run times for those kernel invocations cannot be matched across engines. These mapping differences lead to different inference latencies across engines for a given NN model on that platform.

The effect of such non-determinism in runtime needs to be ana-

Engine1	Engine2	Engine3
229.29(15.34)	226.19(13.42)	124.40(1.27)
824.41(766.25)	111.09(1.46)	502.18(440.2)
109.62(1.92)	212.03(321.8)	110.99(2.72)
111.27(2.83)	207.92(312.2)	110.21(3.41)
249.87(440.16)	111.70(6.62)	195.75(276.79)
448.51(733.85)	149.81(2.28)	196.77(271.18)
266.11(369.92)	149.78(1.51)	
320.05(385.71)	201.95(170.58)	
296.87(467.08)		
9 calls	**8 calls**	**6 calls**

TABLE XIII: Number of invocations and run time (in μs) per invocation of a CUDA kernel in *inception-v4* on AGX platform.

lyzed in terms of application requirements, e.g. Worst Case Execution Time (WCET) analysis in real time tasks. We will discuss one such application with real time requirements, namely Advanced Driving Assistance Systems (ADAS), in Section VI.

Finding 6: TensorRT optimizes NN models and maps the optimized network to CUDA kernels based on hardware architecture. As analyzed above, this process of TensorRT engine generation comprising optimization and mapping, is not deterministic. For the same platform and across platforms, with same NN model input, the generated TensorRT engine comprises different CUDA kernels and have different inference latencies, with sometimes worse inference latencies on more powerful hardware platforms.

VI. Implications of Findings

TensorRT optimizes a trained NN model by layer fusion, quantization etc. and maps the optimized network to CUDA computational kernels for optimal performance on a given hardware architecture. Table XIV lists the positive and unpredictable impacts TensorRT engines can have on NN inferences, which are based on our experiment based findings in the previous sections. We next discuss some implications of these findings on real life applications and computer architecture research.

Finding	Summary	Impact
Maintain task accuracy	TensorRT optimizations lead to less over-fitting, thus can have same or slightly higher accuracy	Positive
Non-deterministic output	TensorRT engines of a given NN model, on same platform and across platforms might not give same output on same given input image	Unpredictable
Throughput gain, higher concurrency	Optimizations such as quantization, layer fusion etc. gives 23-27x FPS gain and can pack upto 36 concurrent NN threads at $>80\%$ GPU utilization	Positive
Non-deterministic inference times	cudamemcpy and some CUDA kernel computations take longer on bigger platforms than smaller; different TensorRT engines for same NN model vary in runtimes on same platform	Unpredictable

TABLE XIV: Summary of empirical findings on TensorRT engines

A. Impact on Applications

The NVIDIA embedded GPU platforms such as Xavier NX or AGX and TensorRT inference engines are not only research oriented hardware-software platforms, sold solely as development kits to experiment with new NN models and frameworks. Instead, they are being extensively adopted by embedded industries, in automotive, medical, agricultural, mining, industrial automation, last mile delivery, construction, retail and other application domains [24]. Whether our empirical findings have any implications for such wide-scale industry adoption of this technology, needs to be carefully analyzed in the context of specific applications.

We discuss two automotive applications in this section, that extensively use NVIDIA's edge computing solutions, namely intelligent traffic intersection control [25] and advanced driving assistance systems (ADAS) [1], [2], [26]. **Intersection control** measures traffic queue length or density in incoming lanes at the intersection and adapts the green and red signals accordingly to optimize transportation metrics, namely intersection throughput or average/worse case wait time for vehicles. They additionally detect rule violations such as vehicles jumping red light or over-speeding, detect the number plates of violating vehicles, classify the number plate into a vehicle number and issue penalty fines. Object detection and tracking to detect rule violations, image classification to read number plates - all these steps use NN model inference. **ADAS systems**, similarly, use object detection and classification tasks to detect pedestrians and other obstacles, for instance to give appropriate commands to the braking or acceleration subsystems. Both these applications can input many camera feeds to a single edge device for NN inferences, as the intersection controller needs many cameras to monitor the various incoming roads and the ADAS instrumented car also needs multiple cameras to monitor its surrounding environment in all directions.

Finding	Positive impact on traffic intersection control and Advanced Driving Assistance Systems (ADAS)
Maintain classification accuracy	Same or slightly better classification accuracy can lead to better reading of number plates for fining rule violating vehicles
Adversarial accuracy gain	Better classification accuracy on adversarial images can give more robustness against malicious attacks [52], [53] for both ADAS and traffic signal control systems
Throughput gain	Higher FPS can process frames in time, even when vehicles travel at high speed, to avoid missing fast approaching obstacles in ADAS or fining over-speeding vehicles in traffic signal control systems
Higher detection concurrency	Self driving cars or intersection polls are fitted with many cameras to look in different directions and at various resolutions. Higher NN concurrency can handle upto 36 camera feeds in one embedded platform

TABLE XV: TensorRT positive impact on automotive applications

Finding	Negative impact on traffic intersection control and Advanced Driving Assistance Systems (ADAS)
Non-deterministic detection output	Obstacles in ADAS or rule violations at signals may or may not be detected, causing unpredictable system outcomes given same camera inputs, if TensorRT engine is rebuilt for the same NN model
Non-deterministic classification output	A number plate might be read as different vehicle numbers, changing which car to fine with legal issues in the rule enforcement system, if TensorRT engine is rebuilt for the same NN model
Slower inference on bigger platform	Companies deploying edge devices might plan an infrastructure upgrade with more expensive hardware platforms (more GPU cores, RAM and LI/L2 cache sizes), only to see slower inference times compared to the earlier cheaper and smaller edge platforms
Non-deterministic inference times	Same NN model on same edge platform can have different inference latencies if TensorRT engine is rebuilt, making Worst Case Execution Time (WCET) analysis tough in real time applications. The detection inference in ADAS might not reach the braking system in time.

TABLE XVI: TensorRT negative impact on automotive applications

Table XV lists the possible advantages of using TensorRT for the two automotive applications, while Table XVI lists the potential disadvantages. Application engineers should analyze the disadvantages and take corrective actions to minimize the negative effects. For example, to reduce non-determinism across multiple TensorRT engines, a single TensorRT engine can be built once. That exact same binary can be deployed on all platforms, instead of rebuilding the engine on each platform. This can keep the computer vision outputs on a given image same across all deployed units, and also take exact same inference latency in real time tasks. Similarly, to predict whether an infrastructure upgrade using AGX vs. NX will benefit in run times, instead of committing a huge budget for the more expensive platforms, small experiments with a few units of AGX is advisable, to see if runtimes improve or degrade compared to NX. Discussions with NVIDIA's internal teams to debug longer latencies on more powerful hardware architectures for particular NN models might be useful too, if possible.

B. Impact on Micro-architecture Performance Modeling

We finally study the implications of our findings on the micro-architecture modeling in computer architecture domain. Most of the existing works in the literature [54]–[58] make use of the GPU performance counter values (obtained using profiling tools such as nvprof) along with microarchitectural parameters such as cache latencies, instruction latencies etc. to model the performance of applications on GPU based platforms. The underlying goal of all these performance models is to estimate the performance of the application on a hardware without actually executing the application on the hardware.

We showcase the impacts of non deterministic behaviour of optimization engines on the hardware performance modeling aspect using a very well known and a simplistic performance model called as Bulk Synchronous Parallel (BSP) model [59]. BSP model captures the essential characteristics of different kinds of hardware platforms as a combination of three important attributes namely *Computation, Communication and Synchronization*. These attributes are then used to determine the performance of the target parallel application on the underlying hardware. The authors in [56] have proposed a BSP inspired performance prediction model to predict the execution times of various CUDA kernels like *Matrix multiplication, Maximum sub-array* on six different GPU platforms. We adopt the model from [56] to elucidate the challenges introduced by the unpredictable nature of the optimization library in modeling the performance of a TensorRT based CNN application.

To begin with, we elaborate the performance model in [56]. This information is quintessential to comprehend the modeling challenges described in this section. The execution time of the kernel comprises of computation time and communication time (data transfer cost to and from global and shared memories). The execution time is defined in Equation 2

$$T = \frac{N \times (Comp + Comm_{GM} + Comm_{SM})}{F \times C \times \lambda} \tag{2}$$

where, T is the predicted execution time of a kernel with N threads, $Comp$ is the computational time, $Comm_{GM}$ and $Comm_{SM}$ are communication cost of global memory and shared memory accesses performed by each thread, F is the operating core frequency and C is the number of cores. The computational time ($Comp$) is the time to execute all the compute instructions and is obtained as ($\#\ instructions \times instruction\ latency$). The $Comm_{GM}$ and $Comm_{SM}$ are defined by the equations given below

$$Comp_{SM} = (ld_s + st_s) \times L_{SM}$$
$$Comp_{GM} = (ld_g + st_g - L1 - L2) \times L_{GM} + L1 \times L_{L1} + L2 \times L_{L2}$$

where ld_s and st_s represent the number of loads and stores performed by all the threads in the shared memory. Similarly, ld_g and st_g are the number of loads and stores performed by all the threads in the global memory. L1, L2 represent the corresponding cache hits. L_{SM}, L_{GM}, L_{L1} and L_{L2} are the access latencies of shared memory, global memory, L1, L2 respectively. The parameter λ captures the effects of thread divergence, shared bank conflicts and coalesced global memory accesses and is defined as the ratio of the predicted execution time of the application with the actual measured execution time. In [56], the authors obtain the λ for a single kernel using ratio of predicted and measured execution time on a single GPU platforms and then use the same λ to predict the execution time of the same kernel on different GPU platforms. Due to the assumption of minimal impact of the hardware on the application performance, they claim that for any kernel, the λ needs to obtained for a single input size and a single board. The same λ should work for all input sizes and other boards having the same microarchitecture.

We now attempt to predict the performance of a TensorRT based CNN application consisting of multiple kernels on the AGX board using the application specific parameters and λs obtained from NX board. Please note that in our case, we need to obtain multiple λs as the application consists of multiple kernels. We use microbenchmarks to obtain the static hardware parameters such as L_{SM}, L_{GM}, L_{L1} and L_{L2} for our experimental hardwares and use *nvprof* profiling tool to obtain the application specific parameters such as ld_s, st_s, ld_g, st_g. The application execution time is obtained by $\sum_{n=1}^{i} T_n * j_n$. Here, T_n is the predicted execution time of the n^{th} kernel using equation 2 and j_n is the number of times the n^{th} kernel was executed during the application execution (obtained using profiling).

Our experimental observations reveal that the above model can result in significantly higher prediction error rates due to several sources of non-determinism. ❶ In the same platform, across different compiled engines, the number of invocations of the kernel varies (refer to Table XIII). ❷ The runtime of each invocation of the same kernel varies (refer to Table XIII). The average execution time across all the invocations is used for computing T_n. ❸ The parameter λ is not able to capture all the optimizations across the platforms. ❹ The optimization engine maps the same application to different number and types of kernels when engines are built on different platforms having same microarchitecture. This inconsistency can lead to incomparable results across platforms. Hence to alleviate this, we use the engine built on NX for all the NN models used in all our experiments.

We build three different engines of *Inception-v4* application compiled on NX and obtain the execution time on AGX using the above prediction model. Table XVII shows the different λs obtained for different kernels across the 3 engines along with the associated prediction error. We tabulate only a subset of the kernels due to the lack of space. We observe that across the engines on the same platform, λ changes and thus the predicted execution time across the three engines for AGX changes leading to variability in the predicted error as highlighted in blue color. There is a significant change of around 2-13% in the prediction error across the three engines. Similarly, Table XVIII shows the results for a representative execution of *Mobilenetv1* CNN application where we build the engine on NX

Representative CUDA Kernels from NN model *inception-v4*	λ1/Error(%)	λ2/Error(%)	λ3/Error(%)
trt_volta_h884cudnn_128x128_ldg8_relu_exp_interior_nhwc_tn_v1	1.6/49.4	1.56/53.19	1.58/51.2
trt_volta_h884cudnn_128x128_ldg8_relu_exp_medium_nhwc_tn_v1	1.47/50.83	1.49/47.4	1.56/40.64
nvinfer1::poolCoalescedC	1.69/23.72	1.685/23.26	1.692/23.58
nchwTonhwc	2.12/25.67	2.1/24.68	2.11/24.9
trt_volta_h884cudnn_256x64_sliced1x2_ldg8_relu_exp_small_nhwc_tn_v1	1.75/32.23	1.72/34.53	1.728/34.1
trt_volta_h884cudnn_256x128_ldg8_relu_exp_medium_nhwc_tn_v1	2.28/7.39	2.19/12.48	2.24/9.46
nchhw2ToNchw	3.36/29.54	3.34/30.6	3.32/31.76
trt_volta_h884cudnn_128x128_ldg8_relu_exp_small_nhwc_tn_v1	2.19/22.64	2.45/30.84	2.56/35.4

TABLE XVII: Different λ values obtained across 3 engines of the *inception-v4* for different kernels along with the associated prediction error in percentage.

Representative CUDA Kernels from NN model *Mobilenetv1*	Error(%)	λ on NX	λ on AGX
cub::DeviceSegmentedRadixSortKernel	55.05	3.36	1.51
trt_volta_fp16x2_hcudnn_fp16x2 _128x32_relu_small_nn_v1	48.96	3.95	2.02
trt_volta_h884cudnn_256x64_ldg8_relu _exp_interior_nhwc_tn_v1	50.88	3.41	1.67
fusedConvolutionReluKernel	56.43	0.15	0.068
genericReformat::copyPackedKernel	60.96	6.24	2.43
nchhw2Tonhwc8	44.17	6.52	3.64
trt_volta_h884cudnn_256x64_sliced1x2 _ldg8_relu_exp_medium_nhwc_tn_v1	60.5	1.5	0.59
prepareSortData	52.08	7.58	3.63

TABLE XVIII: Prediction Error(%) for some kernels of *Mobilenetv1* along with two different λs (one obtained using metrics of NX and other using metrics of AGX)

and predict the performance on AGX. We observe error rates for some kernels as high as 60%. This error rate is the resultant prediction error that occurs when using the λs for various kernels from NX to predict the runtime on AGX. The fourth column in the table gives the actual λ for each kernel obtained from AGX that will result in zero prediction error. This gives an idea as to how far the λ obtained from NX is from the actual λ of AGX. The high error rates of individual kernels result in very low application runtime prediction accuracy. *This result corroborates with our Finding 4 that even with the increasing number of hardware resources such as number of cores, clock frequency etc, the execution time of an application can be higher leading to the unpredictability of run time using the performance model.* While we take BSP model for quantifying λ values and prediction errors, most micro-architecture based performance models [54], [55], [57] make similar assumptions about software determinism. Thus, this behaviour of TensorRT optimized NN applications will hold true for any performance model due to the non-determinism introduced by the TensorRT optimizations.

VII. Related Work

Recently, there has been extensive research on compiler optimizations [60]–[66] for NN compute graphs. For instance, Nakandala et.al [66] propose a tensor compiler that compiles NN models into a set of tensor operations. On the similar lines, [65] propose a DNN compiler design that optimizes the execution of DNN workloads on parallel accelerators. These compilers and their optimizations are now part of many NN frameworks, to create the most optimal NN architecture to speed up both NN training and inference. However, in this paper, we focus on the characterization of NN inference engines specifically optimized for embedded or edge devices, which *to the best of our knowledge is the first work to do so.*

While ours is the first paper to examine the inference engines for edge devices in level 6 of Figure 1, there have been recent prior works in characterizing level 4 frameworks and level 3 NN models on some level 8 hardware platforms for NN inferences at the edge. Pena et.al [67] investigate the inference time and energy consumption of five CNN models on three hardware platforms: NCS, Intel Joule 570X, and Raspberry Pi 3B, with frameworks TensorFlow, Caffe, NCSDK, and OpenBlas. They claim that the Raspberry Pi with Caffe uses the least amount of power. [27] characterizes several edge devices on well-known frameworks using well-known CNN workloads. They monitor the energy consumption per inference as well as the temperature behaviour of edge devices. The authors in [68] compare the latency, memory footprint, and energy consumption of TensorFlow, Caffe2, MXNet, PyTorch, and TensorFlow Lite for CNN inference on the MacBookPro, Intel's Fog Reference Design, Jetson TX2, RaspberryPi 3B+, and Huawei Nexus 6P. According to the findings of this study, TensorFlow runs larger models faster than Caffe2, and vice versa for smaller models. Our work builds in this recent research direction of empirical characterization of NN based edgeML applications on embedded platforms. However, we examine a previously unexplored software layer of inference engines, that are becoming increasingly prevalent in edge devices. We focus on accuracy and performance (throughput, latency) metrics in this paper, and present interesting findings with implications on applications.

VIII. Conclusion and Future Work

Edge devices are being increasingly used for running state of the art deep neural networks (DNNs) for various applications in different domains (automotive, medical, agricultural, mining, industrial automation, last mile delivery, construction, retail etc.). Various software optimizations have been proposed for fast NN inferences on edge platforms. In this paper, we characterize these software frameworks on GPU-based edge devices and present a number of interesting findings. We see significant performance and accuracy gains from software optimizations, with some additional highly unexpected non-deterministic behaviors, such as different outputs on the same inputs or increased execution latency for the same NN model on more powerful hardware platforms. We further discuss the impact of these analyses on real-world examples such as intelligent traffic intersection control and Advanced Driving Assistance Systems (ADAS). Furthermore, we conclude by discussing the implications of our findings for micro-architecture modeling-based application performance prediction. For the future work, we intend to expand these investigations for ARM and Intel's inference engines and check whether those engines have similar performance gains, coupled with non-deterministic behaviors as NVIDIA TensorRT. We will additionally examine the thermal and power characteristics of these NN inference engines on the edge platforms, and their trade-offs with accuracy and performance metrics, if any.

IX. Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments. We also thank our shepherd Jason Lowe-Power for providing valuable feedback that helped us to immensely improve our paper.

REFERENCES

[1] R. Yasrab, N. Gu, and X. Zhang, "An encoder-decoder based convolution neural network (cnn) for future advanced driver assistance system (adas)," *Applied Sciences*, vol. 7, no. 4, p. 312, 2017.

[2] M. Aladem and S. A. Rawashdeh, "A single-stream segmentation and depth prediction cnn for autonomous driving," *IEEE Intelligent Systems*, 2020.

[3] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J.-M. Frahm, "Re-thinking cnn frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge," in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 305–317, IEEE, 2019.

[4] J. Lee, S. Kang, J. Lee, D. Shin, D. Han, and H.-J. Yoo, "The hardware and algorithm co-design for energy-efficient dnn processor on edge/mobile devices," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 10, pp. 3458–3470, 2020.

[5] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.

[6] D. Moolchandani, A. Kumar, and S. R. Sarangi, "Accelerating cnn inference on asics: A survey," *Journal of Systems Architecture*, p. 101887, 2020.

[7] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving dnns like clockwork: Performance predictability from the bottom up," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 443–462, 2020.

[8] Microsoft, "Edge machine learning library," 2021. [Online]. Available: https://github.com/Microsoft/Ell. [Accessed: 1 July 2021].

[9] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2018.

[10] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the regularity of sparse structure in convolutional neural networks," *arXiv preprint arXiv:1705.08922*, 2017.

[11] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5687–5695, 2017.

[12] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher, "DeepIoT: Compressing deep neural network structures for sensing systems with a compressor-critic framework," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pp. 1–14, 2017.

[13] D. Dennis, C. Pabbaraju, H. V. Simhadri, and P. Jain, "Multiple instance learning for efficient sequential data classification on resource-constrained devices," in *Advances in Neural Information Processing Systems*, pp. 10953–10964, 2018.

[14] C. Gupta, A. S. Suggala, A. Goyal, H. V. Simhadri, B. Paranjape, A. Kumar, S. Goyal, R. Udupa, M. Varma, and P. Jain, "ProtoNN: Compressed and accurate kNN for resource-scarce devices," in *International Conference on Machine Learning*, pp. 1331–1340, 2017.

[15] A. Kumar, S. Goyal, and M. Varma, "Resource-efficient machine learning in 2 KB RAM for the internet of things," in *International Conference on Machine Learning*, pp. 1935–1944, 2017.

[16] A. Kusupati, M. Singh, K. Bhatia, A. Kumar, P. Jain, and M. Varma, "FastGRNN: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network," in *Advances in Neural Information Processing Systems*, pp. 9017–9028, 2018.

[17] O. Saha, A. Kusupati, H. V. Simhadri, M. Varma, and P. Jain, "RNNPool: Efficient non-linear pooling for RAM constrained inference," *arXiv preprint arXiv:2002.11921*, 2020.

[18] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pp. 176–189, 2016.

[19] M. Mathew, K. Desappan, P. Kumar Swami, and S. Nagori, "Sparse, quantized, full frame cnn for low power embedded devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 11–19, 2017.

[20] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 548–560, 2017.

[21] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, pp. 1135–1143, 2015.

[22] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.

[23] S. Migacz, "8-bit inference with tensorrt," in *GPU technology conference*, vol. 2, p. 5, 2017.

[24] "Nvidia jetson agx xavier: The ai platform for autonomous machines." https://www.nvidia.com/en-in/autonomous-machines/jetson-agx-xavier/.

[25] Advantech, *Traffic Intersection Monitoring*.

[26] Nvidia, *Self Driving Cars*.

[27] R. Hadidi, J. Cao, Y. Xie, B. Asgari, T. Krishna, and H. Kim, "Characterizing the deployment of deep neural networks on commercial edge devices," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 35–48, IEEE, 2019.

[28] "Jetson xavier nx." https://www.nvidia.com/en-in/autonomous-machines/embedded-systems/jetson-xavier-nx/.

[29] "Jetson xavier agx." https://www.nvidia.com/en-in/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[30] "Device query utility for gpu platforms." https://docs.nvidia.com/cuda/cuda-samples/index.html.

[31] 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.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[33] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[34] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, 2017.

[35] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

[36] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2818–2826, 2016.

[37] "Detectnet coco-dog." https://neuralet.com/models/jetson-nano/detectnet-coco-dog/.

[38] M. Ullah, A. Mohammed, and F. Alaya Cheikh, "Pednet: A spatio-temporal deep convolutional neural network for pedestrian segmentation," *Journal of Imaging*, vol. 4, no. 9, p. 107, 2018.

[39] Z. Yi, S. Yongliang, and Z. Jun, "An improved tiny-yolov3 pedestrian detection algorithm," *Optik*, vol. 183, pp. 17–23, 2019.

[40] F. Schroff, D. Kalenichenko, and J. Philbin, "Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 815–823, 2015.

[41] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[42] Y. Kao, R. He, and K. Huang, "Visual aesthetic quality assessment with multi-task deep learning," *arXiv preprint arXiv:1604.04970*, vol. 5, 2016.

[43] A. B. Labao and P. C. Naval, "Weakly-labelled semantic segmentation of fish objects in underwater videos using a deep residual network," in *Asian Conference on Intelligent Information and Database Systems*, pp. 255–265, Springer, 2017.

[44] "Jetson zoo." https://elinux.org/Jetson_Zoo.

[45] "Nvprof profiler." https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[46] "Tegrastats." https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html#page/Tegra%2520Linux%2520Driver%2520Package%2520Development%2520Guide%2FAppendixTegraStats.html%23wwpID0EJHA.

[47] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[48] "Adversial image dataset." https://zenodo.org/record/2235448#.YOGYKX7hXIU.

[49] M. S. Chauhan, A. Singh, M. Khemka, A. Prateek, and R. Sen, "Embedded cnn based vehicle classification and counting in non-laned road traffic," in *Proceedings of the Tenth International Conference on Information and Communication Technologies and Development*, pp. 1–11, 2019.

[50] Y. H. Chou, C. Ng, S. Cattell, J. Intan, M. D. Sinclair, J. Devietti, T. G. Rogers, and T. M. Aamodt, "Deterministic atomic buffering," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 981–995, IEEE, 2020.

[51] Nvidia, *TensorRT engine Portability*.

[52] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *Ieee Access*, vol. 6, pp. 14410–14430, 2018.

[53] A. Kurakin, I. Goodfellow, S. Bengio, Y. Dong, F. Liao, M. Liang, T. Pang, J. Zhu, X. Hu, C. Xie, *et al.*, "Adversarial attacks and defences competition," in *The NIPS'17 Competition: Building Intelligent Systems*, pp. 195–231, Springer, 2018.

[54] Q. Wang and X. Chu, "Gpgpu performance estimation with core and memory frequency scaling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 12, pp. 2865–2881, 2020.

[55] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in gpgpu applications," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pp. 11–22, 2012.

[56] M. Amaris, D. Cordeiro, A. Goldman, and R. Y. De Camargo, "A simple bsp-based model to predict execution time in gpu applications," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pp. 285–294, 2015.

[57] E. Konstantinidis and Y. Cotronis, "A practical performance model for compute and memory bound gpu kernels," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 651–658, IEEE, 2015.

[58] M. Amaris, R. Y. de Camargo, M. Dyab, A. Goldman, and D. Trystram, "A comparison of gpu execution time prediction using machine learning and analytical modeling," in *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pp. 326–333, IEEE, 2016.

[59] L. G. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[60] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 47–62, 2019.

[61] J. Roesch, S. Lyubomirsky, M. Kirisame, L. Weber, J. Pollock, L. Vega, Z. Jiang, T. Chen, T. Moreau, and Z. Tatlock, "Relay: A high-level compiler for deep learning," *arXiv preprint arXiv:1904.08368*, 2019.

[62] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 578–594, 2018.

[63] N. Rotem, J. Fix, S. Abdulrasool, G. Catron, S. Deng, R. Dzhabarov, N. Gibson, J. Hegeman, M. Lele, R. Levenstein, *et al.*, "Glow: Graph lowering compiler techniques for neural networks," *arXiv preprint arXiv:1805.00907*, 2018.

[64] M. Li, Y. Liu, X. Liu, Q. Sun, X. You, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "The deep learning compiler: A comprehensive survey," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 708–727, 2020.

[65] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling holistic deep learning compiler optimizations with rtasks," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 881–897, 2020.

[66] S. Nakandala, K. Saur, G.-I. Yu, K. Karanasos, C. Curino, M. Weimer, and M. Interlandi, "A tensor compiler for unified machine learning prediction serving," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 899–917, 2020.

[67] D. Pena, A. Forembski, X. Xu, and D. Moloney, "Benchmarking of cnns for low-cost, low-power robotics applications," in *RSS 2017 Workshop: New Frontier for Deep Learning in Robotics*, pp. 1–5, 2017.

[68] X. Zhang, Y. Wang, and W. Shi, "pcamp: Performance comparison of machine learning packages on the edges," in *{USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.