https://doi.org/10.2352/EI.2022.34.16.AVM-118

Open-Source Deep Learning Inference Libraries for Autonomous Driving systems

Kumar Desappan, Anand Pathak, Pramod Swami, Mihir Mody, Yuan Zhao*, Paula Carrillo*, Praveen Eppa and Jianzhong Xu*
Embedded Processors Business, Texas Instruments (India and *USA)

Abstract

Automated driving functions, like highway driving and parking assist, are increasingly getting deployed in high-end cars with the goal of realizing self-driving car using Deep learning (DL) techniques like convolution neural network (CNN), Transformers etc. Traditionally custom software provided by silicon vendors are used to deploy these DL algorithms on devices. This custom software is optimal for given hardware, but supports limited features, resulting in-flexible for evaluating various deep learning model architectures tradeoffs by means of rapid prototyping. This paper proposes usage of various open-source deep learning inference frameworks to quickly deploy any model architecture without any performance/latency impact. The proposed solution consists of automatic Graph Partitioning, Post-Training Quantization and Optimal Tensor Management. The proposed solution has been implemented with three open-source inference frameworks namely Tensorflow-Lite, TVM/Neo-AI-DLR and ONNX Runtime running on Linux OS. The proposed solution using open-source frameworks provides ease of use and improved coverage for network types due to fall back for unsupported features from custom software provided by silicon venders.

Introduction

The Automated driving (AD) systems implement highway driving and value parking functions at multiple automation levels (L2-L5) [1-2]. The figure 1 shows the functional block diagram to achieve these functions. The key blocks for automated driving are namely Perception, Localization, Fusion, Driving Policy, Motion Planning and Control. The multi-modality perception (Camera, radar and Lidar) is used to gather environment information around car [3]. 'Fusion' module is used to give most reliable environment using Bayesian filtering among all modalities [4]. The 'localization' module is used to find exact position of vehicle in real world co-ordinates using High definition-HD Maps and perception data. The resulting environment model is used by 'Driving Policy' module to take decision e.g. stay in lane, lane change, yield, merge etc. The decision of Driving policy module is translated in actual car movement with 'Motion planning' module accounting kinematics and passenger's comfort. Lastly, typical 'control' module is used to track actual trajectory with respect to reference by controlling actuators.

As shown in figure 1, many of these modules are built with Deep Learning (DL) based approaches either completely or partially. With the pervasive used of DL, realization of autonomous systems includes one or more purpose-built processors for DL acceleration along with general purpose processors like Cortex ARM. As these DL engines (Hardware accelerators or DSPs) are purpose built, the silicon vender would provide custom software APIs to deploy DL model on these engines. DL algorithms in ADAS or Autonomous driving system are used for various kind of modalities (Camera, RADAR, LiDAR) and algorithms (Perceptions, Planning, Fusion), so a wide range of DL model architectures needs to be deployed/evaluated on these DL engines. These purpose-built HW engines would come with fixed set of functionally (DL Layer/Operator types) and software support. With rapidly evolving DL model architectures, development of these Automated Driving Systems would need to evaluate new models which may not be supported by exiting DL offering.

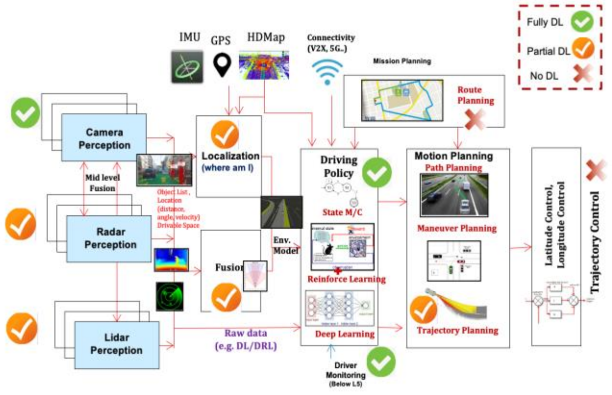

Figure 1. Computation blocks of Autonomous driving system.

Deep Learning Model Development Workflow

The Figure 2 describes the typical workflow in DL algorithms deployment. User would be training the DL models on CPU/GPUs using open-source frameworks like Tensorflow/Pytorch etc. Deployment of trained model on embedded system involves two steps. As a first step these models would be complied for a given target device with the use of offline PC tools provided by the DL engine provider. In the second step, artifacts generated in the compilation are used to run the DL inference on the target device in real-time. These traditional DL offerings lack in below vectors to address the of Automated Driving Systems.

- Ease of use to quickly deploy model on device
- Flexibility/Scalability to support wide range of DL model architectures

Recently multiple open-source DL inference frameworks (Tensorflow Lite, ONNX runtime, TVM, NEO-AI-DLR etc.) [5-8] have been developed to improve the ease of DL model deployment. These inference frameworks are primarily optimized for generic CPU cores like Cortex-A ARM. Some of these frameworks also provide options/APIs to accelerate/offload the inference using DL engines (HWA/DSP). Tensorflow Lite provides *Delegate* API to accelerate/offload supported operators of the model as a subgraph to DL HWA / DSP. Similarly, ONNX runtime provides *Execution Provider* API to achieve the same purpose.

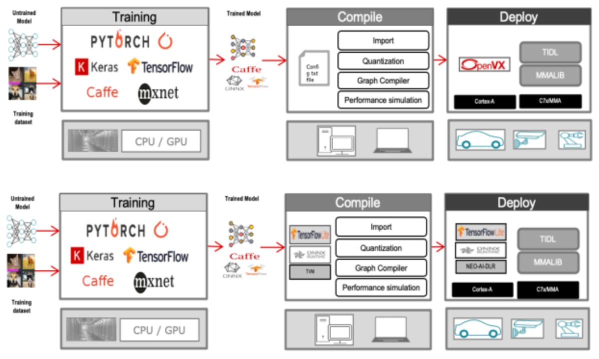

Figure 2. DL Workflow with open-source inference frameworks.

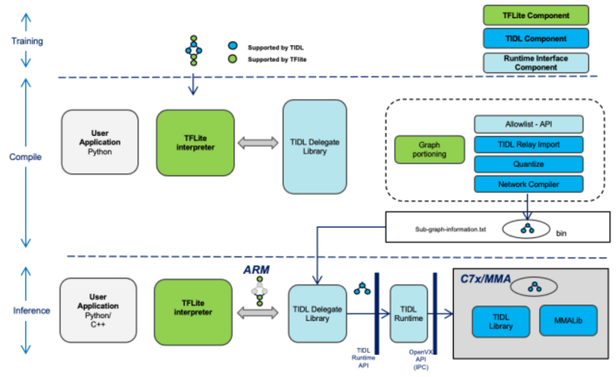

Figure 3. Inference acceleration by offloading to DL HWA

This requires to address the below challenges to fully embrace the open-source frameworks on resource constrained (Power, Latency, Cost etc.) target device. (1) *Latency/ Inferences Per Second*: Time required to run inference on these open-source DL inference frame gets impacted. Input and output tensor copy required between the ARM and the DL engine. Graph transformation and operator fusion offered by custom interfaces are NOT supported. (2) *Accuracy* - DL engines are mostly developed with fixed point (8-Bit) computation for cost efficient implementation. We would be calibrating floating point models when they need to be deployed DL engines. Enabling calibration of models for fixed point computation using open-source DL inference is the important challenge to be addressed.

Open-source Inference Frameworks

The traditional DL SW solution would enable model deployment when complete DL model (all layers) are supported by DL engine. If there are any layers in the DL model which are not supported by DL engine, then the user would NOT be able to deploy the model without silicon vendor's support. Figure 3 describes the DL SW offering modules and user interfaces for Tensorflow lite inference library. Deployment of these models are supported by Tensorflow lite runtime with the use of delegate API. Below are the major components of our proposed solution which make the deployment of user model on given DL HWA possible.

- Allow Listing - Graph Partitioning
- Post Training Quantization (PTQ)
- Optimal Memory and Tensor Management

The following sections explain the details of these major components. Similar methods are used in ONNX runtime (Execution Provider) and TVM Runtimes (Bring Your Own Codegen) to enable acceleration using DL engines.

The figure 4 describes the programming model of proposed solution for model deployment on the device. As described in the figure, it is a two-step process. The first step which is model compilation is performed on host PC using python APIs. This process identifies the operators those can be offloaded and optimizes them as sub graphs for given DL HWA/DSP. The compilation step also handles the calibration of model for fixed point computation with reduced bit-depth. The second step is the model inference in which the optimized model artifacts created during compilation step are used to perform the inference using DL HWA/DSP on target device. This inference step can be optionally simulated on host PC as well for validation or software in loop test.

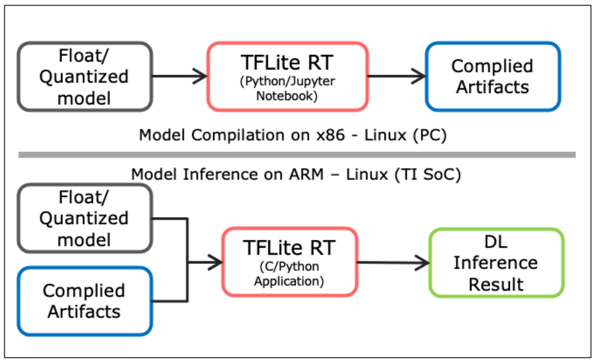
Float/ Quantized model
TFLite RT (Python/Jupyter Notebook)
Complied Artifacts
Model Compilation on x86 - Linux (PC)
Model Inference on ARM – Linux (TI SoC)
Float/ Quantized model
Complied Artifacts
TFLite RT (C/Python Application)
DL Inference Result

Figure 4. Tensorflow Lite based Programming model

Allow Listing - Graph Partitioning

This is the first module that interacts with open-source inference library. All the operators in the given model are queried against support for the same in DL HWA. Based on the supported list of operators, the entire model is partitioned into multiple sub graphs; the subgraphs with supported nodes executed on DL HWA and remaining subgraphs executed on ARM Cortex cores. The allow-list is prepared to have the graph partitioning such that over all execution of the model is optimal.

DL models can have a few operators which are not independently supported by the DL HWA but can be fused to form a supported operator (e.g. reshape-transpose-reshape layer combination can be fused to a single shuffle channel layer). Such potential layer combinations are identified at the time of allow-listing and delegated to DL HWA to be fused as part of custom model compilation. This implementation poses challenge due to the fact that, at a time, only one node is available to the allow-listing API and requires maintaining an active history of combination checks to search for layer combinations. This solution has been optimally implemented as part of our offering.

Object detection (OD) models are a very widely used class of models in autonomous driving applications. These models consist of a backbone part followed by post processing part. For several networks, the post processing (PP) part consists of a large number of operators which may not be supported by the DL HWA. This results in a large number of subgraphs formed which are not supported by DL HWA which is not optimal for latency/performance. We solve this issue with the use of a meta architecture format understandable by our model compilation tool to capture the required post processing information based on model type (e.g. SSD, Retina Net, Yolo, etc.) and implement it as a custom post processing layer supported on DL HWA. This requires allowing all the post processing layers to be delegated to DL HWA where they are then replaced by customized supported layer as part of model compilation. This is done by identifying the backbone part of the model using a backwards DFS on the model with the convolution heads of OD network as roots for DFS. The nodes not a part of backbone (which constitute post processing) are marked allowed by the allow-listing API without any further condition check. This solution results in significant performance improvement for OD models over existing open-source DL offerings. Supported Models can be found from https://github.com/TexasInstruments/edgeai-modelzoo and https://github.com/TexasInstruments/edgeai-yolov5

Post Training Quantization

The model compilation and inference being part of the same eco-system poses a challenge for calibration. (1) Calibration for quantization [9][10] cannot be done during importing of network nodes (as is the current implementation in custom DL inference SW cannot directly call existing Calibration APIs) since inputs are not available in the initialization phase of runtime. (2) Needs multiple images for calibration, interpreter invoke is called for individual images, so all images required for calibration are not available at the same time.

The Figure 5 describes proposed solution for this PTQ calibration. (1) For each of the first 'N' images ('N' – number of frames used for calibration), the floating-point mode of a reference DL inference SW is invoked, and inputs of all subgraphs are saved in file system. Floating point mode 32-bit float mode reference DL inference SW is used here, so no calibration needed. At the invoke of Nth frame, the saved floating-point inputs of all the N frames needed for calibration are available. (2) Then the DL calibration is performed using the 8/16 bit (number of bits provided by the user based on his/her inference requirement) fixed point mode of DL. (3) As a final step, calibrated network is saved to the file system to run the inference of all the remaining images. These 3 steps done as part of Subgraph Initialization. For frame 'N+1' onwards, read input image and perform inference using the saved calibrated subgraph (Subgraph inference).

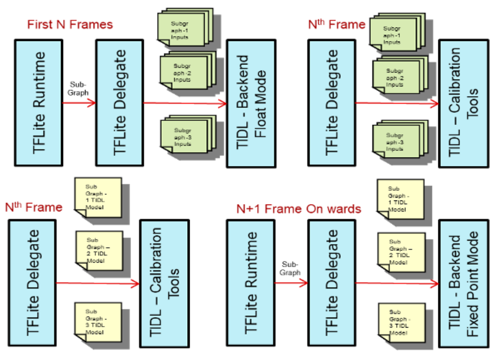

Figure 5. PTQ calibration flow

Memory and Tensor Management

The input and output tensor of the DL models are usually allocated in the user's space memory of the Linux system. This memory space would not be accessible in DL HWA. We have enabled APIs for allocating the input and output tensor buffer in shared heap memories. This avoids one round trip of buffers to the external memory and thus reduces overall memory bandwidth / power requirements. During vision-based DL model training the input image is normalized and resultant float input tensor is used as input for model. The float tensor would need 4 bytes (32-bit) for each element compared to 1 byte of the element from camera sensor output which is unsigned 8-bit integer. We propose to update model offline to change this input to 8-bit integer and push the required normalization parameters as part of the model. This figure 6 shows the example of such original model with float input and an updated model with 8-bit integer. The operators inside the dotted box are additional operators. This model is functionally exactly same as original but would require less memory bandwidth compared original. The additional operators also would be merged into the following convolution layer to reduce overall DL inference latency.

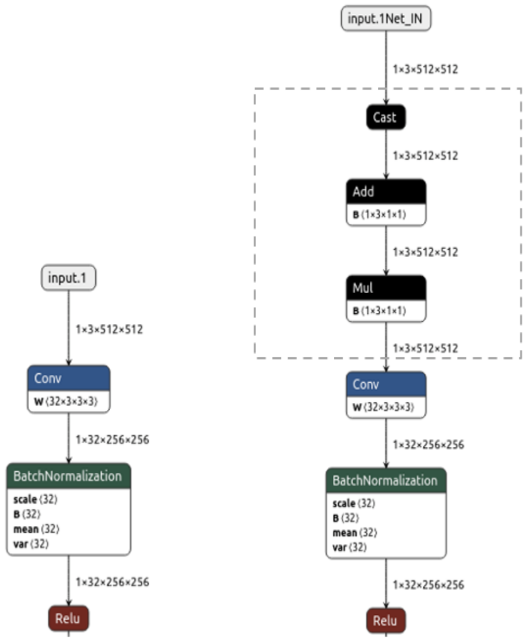

Figure 6. Float to uint8 conversion of Input tensor

Results/Discussion

Table I below shows latency in milliseconds(ms) for a few well-known DL models when executed on our offering. ARM only latency values are reported with 2 threads used for execution. Our heterogeneous solution is seen to provide up to 50 to 150x improvement over ARM only solution for open-source frameworks.

Table I: Inference latency improvement

Model name	ARM only (ms)	ARM+DSP (ms)	Improvement ratio
mobilenet_v1	89	1.45	**61**
inceptionnetv1	118	2.31	**51**
deeplabv3_mnv2	1079	10.22	**105**
ssd_mobilenet_v1	442	2.71	**163**

Table II provides comparison on a variety of factors to demonstrate the wide coverage and interface options provided by our solution based on open source frameworks when compared with custom API.

Table II: Custom and Open-source Framework Comparison

Features	Custom API	TVM, Tensorflow-Lite and ONNX runtimes
Programing interface	C/C++	Python, C/C++
Number of layer types	~30	All the (150+) TFLite/ONNX operators
Number of networks	50+	300+
Model formats	Caffe, ONNX, Tensorflow	MXNet, ONNX, Pytorch, Tensorflow, Caffe2, Keras, etc.
Model Architecture	Classification, Object detection Segmentation	Can run any model that Open-source inference libraries can support

As explained in section on “Allow Listing-Graph Partitioning”, our solution to implement post processing of object detection networks on DL HWA with the use of meta architecture is able to provide a large performance improvement. Table III shows latency in ms for (i) Optimized solution - OD backbone executed on DL HWA and post processing on ARM core (ii) Further optimized solution – OD backbone and post processing both executed on DL HWA. Results show we are able to extract about 10-15 times additional improvement in performance using the meta-architecture based solution.

Table III: Latency improvement for object detection models

Model name	Backbone in DSP and PP in ARM	End to End Model in DSP	Improvement ratio
Mobilenet V2 + SSD	59	4.8	**12.3**
Retinanet + RegnetX	352	19.5	**18.1**
Yolov3 + RegnetX	118	12.1	**9.75**

Conclusion

The paper proposed solution using open-source frameworks, that enabled ease of use and improved coverage for network types using fall back for unsupported features from the custom software. The proposed solution implemented heterogeneous execution on ARM based Linux platform and DL accelerators using open-source inference frameworks - TVM/NEI-AI-DLR, ONNX Runtime and Tensorflow-lite runtime. The proposed solution provides optimal performance and latency without compromising latency. The proposed solution consists automatic splitting model into multiple subgraphs automatically, post-training quantization and Optimal Tensor Management. All tools and examples of this implementation are made available on opensource repository from Texas Instruments.

References

[1] Mody, Mihir, Jason Jones, Kedar Chitnis, Rajat Sagar, Gregory Shurtz, Yashwant Dutt, Manoj Koul, M. G. Biju, and Aish Dubey. "Understanding vehicle e/e architecture topologies for automated driving: System partitioning and tradeoff parameters." *Electronic Imaging* 2018, no. 17 (2018): 358-1

[2] SAE J3016, “Taxonomy and Definitions for terms related to on-road automated motor vehicles”

[3] Mihir Mody, Niraj Nandan, Shashank Dabral, Hetul Sanghvi, et.al, “Image Signal Processing for Front Camera based Automated Driver Assistance System”, IEEE International Conference on Consumer Electronics, (ICCE) , Berlin, 2015

[4] Shyam Jagannathan, Mihir Mody, Jason Jones, Pramod Swami and Deepak Poddar, “Multi-sensor fusion for Automated Driving: Selecting Model and Optimizing on Embedded Platform”, AVM track, Electronic Imaging, 2018

[5] Abadi, Martín, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin et al. "Tensorflow: A system for large-scale machine learning." In 12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16), pp. 265-283. 2016.

[6] ONNX Runtime - https://onnxruntime.ai/docs/execution-providers/

[7] T. Chen et al., “TVM: An automated end-to-end optimizing compiler for deep learning,” in Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI), Carlsbad, CA, USA, 2018, pp. 578–594.

[8] Neo-AI DLR - https://github.com/neo-ai/neo-ai-dlr

[9] K. Desappan, M. Mody, M. Mathew, P. Swami and P. Eppa, "CNN Inference: Dynamic and Predictive Quantization," 2018 IEEE 8th

International Conference on Consumer Electronics - Berlin (ICCE-Berlin), 2018, pp. 1-4.

[10] M. Mody, D. Kumar, P. Swami, M. Mathew and S. Nagori, "Low cost and power CNN/deep learning solution for automated driving," 2018 19th International Symposium on Quality Electronic Design (ISQED), 2018, pp. 432-436.

Author Biography

Kumar Desappan is Senior Member of Technical Staff (SMTS) at Texas Instruments (TI) Incorporated. His domains of interest are Machine/Deep learning, image processing and computer vision algorithms with a focus on software solution for edge devices. He received Bachelor of Engineering (BE) from Anna University - Chennai in 2005

Anand Pathak is a software engineer at Texas Instruments (TI) Incorporated. His domains of interest are Machine/Deep learning, image processing and computer vision algorithms. He received Bachelor of Technology (B.Tech.) and Master of Technology (M.Tech.) degrees in Electrical Engineering from Indian Institute of Technology (IIT), Bombay in 2018

Pramod Swami is Senior Principal Engineer (SMTS) and responsible for software product solution for TI's (Texas Instrument) processor business in field of imaging and analytics. His domains of interest are digital signal processing, computer vision, deep learning and Video coding. He received his bachelor's degree in electronics and communication engineering from Malviya National Institute of Technology, Jaipur in 2001

Mihir Mody is SoC Architect lead (DMTS) responsible for roadmap and chip definition for Sitara MCU business in Texas Instrument (TI). His domains of interest are real time control, image processing, computer vision, deep learning and Video coding. He received his master's in electrical engineering from Indian Institute of Science (IISc) in 2000

Yuan Zhao is Member of Group Technical Staff (MGTS) at Texas Instruments (TI). His domains of interest are compilers, high performance computing, programming models with ease of use, and lately compilers for machine learning. He received PhD in Computer Science from Rice University in 2006

Paula Carrillo is a software developer at Texas Instruments, Inc (TI). Her domains of interest are in digital signal processing, machine learning and codecs. She received her Master of Science in Electrical Engineering from Florida Atlantic University in 2008

Praveen Eppa graduated with Bachelor of Technology (B. Tech.) in Electronics and Communications Engineering from Jawaharlal Nehru Technological University, Hyderabad in 2008. He joined Texas Instruments TI as contractor and is currently working as part of TI Deep Learning Library (TIDL) team in Embedded Processors group

Jianzhong Xu is a Software Applications Engineer at Texas Instruments (TI). His domains of interest are digital signal processing, high performance computing and machine learning. He received his Master of Science in Electrical Engineering from University of Maryland in 1997