PerfNetRT: Platform-Aware Performance Modeling for Optimized Deep Neural Networks

Ying-Chiao Liao
National Taiwan University
Taipei, Taiwan
d08922004@ntu.edu.tw

Chuan-Chi Wang
ADLINK Technology Inc.
New Taipei, Taiwan
chuan-chi.wang@adlinktech.com

Chia-Heng Tu
National Cheng Kung University
Tainan, Taiwan
chiaheng@ncku.edu.tw

Ming-Chang Kao
ADLINK Technology Inc.
New Taipei, Taiwan
fencer.kao@adlinktech.com

Wen-Yew Liang
ADLINK Technology Inc.
New Taipei, Taiwan
william.liang@adlinktech.com

Shih-Hao Hung
National Taiwan University
Taipei, Taiwan
hungsh@csie.ntu.edu.tw

2020 International Computer Symposium (ICS) | 978-1-7281-9255-0/20/$31.00 ©2020 IEEE | DOI: 10.1109/ICS51289.2020.00039

***Abstract*—As deep learning techniques based on artificial neural networks have been widely applied to diverse application domains, the delivered performance of such deep learning models on the target hardware platforms should be taken into account during the system design process in order to meet the application-specific timing requirements. Specifically, there are neural network optimization frameworks available for boosting the execution efficiency of a trained model on the vendor-specific hardware platforms, e.g., OpenVINO [1] for Intel hardware and TensorRT [2] for NVIDIA GPUs, and it is important that system designers have access to the estimated performance of the optimized models running on the specific hardware so as to make better design decisions. In this work, we have developed *PerfNetRT* to facilitate the design making process by offering the estimated inference time of a trained model that is optimized for the NVIDIA GPU using TensorRT. Our preliminary results show that PerfNetRT is able to produce accurate estimates of the inference time for the popular models, including LeNet [3], AlexNet [4] and VGG16 [5], which are optimized with TensorRT running on NVIDIA GTX 1080Ti.**

***Index Terms*—machine learning, benchmark, performance prediction, machine learning accelerators.**

I. Introduction

Deep learning (DL) models are developed to learn the patterns within the given data so as to provide answers to the specific questions with the learned knowledge. When designing a DL model, there may be more than one metric for evaluating its performance, such as accuracy and inference time, and the model training process is to find a network architecture and the parameters satisfying the application criteria of the performance metrics. Unlike accuracy, which can be calculated during an ordinary model training process, inference time of a given DL model is hard to measure during the training process because of the discrepancy of the hardware platforms for training and inference, i.e., the hardware used for the model training is different from that for the model inference. Furthermore, the inference time depends on different hardware devices to be deployed. It is inapplicable to completely measure a huge amount of hardware platforms to get their respective inference time in order to meet the application timing constraints. Thus, without a proper estimate of the inference time, the trained model, which has sufficient model accuracy, could lead to a prolonged inference time violating the application timing constrains, especially for the timing-sensitive systems.

While several works have been developed to estimate the execution time of a DL model [6]–[8], there is no prior work to predict the inference time of a given model that is optimized by the acceleration mechanism on specific hardware, such as TensorRT for NVIDIA GPUs and OpenVINO for Intel-based hardware. In recent year, the neural networks optimization frameworks get more popular for real-case scenarios, application developers are prone to adopting these frameworks to optimize their models for unleashing the computing power available from their target platforms so that the performance improvement of the optimized models on such platforms becomes a significant issue. The performance prediction method, which helps understand the performance improvement in advance, obviously alleviates several burdens of the software/hardware developers.

In this work, we propose *PerfNetRT* using the machine learning (ML) based method to predict the inference time of a DL model that is optimized with TensorRT for NVIDIA GPUs. In particular, PerfNetRT uses the multi-layer regression and the customized loss function to make accurate predictions for the unbalanced distribution of the input data. The input of *PerfNetRT* is the architecture of unoptimized DL model, and its output is the predicted total execution time of optimized model on the NVIDIA GPU. PerfNetRT saves the time for optimizing DL models, and the time taken for the experimental environment setup and the measurement of the optimized model on the physical systems. With PerfNetRT, the inference time of a given model can be calculated on-the-fly during the model training process, which further facilitates the design of a better DL model. The contributions of the proposed work are summarized as follows.

978-1-7281-9255-0/20/$31.00 ©2020 IEEE
DOI 10.1109/ICS51289.2020.00039

- An optimizer-based[1] performance modeling method is proposed to significantly increase the productivity of application and system developers since the proposed method facilitates the hardware/software co-design process by revealing the delivered performance of a trained model without actually implementing and deploying it onto the physical system.
- Our results show that PerfNetRT is useful and applicable for DL models with different levels of complexity, from shallow to deep models, i.e., LeNet, AlexNet, and VGG16.

The remaining of this paper is organizing as follows. Section II introduce the prior works that are related to prediction the exeuction time of a trained model. Section III presents the design of PerfNetRT. The explanation of how we collect and prepare the data to train the proposed MLR predictor are described in Section IV. Experimental results are shown to demonstrate the effectiveness of PerfNetRT in Section V. Finally, we conclude the paper and describe future working directions.

II. Related Work

In this section, we describe the prior work which is dedicated for estimating the execution time for the inference operations of a given deep learning model. PALEO [6] was proposed to model the total inference time by aggregating the estimated computation and communication time. PALEO adopts the two empirical parameters, PPP_{comp} for computational platform percentage of peak performance and PPP_{comm} for communication platform percentage of peak performance, as indicators of the average GPU efficiencies, in terms of computations and communications, respectively. More specifically, PPP_{comp} is used as an indicator of the average GPU utilization achieved by the given model against the peak FLOPS of the GPU, which is used to derive the computation time based on the assumption that the number of operations defined in a given model is proportional to its execution time. PPP_{comm} follow the similar procedure to estimate the communication time. As a result, the predictions reported by PALEO is highly depending on the setting of PPP, and the reported data could be misleading when improper PPP values are adopted. Justus et al. [7] proposed a DL-based performance prediction method targeting a variety of hardware platforms. Nevertheless, the proposed method always overestimates the total execution time because communication time from GPU device to the host system for each individual layer is calculated redundantly.

PerfNet [8] is a machine learning based method to predict the model execution time, where the profiling tool is used to obtain the performance characteristics during the model inference in fine-grained time intervals and the profiled data are served as the features for training the ML based method. In particular, each type of neural network layers is split into three phases and a multi-layer regression model is used to learn the execution time of each phase based on the profiled data. With the ML-based three-phase modeling approach and the fine-tuned loss function, *MAPLE*, the error rates of PerfNet are small enough for guiding the system design decisions. In the case of VGG16 on GTX 1080Ti, PerfNet yields a mean absolute percentage error of 20%. However, PerfNet is incapable of predicting performance of a TensorRT program, which is the major goal of this work.

[1]The optimizer refers to the optimizing library for the specific computing hardware, e.g., TensorRT for NVIDIA GPUs.

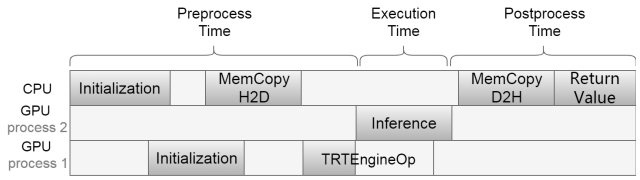

Fig. 1. The execution flow of a TensorRT-based program.

Neural architecture search (NAS) [9] is a popular approach to explore a good neural network architecture/model that suits the application constrains. Conventional research works on NAS focus on identifying good architectures/models with the highest model accuracy [9]–[11]. More recently, multiple-objective optimization approaches [12]–[14] are used to search for proper candidates optimized for multiple targets, e.g., accuracy, energy, and inference time. For example, FBNet [12] uses a look-up table keeping the time required by the neural network operators and computes the overall execution time by adding up the times for all the involved operators. With the model latency in mind, FBNet is able to find good models with high accuracy and low latency. Unfortunately, when targeting different hardware platforms, this platform-aware approach requires redundant efforts to collectively gather the performance data of all the possible operators for a deep learning model, so as to find the platform-aware model designs. On the contrary, our approach learns and estimates the inference latency at the neural network layer level, which significantly reduces the efforts of collecting performance data on physical systems. Besides, our approach can be easily extended to a variety of deep learning accelerators (DLAs) and hence, it can enlarge the search space of NAS by searching neural architectures for all candidate DLAs simultaneously.

III. Methodology

This section presents the modeling approach of PerfNetRT. In particular, the overall workflow of our layer-by-layer modeling is presented in Section III-A, and the formal definition of our modeling approach is given in Section III-B with a concrete example.

A. Workflow

Figure 1 illustrates the typical execution flow of a TensorRT-optimized neural network model on the hardware platform with one CPU and one GPU. There are six major steps for a model inference procedure, which are detailed below, and the executions of these steps are likely to be overlapped.

- Initialization: the time required for loading the model arguments and variables, where the initialization is performed on both CPU and GPU sides concurrently.
- MemCopyH2D: the time cost by copying the input data to GPU for the model inference.
- TRTEngineOp: The elapsed time for converting the serialized model subgraph to the TensorRT (TRT) computation engine.
- Inference: the computation time required by the TRT engine for the inference procedure on the input data.
- MemoryD2H: the time cost by copying back the inference result from the GPU to the CPU.
- ReturnValue: the time for returning the inference result from the TensorFlow framework to the application.

We further condense the six execution steps into three execution phases to facilitate the modeling: 1) preprocessing, 2) execution and postprocessing phases, there the second phase, *execution*, is the Inference step mentioned above. The definition of the three phases are given as follows. It is important to note that the three-phase modeling approach is different from the prior work, where PALEO [6] adopts the utilization ratios, PPP_{comp} and PPP_{comm}, for performance modeling, and Justus et al. [7] uses a coarse-grained approach to model the communication overhead, which often overestimates the overall execution time.

- Preprocess time (T_{pre}): the time for the initialization and the data transferring to the GPU.
- Execution time (T_{exe}): the time for executing the network operations on the GPU.
- Postprocess time (T_{post}): the time for return the inference result form GPU to CPU (i.e., to the application requesting the TensorFlow service).

TensorFlow Profiler is used to microbenchmark the timing for each of the six steps of a TensorRT-based model, and the profiled timing information of the six steps further falls into the corresponding phases; for example, the preprocess time is the summation of the time for Initialization, MemCopyH2D, and TRTEngineOp.

B. Formulations

The predicted inference time for a single batch DL model with k network layers is denoted as T_{single_batch} and is formulated by Equation 1.

$$T_{single\_batch} = T_{pre_0} + \sum_{i=0}^{k} T_{exe_i} + T_{post_k} \quad (1)$$

The estimated time for completing an epoch is modelled as the time of performing *n* inference tasks with batch size *m* is denoted as T_{batch}(n, m), which is estimated by Equation 2. It is important to note that $T_{single\_epoch}(n, m)$ is approximately equal to T_{single_batch} multiplied by the number of batches since the overlapped execution time among batches is negligible according to our empirical studies on the DL execution framework.

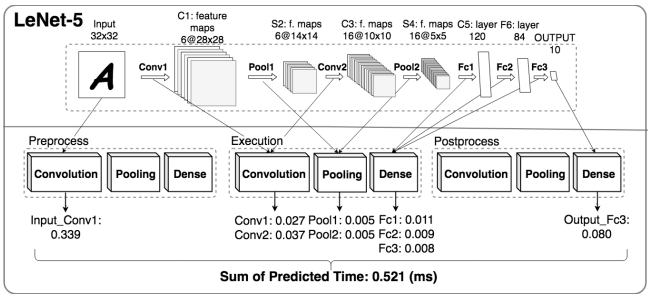

Fig. 2. Inference time estimation (ms) using PerfNetRT for LeNet-5 with batch size 1 on NVIDIA GTX 1080Ti.

$$T_{single\_epoch}(n, m) = n/m * T_{single\_batch} \quad (2)$$

Figure 2 illustrates the prediction workflow using LeNet-5 as an example. In the first phase, the input data is transferred from CPU to GPU and the first layer (Conv1) is initialized, so that we use the preprocess time model (T_{pre_0}) to predict this situation. Secondly, since the required data is already on the GPU and the other layers can be initialized asynchronously while the previous layer executes the TensorRT operation, we could use execution time model (T_{exe_i}) to evaluate the execution time of each layer$_i$. In this example, the data preprocess time (Input_Conv1) is predicted as 0.339ms, and the corresponding execution time of LeNet-5 for each layer are Conv1: 0.027ms, Pool1: 0.005ms, Conv2: 0.037ms, etc. In the third phase, we utilize the postprocess time model (T_{post_k}) to predict the time interval from GPU to CPU, which is Output_Fc3: 0.08ms. Finally, the full model predicted result could be represented as the sum of each phase which is 0.521ms by Equation 2.

IV. Training Features and Model

This section describes the features for training our ML based model in IV-A. The method that we used to collect and prepare the training data is given in Section IV-B. The proposed multi-layer regression model and loss function are introduced in IV-C.

A. Features

The features are extracted from TensorFlow 1.13 APIs and TensorRT 5.0.2, and they are categorized by the three types of the neural network layers, including convolution, pooling, and dense layers, which are described as follows.

- **Convolution layer features.** The features within this category, as shown in Table I, are extracted from the parameters of the `tensorflow.layers.conv2d` API. Each feature has its own range of numerical values, where the ranges are configured based on empirical experiences and are expected to cover a variety of the commonly used convolution layer configurations. For example, *Matrix Size* is ranging from 1 to 512 to take into account different input data sizes, e.g., 32 for CIFAR-10 and 224 for ImageNet. Moreover, *Batch Size* is usually set to the value

TABLE I
CONVOLUTION LAYER FEATURES.

Name	Description	Range
Batch Size	The number of parallel processed data.	1-64
Matrix Size	The width and height of the input data.	1-512
Kernel Size	The filter size applied to the input data composed of several channels.	1-7
Channel Input	The number of input data channels.	1-9999
Channel Output	The number of output data channels.	1-9999
Strides	The factor to controls the window shifts for the cross-correlation of the input data with kernels.	1-4
Padding	The number for controls the amount of implicit zero-paddings on both sides of input data. 0: Valid, 1: Same.	0-1
Activate Function	The number for representing whether use activation function. 0: No activate function. 1: Relu activate function.	0-1
Bias	The boolean number for determining whether adds a learnable bias to the output.	0-1

TABLE II
POOLING LAYER FEATURES.

Name	Description	Range
Batch Size	The number of parallel processed data	1-64
Matrix Size	The width and height of the input data.	1-512
Pooling Size	The size of the window for scaling down the input of several channels data.	1-7
Channel Input	The number of input data channels.	1-9999
Strides	The factor to controls the window shifts for the cross-correlation of the input data with kernels.	1-4
Padding	The number for controls the amount of implicit zero-paddings on both sides of input data. 0: Valid, 1: Same.	0-1
Activate Function	The number for representing whether use activation function. 0: No activate function. 1: Relu activate function.	0-1

smaller than 64, and we choose 64 as the maximum value for it.

- **Pooling layer features.** The parameters of the `tensorflow.layers.max_pooling2d` API are used to characterize the performance of the pooling layer, as shown in Table II. Almost all of the value ranges of the pooling-layer features are identical to those of the convolution-layer features, except for the *Pooling Size*. The maximum value of the size is set to seven based on our empirical experiences.
- **Dense layer features.** The dense layer represents a fully-connected layer found in a neural network. The features are obtained from the `tensorflow.layers.dense` API. The number of dense layer features is relatively smaller than that of the convolution layer features because of its functionality. It is important to note that we set the value ranges between 1 and 4,096 for both *Dimension Input* and *Dimension Output*, so that PerfNetRT can handle a complex multi-class classification problem.

TABLE III
DENSE LAYER FEATURES.

Name	Description	Range
Batch Size	The number of parallel processed data.	1-64
Dimension Input	The size of each input sample.	1-4096
Dimension Output	The size of each output sample.	1-4096
Activate Function	The number for representing whether use activation function. 0: No activate function. 1: Relu activate function.	0-1
Bias	The boolean number for determining whether adds a learnable bias to the output.	0-1

B. Data Collection and Preparation

Our proposed ML model is used to learn the co-relations between the features of a neural network layer and its respective execution time. In particular, we create the samples from the large design space of each layer type, where there are 7.33×10^{14}, 7.33×10^{10}, and 2.14×10^{9} possible configurations for the convolution, pooling, and dense layers, respectively. Each of the samples represents a specific configuration of the parameters for a neural network layer. We use TensorFlow to generate the 25,000 samples with different parameter settings for each layer type, where there are 75,000 samples in total for all three layer types. The samples are run and profiled by TensorFlow Profiler to collect the execution time information. The collected performance data is split into a training data set (80%), and the remains as test data set (20%).

C. Regression Model and Loss Function

The architecture of the proposed model is illustrated in Figure 3, which is based on the design of PerfNet [8]. There are four consecutive dense layers to capture the characteristics of the input data with the output sizes, ranging from 32 to 256. The mean absolute percentage logarithmic error (MAPLE) is used as the loss function to update the loss value with balanced weights in each epoch during training. MAPLE is important while training with unbalanced input data to alleviate the errors introduced by the outliers. MAPLE function is defined as equation below.

$$MAPLE = \frac{1}{n}\sum_{i=0}^{n}\left|\frac{log(1+\hat{y_i}) - log(1+y_i)}{log(1+y_i)}\right| \quad (3)$$

The values of the features are normalized with, for example, Z-score normalization, so as to have a faster converging rate. In addition, the normalized values are also served as the hyperpameters for the model training. The number of training epoch is set up to 100 with batch size of 128. The initial learning rate is set to 0.1, then divided by 2 every 40 epoches to achieve a good accuracy.

V. EXPERIMENTAL RESULTS

In this section, we give the experimental setup in Section V-A, and the performance metrics used to evaluate the efficiency of our method are introduced in Section V-B. We

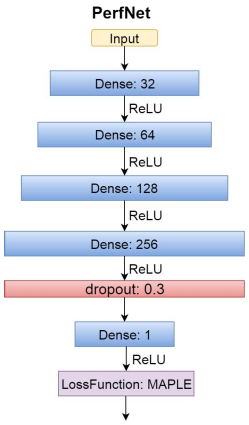

Fig. 3. The adopted multi-layer regression model.

present the prediction accuracy for each phase of the layers and compare the performance of *PerfNetRT* with that of the traditional ML based method in Section V-C. In Section V-D, we show the predictions results of the entire model on the popular neural networks, i.e., LeNet, AlexNet, and VGG16, to demonstrate the applicability of PerfNetRT on real-world cases.

A. Experimental Setup

Our experiments are performed on the Linux-based system as described below. Our testing samples are written by the TensorFlow framework and optimized with TensorRT to run on the NVIDIA GPU. The samples are profiled with the built-in profiler of the TensorFlow framework. The training of the proposed model is also done on the NVIDIA GPU.

- Operating System: Ubuntu 18.04.4 LTS
- Linux Kernel: 5.4.0-42-generic
- Programming Language: Python 3.6
- DL Framework: TensorFlow 1.13.1
- DL Optimizer: TensorRT 5.0.2.6
- CPU model: Intel i7-7700 (3.60 GHz)
- Main Memory Size: 8 GB
- GPU model: NVIDIA GTX 1080Ti (1481 MHz)

B. Performance Metrics

To evaluate the proposed mechanism, 4 performance metrics are set to check the goodness of fit for each model – mean absolute error (MAE), mean absolute percentage error (MAPE), root mean squared error (RMSE), and coefficient of determination (R^2). MAE is used to measure the closeness of the prediction result and the real elapsed time. However, MAE may not be enough, because the data distribution is unbalanced. For example, the longest convolution layer elapsed time on NVIDIA GTX1080Ti is 60.34 ms, but the shortest one is 0.01 ms. The time which is less than 10 ms appears up to 90 % of the overall distribution. If the prediction models only fit good at long elapsed time data and are defective at the short one, the model would not work in a real situation with high MAE value. In this case, MAPE will be a more proper measure than MAE by measuring every data with the same weight. RMSE is used to check the standard deviation of the differences between the predictions and targets. Lastly, R^2 represents a quick relationship value for the user. R^2 value above 0.60 is seen as worthwhile and is equals to 1.0 which indicates a perfect fit. Researchers can select the model according to the standard they want easily by checking these 4 metrics. In this study, we would like to pay more attention to MAPE from the perspective of fairness.

TABLE IV
PERFNETRT: INDIVIDUAL LAYERS PREDICTION ON GTX 1080TI WITH TENSORRT.

Layers	Phase	MAE(ms)	MAPE	RMSE(ms)	R^2
Convolution	preprocess	0.962	7.41%	1.682	0.99
	execution	0.327	17.53%	0.785	0.96
	postprocess	0.152	7.0%	0.452	0.99
Pooling	preprocess	1.133	8.64%	2.072	0.99
	execution	0.206	19.68%	0.513	0.95
	postprocess	0.142	7.27%	0.331	0.99
Dense	preprocess	0.035	5.62%	0.055	0.95
	execution	0.048	36.88%	0.103	0.95
	postprocess	0.013	19.29%	0.017	0.57

TABLE V
POLYNOMIAL REGRESSION: INDIVIDUAL LAYERS PREDICTION ON GTX 1080TI WITH TENSORRT.

Layers	Phase	MAE (ms)	MAPE	RMSE (ms)	R^2
Convolution	preprocess	0.595	15.99%	1.412	0.99
	execution	0.993	316.54%	1.758	0.84
	postprocess	1.743	318.73%	2.739	0.91
Pooling	preprocess	0.546	10.23%	0.828	0.99
	execution	0.629	312.78%	1.213	0.76
	postprocess	1.600	291.19%	2.324	0.90
Dense	preprocess	0.037	6.30%	0.057	0.95
	execution	0.090	594.68%	0.146	0.91
	postprocess	0.014	19.74%	0.017	0.56

C. Layer-level Prediction Results

The phase-wise performance data for each of the three layers generated by PerfNetRT and the polynomial regression are listed in Table IV and V, respectively. As the three types of the layers are the building blocks of a complete neural network model, it is essential that our method can give the accurate layer-level estimates. Based on the calculated R^2 values, which are above 0.95 for almost all data points, except for the postprocess phase of the dense layer, it is suggested that our method gives a good fit. On the other hand, the traditional regression based solution is not as stable as our approach and has higher prediction errors; for example, the RMSE values of the execution phase predictions for the three layers shown in Table V are larger than those shown in Table IV, which suggests our prediction results are more closer to the measured data. The MAPE values on the two tables further give quantitative comparisons exhibiting the effectiveness of PerfNetRT, which performs 10x better than the polynomial regression method.

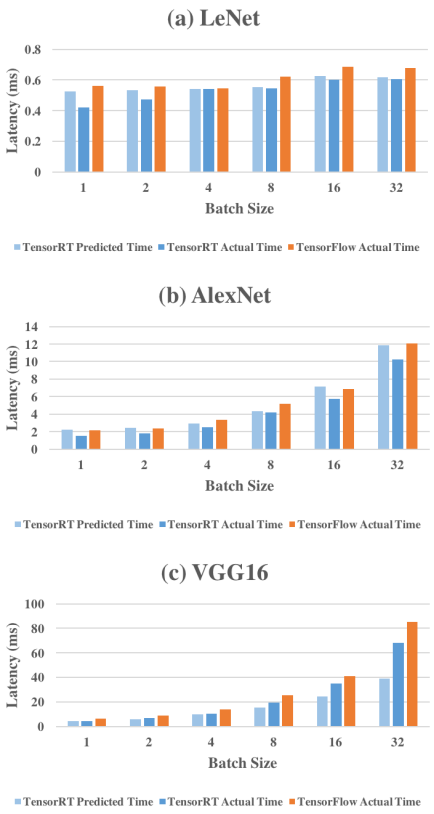

Fig. 4. Model-level performance results of (a) LeNet, (b) AlexNet, and (c) VGG16 on NVIDIA GTX1080Ti.

D. Model-level Prediction Results

While PerfNetRT is able to predict the DNN models composed of convolutional, pooling, and dense layers, as defined in Section IV-A, we present the prediction results of classic models in this section. Figure 4(a) shows the performance data of LeNet across different batch sizes from 1 to 32. TensorRT Predicted Time is reported by PerfNetRT, whereas the two actual times are measured on the physical hardware for the TensorRT optimized code and the TensorFlow (unoptimized) code. The averaged relative error for LeNet is 7.53% of all batch sizes, and the best result is achieved (0.29% error) when the batch size is four. As for the performance data plotted in Figure 4(b), the averaged relative error for all batch sizes is 23.54%, and the best result is 3.54% when the batch size is 8. Figure 4(c) shows the VGG16 prediction results, the most accurate result is batch size of 1, which yields 1.43% relative error. The averaged relative error of VGG16 for all batch sizes is 19.89%. Finally, the MAPE value of the three models is 18.9%, which is a promising result and turns out that the three-phase performance modeling approach can be extended to model the optimized neural networks.

VI. Conclusion

In this work, we have proposed a performance modeling method, *PerfNetRT*, which predicts the performance of an optimized DL model on the GPU-based system. In our experiments, we present the accuracy of our predictions at both the layer- and model-level. The results show that *PerfNetRT* can accurately predict the neural networks with different architecture complexities. While we use TensorFlow as our base framework in case studies, PerfNetRT is able to be migrated to some other ML frameworks, such as Pytorch, and MXNet. *PerfNetRT* is useful and efficient for the software developers and hardware designers so that we plan to perform similar works on other optimization frameworks and the related hardware platforms, e.g., OpenVINO for Intel's VPU [15], to facilitate the DL system designs with different types of hardware accelerators.

Acknowledgment

This work was financially supported by the Ministry of Science and Technology of Taiwan under Grants MOST 108-2218-E-002 -073 -, and sponsored by National Center for High-Performance Computing and Adlink Inc., Taipei, Taiwan.

References

[1] Intel OpenVino: https://software.intel.com/en-us/openvino-toolkit
[2] NVIDIA TensorRT: https://developer.nvidia.com/tensorrt
[3] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner: gradient-based learning applied to document recognition. In: Proceedings of the IEEE, 1998.
[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E.Hinton: ImageNet classification with deep convolutional neural networks. In: Proceedings of the 25th International Conference on Neural Information Processing Systems, 2012.
[5] Karen Simonyan, Andrew Zisserman: Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556, 2014.
[6] Hang Qi, Evan R. Sparks and Ameet Talwalkar: Paleo: A performance model for deep neural networks. In: 5th Proceedings of International Conference on Learning Representations, 2017.
[7] Daniel Justus, John Brennan, Stephen Bonner, Andrew Stephen McGough: Predicting the Computational Cost of Deep Learning Models. In: Proceedings of the IEEE International Conference on Big Data (2018)
[8] Chuan-Chi Wang, Ying-Chiao Liao, Ming-Chang Kao, Wen-Yew Liang, Shih-Hao Hung: PerfNet: Platform-Aware Performance Modeling for Deep Neural Networks. In: Proceedings of the Conference on Research in Adaptive and Convergent Systems, 2020.
[9] Barret Zoph, Quoc V. Le: Neural Architecture Search with Reinforcement Learning. arXiv:1611.01578 (2017)
[10] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, Quoc V. Le: Learning transferable architectures for scalable image recognition. arXiv:1707.07012 (2017)
[11] Bowen Baker, Otkrist Gupta, Nikhil Naik, Ramesh Raskar: Designing Neural Network Architectures using Reinforcement Learning. arXiv:1611.02167 (2017)
[12] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, Kurt Keutzer: FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. arXiv:1812.03443 (2018)
[13] Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, Peter Vajda, Joseph E. Gonzalez: FBNetV2: Differentiable Neural Architecture Search for Spatial and Channel Dimensions. In: Computer Vision and Pattern Recognition (2020)
[14] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, Quoc V. Le: MnasNet: Platform-Aware Neural Architecture Search for Mobile. arXiv:1807.11626 (2019)
[15] Intel Movidius VPU: https://www.intel.com/content/www/us/en/artificial-intelligence/movidius-myriad-vpus.html