Fast and Accurate: Machine Learning Techniques for Performance Estimation of CNNs for GPGPUs

Christopher A. Metz[𝕌] Mehran Goli[𝕌] Rolf Drechsler[𝕌,⋆]
[𝕌]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[⋆]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{cmetz, mehran, drechsler}@uni-bremen.de

***Abstract*—High performance and on-time calculations of *Machine Learning* (ML) algorithms are essential for emerging technologies such as autonomous driving, *Internet of Things* (IoT) or edge computing. One of the major algorithms used in such systems is *Convolutional Neural Networks* (CNNs), which require high computational resources. That leads designers to leverage ML accelerators like GPGPUs to meet design constraints. However, selecting the most appropriate accelerator requires *Design Space Exploration* (DSE), which is usually time-consuming and needs high manual effort.**

In this paper, we present a novel automated approach, enabling designers to fast and accurately estimate the performance of CNNs for GPGPUs in the early stage of the design process. The proposed approach uses static analysis for feature extraction and Decision Tree regression analysis for the performance estimation model. Experimental results demonstrate that our approach can predict CNNs performance with an absolute percentage error of 5.73% compared to the actual hardware.

***Index Terms*—Edge Computing, Internet of Things, Convolutional Neural Networks, Performance Estimation**

I. Introduction

Convolutional Neural Networks (CNN) can handle massive, unstructured data and are widely used in emerging technologies such as application-specific *Internet of Thing* (IoT) devices or Edge Computing Server to perform various tasks ranging from image and video recognition, and image segmentation to natural language processing [1], [2]. CNN is one of the most popular *Deep Learning* (DL) techniques used in autonomous driving where real-time and high-performance computations are essential [3]. However, they require high computational resources, e.g., the convolutional layers, made up of 4-dimensional convolutions, are responsible for over 90% of the computation and require massive processing amounts of data with potentially trillions of computations per second [4]. That leads designers to use *Machine Learning* (ML) accelerators like *General Purpose Computation on Graphics Processing Units* (GPGPUs) to gain performance and meet the time-to-market constraints.

In general, the DL life-cycle has two main phases, which are 1) training, where the DL models are trained based on the training data set, and 2) inferencing, where the finally trained DL models (e.g. a CNN) are executed on real hardware. Since the training phase is offline, high-performance computing devices (e.g., powerful data centers GPGPUs like the Nvidia V100 and Nvidia A100) are usually used without restrictions on non-functional design aspects. In contrast, the inferencing phase is an online process where the non-functional design aspects are vital in defining the overall design constraints. For example, in the case of (small) IoT devices, the selected accelerator directly impacts the design constraints such as low latency and cost of the final product [5].

Hence, selecting the right DL accelerator (e.g., GPGPUs) in this phase is of utmost importance to perform on-time computations (e.g., in the case of autonomous driving) and meet design constraints. Without a fast and automated approach, designers must build several prototypes and test numerous hardware platforms to find the right accelerator for the inferencing phase, which is very time- and cost-intensive.

To overcome this issue, several automated methods have been developed that can be divided into two main categories: GPGPU simulators [6], [7] and ML-based estimation methods [8]–[10]. GPGPU simulators such as GPGPU-Sim [6], or GPU-ocelot [7] are usually used to perform *Design Space Exploration* (DSE) and obtain the performance of a given application without the need for actual hardware execution. They use a combination of performance counters and specific hardware details to measure the performance of applications. The obtained results have an accuracy between 10% to 20% compared to the actual hardware execution [10]. However, these simulators require a significant execution time to obtain the results and thus are significantly slower than actual hardware execution. On the other hand, ML-based estimation methods provide designers with a fast solution to obtain the design parameters of a given application, such as performance. However, they either require specific performance counters, kernel settings [9], [10], or detailed platform descriptions and the scheduling of different CNNs operators on different platform processing [8], which may only sometimes be available.

This paper focuses on the performance estimation of CNNs for GPGPUs, one of the most popular DL models in various domains. We present a novel approach, allowing designers to predict a given CNN's performance for GPGPUs quickly. In contrast to the existing methods that rely on specific setups (performance counters or kernel settings), the proposed approach is developed based on a simple ML model and easy-to-extract features, namely CNNs number of trainable parameters, number of *Parallel Thread Execution* (PTX) instructions, and

This work was supported by the Data Science Center of the University of Bremen (DSC@UB)

GPGPUs architectural information.

The experimental results demonstrate the effectiveness of our approach in estimating the performance of CNNs for GPGPUs where a *Mean Absolute Percentage Error* (MAPE) of 5.73% with an R^2 of 0.45 and an adjusted R^2 of 0.19 in comparison to the actual hardware execution is obtained. Moreover, our proposed approach is significantly faster.

In summary, the main contributions of this paper are as follows:

- proposing a quick and highly accurate performance prediction model of CNNs for GPGPUs with a minimal dependency on the runtime performance counter compared to the state-of-the-art methods. The proposed approach has no runtime dependency for the final prediction,
- supporting the cross-platform prediction due to the consideration of hardware features,
- comparing different ML algorithms to obtain the best performance predictive model (i.e., Decision Tree regression analysis),
- evaluating the proposed approach on estimating the performance of different standard CNNs for various GPGPUs such as Nvidia 1080Ti, and V100S.

The rest of this paper is organized as follows. Section II discusses the related works on performance estimation for GPGPUs. Section III describes the preliminaries for CNNs and *Parallel Thread Execution* (PTX). Section IV lays the basic methodology of our approach. Section V discusses the experimental results. Finally, Section VI concludes this paper and provides an outlook.

II. Related Work

Performance estimation significantly impacts performing DSE in various domains, such as embedded software and hardware/software co-design. In general, the performance estimation (number of *Instructions Per Cycle* (IPC)) of a given application (e.g. CNN) for a target device (e.g. GPGPU) can be divided into two main categories, which are 1) ML-based methods which learn the relationship and complicated rules from a set of training data in the training phase [10]–[14], and 2) statistical analysis of the application and devices [8], [15], [16]. In the following, we explain some of the most relevant literature to our work.

The method in [15] predicts the performance of the CNNs training process on a single GPGPU. Since, in the training process, the estimation of neural networks performance has different aspects and features, the method cannot predict the required number of cycles for a given CNN running on a given GPGPU. The method in [15] uses linear regression and the so-called *Per Layer Model*, which determines the computing complexity of CNN training on a single GPGPU. In contrast, we focus on estimating the performance of a trained CNN running on different GPGPUs, often referred to as inferencing.

The method presented in [13] uses ML-based algorithms to predict the performance of a CNN on a given GPGPU. The method is applied to five different ML-algorithm: Multiple Linear Regression, Multi-Layer Perceptions, Support Vector Regression, Random Forest, and extreme Gradient Boosting. It enables designers to estimate the performance of new CNNs on a GPGPU. As predictors, a set of CNN attributes ranges from the number of hidden layers, filter, and input size to the total number of FLOPs. However, the method is limited to a single device, with no device-specific features and specifications as predictors. Therefore, for a given device, a new model needs to be trained that can only predict the performance of CNNs on this specific device.

Moreover, a new training dataset concerning the new device is also required. Thus, several models must be trained to cover, e.g., a decent design space. Hence, the main advantage of our proposed approach is the support of cross-platform estimation since we consider device features and specifications as predictors like base frequency or L2 cache.

In [10], an estimation model based on various applications run at different GPGPU configurations is introduced. The model learns how applications to scale as the GPGPU's configuration changes from the measured performance and power data. However, considering a general solution to estimate the performance and power consumption of all application types can reduce the accuracy of the final result. This is mainly because applications from various domains have different features and characteristics. Hence a single predictive model usually needs clarification to find the relationship.

The method in [11] predicts the power consumption of CNNs on GPGPUs with neural networks. The PTX instructions are categorized, and the total number per category, and the GPGPUs' architectural details, are used as predictors. The method in [17] uses a *K-Nearest Neighbors* (K-NN) regression and reduced feature space for power prediction. Due to the feature selection technique, a similar accuracy with a faster training time in comparison to [11] is achieved. However, neither of the approaches can support performance estimation.

The ALOHA method [8] presents a statistic approach for power and performance estimation of various applications considering the impact on *Neural Architecture Search* (NAS) on heterogeneous systems (e.g. GPGPUs and FPGAs). For a given device and CNN, it can estimate operations and data transfers and their deployment of computing and communication resources. This provides designers with important information for NAS. Moreover, it reports CNN's latency and energy consumption on the platform. However, this method requires an execution model for the different heterogeneous systems provided; hence, the platform and the scheduling of various CNN operations may only be available.

In [16], a time analysis framework for CNNs on multi-GPU scenarios is introduced. The analysis framework can predict the CNNs' training time on multi-GPU parallel scenarios. As mentioned earlier, features for analyzing a given CNN training time are different from the execution (inferencing) of the CNN on GPGPUs. Hence, the method cannot support CNNs performance estimation.

Overall, although the results of the previously proposed performance estimation methods are complementary to our approach, they have some limitations in terms of lacking

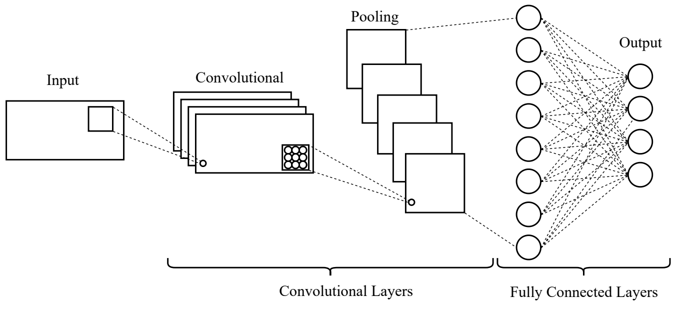


Fig. 1: The general architecture of a CNN model adjusted from [19].

the support of multi-threaded systems (e.g. GPGPUs), cross-platform prediction (i.e., the number of IPC for different types of CNNs as well as different types of GPGPUs), the need of details platforms scheduling as well as different CNN operations or focusing on the training process of the CNN. Hence, the main goal of this work is to overcome the limitations mentioned earlier.

III. Preliminaries

In this section, we explain some essential preliminaries and introductory concepts of CNNs algorithm and structure, i.e., trainable parameters, convolutional layers, stages, and *Parallel Thread Execution* (PTX) code that are necessary to understand the proposed approach and to make the paper stand alone.

A. Convolutional Neural Networks

CNNs are mainly designed for image classification or recognition but are used in many more areas. They differ in size, accuracy, and complexity depending on the use case, the number of layers, and neurons per layer [1], [2]. The so-called trainable parameters are one option to describe the complexity of a CNN. They are the number of connections of a neural network [18]. Fig. 1 illustrates the general architecture of a CNN model with n input neurons and m output neurons. In a fully connected neural network, each layer node has n connections, one to each node of the following layer. Consequently, the fully connected layers at the end of the CNN in Fig. 1 has $n \times m$ weighted connections. The connections between each model layer must be considered to obtain the CNN's total number of trainable parameters.

In the case of a convolutional layer, the trainable parameters depend on the number and size of the kernel used in the convolutional layer. A convolutional layer can have three stages which are 1) convolution, 2) activation, and 3) pooling. In the first stage, several convolutions are executed in parallel. The second stage applies the linear activation function, such as *Rectified Linear Unit* (ReLU) activation function. The third stage performs a pooling function such as the max-pooling [19]. This stage is optional and not included in all convolutional layers. For example, the Alexnet has three max-pooling layers and six convolution layers [20]. The max-pooling function selects the highest value from a kernel or

```
1   // Generated by LLVM NVPTX Back-End
2   .version 6.0
3   .target sm_61
4   .address_size 64
5   ...
6   .reqntid 256, 1, 1{
7    .reg .pred  %p<14>;
8    ...
9    mov.u32  %r13, %ctaid.x;
10   mov.u32  %r14, %tid.x;
11   shl.b32  %r15, %r13,  10;
12   shl.b32  %r16, %r14,  2;
13   or.b32   %r1, %r16, %r15;
14   setp.lt.u32  %p1, %r1,  718296;
15   @%p1 bra  LBB0_2;
16   bra.uni  LBB0_1;
17   LBB0_2:
18   ld.param.u64  %rd10, [fusion_135_param_0];
19    ...
20   LBB0_1:
21   ret;}
22  ...
```

Fig. 2: Part of the PTX file of a CNN model.

window with $n \times m$ size. Technically, the second and third stages do not have trainable parameters. However, to calculate the trainable parameters for a fully connected layer following a convolutional or pooling layer, it is necessary first to calculate the output dimension of the convolutional or pooling layer. Consequently, the architectural parameters of the pooling layer are essential too.

B. Parallel Thread Execution (PTX)

Nvidia provides the *Compute Unified Device Architecture* (CUDA) to execute general-purpose code on *Graphic Processing-Units* (GPUs) [21]. Every application written in CUDA is executed in so-called kernels[1] on the GPGPU. These are defined in the PTX code generated by compiling the CUDA code with the nvcc compiler. The PTX is an *Instruction Set Architecture* (ISA) for GPGPUs. It contains all computational instructions (such as ADD, MUL, FMA, etc.) and all memory accesses (write and read). One important factor in defining a CNN's complexity is considering the total number of PTX instructions. However, the exact number of executed instructions cannot be statically obtained from the PTX code. Fig. 2 shows a part of the PTX file generated from a given CNN. As illustrated in this figure, lines 15 and 16 demonstrate jumping operations (e.g. bra) in the PTX file. Hence, a static analyzer cannot identify whether a block of instructions is executed once or multiple times (due to the dynamic dependency). Consequently, to obtain the exact number of executed instructions, execution of the PTX file on a real GPGPU, with symbolic execution or with a simulator, is inevitable.

IV. Performance Estimation of CNNs

The proposed methodology comprises two main phases illustrated in Fig. 3. In the following, we explain each phase of the proposed approach in more detail.

[1]Please note that these kernels are not related to the kernels of the CNN mentioned above

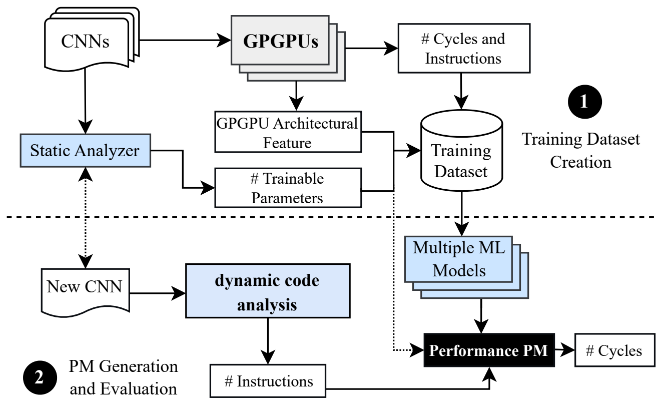


Fig. 3: Performance estimation methodology.

A. Training Dataset Creation

The performance (accurate number of IPC) of a given CNN running on a GPGPU is significantly related to two main factors: the architectural features of the GPGPU and the complexity of the CNN model. The architectural features (CUDA cores, memory, registers, or L2 cache) of a GPGPU are defined based on the type, size, and number of components the GPGPU consists of. Hence, the performance of a CNN running on different GPGPUs is not identical and varies. We extract these architectural features of GPGPUs that impact the required number of IPC when a CNN runs on them. When the training dataset is built, this information is used as a predictor (inputs). This work considers two different GPGPUs, the NVIDIA V100S and the NVIDIA GTX 1080Ti, for the training phase. They have different specifications and architectures to cover a variety of features.

The number of trainable parameters and the total number of PTX instructions can specify the complexity of neural networks. A trainable parameter is a weighted connection between the neurons, meaning more trainable parameters needs more calculations to produce the final output. To obtain the trainable parameters for a given CNN, we perform a static analysis using the *Static Analyzer* module (Fig. 3), where first, the trainable parameters for each convolutional layer are calculated. Next, based on the number of layers, the total number of trainable parameters for the CNN is achieved.

The *Static Analyzer* module performs all calculations for all 32 CNNs used during the first phase illustrated in Fig. 3. The results of the *Static Analyzer* module are unified with all other predictors in the training dataset. Modern frameworks like Pytorch and Tensorflow supply functions that can be used to calculate the trainable parameter fast and accurately.

Tab I provides an overview of the CNNs used for all experiments. They differ in various aspects, like the number of layers, neurons, or input layer size. As the table shows, most CNNs have the same input size. That is because most are trained on the ImageNet data set. However, we ensure that also CNNs with different input sizes are considered.

TABLE I: An overview of CNN models used in the experiments

Model name	Input Size	Layers	Neurons	Trainable Parameters
m-r50x1	224×224	50	15,903,016	25,549,352
m-r50x3	224×224	50	143,111,080	217,319,080
m-r101x3	224×224	101	25,3408,168	387,934,888
m-r101x1	224×224	101	28,158,248	44,541,480
m-r154x4	224×224	154	611,981,544	936,533,224
resnet101	224×224	101	55,886,036	44,601,832
resnet152	224×224	152	79,067,348	60,268,520
resnet50v2	224×224	50	31,381,204	25,568,360
resnet101v2	224×224	101	51,261,140	44,577,896
resnet152v2	224×224	152	75,755,220	60,236,904
nasnetmobile	224×224	771	27,690,705	5,289,978
nasnetlarge	331×331	1041	290,560,171	88,753,150
densenet121	224×224	121	49,926,612	7,978,856
densenet169	224×224	169	60,094,164	14,149,480
densenet201	224×224	201	77,292,244	20,013,928
mobilenet	224×224	28	16,848,248	4,231,976
inceptionv3	299×299	48	32,554,387	23,817,352
vgg16	224×224	16	15,262,696	138,357,544
vgg19	224×224	19	16,567,272	143,667,240
efficientnetb0	224×224	240	25,117,095	5,288,548
efficientnetb1	240×240	342	40,150,331	7,794,184
efficientnetb2	260×260	342	50,908,981	9,109,994
efficientnetb3	300×300	387	87,507,971	12,233,232
efficientnetb4	380×380	477	180,088,531	19,341,616
efficientnetb5	456×456	579	358,290,427	30,389,784
efficientnetb6	528×528	669	605,671,091	43,040,704
efficientnetb7	600×600	816	1,046,113,195	66,347,960
Xception	299×299	71	62,981,867	22,855,952
MobileNetV2	224×224	53	21,815,960	3,504,872
InceptionResNetV2	299×299	164	81,201,907	55,813,192
alexnet	227×227	8	650,000	58,325,066

We use dynamic code analysis and symbolic execution to calculate the number of PTX instructions needed to execute a specific CNN. Therefore, our dynamic code analysis module parses the PTX code of a CNN and generates a dependency graph $G = \{E, V\}$ structure, showing all the data dependencies inside the code. Each node $n \in V$ represents one instruction of the belonging PTX code. An edge $e \in E$ is added between to nodes n when a data dependency is detected between the belonging PTX instructions. Based on these data dependencies, a control flow is generated. Afterward, the dynamic code analysis slices the instructions (e.i., subgraph $G_{v^*} = (V', E')$ of $G = (V, E)$) that need to execute to verify which path at a branch to choose. By this, we overcome the lack of speed of a traditional simulator since we only execute a small part of the code (e.i., G_{v^*}). Moreover, this enables our approach to calculate the total number of PTX instructions for any CNN without executing it on an actual GPGPU. The total number of PTX instructions – calculated by the dynamic code analysis – is used as predictors (inputs) for the training data set.

By running CNNs on GPGPUs, we can obtain the accurate number of IPC considered as the training dataset's response (output). Thus, we are executing all of our 32 test CNNs on different GPGPUs while measuring the number of IPC with the nvprof profiler provided by NVIDIA. We ensure that the 32 CNNs, used for the dataset generation have different complexities and sizes.

An item of the final training dataset D is formally denoted by a vector:

$$d = (y, p, c_1, .., c_m, t) \tag{1}$$

Where for each observation d, the input parameters p, c, t identify the total number of instructions, the GPGPU architectural features, and the CNN trainable parameters, respectively. The output parameter y denotes the measured performance (number of IPC) of each CNN running on GPGPUs. D is the set of all measured data points n and $d_i, 1 \leq i \leq n$ denotes the i-th training data. The training data set D is split into D_t containing 70% of the data points for training and D_v containing 30% of all data points for evaluation. Moreover, we ensure that no data points exist in both data sets. Consequently, all evaluation data points are completely new to the trained model.

B. Predictive Model Generation and Evaluation

We evaluate five different standard ML algorithms, namely *Decision Tree*, *K-Nearest Neighbors* (K-NN), *Random Forest Trees*, *Linear Regression* and *eXtreme Gradient Boosting* (XG-Boost). Machine learning aims to locate patterns in the given set that provide the most straightforward explanation possible of the phenomenon. That follows Occam's razor, which states that if several theories explain a given phenomenon, the one making the least assumptions probably is the right one [22].

We selected the Linear Regression to justify if there are linear dependencies between the output (e.g., number of IPC) and the predictors. Moreover, we selected K-Nearest Neighbors and Decision Tree regressions to consider algorithms that can show non-linear dependencies. Since the Random Forest Tree is an ensemble of Decision Trees, we consider them an advanced method of the Decision Tree. Since the execution time of our predictive model is important to speed up the DSE, we take the XGBoost into account, a frequently used boosting system, to improve execution time and accuracy of tree classifications and regressions [23]. Furthermore, the runtime of KNN is significant depending on the dataset since the execution time increases linearly proportional to the number of data entries in the training data set, which can cause the necessity of faster techniques. Because the training dataset is small, we do not consider neural networks for prediction. To train neural networks, large training datasets are usually needed.

As shown in Fig. 3, for a given new CNN (in the evaluation step of the second phase), the GPGPU architectural features and the CNN trainable parameters (inputs of the predictive model) are extracted using the *Static Analyzer* module. The total number of PTX instructions is extracted from abstract PTX files by the dynamic code analysis module (as the PTX contains dynamic information such as the length of loops or jump instructions based on the comparison). Consequently, no execution of the CNN on real hardware or cycle-level simulators (which take much longer than real devices) is required.

We evaluate our experimental results using the *Mean Absolute Percentage Error* (MAPE) and the R^2 coefficient. That also enables us to compare our results to state-of-the-art research. The R^2 coefficient reports the correlation between the predictive model and underlying data. It returns a value between 0 and 1. An R^2 near 1 means a high correlation between the model and underlying data.

TABLE II: Comparison of four different ML-Regression algorithms in terms of accuracy and execution time

Regression Model	MAPE	R^2	adj. R^2
Linear Regression	8.07%	-0.0034	-0.4439
K-Nearest Neighbors	5.94%	0.34	0.08
Random Forest Tree	7.12%	0.22	-0.12
Decision Tree	5.73%	0.45	0.19
XG Boost	7.59%	0.14	-0.24

TABLE III: Predictors descriptions used by the Decision Tree

Features	**Brief description**	**Importance**
Executed Instructions	Number of instruction to be executed	0.0141
trainable params	Number connections between neurons	0.2599
Memory Bandwidth	Available memory bandwidth	0.72583

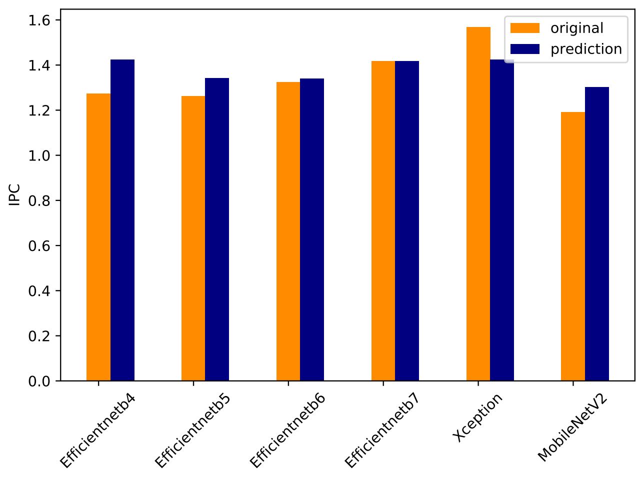
V. Experimental Results

Our experimental results demonstrate that the performance prediction based on GPGPU architectural features, the number of CNN instructions, and trainable parameters is promising. We evaluate five ML-algorithm: Linear Regression, K-Nearest Neighbors, Random Forest Tress, Decision Tree, and XG-Boost. Table II illustrates an overview of the experimental results. The Linear Regression achieves the worst results with a MAPE of 8.07% followed by XG Boost with a MAPE of 7.59%. The Random Forest Tree, the KNN, and Decision Tree are close with a MAPE of 7.12%, 5.94%, and 5.73%, respectively.

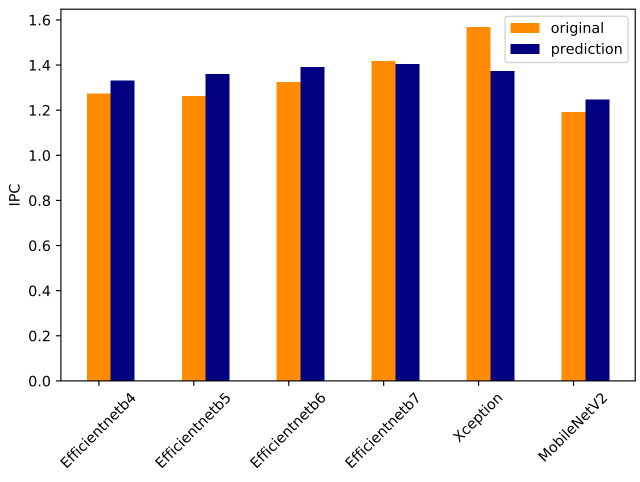
The R^2 and adjusted R^2 of the Linear Regression indicate no linear dependencies between output and predictors. The other interesting point in this experiment is that the results of the Decision Tree are better than Random Forest Trees. The main reason could be that the decision is taken based on the average value of all included Decision Trees for Random Forest Tree. Therefore, the results could be distorted if decision trees exist with poor accuracy.

All regression models considering nonlinear dependencies show promising results based on the experimental results. Due to the obtained results, we decided to build the predictive model based on the Decision Tree algorithm. However, these results can be improved by considering a more extensive range of GPGPUs for the generation of training data sets.

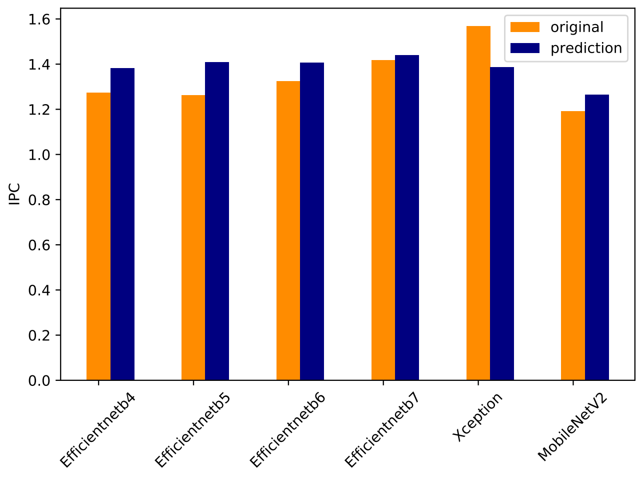
Fig. 4 shows the predicted and original performance of six randomly selected standard CNNs [20], [24], [25] (which are entirely independent of the training phase) on our final Decision Tree (predictive model). The results for the Decision Tree are illustrated in 4a, for the KNN in 4b, for the XG Boost in 4c and for the Random Forest Tree in 4d. As the figures illustrate, all predictive models' predictions are close to each other and do not differ significantly. Compared to the real hardware, namely Nvidia GTX 1080Ti, the proposed approach achieves a MAPE of 5.73%, and in the best case, the exact IPC value is predicted by the Decision Tree for the EfficientNetB7. Table III reports the three predictors with the most impact on the model. The Decision Tree predictors are chosen based on performing the Gini Coefficient at the predictive model training phase. As shown in this table III, we use one GPGPU architectural predictor i.e. Memory Bandwidth and two CNN-related predictors, i.e. number of executed instructions and



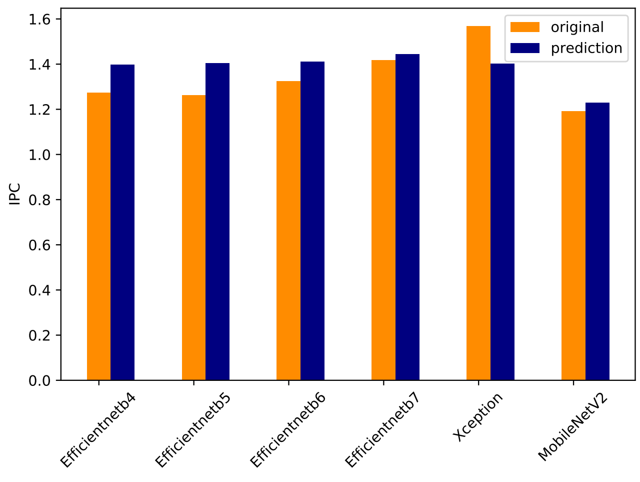
(a) Decision Tree predicted



(b) KNN predicted



(c) Gradient Boosting predicted



(d) Random Forest Tree predicted

Fig. 4: Predicted performance compared to original performance

trainable parameters. Based on our analysis, the Memory Bandwidth has the highest impact on estimating the number of cycles.

Compared to the recent method presented in [13] with a MAPE of 14.73%, our proposed approach provides designers with 2.5 times better accuracy. This comparison also shows that performance prediction based on CNNs topology, the number of instructions, and GPGPU architectural information with the Decision Tree algorithm achieves better results than [13]. Moreover, as [13], [15] do not consider hardware details as features for prediction; they cannot perform the cross-platform estimation. Therefore, their models are limited to a single GPGPU.

Assume a DSE scenario (as an application of the proposed approach) where the goal is to obtain the performance of a given CNN for n GPGPUs, the execution time of the proposed approach is defined as $T_{est} = t_{dca} + (n \times t_{pm})$ where t_{dca} and t_{pm} denote the time for our dynamic code analysis and execution time of PM, respectively. In contrast, the total time with real GPGPUs to obtain similar results (naive approach) is defined as $T_{measur} = t_p \times n$ where t_p denote the profiling time (e.g., with nvprof). Since both t_{pm} and t_{dca} are smaller than t_p (seconds vs minutes), T_{est} is in most cases almost equal to t_p or smaller. In this case, compared to the execution time of the naive approach T_{measur}, the proposed approach is significantly faster. That enables designers to estimate the performance of a given CNN at the early stages and perform a fast DSE. To prove the concept, Table IV illustrates the measured execution time of seven standard CNNs and their profiling with nvprof on seven different GPGPUs (e.g., Nvidia GTX 1080Ti, Nvidia V100S, and Nvidia Quadro P1000) for the naive approach and our proposed approach. Our approach achieves an average speedup of 33 times for one single GPU. The speedup is even higher for more larger numbers n of GPGPUs.

TABLE IV: Execution time comparison of the proposed approach versus the naive approach for eight different CNNs in the case of various GPGPUs

CNN	**Naive Approach (s)**								**Ours (s)**								
	t_p	n=1	n=2	n=3	n=4	n=5	n=6	n=7	t_{pm}	t_{dca}	n=1	n=2	n=3	n=4	n=5	n=6	n=7
efficientnet b3	663	663	1,326	1,989	2,652	3,315	3,978	4,641	11	24.8	35.8	46.8	57.0	68.8	79.8	90.8	101.8
efficientnet b4	778	778	1,556	2,334	3,112	3,890	4,668	5,446	9	24.0	33.0	42.0	51.0	60.0	69.0	78.0	87.0
efficientnet b5	950	950	1,900	2,850	3,800	4,750	5,700	6,610	8	40.3	48.3	56.3	64.3	72.3	80.3	88.3	96.3
efficientnet b6	936	936	1,872	2,808	3,768	4,680	5,616	6,552	8	60.2	68.2	76.2	84.2	92.2	100.2	108.2	116.2
efficientnet b7	1,037	1,037	2,074	3,111	4,148	5,185	6,222	7,259	1	6.8	7.8	8.8	9.8	10.8	11.8	12.8	13.8
Xception	314	314	628	942	1,256	1,570	1,884	2,198	8	23.6	31.6	39.6	47.6	55.6	63.6	71.6	79.6
MobileNet V2	343	343	686	1,029	1,372	1,715	2,058	2,401	8	12.2	20.2	28.2	36.2	44.2	52.2	60.2	68.2

VI. Conclusion

In this paper, we proposed a novel ML-based approach to estimate the performance of CNNs for GPGPUs. We illustrated how the performance of a given CNN for GPGPUs can be estimated by analyzing the CNNs topology and instructions as well as GPGPUs' architectural information. Experimental results sound promising. Our predictive model achieves a MAPE of 5.73% in performance prediction (accurate number of IPC) compared to real GPGPUs. Compared to the state-of-the-art methods, the accuracy of our predictive model is up to 2.5 times better. Moreover, it empowers designers to predict the performance of a CNN on different GPGPU architectures without the need for retraining. The proposed approach can also help designers perform NAS with hardware/software co-design to predict the performance of different generated CNN architectures for a wide range of GPGPUs' architectures without the need to execute the CNN on all of them. Hence, the DSE process can be sped up significantly by using our proposed approach.

As part of our future research, we plan to study other relevant features of CNNs such as the FLOPs or *Multiply-Accumulate* (MAC) operations and dynamic frequency scaling. Moreover, we work on preparing more standard CNNs and variations of well-known CNNs and GPGPUs to expand our training dataset.

References

[1] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, B. C. Van Esesn, A. A. S. Awwal, and V. K. Asari, "The history began from alexnet: A comprehensive survey on deep learning approaches," 2018.

[2] L. Sekanina, "Neural architecture search and hardware accelerator co-search: A survey," *IEEE Access*, vol. 9, pp. 151337–151362, 2021.

[3] S.-W. Kim, K. Ko, H. Ko, and V. C. M. Leung, "Edge-network-assisted real-time object detection framework for autonomous driving," *IEEE Network*, vol. 35, no. 1, pp. 177–183, 2021.

[4] M. Fingeroff, "Machine learning at the edge: using hls to optimize power and performance," in *The Mentor - A Siemens Business (white paper)*, 2021.

[5] A. H. Shamsan and A. R. Faridi, "Issues and challenges of using machine leaning in iot," in *2021 8th International Conference on Computing for Sustainable Global Development (INDIACom)*, pp. 232–237, 2021.

[6] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 163–174, 2009.

[7] gatech, "GPU ocelot." https://gpuocelot.gatech.edu/doxygen/, 2012.

[8] P. Busia, S. Minakova, T. Stefanov, L. Raffo, and P. Meloni, "Aloha: A unified platform-aware evaluation method for cnns execution on heterogeneous systems at the edge," *IEEE Access*, vol. 9, pp. 133289–133308, 2021.

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

[10] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou, "GPGPU performance and power estimation using machine learning," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 564–576, 2015.

[11] C. A. Metz, M. Goli, and R. Drechsler, "Early power estimation of cuda-based cnns on gpgpus: Work-in-progress," in *International Conference on Hardware/SoftwareCodesign and System Synthesis (CODES+ISSS)*, pp. 29–30, 2021.

[12] C. A. Metz, M. Goli, and R. Drechsler, "Towards neural hardware search: Power estimation of cnns for gpgpus with dynamic frequency scaling," in *Proceedings of the 2022 ACM/IEEE Workshop on Machine Learning for CAD*, MLCAD '22, (New York, NY, USA), p. 103–109, Association for Computing Machinery, 2022.

[13] H. Bouzidi, H. Ouarnoughi, S. Niar, and A. A. E. Cadi, "Performance prediction for convolutional neural networks on edge gpus," in *ACM International Conference on Computing Frontiers*, p. 54–62, 2021.

[14] L. Braun, S. Nikas, C. Song, V. Heuveline, and H. Fröning, "A simple model for portable and fast prediction of execution time and power consumption of gpu kernels," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 1, pp. 1–25, 2021.

[15] E. Gianniti, L. Zhang, and D. Ardagna, "Performance prediction of gpu-based deep learning applications," in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pp. 167–170, 2018.

[16] Z. Pei, C. Li, X. Qin, X. Chen, and G. Wei, "Iteration time prediction for cnn in multi-gpu platform: Modeling and analysis," *IEEE Access*, vol. 7, pp. 64788–64797, 2019.

[17] C. A. Metz, M. Goli, and R. Drechsler, "Ml-based power estimation of convolutional neural networks on gpgus," in *IEEE International Symposium on Design and Diagnstics of Electronic Circuits and Systems*, pp. 166–171, 2022.

[18] S. Mallick and S. Nayak, "Number of parameters and tensor sizes in a convolutional neural network (CNN)." https://learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network, 2018.

[19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems* (F. Pereira, C. Burges, L. Bottou, and K. Weinberger, eds.), vol. 25, Curran Associates, Inc., 2012.

[21] Nvidia, "CUDA toolkit documentation." https://docs.nvidia.com/cuda/, 2022.

[22] D. M. Hawkins, "The problem of overfitting," *Journal of chemical information and computer sciences*, vol. 44, no. 1, pp. 1–12, 2004.

[23] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, (New York, NY, USA), p. 785–794, Association for Computing Machinery, 2016.

[24] M. Tan and Q. V. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning*, vol. 97, pp. 6105–6114, 2019.

[25] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1800–1807, 2017.