Hardware Execution Time Prediction for Neural Network Layers

Adrian Osterwind[1] [0000-0002-0752-8698], Julian Droste-Rehling[2], Manoj-Rohit Vemparala[3], and Domenik Helms[1] [0000-0001-8570-8363]

[1] German Aerospace Center (DLR)
Institute of Systems Engineering for Future Mobility
`firstname.lastname@dlr.de`
https://www.dlr.de/se/
[2] Siemens AG (Bremen)
`firstname.lastname@siemens.com`
[3] BMW Autonomous Driving
`firstname.lastname@bmw.com`

Abstract. We present an estimation methodology, accurately predicting the execution time for a given embedded Artificial Intelligence (AI) accelerator and a neural network (NN) under analysis. The timing prediction is implemented as a python library called Model of Neural Network Execution Time (MONNET) and is able to perform its predictions analyzing the Keras description of an NN under test within milliseconds. This enables several techniques to design NNs for embedded hardware. Designers can avoid training networks which could be functionally sufficient but will likely fail the timing requirements. The technique can also be included into automated network architecture search algorithms, enabling exact hardware execution times to become one contributor to the search's target function.
In order to perform precise estimations for a target hardware, each new hardware needs to undergo an initial automatic characterization process, using tens of thousands of different small NNs. This process may need several days, depending on the hardware.
We tested our methodology for the Intel Neural Compute Stick 2, where we could achieve an root mean squared percentage error (RMSPE) below 21 % for a large range of industry relevant NNs from vision processing.

Keywords: Execution time · Prediction · Neural Networks · Analytical Model.

1 Introduction

With the constant rise of Artificial Intelligences (AIs) and neural networks (NNs) in the industry it becomes important to obtain definitive data about execution constraints of these algorithms. An algorithm must be verified to be able to work within a certain set of hardware and application constraints. As an example the execution time may not exceed a certain threshold in an autonomous vehicle, so that it is still capable of reacting to the input within safety limits.

The best way to determine this execution time is to measure it on the target hardware. For complex neural networks and difficult to obtain hardware this is not always feasible, for example in an automated network architecture search (NAS). One way of performing hardware execution time aware NAS is to rely on readily available metrics such as MAC count or number of parameters, which leads to suboptimal results [10]. Another is to obtain the execution metrics through hardware in the loop measurements, where feasible [3]. An alternative is to estimate the execution time of the neural network.

The goal of this work is to develop a gray box modeling methodology, which is capable of estimating the latency of a given NN, when running on a specific hardware. This library will be called Model of Neural Network Execution Time (MONNET). After an initial time intensive analysis (characterization) of the hardware, the estimator has to be able to run independently of the hardware itself and within an execution time, small enough to allow comparing different solutions in a design space exploration or network architecture search conveniently.

The only input parameters of the final timing estimator have to be the topology of the NN graph and the characterization data for the target hardware. Thus, it will be possible to apply the estimator directly after specification, avoiding time expensive training of solutions, which do not meet given constraints.

Another design constraint is the reduction of hardware knowledge needed to port the estimator to a different target platform. It should be possible to define a model of a layer type and use it on multiple hardware types. The only hardware related knowledge required should be how to deploy and benchmark a neural network on the target hardware.

The rest of this work is organized as follows. In section 2 similar and flanking work is discussed. This results in a new approach to execution time estimation in section 3. The timing model is leveraged to estimate the execution time in section 4. Experimental results of the approach and an evaluation of memory modeling are discussed in section 5. Section 6 summarizes the work and lays out some future directions, where this work can be taken.

2 Related Work

Execution time and power consumption modeling is a topic of much research in literature. NVIDIA uses performance and energy consumption estimation to inform design decisions in the development of deep NN accelerator hardware [4].

For traditional algorithmic software, there are multiple approaches in literature. In the area of power measurement different levels of abstraction are used to represent the modeled process. These are in order from least to most abstracted gate-, register-transfer-, transaction- and function level modeling. [8]

The advantage of lower abstraction levels is higher accuracy in the estimations. Function level and higher abstracted models on the other hand need less in depth knowledge about the exact behaviour of the underlying hardware. This allows for easier portability of the model to different platforms. [8]

A useful aspect for time and power modeling is knowledge about the memory utilization and caching strategies utilized. This allows integration of memory latency into the estimation. In general purpose Central Processing Units (CPUs) there are different strategies to manage caching. Direct mapped caches allow writing of memory blocks to predefined locations in the cache. Increasing the associativity of the cache allows distribution of the cached content to different places. [5]

NN accelerators such as the herein used Neural Compute Stick 2 (NCS2) use application specific caching strategies. [6] describes some common memory caching strategies. Different approaches use different amounts of cache for the same NN layers, since they change the hierarchy of caching. This would lead to different amounts of cache accesses in each scheme.

Runtime optimization and complexity estimation of NNs is often done by comparing either the number of parameters or the number of floating point operations (FLOPs) for a given NNs. This does not accurately match the execution time of the NN as shown in [10]. A better approach is shown in [9]. Here the authors use an interpolation driven approach to capture the timing behaviour of various NN layers. It uses little hardware knowledge to estimate layer timing.

The contribution of this paper uses a similar approach to the one in [9]. It simplifies the estimator at the cost of a need for a higher amount of samples to create the model compared to [9]. This should allow for easier use in NAS approaches [1].

[1] shows several hardware aware NAS-systems. These utilize different metrics to determine hardware timing. The simplest method is the integration of hardware in the loop measurements. Others use models to estimate execution time ranging from lookup tables to meta-AIs, which learn the timing behaviour of the hardware.

MONNET, which is presented in this work, leads to better abstraction from hardware and framework artefacts. This in turn leads to overall higher accuracy and better transferability to other hardware accelerators and thus simpler application in dependant applications such as NAS.

3 Characterization and model building

The timing modeling approach can be separated into two general steps. At first the model needs to be created and characterized, which is discussed in this section. A model needs to be defined once for each neural network layer type to be supported. The characterization needs to be done once per target hardware.

3.1 Model creation

The general model used in this approach is shown in Equation 1.

$$t_l = t_{\mathrm{op}_l} \cdot n_{\mathrm{op}_l} \tag{1}$$

Here the execution time per layer t_l is modeled as the number of operations in one layer n_{op_l} (the layer complexity) multiplied by the time required per operation t_{op_l} (the efficiency). The number of mathematical operations in a layer, which is the same for each hardware, is separated from the actual hardware-specific execution time. This way only the t_{op_l} needs to be heuristically determined (i.e. measured on the target hardware), with n_{op_l} being mathematically derived from the layer parameters.

The efficiency t_{op_l} is not a constant, but depends on the actual layer configuration. t_{op_l}, as a function of the input parameters, is thus depending on and reflecting the influence of the hardware itself as well as configuration and artifacts of the neural network library and hardware deployment frameworks. Due to this, it has to be sampled over a large range of parameters for each layer type.

Applying this to one of the most time-consuming and most used layers in a convolutional neural network, the Convolutional 2D (Conv2D) layer, the complexity can be calculated as follows:

$$n_{\mathrm{op}_l} = k_x \cdot k_y \cdot c \cdot x' \cdot y' \cdot f \tag{2}$$

The number of operations for each filter is the number of outputs as $x' \cdot y'$ multiplied by the kernel size $k_x \cdot k_y$ and the number of input channels as c. Multiplying this by the number of filters f results in the number of operations for each layer.

x' and y' themselves are functions of the input size x and y, the stride and if no padding is applied the kernel size. They are calculated as shown in Equation 3, with y' being calculated similarly.

$$x' = \begin{cases} \lfloor \frac{x-k_x}{s_x} + 1 \rfloor, & \text{if padding} = 0 (valid) \\ \lfloor \frac{x-1}{s_x} + 1 \rfloor, & \text{if padding} = \lfloor \frac{k_x}{2} \rfloor (same) \end{cases} \tag{3}$$

To estimate the hardware and deployment framework dependent efficiency for a given layer, a dataset needs to be collected, containing samples at different complexities in different configurations. At the time of writing, the sample locations are determined using manual testing to detect the limits of the hardware and use case fitting, through evaluation of the test networks and determining the upper bounds of the network sizes.

For characterization and timing estimation, Equations 1-3 are used. For each layer the execution time t_l can thus be inferred using the base cost per operation t_{op_l} and the complexity n_{op_l}.

The modeling methodology generally relies on the fact, that the inference time of an entire neural network is the sum of the inference times of all its layers. This is an assumption which is in general valid and was already introduced by [4].

During initial measurements it turned out, that separating single NN layers for a characterization can nevertheless have a significant impact on the timing, measured in hardware. E.g. a Conv2D layer with a 3×3 kernel and 32 filters

Table 1: Result of per layer measurement in the MONNET-tool. Full HW measurement refers to the measurement of the per layer timing, if the layer is still executed in sequence with all other layers (here in the DenseNet121). Isolation mode 0 is the measurement of the same layer with identical parameters, but isolated and running standalone. This huge difference is typical and would render all layer wise modeling impossible. Thus better isolation techniques, mode 1 and mode 2 (see Figure 2) with better per layer isolation had to be developed.

Measurement type	Execution time in mS
Full HW measurement	23±0
Isolation mode 0	45.2 ± 1.46
Isolation mode 1	26.1 ± 0.696
Isolation mode 2	24.8 ± 0.533

working on $7 \times 7 \times 128$ input data has an execution time of 23.0 ± 0.0 ms (see Table 1), when measured within a DenseNet121. Cutting out this layer and synthesizing it standalone on hardware will increase the execution time to 45.2 ± 1.46 ms. A characterization has to be independent of a specific NN, so that the approach can be transferred. To eliminate this separation effect, the layers need to be embedded in a representative testing NN.

In order to gather enough data for a characterization of the estimation model, an automated synthesis flow is used. The flow starts with the host system, specifying a benchmark or characterization NN for the given layer configuration. Then it synthesizes the NN for the target hardware, after which it is executed and measured. Keras was used in order to generate a protobuf description and followed by the OpenVINO™ [2] toolchain to convert this into a hardware-agnostic, yet runtime optimized intermediate XML representation. From there, the NN could be compiled and flashed onto the hardware, using the OpenVINO inference engine. This toolchain is shown in Figure 1.

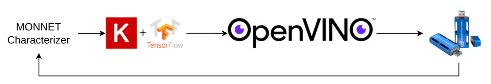


Fig. 1: MONNET Toolchain

In order to prevent the layer isolation issue presented in Figure 2, several methods were developed and tested to properly embed the layer under test. The embedding mode, which was determined to be the best, is to have a feeding layer and a consuming layer, both of type Conv2D with a standardized configuration and to measure only the timing of the middle layer under test, seen in Figure 2.

From Equations 2 and 3 the following parameters of a layer, which can be directly influenced can be extracted: x, y, c, k_x, k_y, f, s_x, s_y and the padding.

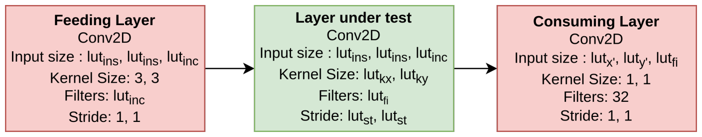

Feeding Layer
Conv2D
Input size : lut$_{ins}$, lut$_{ins}$, lut$_{inc}$
Kernel Size: 3, 3
Filters: lut$_{inc}$
Stride: 1, 1
Layer under test
Conv2D
Input size : lut$_{ins}$, lut$_{ins}$, lut$_{inc}$
Kernel Size: lut$_{kx}$, lut$_{ky}$
Filters: lut$_{fi}$
Stride: lut$_{st}$, lut$_{st}$
Consuming Layer
Conv2D
Input size : lut$_{x'}$, lut$_{y'}$, lut$_{fi}$
Kernel Size: 1, 1
Filters: 32
Stride: 1, 1

Fig. 2: When measuring a single neural network layer in isolation, the timing may significantly differ from a measurement of the same layer inside a larger neural network. In order to measure each layer independently, but in a typical environment, in characterization, the layer under test is embedded between a feeding layer (mode 1 and 2) and a consuming layer (mode 2)

To decrease the amount of measurements needed, the following assumptions are made for the characterization of the efficiency term: $x = y =: i$, and $s_x = s_y =: s$. In the model application, independent input sizes and strides can be described via the complexity term. This focuses on the most prevalent networks, which are benchmarked. In those the inputs of the layer are mostly square. Furthermore, the padding was so far set to same, meaning, that k_x and k_y have no influence on the output size as seen in Equation 3. This omission is automatically accounted for by the complexity term, which calculates the amount of mathematical operations based on the output size. As a result the characterization space for Conv2D layers can be described as a hypercube with six dimensions. The characterization space is the same for Separable Convolutional 2D (SepConv2D) layers.

To ensure reliable data for the characterization, each measurement is repeated until a likelihood of above 95 % of being within the 95 % confidence interval of the unknown real mean value is reached. Numpy's build in statistic tools are used to compute the probabilities after the fifth measurement first and then again after each further measurement until measurements converge.

The open source code from OpenVINO was adapted to allow for a repeated measurement and a stopping condition. As a result of this, the metric to determine the convergence of the execution time could be tightly integrated into the measurement process. Before a reflashing of the device was necessary for every measurement. This step can now be removed, speeding up the characterization process.

The multilinear interpolation, which is used for the estimation and, which will be explained in section 4 requires the sampling points to be on a regular grid in a hypercube. For hardware related reasons not every parameter combination can be synthesized or executed. If for example the layers are too large and have atypical parameter combinations such as highest values of input size, channels and filters at the same time, the hardware might not have enough memory to execute a layer. In other cases the synthetization requires too much memory on the host platform. This results in missing sample points.

To mitigate their impact those are automatically determined by interpolation after hardware characterization. For this a slightly different approach is taken than is used in the final interpolation for end use. If neighboring values were validly sampled, the value is in one axis interpolated. Otherwise, it is set to a default value, which is determined by the lowest value yet seen in the dataset.

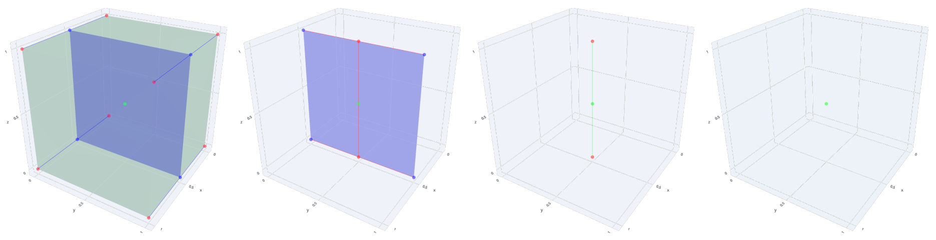
Even though this might seem arbitrary, it is well-chosen and leading to the best final estimation results. Missing data points typically exist for large and untypical parameter combinations such as input sizes, channels and filters all in the several thousands. For large regular tensors, the efficiency of the hardware tends to flatten out towards the maximum hypothetical efficiency as defined by the memory bandwidth and / or FLOPs rating.

An attempt was made to replicate the memory modeling from [9], which is integrated in their approach. Some studies were performed to determine the viability of automating this on a hardware-agnostic level. Attempts were made to map the memory models in [7] to our hardware. This led to no usable results as will be shown in section 5.

4 Timing prediction

Section 3 discussed the creation of the model. To predict an execution time, which is not within the characterization dataset, a multilinear interpolation is used.

In this approach the sample space is viewed as an n-dimensional hypercube. For the Conv2D and SepConv2D layers it has six dimensions as discussed in section 3. Two-dimensional activation layers can be reduced to a three-dimensional characterization space, with the dimensions being in_s, in_c and activation function.



(a) 3D starting point (b) 2D reduction (c) 1D reduction (d) Target value

Fig. 3: A multidimensional linear interpolation can be performed for any dimensionality of the to be evaluated layer model by recursively doing linear interpolations for pairs of neighbouring points along one of the dimensions, reducing the problem dimensions by one.

The interpolation is done stepwise as shown in Figure 3. At each step the dimensionality of the hypercube is reduced by one. This is done by interpolating the missing value along the axis linearly. As soon as it reaches a 0-dimensional state, the value left is the scalar corresponding to t_{op_l}. Obtaining the execution time of the layer requires calculation of the layer complexity and multiplication of this by t_{op_l}.

This approach assumes that t_{op_l} changes at a local level nearly linearly, meaning the characterization in section 3 needs to be granular enough to ensure this. Another approach could be to interpolate the dataset using a piece-wise defined polynomial, which passes through all sample points. Using this a slope in the data could be easily modeled. This was attempted but resulted in the following problem. An erratic behavior was observed if the sample points are too close together while having a high deviation. This can occur if the data is not continuous but as observed stepwise. For this see section 5. This resulted in a worse performance than the linear approximation, which is only influenced by two sample points and only affects values in between.

5 Evaluation

For the evaluation of the timing estimation common NN architectures are used. These can show the strengths and weaknesses of the current model. Specifically the networks from Table 2 were used.

Table 2: Benchmarking Networks

Network Name	Number of layers	Number of parameters
AlexNet	34	25730506
DenseNet121	429	8062504
DenseNet169	597	14307880
DenseNet201	709	20242984
InceptionResNetV2	782	55873736
InceptionV3	313	23851784
MobileNet	91	4253864
NASNetLarge	1041	88949818
NASNetMobile	771	5326716
ResNet101	347	44707176
ResNet152	517	60419944
ResNet50	177	25636712
VGG16	23	138357544
VGG19	26	143667240
Xception	134	22910480

The characterization and testing is performed on the NCS2, which is a NN hardware accelerator developed by Intel®. It uses the Intel Movidius™ Myriad™ X architecture, serving as a Vision Processing Unit. Due to the usage of the OpenVINO toolchain, multiple NN libraries can be used. In this work the decision was made to use the TensorFlow Keras libraries, which allows usage of the predefined models in the Keras applications.

Table 3: Search space for Conv2D-characterization

Input size	1, 2, 4, 7, 14, 28, 56, 112, 224
Input channels	2, 4, 8, 16, 32, 64, 128, 256, 512, 768, 1024, 2048, 4096
Kernel x	1, 3, 5, 7, 11
Kernel y	1, 3, 5, 7, 11
Filters	2, 4, 8, 16, 32, 64, 128, 256, 512, 768, 1024, 1536
Stride	1, 2

Table 4: Search space for SepConv2D-characterization

Input size	1, 2, 4, 7, 14, 21, 28, 42, 56, 112, 168
Input channels	2, 4, 8, 16, 32, 64, 128, 256, 512, 513, 768, 1024, 1280, 1536
Kernel x	3, 5
Kernel y	3, 5
Filters	2, 4, 8, 16, 32, 64, 128, 192, 256, 512, 513, 768, 1024, 1536, 2048
Stride	1, 2

The comparison focuses primarily on the measured execution time of the modeled layers compared to the estimated execution time. As of the writing of this work the only modeled layers are of type Conv2D and SepConv2D. Other layers such as Activation layers and Pooling layers are being worked on, but the isolation of the layer still needs work, since the measured timings of layers from real NNs strongly deviate from the extracted versions.

In most convolutional neural networks (CNNs) the Conv2D-layers require the highest amount of time to execute. Yet depending on the target application other layer-types need to be modeled as well, to estimate the timing accurately.

For evaluation purposes a characterization with the parameters in Table 3 and Table 4 was performed for Conv2D and SepConv2D layers respectively.

This results in 70,200 measurements taken for the Conv2D-layers, which is around twice as much as in [9] and 18,480 for the SepConv2D. This is seen as a reasonable tradeoff, since the measurements need to be taken only once. The search space is adapted to the target application, to increase the relevance of the measurements taken, but could be expanded upon in different target applications. Networks from the Tensorflow Keras Applications library were used for benchmarking the timing estimation approach (see Table 2).

Using this characterization the results for the benchmarking networks are shown in Figure 4. Blue shows the mean average percentage error (MAPE) for the benchmark networks. Red is the deviation of the estimated execution time from the real execution times of the layers under test. This results in a root mean squared percentage error (RMSPE) of 19.02 % for all Conv2D- and 26.38 % and for all SepConv2D-layers. [9] in comparison achieves 42.6 % RMSPE for all Conv2D-layers on a different set of evaluation networks on the same hardware.

Most of the network estimations stay within a 20 % error margin. The exceptions are Xception and DenseNet201. In the case of Xception the fault is within the estimation of the SepConv2D layers. This could be mitigated by a larger characterization space for SepConv2D. The DenseNet201 total deviation seems to occur, since it has many repeating layers. Some of these repeating layers are not well estimable by MONNET. This results in an accumulation of errors over the entire network. By including these within the measurement space, the error could be mitigated. This shows the need for automated selection of the sampling points.

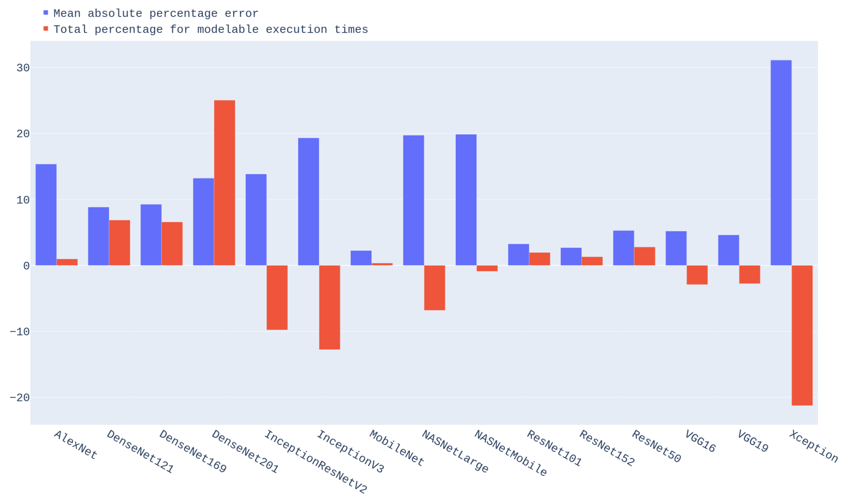


Fig. 4: Benchmark results showing the MAPE for all Conv2D and SepConv2D layers(blue). These stay mostly under 20 % with the notable exception of the Xception network which has numerous SepConv2D layers (34 SepConv2D to the 6 Conv2D layers). The red bars show the total deviation from the estimable execution time. Here DenseNet 201 stands out, as it has several repeating layers, on which the estimation performs poorly.

To evaluate the viability of independent memory modeling several measurement sweeps were performed. Figure 5 shows sweeps varying the input size in y direction, the kernel size in x and y and the amount of input channels.

All measurements show a general linear rise according to the increase in complexity. However, at certain parameter values, the execution time jumps (e.g., for 1024 channels on a (17, 25) input at a (1, 1) kernel [blue curve]). These jumps are not separable per dimension, but depend on all other parameters, too. With just one less row of the input size in y direction [green curve], the step size is significantly reduced from 201 % to 42 %.

Increasing the kernel size from (1, 1) to (3, 3), the curve shows a completely counterintuitive behavior between 670 and 800 channels, rising in timing by

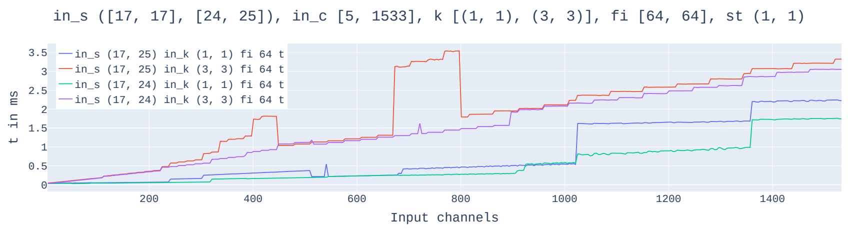


Fig. 5: Execution time as a function of number of input channels for various parameter combinations. A similar behavior can be observed, when measuring over the input size of the filter count.

almost 140 % and then falling back to the linear progression. This behavior is counteracted by decreasing the input size in y direction again to 24.

The results show, that while there are certain regularities, which could be modeled, large amounts of data would be needed to model the memory behavior more accurately. At the time of writing the leading theory is, that the system is layered in such a way, that the OpenVINO toolkit performs different optimizations on different levels, adding to the size dependent behavior of the memory. The latter was described in [7].

Due to this the added complexity of a memory modeling approach was left out of this work, allowing for a simpler model, which could be used for hardware aware NAS-approaches, while being less hardware dependent than a handcrafted memory model.

6 Conclusion

In this work MONNET, a timing estimation approach, was presented, which does not create the need for regarding programming artifacts of the synthesis flow and/or hardware artifacts of the embedded AI accelerator. Instead, the model averages over such artifacts, leading to an unavoidable average error for a concrete evaluation, but on the other hand leading to a much more steady (and differentiable) description of the general behaviour of the hardware, which can be used to control manual or automatical architecture searches. The deviation in the range of 20 % can be corrected by a single hardware measurement after the neural networks' topology was defined, and the network was trained. This work thus introduces a significant improvement over the designer's best guess or a MAC and parameter count based timing optimization.

Future work will entail usage of the modeling approach in a hardware aware NAS loop as shown in [1]. Furthermore, the approach needs to be validated on other hardware types such as Graphics Processing Units and Field Programmable Gate Arrays. In the future a system could be created, which automatically determines the upper limits of the hardware capabilities.

To circumvent problems regarding inter-layer optimizations which could be performed by the NN compiler, future work could focus on modeling these optimizations. This would work similarly to [9], modeling whether a layer is optimized out or by mapping the higher level operations to lower level hardware operations.

Acknowledgment This publication was created as part of the research project KI Delta Learning (project number: 19A19013K) funded by the Federal Ministry for Economic Affairs and Energy (BMWi) on the basis of a decision by the German Bundestag.

References

1. Benmeziane, H., Maghraoui, K.E., Ouarnoughi, H., Niar, S., Wistuba, M., Wang, N.: A comprehensive survey on hardware-aware neural architecture search (2021). https://doi.org/10.48550/ARXIV.2101.09336, https://arxiv.org/abs/2101.09336
2. Intel®: Openvino™, https://docs.openvino.ai/latest/get_started.html
3. Mori, Pierpaoloand Vemparala, M.R., Fasfous, N., Mitra, S., Sarkar, S., Frickenstein, A., Frickenstein, L., Helms, D., Nagaraja, N.S., Stechele, W., Passerone, C.: Accelerating and pruning cnns for semantic segmentation on fpga. In: Design Automation Conference (DAC) (2022)
4. Parashar, A., Raina, P., Shao, Y., Chen, Y.H., Ying, V., Mukkara, A., Venkatesan, R., Khailany, B., Keckler, S., Emer, J.: Timeloop: A systematic approach to dnn accelerator evaluation. pp. 304–315 (03 2019). https://doi.org/10.1109/ISPASS.2019.00042
5. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design: The Hardware/Software Interface. Fifth edition edn.
6. Siu, K., Stuart, D.M., Mahmoud, M., Moshovos, A.: Memory requirements for convolutional neural network hardware accelerators. In: 2018 IEEE International Symposium on Workload Characterization (IISWC). pp. 111–121 (2018). https://doi.org/10.1109/IISWC.2018.8573527
7. Siu, K., Stuart, D.M., Mahmoud, M., Moshovos, A.: Memory requirements for convolutional neural network hardware accelerators. In: 2018 IEEE International Symposium on Workload Characterization (IISWC). pp. 111–121 (2018). https://doi.org/10.1109/IISWC.2018.8573527
8. Sotiriou-Xanthopoulos, E., Percy Delicia, G.S., Figuli, P., Siozios, K., Economakos, G., Becker, J.: A power estimation technique for cycle-accurate higher-abstraction systemc-based cpu models. In: 2015 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS). pp. 70–77 (2015). https://doi.org/10.1109/SAMOS.2015.7363661
9. Wess, M., Ivanov, M., Unger, C., Nookala, A., Wendt, A., Jantsch, A.: Annette: Accurate neural network execution time estimation with stacked models. IEEE Access **PP**, 1–1 (12 2020). https://doi.org/10.1109/ACCESS.2020.3047259
10. Yao, S., Zhao, Y., Shao, H., Liu, S., Liu, D., Su, L., Abdelzaher, T.: FastDeepIoT: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In: Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems. ACM (nov 2018). https://doi.org/10.1145/3274783.3274840, https://doi.org/10.1145%2F3274783.3274840