This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3101936, IEEE Access

Digital Object Identifier

Blackthorn: Latency Estimation Framework for CNNs on Embedded Nvidia Platforms

MARTIN LECHNER[1,2], and AXEL JANTSCH[1,2] (Senior Member, IEEE)
[1]Institute of Computer Technology, TU Wien, 1040 Vienna, Austria
[2]Christian Doppler Laboratory for Embedded Machine Learning, Institute of Computer Technology, TU Wien, 1040 Vienna, Austria

Corresponding author: Martin Lechner (e-mail: martin.lechner@tuwien.ac.at).

This work was supported in part by the Austrian Federal Ministry for Digital and Economic Affairs, in part by the National Foundation for Research, Technology and Development, and in part by the Christian Doppler Research Association.

ABSTRACT With more powerful yet efficient embedded devices and accelerators being available for Deep Neural Networks (DNN), machine learning is becoming an integral part of edge computing. As the number of such devices increases, finding the best platform for a specific application has become more challenging. A common question for application developers is to find the most cost-effective combination of a DNN and a device while still meeting latency and accuracy requirements. In this work, we propose Blackthorn, a layer-wise latency estimation framework for embedded Nvidia GPUs based on analytical models. We provide accurate predictions for each layer, helping developers to find bottlenecks and optimize the architecture of a DNN to fit target platforms. Our framework can quickly evaluate and compare large amounts of network optimizations without needing to build time-consuming execution engines. Our experimental results on Jetson TX2 and Jetson Nano devices show a per-layer estimation error of 6.104% Root-Mean-Square-Percentage-Error (RMSPE) and 5.888% RMSPE, which significantly outperforms current state-of-the-art methods. At network level, the average latency error is below 3% for the tested DNNs.

INDEX TERMS Artificial neural networks, Estimation, Neural network hardware

I. INTRODUCTION

Deep Neural networks (DNNs) are widely adopted as key components in many use-cases like vision and speech processing solutions. Typical vision applications range from the automotive industry [1], [2] to medical use cases [3], [4] and consumer-focused applications like Google Lens. Until a few years ago, the computationally demanding DNNs were executed in the cloud requiring a stable network connection all the time. The increasing capabilities of embedded devices like the Graphic Processing Units (GPUs) in the Nvidia Jetson family and hardware accelerators like Intel's NCS2 or the Gyrfalcon Lightspeeur allow designers to run larger DNNs directly on mobile platforms. Such edge devices play a significant role in areas where manufacturers are interested in advanced driver assistance systems (ADAS) like the railway or construction industry since a stable network connection cannot be ensured [5].

Despite the fact of embedded devices are getting more powerful, optimization of DNNs is still a must to achieve feasible performance. In product development, computer vision and machine learning engineers are often interested in quick estimations like: Can their neural network run on a specific hardware platform with a given latency? What effect does a change of parameters due to optimization or a larger input image have on latency? In order to meet latency requirements on resource-limited embedded platforms, compression techniques like quantization [6], pruning [7], and shunt connections [8] are utilized. Pruning is a technique where the size of individual layers, i.e., the number of filters of a layer, is reduced. Shunt connections, on the other hand, replace larger sections of a neural network, e.g., multiple residual blocks as used in MobileNetV2 [9], with smaller, more efficient blocks. However, analyzing and comparing different setups, e.g., different optimization and compression techniques applied to multiple scales of several DNN architectures, is usually extremely time-consuming. It often results in retraining the network, and most platforms require a building or compiling step before execution to achieve optimal performance, increasing the time to test a single network. This compiling step typically requires up to a few hundred seconds, clearly

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3101936, IEEE Access

overtopping the inference time, which is in the range of tens of ms for state-of-the-art networks.

To skip the time consuming compiling step, DNN latency prediction techniques based on analytical or statistical models have been put forward. They target either large desktop-grade GPUs [10], [11] embedded Central Processing Units (CPUs) [12] but not more powerful embedded devices. Methods like [10] designed for desktop GPUs rely on the Nvidia System Management Interface (Nvidia SMI), which is not available on mobile GPUs. CPUs, on the other hand, feature a much smaller number of parallel computation units allowing linear models to accurately predict latency [12]. Estimation frameworks designed for Application Specific Integrated Circuits (ASIC) or Field Programmable Gate Arrays (FPGA) using white-box approaches rely on knowledge of the underlying hardware [13], [14]. This information is often not available for off-the-shelf embedded platforms.

Collecting the data for building estimation models is critical for achieving high accuracy. Since compiling a network architecture takes up to a few hundred seconds, a dense benchmark of the design space with more than 10^3 possible points in each of the multiple dimensions (e.g., the number of input channels and the width and the height of the input) is not feasible. A naive (dense) approach on a single device could then take hundreds of years. Improved implementations using a reduced set of pre-selected points can reduce the benchmarking time down to a couple of weeks. However, one has to carefully select these points to avoid overseeing patterns in the inference time. Some implementations like [10] use a dataset of layers obtained from state-of-the-art Convolutional Neural Networks (CNNs). While being fast, the generalization to unseen and uncommon layers is a big concern due to the small number of samples.

To overcome the limitations mentioned above and fill the gap of embedded GPU platforms, we propose **Blackthorn**, a layer-wise latency estimation framework for CNNs on embedded Nvidia GPUs. We eliminate the data collection issues by selecting the benchmarking points that provide the most information to minimize the required measurements. We use analytical layer models based on function templates to improve the generalization ability of our estimator while keeping the underlying dataset sparse. To the best of our knowledge, this is also the first work that maps a small set of functions, namely step, and linear functions, to the measured latency to generate layer-wise models. Blackthorn can be used to estimate the effects of network optimizations, to guide compression techniques to utilize a platform better, e.g., platform-aware pruning, and for fast network evaluation (Network Architecture Search or NAS).

In the context of this paper, the term "platform model" always refers to a combination of the hardware and a specific version of the provided framework, e.g., an Nvidia Jetson Nano with Jetpack 4.3.

Specifically, this paper makes the following main contributions:

- Blackthorn, a model-based framework to estimate the execution time of convolutional neural networks on embedded Nvidia platforms;
- An estimation method based on an analytical approach using a combination of linear and step functions;
- Fast platform benchmarking by finding optimized measurement points and thus minimizing their number.

The rest of the paper is organized as follows. Section III describes our overall architecture with more details presented in Section IV. The estimation results and comparisons to the state-of-the-art are shown in Section VI with a conclusion drawn in Section VII.

II. RELATED WORK

Recently, several studies have been published evaluating the performance of common DNNs on various hardware platforms helping developers select the best target platform for their needs. DawnBench [15], [16] is a benchmark suite for measuring training and inference time targeting mainly cloud and server-related hardware. In EMBench [17], multiple state-of-the-art DNNs are tested on a broader range of platforms, including desktop GPUs (Nvidia RTX 2080Ti), embedded GPUs (Nvidia Xavier), specialized DNN accelerators (Intel NCS2), as well as mobile and server CPUs. They also provide insights into how different layers, e.g., convolution, fully-connected, and pooling layers, perform on the selected platforms. It turns out that some platforms are more efficient for specific layer types than others. While convolution layers always take the most time, their share of the total runtime varies from 65% to 89%. MLPerf [18] is an industry-backed benchmark suite. It defines a set of rules and best practices enabling fair and comparable benchmarks across different hardware platforms and offers an extensive database containing inference times for different networks on multiple platforms. However, all these benchmark suites share the disadvantage that they cannot predict the inference time when one adjusts the size of the input image or some layers, e.g., by pruning the network.

Layer-level modeling of DNNs is one approach to fill this gap. A simple way to create layer models relies on the number of computations of a layer. Rouhani et al. [19] take the number of layers and the number of neurons per layer as input and perform micro-benchmarks to estimate the costs of certain operations (e.g., multiple-add and activation function). A more advanced framework is Paleo [11]. It models the execution time of a single layer using an analytical model based on the time to fetch the input, to perform the computation, and to write the outputs to the local memory. The computation model is based on the heuristics implemented in CuDNN, and thus, it cannot be used for estimating other platforms than Nvidia GPUs. NeuralPower [10] proposes an estimation method relying on a polynomial regression model. It decreases the estimation error on an Nvidia TitanX for a set of CNNs (AlexNet [20], VGG16 [21], Nin [22], Overfeat [23], and CIFAR10-conv6) from 23.12% to 7.96% and the Root-Mean-Square-Percentage-Error (RMSPE) for convolution layers from 58.29% to 39.97% compared to Pa-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3101936, IEEE Access

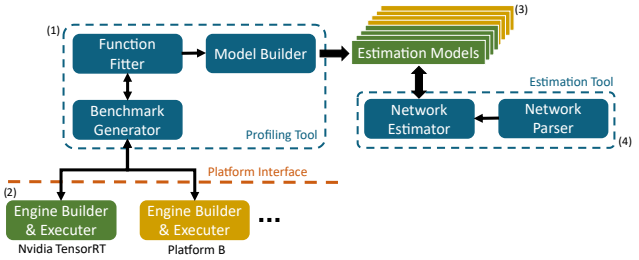


FIGURE 1. Overview of the system architecture: The profiling tool (1) builds estimation models based on measurement results obtained from a platform using the vendor specific inference frameworks (2). The resulting estimation models (3) are then utilized by the estimation tool (4) to predict a network's inference time.

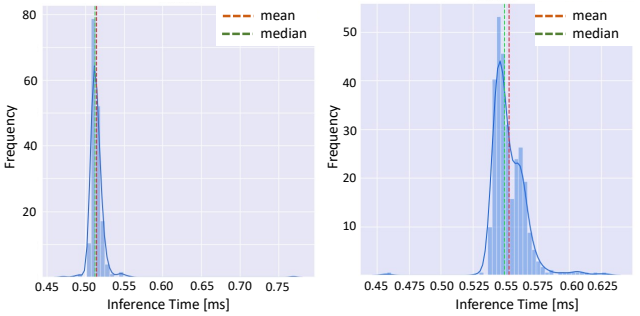


FIGURE 2. Two sample histograms showing the non-deterministic behavior when running the same layer multiple times.

leo. NeuralPower also includes models for power and energy estimation. However, it relies on the Nvidia System Management Interface (SMI), which is not available on Jetson platforms. PreVIous [12] presents a similar approach using linear regression models. It targets embedded CPU platforms like a Raspberry Pi3 and an Odroid-XU4 and reports an average error of 3.24% for the tested networks. MLPAT [14] is a white-box approach for power, area, and timing estimation of machine learning accelerators at chip architecture level. They report an error of less than 10% for a TPU-v1. A similar approach is DNN-Chip predictor [13], an analytical performance estimator targeting FPGA and ASIC machine learning accelerators. The maximum prediction error is 17.66% across several DNN models, hardware architectures, and dataflows.

Besides the latency and power estimation of neural networks, much effort has been put into predicting performance and power on lower levels of abstraction. On a GPU, every neural network is broken down into a sequence of multiple kernels, where every layer consists of one or more individual kernels. In [24] the authors present a model to predict execution time and power consumption of a set of GPU kernels extracted from multiple benchmarking suites. The developed models are based on a random forest approach using only architecture independent features. The achieved Mean-Absolute-Percentage-Error (MAPE) for prediction the execution time is in the range of 8.86% to 13.86% for server-grade GPUs. However, on consumer-grade GPUs, the MAPE increases to 52% as they do not support setting a fixed clock frequency. The effect of enabled dynamic voltage and frequency scaling (DVFS) on estimating the performance of GPU kernels is analyzed and modeled in [25]. The models are based on detailed investigations of the execution pipeline and memory accesses of different kernels. Over a set of 12 GPU kernels and 49 frequency settings, a MAPE of 3.8% is reported.

III. SYSTEM ARCHITECTURE

The overall system architecture is shown in Figure 1. The **Profiling Tool** builds a platform model based on micro-benchmarks of individual layers and layer combinations, e.g., stacked convolution and pooling layers. Determining the specific benchmarking points at run-time allows us to find the points with a significant expected improvement for function fitting, which efficiently decreases the number of required measurements. Consequently, the overall run-time of the profiling tool wanes as well. The benchmark generator is responsible for managing the measurement points and communicates with a selected backend.

The profiling tool requests latency measurements from a device using a simple platform interface communicating with a platform-specific backend. This backend always relies on the currently available optimizer and inference engine provided by the manufacturer. Our experiments target the Nvidia Jetson family of embedded GPUs. Thus, we rely on TensorRT in this work. However, backends for other platforms can easily be plugged in in future work.

The output of the profiling tool is a collection of **Estimation Models** for different layers, devices, and platforms. The **Estimation Tool** reads in a given network description and predicts the inference time based on the previously generated estimation models.

In Section IV, we describe our approach to profile a platform and explain how the model builder uses these results to create the estimation models.

IV. PROFILING TOOL

The profiling tool (Figure 1) consists of three main components, the function fitter, the benchmark generator, and the model builder. In the following section we explain the individual modules and our approach to profiling embedded Nvidia GPU platforms in more depth.

A. BENCHMARK GENERATOR

The benchmark generator manages the communication with the hardware platform to build and execute a layer and to collect latency measurements. This data is then used as an input for our function fitting module. We utilize the event management tools included in the CUDA runtime API [26] to measure the inference time of individual layers. Since the Nvidia Jetson devices are general-purpose devices, the inference time for a single layer is not as deterministic as using a specialized neural network accelerator built on top of an Application Specific Integrated Circuit (ASIC). Figure 2

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3101936, IEEE Access

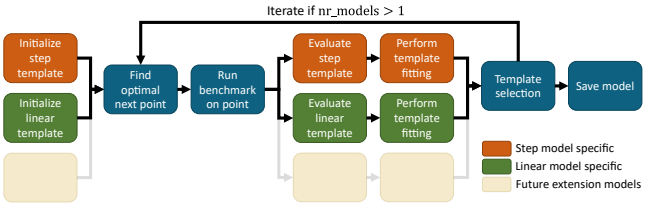


FIGURE 3. Process of function selection and fitting. Blue colored boxes show components which are common for all function templates.

shows the histograms of two different layers each covering 500 individual runs. We use the median value of a set of 500 measurements to compensate for uncertainty introduced here. As a result, our models estimate an average runtime for a given neural network. As an upper bound can be important in some applications, we record other statistical metrics like the 75^{th} as well as the 97.5^{th} percentile, enabling estimation models for the upper bound in future extensions.

B. FUNCTION FITTER

The function fitter is one core element of the profiling tool. Figure 3 illustrates the process of finding models representing a single dimension of a layer's design space. The dimensions of the design space are equal to the parameters describing a layer. In case of a convolutional layer, the dimensions of the design space are: the input width w_{in} and height h_{in}, the number of input channels d_{in}, the number of filters k, the size of the filter $kernel\_size$, and the stride s. The single-dimension models are built on top of parameterized function templates, each responsible for a single function type. In this work, we focus on linear and step functions, but other templates can easily be added.

The process of function fitting (Figure 3) can be outlined as follows: First, two measurement points initialize each function template, where each template can hold multiple sets of parameters. Then the fitter selects an optimal next benchmark point and requests the measurement over the platform interface by utilizing the benchmark generator. Next, the individual templates optimize their parameter sets to adjust to the new benchmark point. The parameter sets of all templates are then sorted and filtered based on their fitting error keeping only promising parameter sets. This process continues until one function template with a single parameter set remains. We sequentially run the function fitter across all dimensions and combine the results into a model for each layer. This process is described in detail in Section IV-C.

1) Function Templates

In our work, we use two function templates, one for linear functions and one for step functions as defined in Equations 1.

$$\begin{aligned} f_{lin}(x) &= mx + b \\ f_{step}(x) &= d + \left\lfloor \frac{x}{w_{step}} \right\rfloor h_{step} \end{aligned} \tag{1}$$

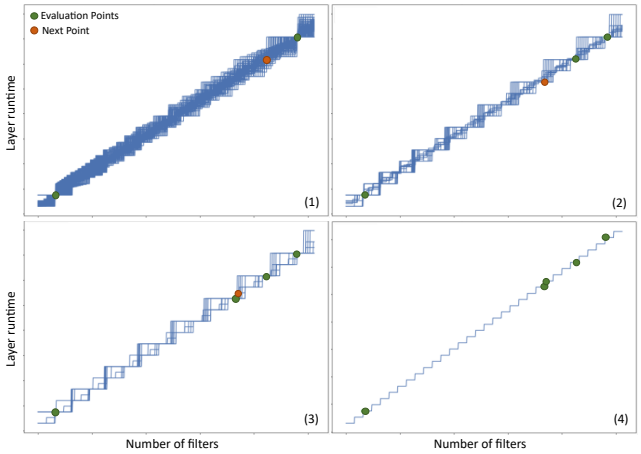


FIGURE 4. Step function selection process. (1): Initialized step fitter template. (2)-(4): Adding additional measurements reduce the size of the ensemble until only one function is left.

While fitting a linear function is simple, step functions require some attention. The step width w_{step} is connected to a single dimension of a CNN layer's design space, e.g., the number of filters or the number of input channels. Since these parameters are always whole numbers, w_{step} is constrained to unsigned integer values. To efficiently find the best-matching step function, we initialize the step-function template with an ensemble of possible step functions through two starting points (see Figure 4 (1)). Each function has a fixed step width ($w_{step} \in (8, 512)$ with w_{step} being a multiple of 4) while the remaining parameters are optimized during template fitting (see Section IV-B5).

2) Next Point Selection

One crucial part of minimizing the required number of measurements is finding an optimal next point. While executing networks on desktop or server platforms using TensorFlow or PyTorch is usually straightforward, platform-specific toolchains often require some optimization steps before execution. TensorRT performs some self-benchmarking jobs before optimizing the network and mapping the layers to the computation cores. This process can take up to three orders of magnitudes longer than the inference for a single layer. Therefore, minimizing the number of measurements is critical to complete the benchmark in a feasible time.

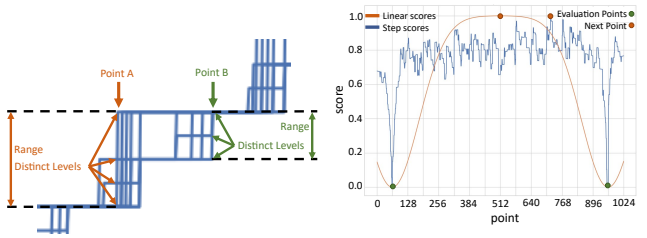
Our method of finding the optimal next point differs between the function templates. For step functions we use a combination of three different metrics:

- The number of distinct unique function values at each point U,
- the range R defined as between the maximum and the minimum value for each point,
- and the distance to already benchmarked points D

Figure 5a explains the first two criteria. Each set of scores is then normalized to the range $[0 - 1]$ before summed up:

$$scores_{step} = norm(\alpha U_{sc} + \beta R_{sc} + \gamma D_{sc}) * c_{step} \tag{2}$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3101936, IEEE Access



(a) (b)

FIGURE 5. Next point selection. (a): Definition of the range and distinct function values criteria. (b): Example of optimal points for a step and a linear function.

where α, β, and γ weigh the individual metrics, U_{sc} being short for $norm(U)$, and c_{step} is a difficulty coefficient. The final score is normalized again. A grid search optimizes the values of the parameters α, β, and γ. Therefore we ran our fitting algorithm on collected ground truth data and selected the values of α, β, and γ in order to minimize the average number of iterations required to terminate the fitting process.

For linear functions, we only use the normalized distance to previous points as decision criteria.

$$scores_{lin} = D_{lin} * c_{lin} \tag{3}$$

The difficulty coefficient ensures that more complex functions, i.e., functions with large ensembles of parameter sets, will be prioritized when selecting the next point. Because we have function templates of which only one contains a set of possible choices (step-function template), we set $c_{lin} = 1$ and $c_{step} = 2$. This way, we respect the fact that the step function template has to eliminate all but one parameter set. Finally, the next point is selected by finding the highest overall score. The selected next point is the index of $max(max(scores_{step}, scores_{lin}))$.

3) Template evaluation

Since a template can contain an ensemble of possible functions, we have to evaluate, rate, and filter them to find the best model describing the data. Therefore, we sort the ensemble of possible functions of each template, e.g., a set of d, w_{step}, and h_{step} for the step-function template based on the sum of squared errors. The error e_{seti} is associated with a single parameter set i, and e_n is the error at a specific measurement point (Equation (4)). The vector $\boldsymbol{e_{set}}$ contains all error values for each template so that $e_{seti} \in \boldsymbol{e_{set}}$ and $\boldsymbol{e_{set}} \in \mathbb{R}^k$ and $i \leq k$ where k is the number of parameter set of a template. The linear function template consists of only one parameter set so that $k = 1$.

$$e_{seti} = \sum_n (x_n - \hat{x_n})^2 = \sum_n e_n \tag{4}$$

To filter the parameter sets in an ensemble, we apply two criteria, a threshold T_e and a ratio R_e. A parameter set is removed from an ensemble if Equation (5) evaluates to *true*.

$$(e_{seti} > T_e) \wedge \left(\frac{e_{seti}}{min(\boldsymbol{e_{set}})} > R_e \right) \tag{5}$$

In our experiments, we use a fixed threshold T_e and a ratio R_e following a linear decay with increasing iterations to force the function selection to terminate. The value of T_e as well as the starting value and the decay per iteration of R_e were selected using a grid search hyper-parameter optimization to minimize the number of iterations.

4) Outlier detection

Outliers occur when the platform behaves unexpectedly (see the two bumps shown in Figure 7b for example). As such points can easily corrupt the function fitting process, especially when using only a few points, we detect and exclude them. We define an outlier as a point where the error dramatically differs from the error at all other points. The error for each point of a single function is e_n (see Section IV-B3) and the vector $\boldsymbol{e}$ collects the error values of all points so that $e_n \in \boldsymbol{e}$. The vector of median-normalized distances between the error at each point n and the median error then is

$$\boldsymbol{e_d} = \frac{|\boldsymbol{e} - median(\boldsymbol{e})|}{median(|\boldsymbol{e} - median(\boldsymbol{e}|)} \tag{6}$$

In case of an ensemble we can collect all error values in an error matrix $\boldsymbol{E}$ so that $e_{n,i}$ is the error at point n of set i. Consequently $\boldsymbol{E_d}$ contains the median-normalized error distances $e_{dn,i}$. We count a measured point n as an outlier when the error distance $e_{dn,i}$ is greater than a threshold $thres$ for more than half of all possible functions i in the current set:

$$outlier = \begin{cases} True \text{ if } \frac{\#(\boldsymbol{e_d} \geq thres)}{\#\boldsymbol{e_d}} > 0.5 \\ False \text{ otherwise} \end{cases} \tag{7}$$

The outlier detection is repeated after each iteration as outliers can turn back into valid points when the set of possible functions gets smaller. This can be the case if, during an early iteration, 75% of the possible functions vote for one point being an outlier. Later on, most of these 75% are removed due to not fitting the additional data well. Then the majority can shift, turning an outlier back into a valid point.

5) Template fitting

We use the Trust Region Reflective curve fitting algorithm (TRF) [27] to fit our templates to the measured data. The TRF algorithm is based on [28] and designed for sparse problems. In our case, the input to the fitting algorithm consists of three to twelve data points taken from a total of 1024 points. By default, the TRF algorithm optimizes all parameters to floating point values. Since the step with in the step function template is constrained to integer values (see Section IV-B1) we have to adjust this template function for curve fitting. Equation 8 forces the step width towards integer values.

$$f(x) = d + \left\lfloor \frac{x}{w_{step}} \right\rfloor h_{step} + (nint(w_{step}) - w_{step})^2 \tag{8}$$

We use $nint()$ to denote the rounding function to the nearest integer.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3101936, IEEE Access

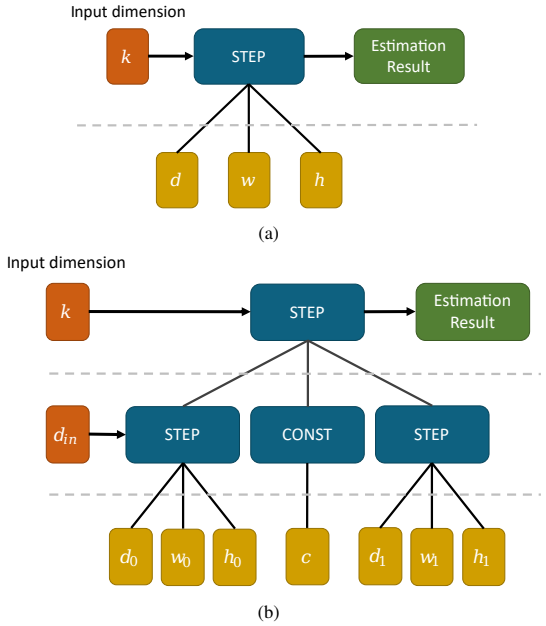


FIGURE 6. Example graph for a model of the number of filters k and the input dimensions d_{in} of a convolution layer. We start by running a fitter in a single dimension (a) to build up a graph modelling convolution layers.

6) Template selection

After fitting our template functions to the measured data, we end up with multiple possible templates to choose from. Similarly to our approach of sorting the ensemble of functions inside a template (Section IV-B3), we sort the templates themselves using the sum of squared errors. To select a template, i.e. decide whether to use a linear or a step function, we only use a ratio R_t and compare the best parameter sets of each template ensemble, i.e. where $e_{set_i} = min(\boldsymbol{e_{set}})$. The value of R_t is determined using a grind-search such that the number of iterations is minimized. For convenience, we define the error of a template j as $e_{tpl_j} = min(\boldsymbol{e^j_{set}})$ and $e_{tpl_j} \in \boldsymbol{e_{tpl}}$ so that $\boldsymbol{e_{tpl}}$ contains the minimal errors of each template. $\boldsymbol{e^j_{set}}$ is the error vector corresponding to template j. Analogous to Section IV-B3, we remove a template if Equation 9 evaluates to **true**. The function fitter terminates and saves the resulting model when only one template containing a single parameter set is left.

$$\left(\frac{e_{tpl\,j}}{min(\boldsymbol{e_{tpl}})} > R_t\right) \tag{9}$$

C. MODEL BUILDER

The design space for neural networks is huge. For a single convolution layer one can vary the input size (width, height, and channels), the number of filters, the size of the filters, and stride and padding settings. Fully-connected layers (input and output neurons) and pooling layers (polling size and stride) have fewer dimensions compared to convolution layers. To solve this multi-dimensional estimation problem, we apply the one-dimensional estimation model sequentially on each dimension. After finding a model for the first dimension, we estimate each model parameter to track how they change in the next dimension. Figure 6 illustrates the process of building the graph for a two-dimensional model. The graph in Figure 6b shows a model for the number of filters k and the input dimensions d_{in} of a convolution layer.

We start by running our function fitter across a single dimension, keeping everything else fixed. This way we obtain a step model for the number of filters k with the parameters d, w and h at one specific location (h_{in}, w_{in}, d_{in}, $kernel\_size$, $stride$) (Figure 6a). Next we run the fitter at different values of d_{in} which results in multiple, slightly different step function models. Running the fitter on the parameters d, w and h allows us to build a model describing how these parameters change depending on d_{in}. The outcome of this step is shown in Figure 6b. The model represented by this graph has two inputs, d_{in} and k, and seven parameters. By repeating this

Algorithm 1: High level description of the function fitting algorithm.

```
FITTER1D(c)
Input: Profiling coordinates c
Output: Fitted template described by params

initial_points ← (64, 960)
m ← runBenchmark(c, initial_points)
nr_templates, params ← updateTemplates(m)
while nr_templates > 1 do
    next_point ← findNextPoint(m)
    m ← m ∪ runBenchmark(c, next_point)
    runOutlierDetection(m)
    nr_templates, params ← updateTemplates(m)
end
return params

PROFILER(d, c)
Input: Dimensions of the layer parameter space d
Input: Initial profiling coordinates c
Output: A set of fitted templates func_params

while ∃ t in nr_templates: t > 1 do
    if d = 1 then
        func_params ← FITTER1D(c)
        break
    else
        params ← PROFILER(d − 1, c)
        foreach p in params do
            runOutlierDetection(p)
            nr_templates[p], func_params[p] ←
              updateTemplates(p)
        end
        c ← findNextPoint(params)
    end
end
return func_params
```

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3101936, IEEE Access

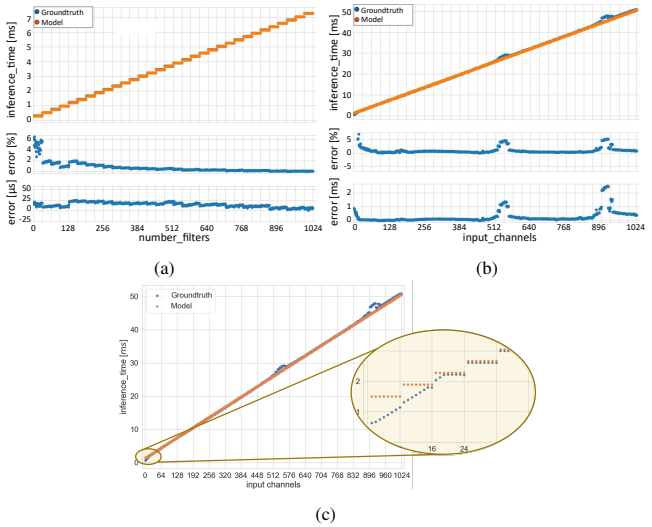


FIGURE 7. Step function fitting results for a single dimension of a convolution layer with a stride of 1 and using a 3×3 filter. (a) Convolution layer evaluated on k with $w_{in} = h_{in} = 64$ and $d_{in} = 128$. (b) Convolution layer evaluated on d_{in} with $w_{in} = h_{in} = 64$ and $k = 1024$. (c) Linear behavior of the Jetson Nano for a small number of input channels.

procedure for the other dimensions of the design space, we finally get a graph model for a convolution layer. The same process is used to model other types of layers like fully-connected and pooling layers, although the resulting models are simpler due to the smaller design space.

For convolution and pooling layers we do not end with a single graph for the entire design space. Some dimensions, like the kernel size, are usually small (typical kernel sizes are 1, 3, 5, 7, or 11 as compared to d_{in}, which can be anywhere in the range (3, 1024) or even larger). Thus we can not utilize the function fitter on such dimensions as there are too few points for fitting a function. As a result, the convolution and pooling layer models consist of multiple sub-graphs, while a fully-connected layer can be expressed in a single graph.

Algorithm 1 provides a high level view of the function fitting routine, where the function *fitter1D* corresponds to the process shown in Figure 3. The result is a set of fitted templates which are further processed in order to build our model graphs as shown in Figure 6.

V. ESTIMATION TOOL

The estimation tool reads a neural network description from an ONNX [29] file and extracts the individual layers. If possible, layer fusion, e.g. combine a convolution and a ReLU (Rectified Linear Unit) layer into a single layer. Layer fusion is done based on the list of supported layer fusions provided by Nvidia [30]. Using the previously generated layer level estimation models, the estimation tool predicts the inference time of each layer. The total run time of the provided neural network is then calculated by the sum of the individual layer execution times.

VI. RESULTS

For our experiments, we used two embedded Nvidia GPUs, the Jetson Nano (based on the Maxwell architecture) and the Jetson TX2 (based on the Pascal architecture). Both are running Jetpack 4.3, which includes TensorRT 6.0.1 and cuDNN 7.6.3. In case of an update of one of these libraries, we have to re-run our benchmarking tool to create a new model to capture possible performance improvements. We use TensorRT to build and execute the engines to get the best performance out of the devices. Additionally, we set the power mode of each device to the maximum and disabled frequency scaling. It has turned out that keeping frequency scaling active has a negative impact on performance and repeatability as TensorRT includes a kernel timing and selection step before building an engine. Depending on the architecture of the platform and the available kernels, the low-level representation of a CNN might change for different platforms. Modifying the frequency during the timing and selection phase can mess up the benchmarking results leading to a sub-optimal inference performance.

Typical use-cases of embedded platforms involve camera-based applications where objects have to be detected on the incoming video stream in real-time, e.g., pedestrian detection or infrastructure monitoring. To fit these use-cases and to achieve the lowest latency possible, we set the batch size in all experiments to 1. Classification and object detection on large datasets where bigger batches are beneficial are typically performed on desktop and server-grade GPUs which is outside the scope of this work. However, including the batch size as an additional dimension would allow an extension to larger batch sizes.

The following sections hierarchically present our estimation results. First, we evaluate our method of function fitting and template selection. Then we continue with the results at layer-level and finally test our framework on complete CNNs. In the last section, we discuss the benefits of estimating the latency using blackthorn compared to executing the neural networks on the hardware platforms.

A. FUNCTION FITTING

Our approach to creating layer and network models relies on detecting function types and fitting the function to the measurements for single dimensions. Thus, we need high reliability and a small error in order to achieve accurate models. We captured 64 full single-dimension sweeps on the Jetson Nano as ground-truth data to evaluate our function fitting module. Each sweep consists of 1024 measurements. The error across this dataset is 2.17% Mean-Absolute-Percentage-Error (MAPE) and 0.341ms Mean-Absolute-Error (MAE). Figure 7 shows two examples for fitted step functions on a single dimension of a convolution layer together with the absolute and relative error for each point. We typically need 8 to 14 measurements for such a dimension where a larger step width ($step_w = 32$ in Figure 7a) tends to require less measured points than a smaller step width ($step_w = 8$ in Figure 7b).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3101936, IEEE Access

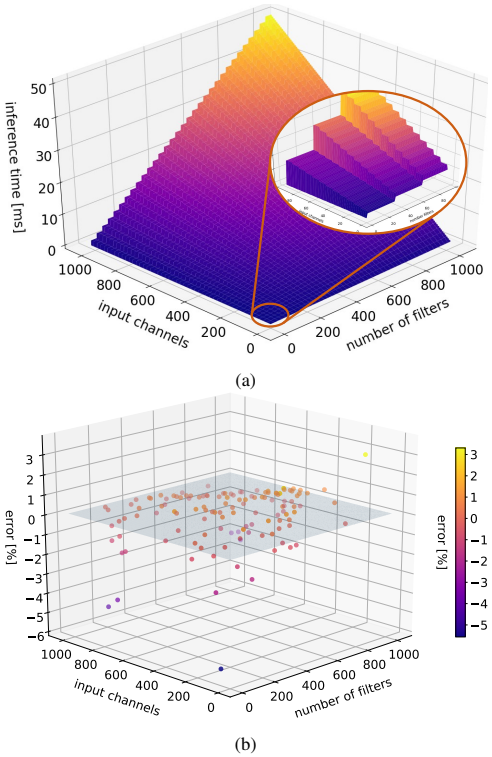


FIGURE 8. 2-dimensional estimation results for a convolution layer with an input size of 32×32. (a): 2D slice through a convolution-layer model. (b): Relative error for 768 measurement points.

The ground truth in Figure 7b (blue) shows two bumps. The step width does not change in both cases, but the measurements do not follow a clean step function. Our model does not capture these irregularities. However, the maximum error in such sections is still around 5%, keeping the impact on the overall accuracy small. Figure 7c shows a corner case where 1024 filters are applied to a varying number of input channels. Besides the two bumps, the ground truth also shows a linear behavior for a small number of input channels (number of input channels < 20). Again, our model does not capture such a characteristic. There are two main reasons for omitting the small-scale features. First, there is always a tradeoff between the estimation accuracy and the model complexity. Including such small-scale features would have little effect on the estimation accuracy at the cost of a dramatically increased model complexity. As the complexity of a model increases, so usually does the number of parameters describing it. Consequently, more adjustable parameters would require more measurement, which is in opposition to our goal of minimizing the number of required benchmarking points.

Figure 8 shows the fitting result for two dimensions corresponding to Figure 6b. The entire plane consists of 1024^2 individual points, of which 112 were used to build the shown model. Figure 8b shows the relative error at 768 randomly sampled locations. Despite a maximum error of around 5%, most errors are in the range of $\pm 1\%$.

TABLE 1. Estimation results for 1000 randomly picked convolution layers

	RMSE [ms]	MPE $\pm$ SD [%]	RMSPE [%]	MINPE [%]	MAXPE [%]
Nano	0.538	4.41 $\pm$ 1.347	7.566	0.051	86.7
TX2	0.265	4.631 $\pm$ 1.47	7.974	0.017	79.2

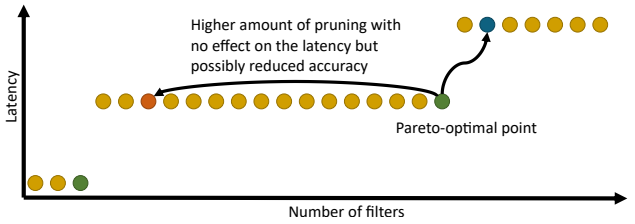


FIGURE 9. Pareto-optimal pruning target.

B. LAYER-LEVEL RESULTS

We evaluate our estimation model for single layers using the Mean-Absolute-Error (MAE), Root-Mean-Square-Error (RMSE) and the RMSPE. Table 1 shows the results for a random sample of 1,000 convolution layers and Table 2 contains the results for the same set of layers as used in [10]. These are all pooling, convolution, and fully-connected layers which can be found in AlexNet [20], VGG16 and VGG19 [21], Faster R-CNN [31], NiN [22], CaffeNet [32], GoogleNet [33], and Overfeat [23]. As our work focuses on embedded platforms, we used different target devices than the state-of-the-art methods mentioned in Table 2. However, the Titan X and the Jetson TX2 are both based on the Nvidia Pascal architecture. The main differences between them are the number of available computation cores and the available memory. Thus, the results should be comparable in terms of Root-Mean-Square-Percentage-Error (RMSPE) due to its relative nature.

The errors shown in Table 1 are slightly larger compared to Table 2. This difference can be explained as the random test set can also contain extensive layers, e.g., convolution with an input of $512 \times 512 \times 1024$ and 1024 3×3 kernels, as well as rectangular inputs, e.g., 64×512. Such layers are usually not used in practice, but including them shows the generalization ability of our model and allows us to predict the run-time of arbitrarily sized layers, e.g., convolution layers with an arbitrary number of input channels and filters. Consequently, we can predict the impact of layer level optimizations such as pruning where the size of a resulting layer might be very different from common choices, e.g., a layer with an uneven number of filters. Furthermore, we can determine Pareto-optimal pruning goals for a given Nvidia platform. Such points lie on the right end of a plateau of the step function described by the number of filters where slightly more pruning does not decrease the latency, but an additional filter increases the latency (Figure 9).

The large Maximum Percentage Error (MAXPE) in Table 1 occurs in case of a convolution layer with a small number

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3101936, IEEE Access

TABLE 2. Estimation results for the test set used in NeuralPower [10]

Work	Platform	RMSE [ms]	MAE [ms]	RMSPE
[11]	Titan X	4.304	-	58.29%
[10]	Titan X	1.019	-	39.97%
This	Jetson Nano	0.33	0.188	5.888%
This	Jetson TX2	0.126	0.087	6.104%

TABLE 3. Estimation results for complete neural networks compared to the state-of-the-art

Work	Platform	Estimation [ms]	Measured [ms]	Error [%]
AlexNet				
Paleo [11]	Titan X	33.16	39.02	15.01
NeuralPower [10]	Titan X	43.41	39.02	11.25
This	Jetson Nano	26.28	27.82	5.54
This	Jetson TX2	10.47	11.22	6.68
VGG16				
Paleo [11]	Titan X	345.83	368.42	6.13
NeuralPower [10]	Titan X	373.82	368.42	1.46
This	Jetson Nano	154.2	154.9	0.45
This	Jetson TX2	62.02	61.16	1.41
ResNet-50				
This	Jetson Nano	48.01	49.15	2.37
This	Jetson TX2	20.45	21.44	4.83
MobileNetV2				
This	Jetson Nano	13.26	13.74	3.61
This	Jetson TX2	6.4	6.7	4.18

of input channels. The inference time in these areas follows a linear function before switching to a step function (Figure 7c). However, the introduced absolute error is typically below $0.25ms$, and such a layer exists only once in a neural network (input layer processing the image data).

C. NETWORK-LEVEL RESULTS

Using the layer-wise estimation results above, we can estimate the latency of entire neural networks. In Table 3 we compare our results with the current state-of-the-art. Our approach outperforms the current state-of-the-art at least by a factor of 6. However, Table 2 shows that the estimation error on layer-level is larger than on network-level. This difference is a consequence of the errors compensating each other when summing up all estimation results.

Figure 10 visualizes a comparison between the measured and the estimated runtime for VGG16 broken down into all layers we have modeled. The remaining layers (single ReLu and Shuffle) have a total runtime of $0.06ms$, which is negligible. Compared to [10], our estimated runtimes match the measured values better. The larger error for the first convolution layer (conv1-1) can be explained by the linear behavior of the Jetson Nano for a small number of input channels (see Figure 7c).

Note that the total latency of VGG16 on a Jetson Nano

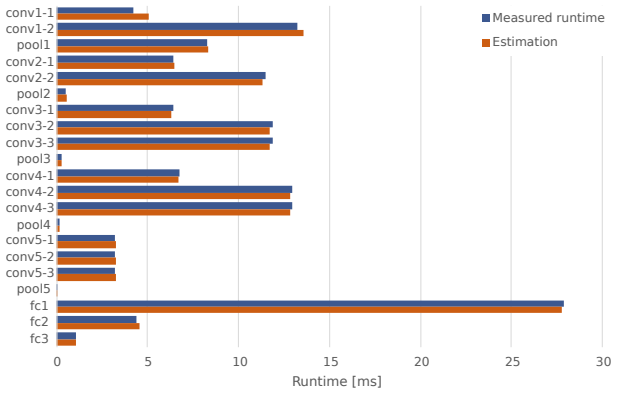


FIGURE 10. Layer-wise runtime prediction for VGG16 on a Nvidia Jetson Nano.

is more than two times lower than the values reported in NeuralPower, with individual layers being up to three times faster. This can be explained by the fact that the authors in [10] used Tensorflow to take the measurements, which is, depending on the platform and the network - or layer - size, three to eight times slower than the pure TensorRT implementation we use.

D. PERFORMANCE ANALYSIS

In this section, we discuss the performance of our framework regarding the profiling time (time required to benchmark a platform and build layer models) and the estimation time (time to predict the runtime of a DNN).

1) Platform profiling

Profiling a platform to capture the required data for building a model is usually an extremely time-consuming task, especially when the design space is huge. By minimizing the required measurements, we can effectively reduce the overall runtime. On a Jetson Nano, the Blackthorn profiling tool takes 6.5 days to complete benchmarking the platform and building the layer models. This time a dominated by the convolution layers due to their dimensionality. Blackthorn typically requires 8-12 measurements per dimension, leading to a total of about 15,000 measurements for convolution layers and another 5,000 for all other types. As discussed in Section **??**, our models are valid for the entire design space including uncommon layer dimensions. By limiting ourselves to only common layer dimensions, we could speed up this process by a factor of three. However, this would result in a loss of generalization, eliminating the possibility of predicting the effect of optimization techniques (pruning can often lead to layer dimensions not used in state-of-the-art DNNs).

2) Estimation

As outlined in the introduction, executing a neural network on an embedded platform often requires a compiling or building step. In case of the Nvidia Jetson devices and the

TABLE 4. Comparison of building and execution times of state-of-the-art neural networks on Jetson devices and the estimation times when using our Blackthorn estimator

Net	Jetson TX2			Jetson Nano			Blackthorn
	Build [s]	Inference [ms]	Total [s]	Build [s]	Inference [ms]	Total [s]	Estimation [ms]
AlexNet	21	11.22	27	47	27.82	59	< 1
VGG16	83	61.16	116	174	154.9	245	< 1
ResNet-50	122	21.44	133	181	49.15	206	1.2
MobileNetV2	109	6.7	113	144	13.74	152	1.2

TensorRT toolkit, an executable engine must be built once before starting an inference session. This step is required for every experiment, which involves changes in the network architecture, including the size of individual layers due to pruning or evaluating different sizes of the input image. Four our measurements, we set the TensorRT workspace size to $1GB$ and disabled the layer timing cache. Table 4 shows a comparison of the execution time of different neural networks on Jetson devices and using the Blackthorn estimator. The total time includes building an engine, running the same 500 inference cycles as our benchmarking tool to get the median latency, and an additional overhead introduced by the TensorRT toolkit. The results show that building an engine dominates the time required for running a single experiment.

To run the Blackthorn estimation tool, we use an off-the-shelf notebook with $8GB$ RAM and an i7-8650U processor. Estimating the latency of a given CNN depends on the number of executed layers with a single layer requiring about $20\mu s$. Comparing the estimation time to the total execution time of a neural network given in Table 4, our estimation tool is faster by a factor of at least $5 * 10^4$. This allows us to quickly evaluate many different network configurations without having to access the hardware platform.

VII. CONCLUSION AND FUTURE WORK

We presented Blackthorn, a latency estimation framework for neural networks on embedded Nvidia GPUs. We built analytical models based on linear and step functions to predict the inference time of a CNN layer by layer. On a set of eight CNNs, we achieved an average estimation error of 2.547% on a Jetson Nano and 2.843% on a Jetson TX2. On the layer-level, we reduced the RMSPE down to 5.888% and 6.104%, respectively, which is significantly lower than other state-of-the-art works. Our layer-level results show that our framework can accurately predict the impact of changes on the CNN on the latency due to optimization (like pruning or shunt connections) and different sized inputs. This also allows us to select optimal goals for pruning since the point right before a step uses the available cores most efficiently. Blackthorn allows a developer to estimate a CNN's latency in around $1ms$ without requiring access to the hardware platform. This is at least $5 * 10^4$ times faster than building and executing the CNN on embedded Nvidia GPUs directly.

As a next step, we plan to extend our framework to include more types of layers as well as other embedded platforms and integrate estimations for power and energy consummation using the same analytical approach. We are confident that Blackthorn can help developers quickly select the best platform and optimizations for their use-case, spending less time and money on subsequent training or retraining.

.

ACKNOWLEDGMENT

REFERENCES

[1] A. Luckow, M. Cook, N. Ashcraft, E. Weill, E. Djerekarov, and B. Vorster, "Deep learning in the automotive industry: Applications and tools," in 2016 IEEE International Conference on Big Data (Big Data), 2016, pp. 3759–3768.

[2] A. V. Devyatkin and D. M. Filatov, "Neural network traffic signs detection system development," in 2019 XXII International Conference on Soft Computing and Measurements (SCM)), 2019, pp. 125–128.

[3] G. Li, L. Bai, C. Zhu, E. Wu, and R. Ma, "A novel method of synthetic ct generation from mr images based on convolutional neural networks," in 2018 11th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), 2018, pp. 1–5.

[4] G. Jakimovski and D. Davcev, "Lung cancer medical image recognition using deep neural networks," in 2018 Thirteenth International Conference on Digital Information Management (ICDIM), 2018, pp. 1–5.

[5] nPerf, "3g / 4g / 5g coverage map," Sep. 2020. [Online]. Available: https://www.nperf.com/en/map/AT/-/-/signal/

[6] C. Gong, Y. Chen, Y. Lu, T. Li, C. Hao, and D. Chen, "Vecq: Minimal loss dnn model compression with vectorized weight quantization," IEEE Transactions on Computers, pp. 1–1, 2020.

[7] F. Tung and G. Mori, "Clip-q: Deep network compression learning by in-parallel pruning-quantization," in 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 7873–7882.

[8] B. Singh, D. Toshniwal, and S. K. Allur, "Shunt connection: An intelligent skipping of contiguous blocks for optimizing MobileNet-v2," Neural Networks, vol. 118, pp. 192–203, oct 2019.

[9] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2018, pp. 4510–4520.

[10] E. Cai, D. Juan, D. Stamoulis, and D. Marculescu, "Neuralpower: Predict and deploy energy-efficient convolutional neural networks," CoRR, vol. abs/1710.05420, 2017.

[11] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in Proceedings of the International Conferenceon Learning Representations (ICLR), 2017.

[12] D. Velasco-Montero, J. Fernández-Berni, R. Carmona-Galán, and Á. Rodríguez-Vázquez, "Previous: A methodology for prediction of visual inference performance on iot devices," arXiv e-prints, p. arXiv:1912.06442, Dec. 2019.

[13] Y. Zhao, C. Li, Y. Wang, P. Xu, Y. Zhang, and Y. Lin, "Dnn-chip predictor: An analytical performance predictor for dnn accelerators with various dataflows and hardware architectures," 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 1593–1597, 2020.

[14] T. Tang, S. Li, Y. Xie, and N. Jouppi, "MLPAT: A power, area, timing modeling framework for machine learning accelerators," DOSSA, 2018.

[15] C. Coleman, D. Kang, D. Narayanan, L. Nardi, T. Zhao, J. Zhang, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "Analysis of DAWNBench, a time-to-accuracy machine learning performance benchmark," ACM SIGOPS Operating Systems Review, vol. 53, no. 1, pp. 14–25, Jul. 2019.

[16] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia, "DAWNBench: An end-to-end deep learning benchmark and competition," NIPS ML Systems Workshop, 2017.

[17] M. Almeida, S. Laskaridis, I. Leontiadis, S. I. Venieris, and N. D. Lane, "Embench: Quantifying performance variations of deep neural networks across modern commodity devices," in The 3rd International Workshop on Deep Learning for Mobile Systems and Applications, ser. EMDL '19. New York, New York: ACM, 2019, pp. 1—6.

[18] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou et al., "Mlperf inference benchmark," arXiv preprint arXiv:1911.02549, 2019.

[19] B. D. Rouhani, A. Mirhoseini, and F. Koushanfar, "Going deeper than deep learning for massive data analytics under physical constraints," in 2016 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2016, pp. 1–3.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," Communications of the ACM, vol. 60, no. 6, pp. 84–90, May 2017.

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," CoRR, vol. abs/1409.1556, 2015.

[22] M. Lin, Q. Chen, and S. Yan, "Network in network," CoRR, vol. abs/1312.4400, 2014.

[23] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," CoRR, vol. abs/1312.6229, 2014.

[24] 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 Trans. Archit. Code Optim., vol. 18, no. 1, Dec. 2021.

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

[26] "CUDA runtime API," May 2021. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/index.html

[27] "ScyPy reference - least squares optimizer," May 2021. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.least_squares.html

[28] M. Branch, T. Coleman, and Y. li, "A subspace, interior, and conjugate gradient method for large-scale bound-constrained minimization problems," SIAM Journal on Scientific Computing, vol. 21, 12 1999.

[29] "Onnx: Open neural network exchange," Feb. 2021. [Online]. Available: https://github.com/onnx/onnx

[30] "Best practices for TensorRT," Feb. 2021. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/best-practices/index.html

[31] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 6, pp. 1137–1149, 2017.

[32] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," ser. MM '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 675–678.

[33] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, pp. 1–9.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3101936, IEEE Access

MARTIN LECHNER received the B.Sc. and M.Sc. degrees from the Department of Electrical Engineering, TU Wien, Vienna, Austria, in 201 and 2018, respectively. He is currently pursuing the Ph.D. degree with the Institute for Computer Technology and is part of the Christian Doppler Laboratory for Embedded Machine Learning at TU Wien, Austria. His current research interests include hardware-accelerated machine learning and applications on embedded systems.

AXEL JANTSCH (M'92) received a Dipl.Ing. and a Ph.D. in Computer Science from TU Wien in 1987 and 1992, respectively. Between 1997 and 2002, he was associate professor at KTH, the Royal Institute of Technology, in Stockholm, and from 2002 to 2014, he was full professor in Electronic Systems Design at KTH. Since 2014 he is professor of Systems on Chips at the Institute of Computer Technology at TU Wien, Vienna, Austria.

His current research interests are Systems on Chips, Self-Aware Cyber-physical systems, and embedded machine learning. He has published 5 books as editor and 1 as author, over 300 peer-reviewed contributions in journals, books, and conference proceedings, and he has given over 100 invited presentations at conferences, universities, and companies.