nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices

Li Lyna Zhang[1] Shihao Han[1,2] Jianyu Wei[1,3] Ningxin Zheng[1] Ting Cao[1] Yuqing Yang[1] Yunxin Liu[4]
[1]Microsoft Research [2]Rose-Hulman Institute of Technology [3]University of Science and Technology of China
[4]Institute for AI Industry Research (AIR), Tsinghua University
{lzhani,ningxin.zheng,ting.cao,yuqing.yang}@microsoft.com,
hans3@rose-hulman.edu, noob@mail.ustc.edu.cn, liuyunxin@air.tsinghua.edu.cn

ABSTRACT

With the recent trend of on-device deep learning, inference latency has become a crucial metric in running Deep Neural Network (DNN) models on various mobile and edge devices. To this end, latency prediction of DNN model inference is highly desirable for many tasks where measuring the latency on real devices is infeasible or too costly, such as searching for efficient DNN models with latency constraints from a huge model-design space. Yet it is very challenging and existing approaches fail to achieve a high accuracy of prediction, due to the varying model-inference latency caused by the runtime optimizations on diverse edge devices.

In this paper, we propose and develop nn-Meter, a novel and efficient system to accurately predict the inference latency of DNN models on diverse edge devices. The key idea of nn-Meter is dividing a whole model inference into *kernels*, i.e., the execution units on a device, and conducting kernel-level prediction. nn-Meter builds atop two key techniques: (i) *kernel detection* to automatically detect the execution unit of model inference via a set of well-designed test cases; and (ii) *adaptive sampling* to efficiently sample the most beneficial configurations from a large space to build accurate kernel-level latency predictors. Implemented on three popular platforms of edge hardware (mobile CPU, mobile GPU, and Intel VPU) and evaluated using a large dataset of 26,000 models, nn-Meter significantly outperforms the prior state-of-the-art.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; **Embedded systems**.

KEYWORDS

deep neural network, inference latency prediction, edge AI

ACM Reference Format:
Li Lyna Zhang[1] Shihao Han[1,2] Jianyu Wei[1,3] Ningxin Zheng[1] Ting Cao[1] Yuqing Yang[1] Yunxin Liu[4] . 2021. nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices. In *The 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '21), June 24-July 2, 2021, Virtual, WI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458864.3467882

1 INTRODUCTION

Deep Neural Networks (DNNs) have been widely used in today's mobile and edge applications [33]. In many applications such as on-device video analytics, face recognition, AR/VR etc., DNN models are constrained by efficiency constraints (e.g., latency). To design a model with both high accuracy and efficiency, model compression [6, 14, 15, 24] and the recent Neural Architecture Search (NAS) [7, 29, 32, 34] consider the inference latency of DNN models as the hard design constraint.

However, measuring the inference latency for DNN models is laborious and expensive. In practice, it requires developers to perform a deployment process on the physical device to obtain the latency. The engineering effort is tremendous for diverse edge devices (e.g., mobile CPU/GPU and various AI accelerators) and different inference frameworks (e.g., TFLite and OpenVINO). Even on a single device, it may be extremely time-consuming to measure a large number of models in NAS tasks (e.g., ProxylessNas [7] explores ~0.3 millions of models in just one round of search). Such a high cost can hinder the scalability and make the measurement-based method practically infeasible to support the fast-growing number of edge devices.

Consequently, approaches have been proposed to predict the inference latency. For example, the FLOPs [1] based method has been widely applied to evaluate the efficiency [15, 22, 23, 30], which is simple but not a direct metric of latency. To predict a model latency, many NAS works [6, 7, 32] build the operator-wise lookup table. Such operator-level methods sum up the latencies of all operators. However, they do not consider the model latency differences caused by runtime optimizations of model graphs. For instance, many frameworks merge multiple operators into one fused operator to accelerate the inference, which impacts the inference latency significantly. Recently, the state-of-the-art BRP-NAS [13] uses graph convolutional networks (GCN) to predict latency of the NASBench201 [12] dataset on various devices. It captures the runtime optimizations by learning the representation of model graphs and corresponding latency. However, this model-graph based approach heavily depends on the tested model structures and may not work for many unseen model structures.

In this work, we propose and develop a novel system called nn-Meter [2] that aims to accurately predict the latency of *arbitrary* DNN models on *diverse* edge devices. The key idea of nn-Meter

[1]The definition of FLOPs follows [35], i.e., the number of multiply-adds.
[2]nn means neural networks.

is dividing a whole model inference into multiple *kernels* that are independent execution units of the model inference on a device. A kernel may be either a single primitive operator or a fusion of multiple operators, depending on the runtime and hardware. nn-Meter builds latency predictors for kernels and predicts the total latency of a model by the latency sum of all kernels of the model.

This design choice of kernel-level prediction are based on two observations. First, kernel is the basic scheduling and execution unit (e.g., GPU kernels) in deep-learning frameworks, particularly on edge devices. Thus, the notion of kernel naturally captures the diverse runtime optimizations including operator fusion, the most important optimization that can largely impact the latency. Second, despite a very large number of DNN models, the kinds of operators and kernels are stable with a relative small set. Any models are just different combinations of operators/kernels. Therefore, kernel-level prediction is generic enough to support unseen new models.

nn-Meter faces two key challenges. The first challenge is how to split a model into a proper set of kernels on various edge devices. Due to the diverse runtime optimizations, the executed kernels are varying on different devices. For example, the Conv-add is a fused operator on mobile GPU, but not on mobile CPU and Intel VPU. Furthermore, many inference frameworks are not open-sourced. Even for the open-sourced ones, it requires hardware expertise to determine the kernels. Second, it is non-trivial to build accurate predictors for the kernels. As we show in Section 5.1, the kernels show non-linearity between latency and prediction features. Moreover, the multiple configurable dimensions of kernels lead to a huge possible searching space for the latency prediction, as huge as billions. Sampling the whole billion-scale configuration space to get labeled training data is infeasible. Thus, how to do efficient sampling while ensuing high prediction accuracy remains a big challenge.

To address the above challenges, we propose two techniques, *automatic kernel detection* and *adaptive data sampling*. To split a model into kernels, nn-Meter employs a kernel detector that automatically detects the possible kernels on various edge devices in a black-box matter. We design a set of test cases to detect whether two operators can be fused or not. A DFS-based rule matching algorithm is designed to search for the maximum fusion unit (i.e., kernel) in a model. To reduce the data sampling cost, nn-Meter uses an adaptive data sampling algorithm that leverages both the model design and hardware latency characteristics. It firstly prunes the kernel configurations that are rarely considered in DNN models. Then, an iterative sampling process is executed to automatically detect the most beneficial configurations to sample, instead of random selection. Finally, we build machine-learning regressors to learn the non-linearity with the sampled data.

To demonstrate the effectiveness of nn-Meter, we further propose and create a large benchmark dataset that contains 26,000 representative Convolutional Neural Network (CNN) models [3]. Unlike previous works that use datasets with a smaller prediction scope, i.e., the NASBench201 dataset, models in our dataset are with various operators, configurations, graphs, and latency ranges. We believe that our dataset sets a new bar for latency prediction of DNN models and it may also be used for other tasks such as NAS and channel search.

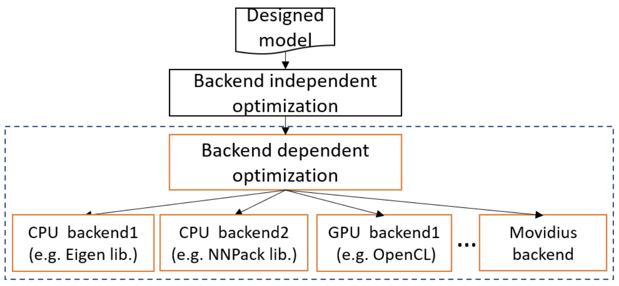

Designed model
Backend independent optimization
Backend dependent optimization
CPU backend1 (e.g. Eigen lib.)
CPU backend2 (e.g. NNPack lib.)
GPU backend1 (e.g. OpenCL)
...
Movidius backend

Figure 1: Graph optimizations of framework.

We implement and evaluate nn-Meter on three popular platforms of edge devices: mobile CPU, mobile GPU, and Intel VPU (a representative AI accelerator for edge devices). Significantly, nn-Meter achieves a prediction accuracy [4] of 99.0%, 99.1%, 83.4% on the CPU, GPU, and VPU, respectively, significantly outperforming the state-of-the-art approaches. We also conduct comprehensive experiments to evaluate the effectiveness of kernel-level prediction and adaptive data sampling, and report nn-Meter system overhead.

While we have obtained promising results of nn-Meter on the three platforms, it requires joint efforts across the community to apply nn-Meter onto many other types of edge devices. To this end, we will open-source [5] our dataset, test cases, and code for other researchers and developers to build latency predictors for their own devices. Collectively, we expect that the community can work together to realize accurate latency prediction of DNN models for a variety of edge devices.

We summarize our key contributions as follows:

- We propose and design nn-Meter, a novel and efficient system to accurately predict inference latency of CNN models on diverse edge devices.
- We design two key techniques for nn-Meter, including automatic kernel detection to capture the diverse operator-fusion behaviors on edge devices, and adaptive data sampling to reduce the cost of building kernel-level latency predicators.
- We create a new and large latency-prediction benchmark dataset that contains 26,000 representative CNN models with various model graphs, configurations, and latency ranges.
- We implement and evaluate nn-Meter on mobile CPU, mobile GPU and Intel VPU, and demonstrate that nn-Meter achieves a significantly better prediction accuracy across different devices and models, compared with existing approaches.

2 BACKGROUND AND MOTIVATION

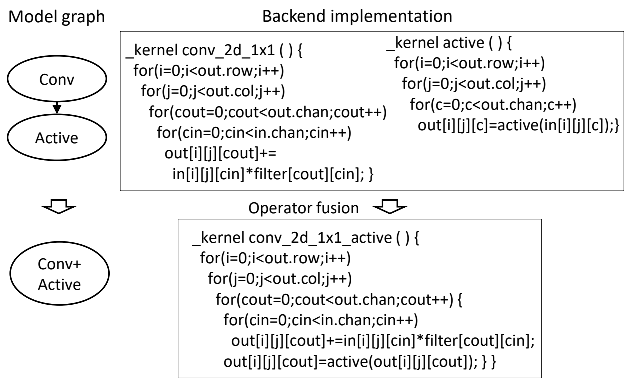
2.1 CNN Model Characteristics

Fast evolving CNN models. Since the milestone work of AlexNet, it takes years of DNN experts' efforts to invent a number of CNN models, including VGG, GoogleNet, ResNet, DenseNet, SqueezeNet, MobileNetv1, etc. Recently, NAS has demonstrated much success in

[3]Due to the poor support of other types models (e.g., NLP ones) on edge devices, we currently focus on CNNs. The techniques of nn-Meter are generic to other models.

[4]Defined as within a prediction error of ±10% [13].

[5]https://github.com/microsoft/nn-Meter



```
_kernel conv_2d_1x1 ( ) {
 for(i=0;i<out.row;i++)
  for(j=0;j<out.col;j++)
   for(cout=0;cout<out.chan;cout++)
    for(cin=0;cin<in.chan;cin++)
     out[i][j][cout]+=
      in[i][j][cin]*filter[cout][cin]; }
```

```
_kernel active ( ) {
 for(i=0;i<out.row;i++)
  for(j=0;j<out.col;j++)
   for(c=0;c<out.chan;c++)
    out[i][j][c]=active(in[i][j][c]);}
```

```
_kernel conv_2d_1x1_active ( ) {
 for(i=0;i<out.row;i++)
  for(j=0;j<out.col;j++)
   for(cout=0;cout<out.chan;cout++) {
    for(cin=0;cin<in.chan;cin++)
     out[i][j][cout]+=in[i][j][cin]*filter[cout][cin];
    out[i][j][cout]=active(out[i][j][cout]); } }
```

Figure 2: Kernel implementation for operator fusion("+" is used to represent fusion in this paper).

	Model	Operator sum		Kernel sum	
	Latency	Latency	Error	Latency	Error
CPU	45.57ms	51.23ms	12.42%	45.41ms	0.35%
GPU	10.18ms	12.31ms	20.92%	9.91ms	2.65%
VPU	22.64ms	33.86ms	49.56%	23.18ms	2.38%

Table 1: MobileNetv2 latency.

automating DNN design for various deep learning tasks [7, 23, 30]. As a result, it's much faster to design a novel CNN model with high accuracy by NAS. For instance, the OFA [6] has produced 50 high-accuracy models with different latency constraints.

Stable primitive operators. Despite the vast amount of DNN models, the operator types are with a small set. Widely applied operators include convolution2d (Conv), depthwise convolution2d (DWConv), activations (e.g., relu, relu6 and hswish), fully connected layer (FC), BatchNorm layer (bn), and element-wise operators (e.g., add). Each operator has a set of parameters to configure in the model, which can result in significant inference latency changes.

Too large space for model-level prediction. We address the difficulties in the model-level prediction. Considering a model with N nodes, where each node has D features (i.e., operator type and configuration), there are maximum $N \times (N-1)$ edges in a model DAG. For each node, the number of operator types is small. But one operator can have an unbounded range of configurations. To this end, the full prediction space is huge with possible $D^N \times N \times (N-1)$ models. Since model-level prediction learns from model graphs, the training data should cover the full set of operator types, edge connections, and various operator configurations. While it's challenging to construct and label such training set from the very large space, model-level prediction usually has poor performance on unseen model graphs.

Unaware of runtime optimization of operator-level prediction. As a result, prediction with fine-grained levels such as the existing operator-level approaches is more promising. However, operator-level prediction can not capture the graph optimizations on the edge devices, and hence is inaccurate. As shown in Table 1, the prediction produces large errors on three devices. In this paper, we propose the kernel-level prediction that is a fine-grained level and optimization-aware approach.

2.2 Optimizations of Inference Frameworks

Inference frameworks usually conduct a series of model graph transformations to optimize the model and reduce inference latency. Fig. 1 shows the framework optimization process. Most of the optimizations are backend independent, such as constant folding (e.g., $add(c1, add(x, c2))$->$add(x, c1 + c2)$) and common expression elimination. These optimizations should be done no matter what the target backend is.

There are also optimizations dependent on the target backend implementation. The major one is **operator fusion** illustrated in Fig. 2. It can fuse operators that satisfy certain rules (e.g., specific operator type and connection) together. This optimization can avoid storing intermediate results into memory to reduce memory access cost. For example, in Fig. 2, rather than calculating all the elements in the convolution kernel and then executing activation, the fused kernel executes activation after each element is calculated. Operator fusion requires the backend to implement the according kernel of the fused operators. Therefore, the fusion rules are various on different backends.

To this end, nn-Meter preprocesses the designed CNN model by backend-independent optimizations using an open-source framework (e.g., Tensorflow). The output model is used as the input of the prediction process. The operator-fusion rules of different backends will be detected by nn-Meter's test cases.

2.3 Rationale for Kernel-level Prediction

To predict a CNN model latency, nn-Meter splits model graph into kernels, and sum up predicted kernel latencies as the model latency.

This method is based on the assumption that kernels run sequentially on each device, even for those ones without dataflow dependency. This assumption is valid on current *edge* AI platforms (rather than server platforms) mainly for two reasons. Firstly, computation resources of edge chips are normally limited. It is not that beneficial to run multiple kernels parallelly. For example, server CPUs can have dozens of cores. Running one kernel at a time cannot utilize all the cores. However, edge CPUs only have several cores, which are unlikely to have spare ones to run multiple kernels at a time. Secondly, even although some AI chips have high computation bandwidth, due to the power and chip area restriction, there is no complex hardware support (e.g., multi-stream of CUDA GPUs) to run multiple kernels parallelly. Very likely, this assumption can be valid for future edge AI devices too.

As far as we have verified, kernels all run sequentially on current edge AI platforms, such as TFLite, SNPE, MNN, NCNN, and MACE. There are also many works before use latency sum of operators as the model latency also based on this assumption [5, 7, 11, 19, 32].

3 NN-METER DESIGN

In this section, we describe the overall architecture of nn-Meter and the benchmark dataset collection.

Overview. Fig. 3 illustrates the system architecture. It shows two core components to realize accurate latency prediction for a DNN model: Kernel Detection and Adaptive Data Sampling. Conceptually,

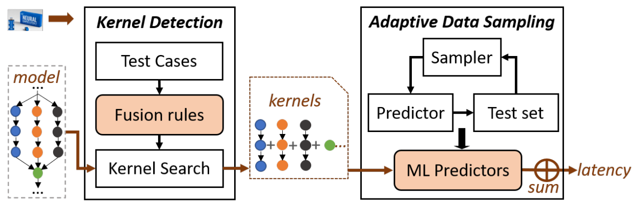


Figure 3: System architecture of nn-Meter. It offline detects fusion rules and builds ML predictors of kernels.

the former automatically divides the target model into a set of kernels, and the latter samples the most beneficial configurations from a large space to build accurate kernel-level latency predictors. Then, for each kernel, we extract the features and predict the latency. nn-Meter sums up predicted kernel latencies as the model latency.

Kernel Detection. It consists of well-designed test cases to detect fusion rules between two operators and an algorithm to search all the kernels in a model. We offline collect all the fusion rules. For online model prediction, the kernel search algorithm recursively applies these rules to the target model to find all the kernels. We present the technical details in Section 4.

Adaptive Data Sampling. It offline builds machine learning predictors for all kernels on the target device. For each kernel, it samples the most beneficial configurations through an iterative sample process. The sampler samples from a prior possibility distribution, that describes the considered kernel configurations in model CNN design. We design a test set for evaluating the quality of sampled data. At each iteration, a new machine learning predictor evaluates the performance with the test set. For data with large prediction errors, we perform fine-grained channel number sampling around them. We describe the details in Section 5.

Benchmark Dataset Collection. To evaluate the effectiveness of nn-Meter on an arbitrary DNN model, we need a representative dataset that covers a large prediction scope. We consider two scenarios. First, the DNN model can be consisting of any type of primitive operators and the various edge connections among them. Second, each type of operator in a model graph can have many possible configurations. For instance, the channel numbers of the Conv can be configured with any positive integer. Existing works [5, 13, 28] evaluate on either a small dataset or the NASBench201. The prediction scope is small, and thus they can not serve our purpose. For example, the operators in NASBench201 models are Conv, Pooling, add, and FC, which are few (14 types in our dataset). Moreover, the operator configurations (e.g., channel number and stride) are configured with fixed numbers.

In this work, we consider the large prediction scope and generate a large dataset that is applicable to channel search and NAS scenarios. First, we collect 12 state-of-the-art CNN models on the ImageNet2012. They are from both manual-designed and NAS-searched models with totally different operator types and configurations. For each model (e.g., AlexNet), we generate 2,000 variants (e.g., AlexNets) by re-sampling the output channel number and kernel size for each layer. Specifically, the new output channel number is randomly selected from $[0.2 \times C_{out}, 1.8 \times C_{out}]$, and the kernel size is sampled from {1, 3, 5, 7, 9}. Besides, we add 2,000 models

Model variants	avg FLOPs (M)	Latency(ms) Mobile CPU min - max	Mobile GPU min - max	Intel VPU min - max
AlexNets	973	7.1 - 494.4	0.4 - 81.7	2.1 - 47.3
VGGs	28422	178.4 - 10289	20.1 - 1278	25.6 - 1467
DenseNets	1794	109.6 - 431.6	26.7 - 69.5	26.4 - 70.7
ResNets	4151	35.9 - 1921.7	7.3 - 329.5	10.7 - 145.5
SqueezNets	1597	42.7 - 524.9	7.5 - 72.2	6.9 - 57.3
GoogleNets	1475	115.5 - 274.6	23.0 - 49.0	12.2 - 24.4
MobileNetv1s	547	27.5 - 140.0	5.5 - 28.8	8.9 - 37.0
MobileNetv2s	392	15.6 - 211.0	3.5 - 37.0	11.3 - 86.1
MobileNetv3s	176	10.4 - 78.4	4.3 - 18.6	17.4 - 70.8
ShuffleNetv2s	307	22.2 - 84.3	-	20.9 - 44.2
MnasNets	327	25.6 - 99.3	5.8 - 24.1	19.8 - 60.9
ProxylessNass	532	34.5 - 195.9	7.9 - 72.2	18.0 - 77.8
NASBench201	97.5	5.6 - 27.9	1.8 - 8.3	2.3 - 6.4

Table 2: The FLOPs and latency of each model variants in our proposed dataset. It covers a wide spectrum.

with the highest test accuracy on CIFAR10 from the NASBench201, where each model has a different set of edge connections.

In total, our dataset contains 26,000 models with various operators (14 types), configurations (144,217 unique points), and edges. It has 2,012 different model graphs, while the remaining 24k models have different configurations. It supports both the 224×224×3 and 32×32×3 input image sizes. As shown in Table 2, the dataset covers a wide spectrum with different levels of FLOPs and latency.

4 KERNEL DETECTION

As discussed in Section 2.2, one key reason of nn-Meter's high prediction accuracy is to incorporate the knowledge of graph optimization of the framework. Out of the optimizations, operator fusion is the backend-dependent one that impacts latency the most. This section will introduce how the test cases of nn-Meter are designed to detect the fusion rules of a target backend and find all kernels of a CNN model.

4.1 Test Case Design

There are two major challenges of finding fusion rules and kernels. The first is that many inference backends for edge platforms are closed source. It is unable to get the kernels from the source code. The second is that there are arbitrary CNN models. To support the prediction of all models, the method to detect the fusion rules should be independent of specific model graphs. To solve the issue, we first analyze the essential features that affect the fusion implementation, then design test cases based on the features to reflect the fusion rules of a backend, and finally recursively apply these rules to a model graph to find all its constituent kernels.

Design principles. Our test case design is driven by two features of a CNN model which impact the fusion rules, i.e., *operator type* and *operator connection.* (We consider each operator as a type here. Type of fused operators will be discussed in Section 4.2.) Operator type can impact fusion rules because the fusion of different operators requires different implementation cost. For example, the kernel code (more precisely, its loop body) of injective operators such as activation functions in Fig. 2 can directly be connected to the code of other operators, and automatically generate a new kernel.

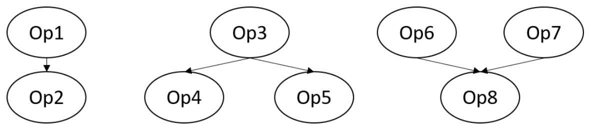


Figure 4: Operator connections: (a) single inbound and outbound; (b) multiple outbounds; (c) multiple inbounds.

Hence, this kind of fusion is widely implemented by backends. For some non-injective operators such as pooling, their codes cannot be easily connected to other operators, and thus the fusion gets less supported.

Besides of operator type, operator connection also impacts fusion rules. This is because improper fusion can not only cause additional time cost, but also cyclic operator dependency. Although the model graphs are arbitrary, they are all composed of three basic connection types as Fig. 4 shows: single in/outbound, multi-outbound and multi-inbound.

Suppose the operator fusion of the predecessor and successor pair in the graph are all supported by the backend. For the single in/outbound connection, Op1 and Op2 will be fused. However, for the multi-outbound connection (Fig. 4(b)), if both (Op3,Op4) and (Op3,Op5) are fused, Op3 will be unnecessarily calculated twice. Even if only (Op3,Op4) is fused, the fusion requires to keep the output tensors of both Op3 and Op4 at the same time, which can increase the memory cost. What is more, if there is an edge from Op5 to Op4, the fusion of (Op3,Op4) causes a dependency cycle in the graph, which breaks the acyclic requirement for a model graph. For the multi-inbound connection (Fig. 4(c)), the fusion of (Op6,Op8) or (Op7,Op8) will not invoke unnecessary cost. Besides, the no-fusion for multi-outbound rule makes sure this fusion will not cause cyclic dependency. Particularly, in the perspective of time saving, there should be no preference for (Op6,Op8) or (Op7,Op8) fusion, because both avoid the write and read of the Op8 input. Then, it is possible the first visited inbound would be fused. To conclude, operator connections can impact fusion rules, which requires test cases to detect them too.

The same principles apply when the number of out/inbounds is bigger than two. Thus, no test cases of nn-Meter is designed specifically for them.

Test cases Based on the analysis above, the test cases cover both operator type and connection to detect the fusion rules of different backends. For operator type, our test cases include the single in-/outbound connection permutation of every two possible operators, to detect whether they can be fused. Then, four fusible operators are selected to compose multi-in/outbound connections as Fig. 4 to detect whether they can still be fused.

The running time difference of connected and separated operators is used as the metric to judge whether fusion happens, since fusion reduces latency by connecting calculation on the same element together. That is, for a single in/outbound connection like Fig. 4(a), if the time of operators follows Inequation 1, they are regarded as being fused as Op1+Op2.

$$T_{Op1} + T_{Op2} - T_{(Op1,Op2)} > \alpha * min(T_{Op1}, T_{Op2}) \qquad (1)$$

In the inequation, T_{Op1} and $T_{(Op1,Op2)}$ mean the measured time of Op1 and (Op1,Op2) connection respectively. α is the empirical coefficient (set to 0.5 in our experiments) as a threshold. Table 3 uses (pool,relu) as an example to show the measured time and the detected rules (note that the VPU timing resolution is μs).

Backend	T_{pool} (μs)	T_{relu} (μs)	$T_{(pool,relu)}$ ($T_{pool} + T_{relu}$)	Rule
VPU	13	26	16 (39)	"pool_relu":True
GPU	5.08	3.50	6.00 (8.60)	"pool_relu":True
CPU	23.60	0.81	24.48 (24.42)	"pool_relu":False

Table 3: A fusion detection example (pool, relu).

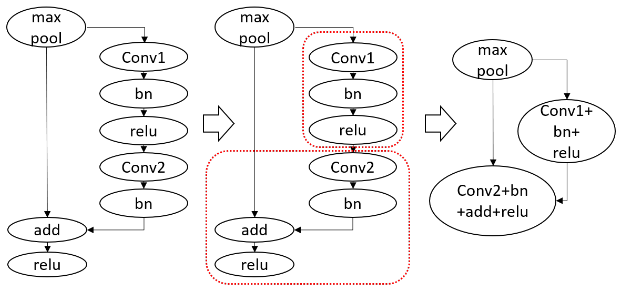


Figure 5: A kernel search example on a subgraph of ResNet18 model. The found kernels are {maxpool, Conv+bn+relu, Conv+bn+add+relu}.

Similarly, for the multi-in/outbound connection like Fig. 4(c), the candidate fusion plan with the closest time cost as the real time is regarded as the detected fusion plan. That is to pick the closest one among $T_{Op6} + T_{Op7} + T_{Op8}$, $T_{Op6+Op8} + T_{Op7}$, and $T_{Op6} + T_{Op7+Op8}$.

The detected fusion rules are recorded in JSON format. It is easy to be read by other algorithms or updated manually, such as adding known rules by framework designers.

4.2 Find All Kernels of a Model

Based on the detected fusion rules, for a model graph, nn-Meter recursively applies the rules to the graph to find all the constituent kernels (i.e., fused operators). The approach is shown in Algorithm 1.

The algorithm traverses the graph in a depth-first order from the root operator (line 22). If two operators (O_{pred},O_{succ}) can be fused according to the rules (line 11), a new fused operator O_{pred}+O_{succ} is created (line 2). The in/outbounds of the new operator are the union of the in/outbounds of O_{pred} and O_{succ}. The type of the new operator is the same as O_{pred}, since generally the kernel code of O_{succ} is appended to O_{pred} (line 5). Then, the new operator O_{pred}+O_{succ} will replace O_{pred} and O_{succ} in the graph. The traverse will continue from this new operator (line 18). The final output of the algorithm is a set of all constituent kernels of the graph.

Take Fig. 5, a subgraph of ResNet18, as an example to illustrate the fusion process. Suppose that the fusion rules related to this graph are as following (detected on the GPU backend). For multi-in/outbound rules, 0, 1, and 2 mean no fusion, fusion with the left-most in/out bound, and fusion with the right-most in/outbound respectively.

```
{"pool_add": True, "add_relu": True,   "pool_conv": False,
 "conv_bn": True,  "conv_relu": True, "conv_conv": False,
 "conv_add": True, "multi-inbound": 1, "multi-outbound": 0}
```

Algorithm 1 Kernel searching

Input: G a CNN model graph; R a set of fusion rules for a backend;
Output: Updated G with fused operators

```
1: function FUSE($O_{pred}$, $O_{succ}$)
2:     $O \leftarrow$ add a new operator in $G$ as $O_{pred}$ + $O_{succ}$
3:     $O.in \leftarrow O_{pred}.in \cup O_{succ}.in - O_{pred}$
4:     $O.out \leftarrow O_{pred}.out \cup O_{succ}.out - O_{succ}$
5:     $O.type \leftarrow O_{pred}.type$
6:     remove $O_{pred}$, $O_{succ}$ from $G$
7:     return $O$
8: end function
9: function DFSTRAVERSE($O_{pred}$)
10:    for $O_{succ} \in O_{pred}.out$ do
11:        if $Rule_{fuse}(O_{pred}.type, O_{succ}.type)$
12:          and (len($O_{pred}.out$)==1 or $Rule_{multiout}()$)
13:          and (len($O_{succ}.in$)==1 or $Rule_{multiin}()$) then
14:            $O_{next} \leftarrow$ FUSE($O_{pred}$, $O_{succ}$)
15:        else
16:            $O_{next} \leftarrow O_{succ}$
17:        end if
18:        DFSTRAVERSE($O_{next}$)
19:    end for
20: end function
21: ▷ Initial traverse input is the root of $G$
22: DFSTRAVERSE($O_{root}$)
```

VPU		GPU		CPU	
kernel	#	kernel	#	kernel	#
Conv+bn+relu	9	Conv+bn+relu	9	Conv+bn+relu	9
maxpool	1	maxpool	1	maxpool	1
Conv+bn	11	Conv+bn+add+relu	8	Conv+bn	11
add+relu	8	Conv+bn	3	add+relu	8
avgpool	1	avgpool	1	avgpool	1
FC	1	FC	1	FC	1

Table 4: Found kernels for ResNet18.

The algorithm visits maxpool first, and checks the fusion rules of (maxpool, add). They cannot be fused due to the failure of the multi-outbound rule. Add operator is visited next. It can be fused with its outbound relu as add+relu. The algorithm then goes back to check the fusion of maxpool with its next outbound Conv1. They cannot be fused due to the multi-outbound and "pool_conv" rules. (Conv1, bn) can be fused as Conv1+bn with an operator type as Conv. Next, since "conv_relu" rule is true, Conv1+bn can be further fused with relu as Conv1+bn+relu. However, it cannot be fused with Conv2 since the "conv_conv" rule is false. Conv2 is fused with its outbound bn as Conv2+bn. Finally, because "conv_add" rule is true, and Conv2+bn is also the first fusable inbound (multi-inbound rule) of add+relu, they can be fused together as Conv2+bn+add+relu. Table 4 shows all the found kernels of ResNet18 on the three backends. The kernels on the CPU and VPU are different from the GPU because their rule "conv_add" is false. Therefore, (Conv2+bn, add+relu) cannot be fused as the GPU backend.

Although the current detection method is only for operator fusion, the idea can be used to detect other graph optimizations too, such as the constant folding example $add(c1, add(x, c2))$->$add(x, c1+c2)$ mentioned in Section 2.2.

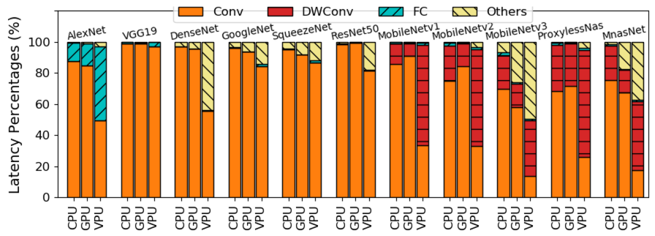


Figure 6: Model latency percentage breakdown. Conv and DWConv are the latency-dominating kernels.

5 LATENCY PREDICTOR

This section introduces the method to build latency predictors for kernels and models. We start by addressing the challenges in non-linear latency pattern and expensive sampling cost.

5.1 Kernel Characterization

Conv and DWConv dominate the latency of a model. By applying the kernel detection to the target model, we get a set of kernels. However, not all kernels account equally for the latency. Fig. 6 shows the model latency percentages by kernel types. We make the following observations: (1) In most models, Conv (Conv+bn+relu) and DWConv (DWConv+bn+relu) account for the main latency percentages. On average, Conv and DWConv take 94.2%, 91.91%, 75.5% of the model latency on the CPU, GPU, and VPU, respectively. (2) The latency of FC and element-wise operators (i.e., Others in Fig. 6) are relative large on the VPU. It's also necessary to estimate these small kernels for accurate prediction. For instance, FC can take 47.4% latency of AlexNet. Among all the detected kernels, Conv is the most challenging one due to the large sample space. We mainly take Conv as the example in the following discussion.

A large sample space of Conv. The possible configurations of a kernel decides the sample space. For the latency-dominating Conv and DWConv kernels, the primary configuration parameter includes: input height H, input width W, kernel size K, stride S, input channel number C_{in} and output channel C_{out}. Since H usually is equal to W for a kernel in CNN models, we encode it as a 5-dimension vector: $(HW, K, S, C_{in}, C_{out})$.

We collect 24 CNN models from PyTorch model zoo and get all the Conv configurations. As shown in Table 5, CNN models configure HW, K, S from a small range of numbers. However, the range of C_{in} and C_{out} are unbound. Among these models, C_{in} varies from minimal 3 to maximum 2160. The full sample space size is the multiplication of the size of every configuration dimension. To this end, the latency-dominating Conv has a vast amount (i.e., $\approx$ 0.7 billion) of configurations to sample.

Non-linear latency pattern. Existing works [5, 27, 28] assume the linearity between operator configurations and the corresponding latency. For instance, Conv with a larger H has a larger latency. However, as shown in Fig. 7 and Fig. 8, we observe that the K, HW, C_{in}, C_{out} show the non-linearity pattern on our measured devices. Instead, HW and C_{out} exhibit the staircase pattern, in which Conv with two different HW/C_{out} may have the same latency. These non-linearities reflect the complexities in hardware optimizations.

Dimension	Sample space
input HW	224, 112, 56, 32, 28, 27, 14, 13, 8, 7, 1
kernel size K	1, 3, 5, 7, 9
stride S	1, 2, 4
C_{cin}	range(3, 2160)
C_{out}	range(16, 2048)

Table 5: Sample space of Conv+bn+relu. It contains ≈ 0.7 billion configurations.

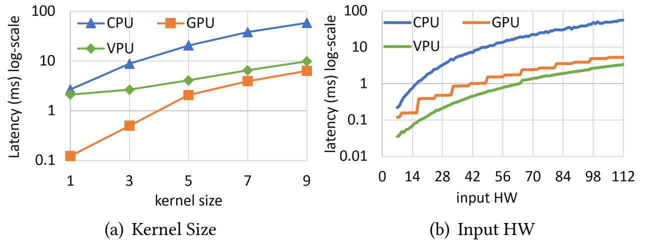


Figure 7: Conv+bn+relu with (a): different kernel sizes (HW=224, C_{in}=3, C_{out}=32, S=1); (b): different input heights/widths. (C_{in}=C_{out}=64, K=3, S=1)

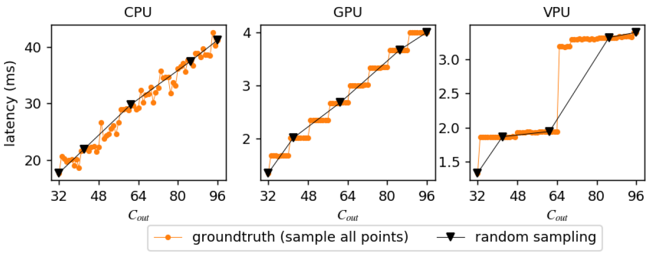


Figure 8: Latency of Conv+bn+relu with different output channel numbers. The groundtruth with sampling all channel numbers shows a staircase pattern on VPU and GPU. (HW=112, C_{in}=32, K=3, S=1)

Random sampling misses hardware-crucial data. To learn the non-linearity between configurations and latency, we need to generate a training set (i.e., variously configured kernels and the latencies) for regression. While it's unfeasible to sample and measure all the configurations of Conv, a direct method is random sampling.

However, we argue that it's difficult to build accurate predictors by random sampling. As shown in Fig. 8, random sampling ignores many important configurations (e.g., C_{out}=66 on the VPU). These crucial data reflect the complex hardware optimizations. Without them, predictors can easily learn an inaccurate latency pattern. To capture the staircase pattern on GPU and VPU, we should sample more data in the channel number dimension.

Main takeaways. Conv and DWConv are the latency-dominating kernels, and the prediction accuracy is most important to final model performance. However, the large sample space of Conv introduces the challenges for sampling and measurement (i.e., labeling). Random sampling ignores many crucial data as shown in Fig. 8.

5.2 Adaptive Data Sampling

Algorithm 2 Adaptive Data Sampling Algorithm

```
Input: $P$ prior possibility distribution from existing model zoo;
$N$ initial data to sample from $P$; $TD$ initial Test set;
$M$, number of data to sample for fine-grained sampling
$e$, the error threshold for regression model performance;
Output: all the sampled data ($X$,$Y$)
1: function FineGrainedSample($X$, $M$)
2:     for $x \in X$ do
3:         $D \leftarrow$ sample $M$ data, we fix the ($HW$, $K$, $S$), channel numbers
   are randomly sampled from range ($0.4C_o$, $1.2C_o$)
4:         $X_{new} \leftarrow X_{new}$+$D$
5:     end for
6:     $Y_{new} \leftarrow$ MeasureLatencyOnDevice( $X_{new}$)
7:     return ($X_{new}$,$Y_{new}$)
8: end function
9:
10: ▷ initialize $N$ data from prior distribution to measure
11: ($X$,$Y$)$\leftarrow$ sample $N$ data from distribution $P$
12: $f \leftarrow$ Construct regression model with ($X_{train}$,$Y_{train}$)
13: $TD \leftarrow TD$ + ($X_{test}$, $Y_{test}$)
14: $e(f) \leftarrow$ test $f$ on $TD$
15: ▷ perform fine-grained sampling for inaccurate data
16: while $e(f) > e$ do
17:     $X^* \leftarrow$ select data with large predict error from $TD$
18:     ($X_i$, $Y_i$)$\leftarrow$FineGrainedSample($X^*$, $M$)
19:     ($X$, $Y$) $\leftarrow$ ($X$, $Y$)+ ($X_i$, $Y_i$)
20:     update regression model $f$ with ($X_{train}$,$Y_{train}$)
21:     $TD \leftarrow TD$ + ($X_{test}$, $Y_{test}$)
22:     $e(f) \leftarrow$ test $f$ on $TD$
23: end while
```

Instead of random selection, the main idea is to sample the most beneficial data from the kernel configuration space. It covers (1) the configuration range in CNN design, and (2) hardware-crucial configurations that reflect the hardware optimizations and can significantly impact the prediction accuracy.

Driven by the two goals, we first prune the rarely-considered configurations by constraining the sampling distribution. This leverages the observation that many configurations are unlikely selected in state-of-the-art CNN models. And, the considered configurations are non-uniformly distributed in the sample space. For instance, due to the efficiency and accuracy, modern CNNs do not consider the Conv with large (224, 1, 1, 2160, 2048) and small (1, 1, 1, 3, 16) configurations. Second, we run an iterative process to sample more data around inaccurate prediction data. These data are treated as the hardware-crucial data. Since Conv has a 5-dimension configuration, we leverage the observation in Fig. 8 to sample in the channel number dimension.

To this end, we propose adaptive data sampling. Algorithm 2 illustrates the main steps. First, to generate sufficient configurations that are likely to be considered in CNN design, we sample by the prior possibility distribution (line 11). Then, to evaluate the sampled data quality, we build the machine learning predictor and design a test set for evaluation (line 12-14). Finally, we perform fine-grained channel number sampling for data with large prediction errors (line 1-8). The iterative process continues until the predictor accuracy meets user's requirements (line 16-23).

Kernel	Features	# Collected Data		
		CPU	GPU	VPU
Conv+bn+relu	HW, C_{in}, C_{out}, K S, FLOPs, params	15824	14040	39968
DWConv+bn+relu	$HW, C_{in}, K, S,$ FLOPs, params	4255	5054	7509
FC	C_{in}, C_{out} FLOPs, params	2000	3700	7065
maxpool	HW, C_{in}, K, S	1200	1366	1264
avgpool	HW, C_{in}, K, S	2575	1523	2179
SE	HW, C_{in}, ratio	2000	2000	2000
hswish	HW, C_{in}	1567	1567	1533
channelshuffle	HW, C_{in}	1000	-	1000
bn+relu	HW, C_{in}	2307	2000	-
add+relu	HW, C_{in}	2000	2000	2262
concat	$HW, C_{in1}, C_{in2},$ C_{in3}, C_{in4}	7674	8513	-

Table 6: Main kernels, features and valid data.

Prior possibility distribution P. It describes the boundary and the possibility of each data to sample. To compute it, we collect the configurations from 24 state-of-the-art CNN models and get the possibility distribution of each kernel dimension. We sample N data from the distributions as the initial data and measure the latency. In our experiment, we set N to 10,000 for Conv, 5,000 for DWConv, and 2,000 for other kernels.
Test set TD. It's crucial to construct the Test set as it evaluates the performance of sampled data and predictor. Since the sample size of the input HW, kernel size K, and stride S are small, we generate all the combinations of (HW, K, S) in Table 5. For C_{in} and C_{out}, we set the numbers that appeared in our collected model zoo. Specifically, the initial set contains 2,800 and 500 points for Conv and DWConv, respectively. To avoid overfitting, we expand 20% of newly sampled data into the test set in each iteration.
Fine-grained sampling around inaccurate data. After evaluating the predictor in each iteration, we pick out the data with large errors and perform fine-grained sampling. For each data, we fix all the other dimensions except the channel number C_o. We random sample M data from $[0.4 \times C_o, 1.2 \times C_o]$. For example, for Conv with (56, 3, 1, 24, 64), we fix the HW, K, S dimension, and sample M new C_{in}, C_{out} from [9, 28] and [25, 76], respectively. We set $M = 10$ in our experiment.

5.3 Kernel and Model Latency Prediction

Predict kernel latency. To learn the non-linearity observed in figs. 7 and 8, we use the Random Forests Regression [21]. Random Forests is ensemble decision tree-based and commonly reported as one of the most accurate learning algorithms. Some works [10] adopt the XGBoost [8]. While XGBoost has many hyper-parameters, Random Forests is much easier to tune for high performance. Table 6 lists out the prediction features and the number of collected data for building the latency predictor. For each kernel, we train and save the predictor for online model latency prediction.
Predict model latency. Finally, we estimate the model latency by the summation of all kernels' predicted latency shown in Equation 2.

	Device	Processor	Framework
CPU	Pixel4	CortexA76 CPU	TFLite v2.1
GPU	Xiaomi Mi9	Adreno 640 GPU	TFLite v2.1
VPU	Intel NCS2	MyriadX VPU	OpenVINO2019R2[17]

Table 7: Evaluated edge devices.

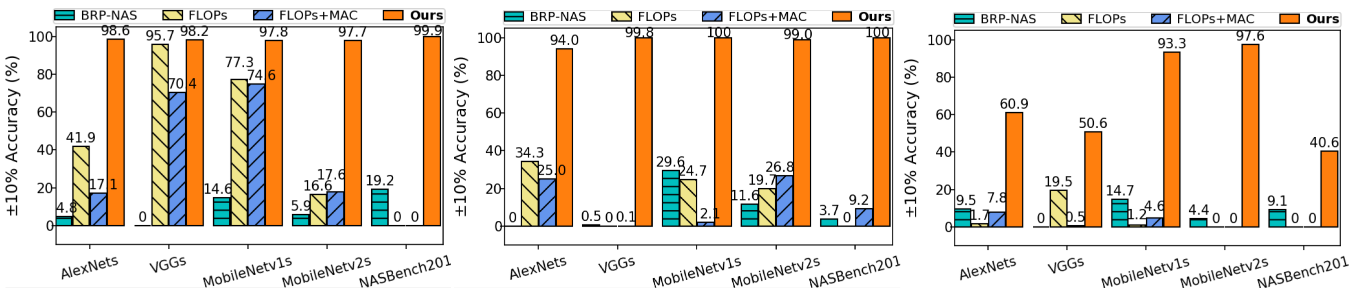
f_o is the ML predictor of kernel o, and x_o is the extracted features.

$$Latency(m) = \sum_{o \in m} f_o(x_o) \tag{2}$$

6 NN-METER IMPLEMENTATION

The entire nn-Meter consists of 18,093 lines of Python code (loc)- Test cases: 2,025 loc, Adaptive data sampling and kernel latency predictors: 8,052 loc, Model latency prediction: 1,291 loc, Benchmark dataset: 2,630 loc, Latency measurement: 4,095 loc.
Latency measurement. nn-Meter currently supports three widely-used edge devices, as shown in Table 7. Different from the mobile CPU, mobile GPU, the Intel NCS2 VPU is a dedicated AI accelerator.

We build an automated measurement platform to measure latency. Given a model/kernel configuration, we generate the graph in both the Tensorflow protobuf and tflite format, which are generally supported by edge inference frameworks. We send the target model to the measurement platform and collect the returned inference latency. To measure the latency on the CPU, we set CPU frequency to the highest 2.42GHz. The latency on the CPU is measured by the TFLite `benchmark` tool. Since TFLite currently doesn't support operator-level profiling for GPU, we implement an operator profiler in TFLite for GPU backend. For VPU latency measurement, we convert the protobuf format into OpenVINO IR, and measure the latency by the OpenVINO^{TM} toolkit. The latency number is the average of 50 inference runs.
Kernel detection. The test cases of nn-Meter cover all possible two-operator combinations for each of the three devices. The numbers of CNN operators are 26 (CPU), 21 (GPU), and 27 (VPU). The detected fusion rules are 668 (CPU), 434 (GPU), and 720 (VPU) respectively. The total found kernels from our dataset are 22 (CPU), 26 (GPU), and 22 (VPU). More kernels are found on the GPU because more fusion rules are supported by the TFLite GPU backend. For example, there is a Conv+bn+add+add kernel found for the GPU since the fusion of these operators is all supported. By comparison, the fusion rules supported by the CPU and VPU are limited to Conv, bn, and relu operators, resulting in less kernels (also refer to Table 4 for the real model example). The detected fusion rules and found kernels are the same as the framework reported results on our CPU and GPU backends. The VPU backend is not open source, and thus cannot directly verify the rules or kernels. However, as Section 7.3 will show, the prediction accuracy on the VPU based on the kernels are much higher (83.4% vs 8.5%) than operator-based prediction. Therefore, the fusion detection and kernel search algorithm are also effective on the black-box VPU.
Latency prediction. In our experiment, we observe the latency difference between Conv and its fused operators (e.g., Conv and relu/relu6, bn, add) is negligible (same as DWConv). For example, the latency of Conv, Conv+bn, Conv+bn+relu with configuration (56, 3, 1, 32, 32) is 0.404ms, 0.404ms, 0,405ms on the GPU, respectively.



(a) CPU (b) GPU (c) VPU

Figure 9: Compared to baseline predictors, nn-Meter achieves much higher ±10% accuracy on unseen models.

Therefore, for Conv and DWConv fused operators, we only build predictor for Conv+bn+relu and DWConv+bn+relu. To collect the data for regression, we manually set the error thresholds and run the adaptive data sampling. We split the collected data (Table 6) into train, validation, and test by 7:1:2, where we use the validation data for hyper-parameter tuning. The hyper-parameters are tuned by the popular NNI [26]. Table 9 lists out main kernels' performance.

7 EVALUATION

7.1 Experiment Setup

We evaluate nn-Meter on the benchmark dataset (Table 2) for CPU, GPU, and VPU (Table 7).

Comparison baselines. We implement 3 baselines for comparison: (1) FLOPs, (2) FLOPs+MAC, (3) BRP-NAS. Baseline (1) and (2) are the widely used latency predictors. Baseline (3) is the latency predictor in BRP-NAS, one of the state-of-the-art model-graph based prediction by GCN on the NASBench201 dataset. For baselines (1) and (2), we use the FLOPs and memory access cost [6] (i.e., MAC) to estimate model latency. We train the predictors by linear regression. For baseline (3), we directly run the BRP-NAS source code [2]. Since BRP-NAS currently implements for cell-based models, it can not directly apply to the non-cell-based models in our dataset. Thus, we modify the graph representation as follows.

The GCN in BRP-NAS takes as input a feature description matrix and a description of the graph structure as an adjacency matrix. BRP-NAS encodes the cell of NASBench201 model for representation. The GCN input is a 9×6 feature matrix and a 9×9 adjacent matrix. However, for the non-cell-based models in our dataset, we should encode the complete model graph. Therefore, we encode all the kernel nodes in a model graph. Besides, we also encode the 5-dimension configuration as the node attributes. Finally, the graph representations are larger than the BRP-NAS. Specifically, the NASBench201 models representation are a 133×22 feature matrix and a 133×133 adjacent matrix.

Metrics. We evaluate the prediction performance by the Root Mean Square Error (RMSE) and the relative Root Mean Square Percentage Error (RMSPE), that are the standard metrics in regression. Besides, we report the ±5% and ±10% accuracy [13], that are the percentage of models with predicted latency within the corresponding error bound relative to the measured latency. In this paper, ±10% error boundary is the maximum acceptable prediction error. We use ±10% accuracy as the default metric. Smaller RMSE/RMSPE and larger ±5%/±10% accuracy suggest better performance.

[6]the size of all feature maps and weights during the inference

7.2 End-to-End Prediction Evaluation

7.2.1 Comparison with Baselines on Unseen Models. In real-world scenarios, a usable predictor must be able to predict unseen models (i.e., a new model). As introduced, nn-Meter requires no model-level data for building the predictors, and can make predictions on models it has not seen before. To demonstrate it, we design a k-fold cross-validation experiment as follows.

Setting. We select AlexNets, VGGs, MobileNetv1s, MobileNetv2s, and NASBench201 for the evaluation. For each model variant, we take it as the testing set (e.g., 2,000 AlexNets), and the remaining 4 model variants (e.g., 8,000 models of VGGs, MobileNetv1s/v2s and NASBench201) as the training set to train the baselines. nn-Meter predicts model latency via the predicted latency sum of all kernels, it requires no model-level training data.

Results. Fig. 9 shows the prediction accuracy achieved by different predictors. Compared with the baselines, nn-Meter is the only approach that consistently achieves accurate predictions on various devices. None of the baselines can achieve comparable performance for unseen models on any device. Specifically, on average, nn-Meter achieves 89.2% accuracy, significantly better than FLOPs (22.1%), FLOPs+MAC (17.1%), and BRP-NAS (8.5%) on the three devices. The FLOPs/FLOPs+MAC predictors achieve better accuracy on the CPU compared to the VPU and GPU. This is because on these accelerators, operator fusion plays a more important role on latency reduction compared to the CPU due to the more serious memory wall issue. However, FLOPs/FLOPs+MAC ignores operator fusion impact. For the BRP-NAS baseline, the performance is consistently poor on three devices. As discussed in Section 2.1, the reason is the model graph differences between training and testing set. GCN learns the representation of model graphs. Although the five model variants have largely overlapped operator types, the operator configurations, edges, and model latency ranges are different.

To further demonstrate the effectiveness of nn-Meter on unseen models, we calculate the kernel configuration overlaps between sampled data and our benchmark dataset. A low ratio indicates a high generalization ability of kernel predictors. Results show

Model variants	Mobile CPU				Mobile GPU				Intel VPU			
	RMSE (ms)	RMSPE (%)	±5% Acc.	±10% Acc.	RMSE (ms)	RMSPE (%)	±5% Acc.	±10% Acc.	RMSE (ms)	RMSPE (%)	±5% Acc.	±10% Acc.
AlexNets	4.02	3.90	81.0%	98.6%	0.93	5.32	72.0%	94.0%	1.17	10.74	23.4%	60.9%
VGGs	185.71	4.84	66.1%	98.2%	12.74	2.97	91.8%	99.8%	85.35	22.25	27.1%	50.6%
DenseNets	7.10	2.76	93.1%	99.9%	1.99	4.52	68.55%	99.9%	2.83	5.89	75.6%	86.3%
GoogleNets	5.69	3.27	85.9%	100%	0.44	1.35	100%	100%	0.94	5.86	39.7%	98.4%
SqueezeNets	7.19	3.59	84.5%	99.9%	1.17	3.85	81.9%	97.9%	1.93	7.08	66.1%	88.5%
ResNets	26.87	4.41	72.3%	98.1%	2.58	3.16	88.8%	99.9%	3.39	7.42	37.9%	84.2%
MobileNetv1s	3.71	4.98	63.8%	97.8%	0.37	2.56	96.9%	100%	1.21	5.90	54.2%	93.3%
MobileNetv2s	3.25	4.84	67.6%	97.7%	0.54	3.93	80.0%	99.0%	1.29	4.26	78.3%	97.6%
MobileNetv3s	2.03	4.34	73.8%	99.0%	0.40	4.02	84.4%	100%	2.47	5.72	47.6%	98.5%
ShuffleNetv2s	2.48	5.01	61.6%	98.3%	-	-	-	-	1.91	6.37	45.6%	91.3%
MnasNets	3.19	5.54	50.9%	99.2%	0.25	1.86	100%	100%	1.76	4.34	77.3%	97.7%
ProxylessNass	3.18	3.44	84.6%	100%	0.61	3.28	95.6%	98.9%	1.97	5.05	65.6%	96.9%
NASBench201	0.44	3.51	82.4%	99.9%	0.12	3.80	75.9%	100%	0.90	18.20	19.3%	40.6%

Table 8: End-to-end latency prediction for 26,000 models on mobile CPU, GPU and Intel VPU.

Kernel	CPU		GPU		VPU	
	RMSE (ms)	±10% Acc.	RMSE (ms)	±10% Acc.	RMSE (ms)	±10% Acc.
Conv+bn+relu	6.24	89.1%	6.77	82.0%	18.74	67.9%
DWConv+bn+relu	0.21	97.4%	0.10	98.7%	0.28	89.4%
FC	0.64	94.3%	0.07	96.2%	0.12	93.9%
maxpool	0.12	89.6%	0.06	97.1%	0.21	89.7%
avgpool	1.94	99.0%	0.01	99.7%	0.26	95.4%
SE	0.45	87.1%	0.39	99.8%	0.44	99.0%
hswish	0.16	98.1%	0.01	100%	0.02	100%
channelshuffle	0.14	99.5%	-	-	0.35	100%
bn+relu	0.85	80.7%	0.01	100%	-	-
add+relu	0.10	93.7%	0.003	98.3%	0.02	98.9%
concat	0.09	89.3%	0.42	77.1%	-	-

Table 9: Performance for main kernel predictors.

that our kernel predictors have seen only 5.9% (CPU), 9.4% (GPU), 5.0%(VPU) configurations in the dataset, but can accurately predict the remaining unseen ones.

7.2.2 nn-Meter Results and Analysis. We now provide results of nn-Meter on the full benchmark dataset (in Table 2). We predict the latency of 26,000 models on each evaluated device. Remarkably, we achieve 99.0%, and 99.1% prediction accuracy on the mobile CPU and GPU, respectively. On the Intel VPU, we can predict 83.4% models within the ±10% error boundary. Table 8 lists out the performance for each model variant on three devices. We can see that the strong performance (small RMSE and high accuracy) generalizes across various devices, which have vastly different latency behaviors. Significantly, >95% of all model variants on the CPU and GPU are with a $< 10\%$ prediction error. nn-Meter even reaches an impressive high ±5% accuracy on the GPU. On the VPU, we notice that nn-Meter achieves relative low accuracy for the AlexNets, VGGs, and NASBench201.

To better investigate the performance on the VPU, we divide the dataset into 4 groups by the model measured latency. As shown in Fig. 10, the relatively large errors are from models with very small (i.e., NASBench201 and AlexNet models <10 ms) or very large (i.e., VGG models >100 ms) latency. Fortunately, these models take a small percentage in our dataset. In a real-world scenario, models with very small latency are more likely with a lower classification accuracy, and models with very large latency (i.e., the average FLOPs of VGGs is 28,422M) are rarely considered for edge devices.

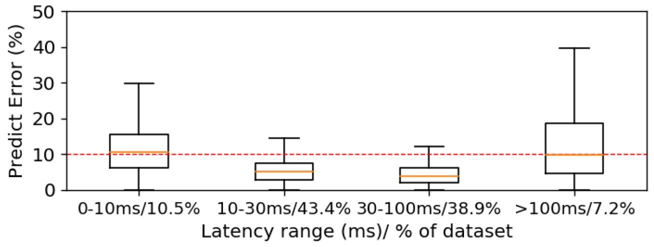
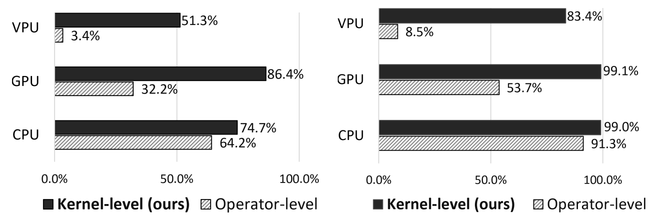


Figure 10: Prediction errors on the VPU. X-axis label: latency range/group size percentages of the dataset.

We now perform manual analysis to reason the failure cases on the VPU. First, model latency is the sum of all kernels' latencies. The accumulated kernel prediction errors sensitively impact the AlexNets and NASBench201, which have low inference latency. To relax the prediction error boundary to 15%, we can reach 86.0% accuracy on AlexNets and 73.3% accuracy on NASBench201. Second, the latency of a single kernel can be significantly different from that in a complete model. For example, for the Conv+bn+relu with a large configuration of (28, 7,1, 819, 768), the latency is 628.7 ms for a single kernel, but becomes 188.6 ms in a VGG model variant. By comparing the execution graphs of Conv within/without a model, we found that VPU performs ad-hoc optimizations that merges the computation of Conv+bn+relu and the next maxpool layer in VGGs. This only happens for very large Conv+bn+relu. We will further discuss it in Section 8.

7.3 Microbenchmarks

Kernel-level prediction. As the core component, kernel detector automatically detects the kernels on each device. The kernel-level prediction diminishes the latency differences caused by operator



(a) ±5% accuracy (b) ±10% accuracy

Figure 11: Operator-level approach achieves much lower ±5% and ±10% accuracy on three devices.

Device	Random Sampling		Adaptive Sampling	
	RMSE	±10% Acc.	RMSE	±10% Acc.
CPU	25.47 ms	21.92%	10.13 ms	71.78%
GPU	1.67 ms	48.70%	1.19 ms	75.34%
VPU	7.87 ms	23.98%	7.58 ms	54.33%

Table 10: Under the same amount of sampled data, we achieve better performance than random sampling.

fusion. To demonstrate the effectiveness, we build the operator-level baseline. It predicts the latency of all operators in a model and sums them as the model latency. Since the latency difference between Conv+bn+relu and Conv is negligible (same for DWConv+bn+relu and DWConv), we use the Conv+bn+relu predictor for Conv. We build extra predictors for relu, bn, and add. The ±10% prediction accuracy is high, which ranges from 87% to 98%.

We test the operator-level baseline on our dataset. Fig. 11 shows the ±5% and ±10% accuracy achieved by two different approaches. On all devices, our kernel-level prediction consistently outperforms operator-level prediction. We observe that operator-level prediction performs unstably on different devices. For the ±10% accuracy, it achieves relative high accuracy on the CPU (91.3%) and GPU (53.7%), but only 8.5% on the VPU. The reason is that these element-wise operators take very small latency percentages (≈ 0.1%-15%) of the model on the CPU and GPU, but high latency percentages (up to 50%) on the VPU. Therefore, the baseline still achieves high prediction accuracy on the CPU and GPU for the Conv-dominating models (e.g., VGGs). However, the operator-level prediction does not work for non-Conv-dominating models. Specifically, it achieves only 65.7%, 48.2% accuracy on the CPU for MobileNetv2s and NAS-Bench201, respectively. The performance is worse on the GPU. Only 2.0% MobileNetv2s and 6.5% ProxylessNass are still within the ±10% error boundary. On the VPU, it has only 8.5% accuracy without considering the impact of operator fusion.

Adaptive data sampling. We now evaluate the performance of adaptive data sampling for its two key tasks: sampling efficiency for Conv and the effectiveness to model prediction. We compare it with random sampling. For each device, we randomly sample the same amount of Conv data as ours (as shown in Table 6).

First, we compare the sampling efficiency for Conv by two different approaches. Due to the large sample space, the sampled data are very different. For a fair comparison, we report the performance on the initial test set that contains 2,800 data (refer to Section 5.2).

Models (measure on Adreno630)	Adreno640 predictor				Adreno630 predictor			
	rmse (ms)	rmspe (%)	±5% Acc.	±10% Acc.	rmse (ms)	rmspe (%)	±5% Acc.	±10% Acc.
AlexNets	8.02	25.40	0.6%	2.3%	0.87	3.61	86.5%	97.9%
VGGs	154.50	24.85	0%	0%	19.47	3.10	89.5%	99.9%
DenseNets	4.44	7.80	18.4%	84.0%	2.41	4.58	67.1%	100%
GoogleNets	6.71	17.03	0%	0%	1.45	3.75	95.8%	100%
SqueezeNets	8.02	19.21	0.4%	3.0%	1.14	3.30	87.3%	100%
ResNets	33.82	21.36	1.3%	11.0%	2.52	2.65	93.4%	100%
MobileNetv1s	0.28	1.92	98.3%	100%	0.23	1.68	99.7%	100%
MobileNetv2s	0.85	5.43	55.6%	97.2%	0.41	3.42	85.7%	99.5%
MobileNetv3s	0.87	8.25	5.2%	87.5%	0.56	6.14	31.5%	99.6%
MnasNets	0.74	5.45	42.2%	99.8%	0.31	2.26	99.9%	100%
ProxylessNass	0.86	4.33	71.5%	100%	0.21	1.11	100%	100%
NASBench201	0.77	14.94	3.2%	17.5%	0.36	6.22	48.1%	90.6%

Table 11: Two different latency predictors for model inference on the Adreno GPU 630.

Table 10 shows the RMSE and ±10% accuracy. Under the same sampling budgets, adaptive data sampling achieves much smaller RMSE and higher accuracy than random sampling. We observe that random sampling generates lots of large but rarely-considered Conv (e.g., configuration of (224, 7, 4, 2141, 1876) has a 750MB size).

Then, we compare the model prediction accuracy achieved by predictors trained with randomly sampled and adaptively sampled data. We evaluate the performance of the 2,000 AlexNets as they are Conv-dominating models. Note that we only change the predictors of Conv and keep others the same. By adopting the Conv predictor trained with randomly sampled data, the accuracy heavily drops to 5.8%, 32.3%, 0% on the three devices.

7.4 Generalization Performance

In previous sections, we build and test latency predictors for the three types of hardware (in Table 7), and demonstrate the high prediction accuracy of nn-Meter. We now discuss the generalization performance for nn-Meter on a new edge platform. The experimental device is a Pixel3XL phone with a mobile Adreno 630 GPU, which is a lower version than the Adreno 640 GPU in Table 7. We measure the inference latency of the dataset in TFLite 2.1 on the Adreno 630 GPU for testing. The evaluation contains two folds.

Firstly, we measure the cross-device generalization performance. We use the existing latency predictors trained for Adreno 640 GPU to predict the model latency on the Adreno 630 GPU. As shown in Table 11, the non-Conv-dominated models (i.e., MobileNet-series, MnasNets, and ProxylessNass that contain both Conv and DWConv kernels) can achieve high prediction accuracy. However, the Conv-dominated model variants (i.e., AlexNets, VGGs, GoogleNets, etc.) achieve much lower prediction accuracy with > 15% RMSPE. The reason is that the Conv kernel runs faster on Adreno 640 GPU than that on Adreno 630 GPU, while the DWConv kernel has a similar inference latency on the two different versions of Adreno GPUs.

Secondly, we rebuild the latency predictors for the Adreno 630 GPU and use them to predict model latency. The results then become very promising as shown in Table 11. In total, nn-Meter

	CPU	GPU	VPU
total measure time	2.5 *days*	1 *day*	4.4 *days*

Table 12: Time cost of nn-Meter.

achieves 99.0% prediction accuracy with the rebuilt latency predictors on the Adreno 630 GPU. The rebuilt cost is acceptable as evaluated in the next section.

7.5 System Overhead

Finally, we evaluate the system overhead. As shown in Table 12, most of the overhead comes from the measurement time. In total, we take 2.5 days, 1 day, and 4.4 days to measure the latency of all sampled kernels on a single CPU, GPU, and VPU device, respectively. The measurement cost can be linearly scaled down by increasing more devices, which indicates it requires low efforts to build predictors for new device by nn-Meter.

8 DISCUSSION

Prediction for language models. Current edge inference backends mainly support CNN models but not language models. For example, TFLite does not support BERT-mini [31] inference. Therefore, nn-Meter is only evaluated on CNN models in this paper. The technique, however, should also be applicable for language models since they are also DAGs composed of operators.

Limitations. There might be some ad-hoc optimizations or implementations in the frameworks for certain unknown conditions, such as specific input size discussed in Section 7.2. For the black-box backends, it is not feasible to find common rules behind these ad-hoc optimizations to design test cases for detection yet. However, these ad-hoc optimizations are rare, and their impact on prediction accuracy is limited.

For new inference backends and significant updates on available backends, the predictor building process should be done again to meet the new implementation. The major cost is the data profiling shown in Table 12.

There are also some backends which generate different kernel codes and search for the fastest one for different input size, such as TVM [9]. nn-Meter could also built predictors for these backends based on the searched kernel implementation. However, these backends are hardly used on edge platforms, because of the large time cost for code generation and search (easily takes hours to run for one configuration). Therefore, nn-Meter has not built predictors for this kind of backends. We leave it for future work.

Current nn-Meter predictors are built offline and will not be updated dynamically during the inference phase. It is possible to integrate more dynamic resource impact in the predictors such as current CPU utilization. This can also be a future research direction.

Concurrent execution. The design of nn-Meter is based on the fact that current kernels run sequentially on edge chips (refer to Section 2.3). For possible future inference where kernels may run concurrently on heterogeneous edge chips or multi-cores, a potential solution is to extend nn-Meter with a static-analysis phase to work out the kernel execution plan first. The predicted model latency will then be the sum of kernel latencies on the longest sequential path, as well as the synchronization cost. The prediction accuracy is possibly lower than that for purely sequential run, since there are more uncertainties introduced by concurrent execution.

Power prediction. It should be straightforward to extend nn-Meter to predict kernel power or energy by training the predictors using measured power or energy data. However, it will be difficult to conduct thermal or heat modeling, since heat dissipation depends on the external environment which is hard to model.

9 RELATED WORK

Unaware of runtime implementation. Current CNN design uses high-level APIs, which are independent of runtime implementation. Besides, most runtimes are closed source. Therefore, many CNN latency predictors only rely on CNN model features but no consideration of runtime implementation. Some works [15, 22, 23, 30] simply use FLOPs and MAC of the model as proxies for latency, or use these as feature inputs of regressors [28] to predict latency. However, these methods are inaccurate because they neglect the runtime behaviour difference of various operators. Similar as our paper, NeuralPower [5] and PALEO [27] predict latency for operators or layers, and sum them up as the model latency. They are inferior to nn-Meter due to the ignorance of model graph optimizations of the runtime.

BRP-NAS [13] can learn both the operator latency and graph optimization by encoding operator type and graph as features to a GCN prediction model. However, as we have shown, its generalization ability is low to new CNN models with diverse number of operators, and the connection distance it can learn is limited.

Prediction on operator implementation. Some operator-latency predictors use machine learning methods to learn latency from low-level implementations.They either use code features and simple regression model to predict operator latency [1, 16], or costly DNN code embedding [19, 25] approach to avoid feature engineering. TVM [10] uses both approaches to accelerate its code search process. Its embedding-based latency predictor recursively uses TreeGRU model to embed a low-level AST into a vector and then map it to predicted latency using a linear layer. The other predictor uses code features like memory accesses, data reuse, vectorization, and unrolling as inputs to an XGBoost model to predict latency. However, since most edge DNN runtimes are closed source, it is infeasible to use these code-based methods.

There are also analytical latency prediction methods generally used by language compilers (e.g., LLVM-MCA [3] and IACA [18]), and cycle-accurate hardware simulation (e.g. gem5 [4] and GPGPU-Sim [20]). These methods require knowledge of the exact mechanisms of the processor, which is also infeasible for black-box edge AI hardware.

10 CONCLUSION

We propose nn-Meter, a kernel-based prediction system that accurately predicts the latency of DNN models on diverse edge devices. nn-Meter introduces kernel detection that captures the various operator-fusion behaviours. By sampling the most beneficial data, nn-Meter efficiently builds latency predictors for kernels. We demonstrate the effectiveness of nn-Meter with experiments on a large dataset and three types of edge devices.

REFERENCES

[1] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to Optimize Halide with Tree Search and Random Programs. *ACM Trans. Graph.* 38, 4, Article 121 (July 2019), 12 pages. https://doi.org/10.1145/3306346.3322967

[2] BRP-NAS authors. 2020. Eagle: Efficient and Agile Performance Estimator and Dataset. https://github.com/thomasccp/eagle.

[3] Andrea Di Biagio. 2018. *llvm-mca: a static performance analysis tool.*

[4] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7. https://doi.org/10.1145/2024716.2024718

[5] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. 2017. *NeuralPower*: Predict and Deploy Energy-Efficient Convolutional Neural Networks. In *Proceedings of the Ninth Asian Conference on Machine Learning (Proceedings of Machine Learning Research)*. PMLR, 622–637.

[6] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once-for-all: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations (ICLR)*.

[7] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *International Conference on Learning Representations (ICLR)*.

[8] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Carlsbad, CA, 578–594.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 3393–3404.

[11] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, et al. 2019. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 11398–11407.

[12] Xuanyi Dong and Yi Yang. 2020. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. In *International Conference on Learning Representations (ICLR)*.

[13] Lukasz Dudziak, Thomas Chau, Mohamed Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas Lane. 2020. BRP-NAS: Prediction-based NAS using GCNs. In *Advances in Neural Information Processing Systems (Neurips)*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 10480–10490. https://proceedings.neurips.cc/paper/2020/file/768e78024aa8fdb9b8fe87be86f64745-Paper.pdf

[14] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *International Conference on Learning Representations (ICLR)*.

[15] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In *European Conference on Computer Vision (ECCV)*.

[16] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. 2010. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression.. In *Advances in Neural Information Processing Systems (NIPS)*.

[17] Intel. 2019. Deploy High-Performance Deep Learning Inference, OpenVINO. https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html.

[18] Gideon Stupp Israel Hirsh. 2019. *Intel Architecture Code Analyzer.*

[19] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, , and Mike Burrows. 2019. Learned TPU Cost Model for XLA Tensor Programs. NeurIPS workshop.

[20] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. 2020. Accel-Sim: An Extensible Simulation Framework for Validated GPU Modeling. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020.* IEEE, 473–486. https://doi.org/10.1109/ISCA45697.2020.00047

[21] Breiman Leo. 2001. Random Forests. In *Machine Learning*. 5–32.

[22] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. In *The International Conference on Learning Representations (ICLR)*.

[23] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations (ICLR)*.

[24] Zechun Liu, Haoyuan Mu, Xiangyu Zhang, Zichao Guo, Tim Kwang-Ting Cheng Xin Yang, and Jian Sun. 2019. MetaPruning: Meta Learning for Automatic Neural Architecture Channel Pruning. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[25] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*. 4505–4515.

[26] Microsoft. 2019. Neural Network Intelligence. https://github.com/microsoft/nni.

[27] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *International Conference on Learning Representations (ICLR)*.

[28] Stefan Reif, Judith Hemp Benedict Herzog, Timo Hönig, and Wolfgang Schröder-Preikschat. 2020. Precious: Resource-Demand Estimation for Embedded Neural Network Accelerators. In *First International Workshop on Benchmarking Machine Learning Workloads on Emerging Hardware.*

[29] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2820–2828.

[30] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[31] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Well-Read Students Learn Better: On the Importance of Pre-training Compact Models. arXiv:1908.08962 [cs.CL]

[32] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 10734–10742.

[33] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, Felix Xiaozhu Lin, Yunxin Liu, and Xuanzhe Liu. 2019. A First Look at Deep Learning Apps on Smartphones. In *The World Wide Web Conference (WWW)*.

[34] Li Lyna Zhang, Yuqing Yang, Yuhang Jiang, Wenwu Zhu, and Yunxin Liu. 2020. Fast Hardware-Aware Neural Architecture Search. In *Proceedings of the IEEE Computer Vision and Pattern Recognition Workshops (CVPRW)*.

[35] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.