Predicting Inference Latency of Neural Architectures on Mobile Devices

Zhuojin Li
University of Southern California
Los Angeles, USA
zhuojinl@usc.edu

Marco Paolieri
University of Southern California
Los Angeles, USA
paolieri@usc.edu

Leana Golubchik
University of Southern California
Los Angeles, USA
leana@usc.edu

ABSTRACT

Due to the proliferation of inference tasks on mobile devices, state-of-the-art neural architectures are typically designed using Neural Architecture Search (NAS) to achieve good tradeoffs between machine learning accuracy and inference latency. While measuring inference latency of a huge set of candidate architectures during NAS is not feasible, latency prediction for mobile devices is challenging, because of hardware heterogeneity, optimizations applied by machine learning frameworks, and diversity of neural architectures. Motivated by these challenges, we first quantitatively assess the characteristics of neural architectures and mobile devices that have significant effects on inference latency. Based on this assessment, we propose an operation-wise framework which addresses these challenges by developing operation-wise latency predictors and achieves high accuracy in end-to-end latency predictions, as shown by our comprehensive evaluations on multiple mobile devices using multicore CPUs and GPUs. To illustrate that our approach does not require expensive data collection, we also show that accurate predictions can be achieved on real-world neural architectures using only small amounts of profiling data.

CCS CONCEPTS

• **General and reference** → **Performance**; **Empirical studies**; • **Computing methodologies** → **Neural networks**; • **Human-centered computing** → **Mobile devices**.

KEYWORDS

Neural Networks, NAS, Latency, Prediction, Mobile, GPU, CPU

ACM Reference Format:
Zhuojin Li, Marco Paolieri, and Leana Golubchik. 2023. Predicting Inference Latency of Neural Architectures on Mobile Devices. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE '23), April 15–19, 2023, Coimbra, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3578244.3583735

1 INTRODUCTION

Due to significant breakthroughs in machine learning (ML), inference tasks using neural networks are being deployed to a growing number of mobile devices (e.g., smartphones, smartwatches, tablets), largely for computer vision and natural language tasks. In comparison with powerful cloud servers, mobile devices have limited resources, which restricts the choice of neural architectures (NAs).

To achieve good tradeoffs between machine learning accuracy and hardware efficiency, state-of-the-art NAs [28, 55, 56] are typically designed through Neural Architecture Search (NAS) [70]. For example, recent work [55, 65] proposes to optimize accuracy under constraints on efficiency metrics (e.g., latency) that are measured directly on a target platform. However, NAs exhibit distinct performance characteristics across platforms [57], and it is impractical to measure the end-to-end latency of every architecture on all platforms during a model search. As an alternative to measurements, existing approaches for evaluating the efficiency of NAs can be categorized as those using (1) proxy metrics [56, 71] (e.g., FLOPs), which are usually platform-independent and cannot accurately reflect the actual performance due to the diversity of platforms [44, 57]; (2) look-up tables [9, 12, 62] of measurements collected for the building blocks of NAs, which require extensive profiling on each platform and cannot cover every possible block configuration; (3) prediction models, which can predict the performance of any block configuration in the search space, broadly relying on machine learning techniques [2, 5, 6, 10, 14, 16, 17, 24, 35, 42, 68], but also including analytical performance models (e.g., accounting for computations [49] and memory access traffic of GEMM-based convolution [40, 43]). However, building accurate prediction models for efficiency metrics on *mobile devices* is difficult due to the following challenges (where we also highlight related work).

(1) Hardware heterogeneity: Existing prediction models mainly focus on Nvidia cloud GPUs [2, 16, 17, 24, 35] or Nvidia embedded GPUs [5, 6]; instead, the heterogeneity of mobile CPUs and GPUs makes their performance predictions more difficult. In particular, inference tasks are frequently performed on mobile devices using CPUs [63], due to the support of a broader set of available operations (e.g., Channel Shuffle [69] is currently unavailable on the TensorFlow Lite (TFLite) [19] GPU Delegate [38]). Modern mobile CPUs typically use the ARM big.LITTLE architecture, which consists of heterogeneous core clusters, e.g., high-performance cores and high-efficiency cores [61]; when an inference task takes advantage of this multicore architecture, the schedule of threads on different cores has a significant impact on performance (Section 3.1.1). In addition, multicore speedups on a given device can vary for different NAs; for instance, MobileNet (with width multiplier of 0.75) and ResNet18 (with width scale of 0.25) achieve comparable inference latency (28.4 ms and 28.1 ms, respectively) on Pixel 4 with one medium core, but differ by 24.6% with three medium cores (11.8 ms and 14.7 ms, respectively). Therefore, it is necessary to evaluate prediction approaches using heterogeneous hardware resources, in

ICPE '23, April 15–19, 2023, Coimbra, Portugal

ACM ISBN 979-8-4007-0068-2/23/04.
https://doi.org/10.1145/3578244.3583735

Device	Platform	CPU Cores	GPU
Google Pixel 4	Snapdragon 855	1x Large (2.84 GHz), 3x Medium (2.32 GHz), 4x Small (1.80 GHz)	Adreno 640
Xiaomi Mi 8 SE	Snapdragon 710	2x Large (2.20 GHz), 6x Small (1.70 GHz)	Adreno 616
Samsung Galaxy S10	Exynos 9820	2x Large (2.73 GHz), 2x Medium (2.31 GHz), 4x Small (1.95 GHz)	Mali G76
Samsung Galaxy A03s	Helio P35	4x Large (2.30 GHz), 4x Small (1.80 GHz)	PowerVR GE8320
Apple iPhone XS	A12 Bionic	2x Large (2.49 GHz), 4x Small (1.52 GHz)	Apple-designed G11P
Apple iPhone 7	A10 Fusion	2x Large (2.34 GHz), 2x Small (1.05 GHz)	PowerVR GT7600 Plus (Custom)

Table 1: Mobile Platforms in Our Study

particular on multicore CPUs; this is *not taken into consideration by existing work* on latency prediction for mobile CPUs [10, 42, 68].

(2) ML framework optimizations: Modern ML frameworks introduce optimizations that can significantly accelerate inference tasks. For example, operator fusion [47] reduces overhead in the invocation of OpenCL/Metal kernels on GPUs: our tests show that disabling kernel fusion in TFLite can lead to an average of 22% performance degradation over 102 real-world NAs on PowerVR GE8320 (Section 3.2.1). Similarly, the choice of algorithms used to implement each operation can considerably affect inference performance: for example, TFLite uses the faster Winograd [37] algorithm for some (but not all) convolution layers on GPUs. *Existing work* on latency estimation for GPUs [5, 6, 10, 17, 35] *does not consider such optimizations* (which are specific to ML frameworks); instead, current literature predicts inference latency only from the features of NAs and hardware platforms.

(3) Neural architecture diversity: During the exploration of the search space by NAS algorithms, the properties of NAs (e.g., the number of operations and their latency) can vary considerably; in addition, novel neural architectures are proposed by manual design [28, 44, 69], prompting the definition of new NAS search spaces. *Existing ML-based performance prediction models use training and test datasets with very similar NAs* [2, 5, 54], *or with a small set of popular NAs* [8, 17, 24]; in contrast, practical applicability to NAS requires accuracy on a large set of *diverse* NAs.

Motivated by these challenges, we first quantitatively assess characteristics of neural architectures and mobile devices affecting inference latency; then, we use our findings to develop a framework to predict end-to-end inference latency on mobile CPUs and GPUs by estimating the latency of NA components through machine learning models. In so doing, we address several shortcomings of related work: (i) we develop a training dataset that is *more representative of real-world NAs*, by including a broader set of NA blocks than current literature [13]; (ii) we measure and predict latency on *different combinations of heterogeneous CPU cores* and *different data representations* (i.e., floating-point or integer quantization [46]), while related work [42, 68] uses only a single CPU core (unrealistic in practice) and floating-point calculations. Notably, our solution explicitly accounts for optimizations applied by TFLite on each NA; in contrast, previous work nn-Meter [68] uses black-box models to estimate the effects of ML framework optimizations (and is limited to a single core for CPUs).

Specifically, the main contributions of our work are as follows.

- By collecting measurements for 102 state-of-the-art NAs (from 25 articles) on 6 mainstream mobile platforms (or SoCs), and based on quantitative evidence, we identify aspects of hardware and ML frameworks that substantially affect the latency of inference tasks on mobile devices. For mobile CPUs, we expose performance characteristics under various settings, including multithreading over ARM heterogeneous core clusters and quantization with lower-bit representations (Section 3.1). On mobile GPUs, we analyze two types of optimization strategies due to ML frameworks: kernel fusion and kernel selection (Section 3.2). As a representative example, we present the principles of both strategies in TFLite, and empirically evaluate resulting speedups to highlight their impact on inference latency.
- Based on the results of our performance study, we develop a framework for estimating end-to-end inference latency on mobile devices by combining accurate latency predictions of individual NA components (Section 4.2). In contrast with complex ML models predicting end-to-end latency from graphs of tensor operations (including parameters of all operations) [2, 14, 16], latency predictors for NA components require less training data and are easier to interpret. To address hardware heterogeneity, we profile execution times of NAs using different sets of CPU cores and different data representations, and we train ML models to predict performance for each combination.[1] For ML framework optimizations, we are able to deduce the OpenCL/Metal kernels that are selected on mobile GPUs, *without deploying and compiling the target NA on the actual hardware* (Section 4.1). After collecting *one-time* training data on each device, we apply ML models to accurately predict the latency of inference tasks under various settings of mobile CPUs and GPUs, which can be used by existing NAS techniques *without* access to the actual hardware.
- Since the existing benchmark dataset NATSBench [13] (studied in [2, 68]) lacks depthwise convolution operations and exhibits limited diversity of operation configurations (see Section 5.6.2 for quantitative analysis), we build a synthetic dataset of 1000 NAs sampled from a NAS space covering a majority of configurations for common operations and building blocks (Section 4.3). For each NA, we comprehensively measure latency under 90 scenarios across 6 mainstream mobile platforms, including multicore combinations and use of integer quantization. In addition to accurate latency prediction, this dataset provides insight (i) to NA developers on how to build efficient NAs and (ii) to mobile developers on how to choose effective optimizations.
- To evaluate how our approach addresses the aforementioned challenges, in addition to the default setting of NAS (Section 5.1), we show that our approach also achieves accurate predictions under

[1] As in existing literature [2, 68] on mobile devices, we collect data and train models for each setting instead of constructing one model to predict inference latency across all devices (e.g., [17] for cloud GPUs) due to the heterogeneity of mobile platforms.

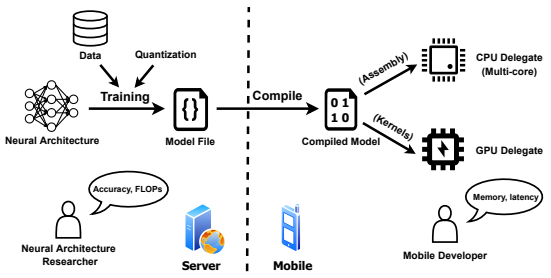


Figure 1: Lifecycle of Neural Architecture Development and Deployment on Mobile Devices

hardware heterogeneity (Section 5.2), neural architecture diversity (Section 5.3), and ML framework optimizations (Section 5.4). To address concerns regarding the cost of training data collection [42], we evaluate prediction accuracy with limited amounts of training data, using multiple ML methods (Section 5.5). Our results highlight that, when trained with latency measurements for a sufficient number of NAs (e.g., 1000 synthetic NAs), powerful ML methods (e.g., GBDT [15]) achieve very accurate predictions for NAs with similar characteristics (e.g., 2.4% and 5.2% average relative error on CPUs and GPUs, respectively); when training and testing data have different characteristics (e.g., training on synthetic NAs and testing on real-world NAs), simple linear models (e.g., Lasso [58]) are robust and still accurate (e.g., 6.5% and 8.3% average relative error on CPUs and GPUs, respectively). When training data is very limited (e.g., 30 synthetic NAs), accuracy is lower (e.g., 8.1% and 8.6% average relative error on CPUs and GPUs with GBDT, respectively) but sufficient for NAS, while profiling time for a target device is negligible compared to deploying and measuring latency of thousands of candidate NAs, as noted in [42].

2 BACKGROUND

As illustrated in Fig. 1, the lifecycle of neural architecture development and deployment on mobile devices consists of (1) designing and training a neural network on cloud servers, and (2) deploying the model on a target mobile device where inference tasks are performed on CPU cores or GPU.

State-of-the-art neural architectures are developed by both manual design [26, 29, 44] and NAS [28, 52, 55, 56]. Due to scarce computing and memory resources, NAs intended for inference tasks on mobile devices are designed not only to maximize prediction accuracy, but also to satisfy performance constraints on end-to-end latency and memory consumption. To achieve these goals, *model quantization* [34, 46] is frequently applied: instead of floating-point values, fixed-width integers are used to represent model parameters and to perform computations with low precision, reducing memory requirements and computation times (as shown in Section 3.1.2).

After training on cloud servers, the identified neural architecture is stored as a model file, which can be distributed to heterogeneous mobile platforms for inference tasks. For instance, in TFLite, a neural architecture is described as a computational graph, where each node represents an operation and each edge represents the flow of

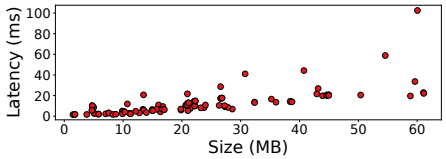


Figure 2: 102 Real-world NAs Evaluated in Our Study

intermediate results between operations; the complete computational graph is included in the `.tflite` model file.

A mobile device can be equipped with multiple hardware accelerators to serve inference tasks (e.g., CPU, GPU, DSP and Edge TPU are available on Pixel 4). To be executed on specific hardware, the model is "compiled" to select an optimized CPU implementation or a platform-specific GPU kernel for each operation of the computational graph. Notably, the same operation can be executed using different algorithms on different devices; for example, the TFLite GPU Delegate can select different kernels for convolution operations on Adreno GPUs vs. Mali GPUs (Section 3.2.2). In addition, the computational graph can be optimized during model compilation; for instance, two consecutive operations can be "fused" and implemented using a single GPU kernel (Section 3.2.1). Eventually, a compiled model is executed on the target hardware: on GPUs, kernels are dispatched to a command queue for execution; on CPUs, operations are executed sequentially, while multithreading is used only to accelerate the execution of individual operations using multiple cores (Section 3.1.1).

3 INFERENCE ON MOBILE DEVICES

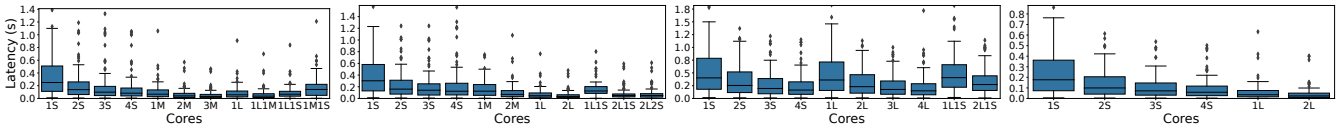
In this section, we present the results of our empirical study on the performance of real-world neural architectures on mobile platforms; in particular, we analyze thread scheduling and model quantization in multicore mobile CPUs (Section 3.1), and kernel fusion and selection in mobile GPUs (Section 3.2), evaluating their impact on inference latency. The insight gained from our results will be used in Section 4 to develop a latency prediction framework.

3.1 Mobile CPUs

3.1.1 Effects of Multithreading. Modern mobile platforms typically adopt the ARM big.LITTLE architecture, which allows multiple types of CPU cores to be integrated into the same system; each group of homogeneous cores is operated as a "core cluster" running at the same clock speed. The "big cores" with higher clock speeds can handle computationally intensive tasks, while the "LITTLE cores" with lower clock speeds require less power. High-priority tasks are usually scheduled on big cores for better performance; non-urgent tasks are assigned to little cores to reduce energy consumption. Table 1 lists the core clusters of the SoCs in our study.[2]

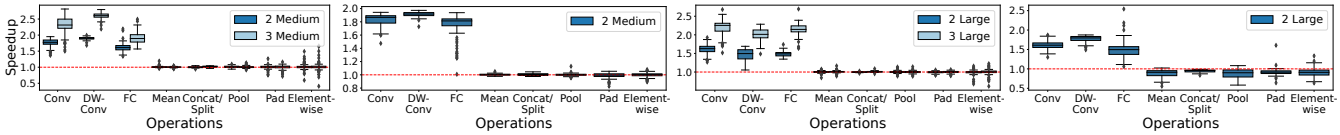
An inference task can be accelerated with multithreading over multiple cores. For Android devices, given a set of CPU cores, we use an equal number of threads and set CPU affinity of these threads to encourage their scheduling on the given set of cores; for iOS devices, we set the *quality-of-service (QoS) class* [3] of each thread to schedule on performance or efficiency cores. Considering the

[2]Since the architectures of Snapdragon 710 and A10 are similar to Snapdragon 855 and A12, respectively, we report their measurements in [41].



(a) Snapdragon 855 **(b) Exynos 9820** **(c) Helio P35** **(d) A12 Bionic**

Figure 3: Effects of Multicore on End-to-end Latency (L, M, S represent Large, Medium, Small cores, respectively)



(a) Snapdragon 855 **(b) Exynos 9820** **(c) Helio P35** **(d) A12 Bionic**

Figure 4: Effects of Homogeneous Multicore on Operation-wise Latency (Speedup over One Core)

limited resources available on mobile devices, we select 102 real-world NAs[3] with up to 18 million parameters from 25 articles (with manual design or NAS) [7, 10, 11, 25–33, 39, 50–53, 55, 56, 59, 60, 62, 64, 66, 67]. Fig. 2 illustrates the combinations of model sizes and end-to-end latencies (on Adreno 640) of these real-world NAs.

Fig. 3 uses boxplots[4] to depict end-to-end latency of these real-world neural architectures with different multicore configurations.[5] Counterintuitively, *using multiple heterogeneous cores can result in performance degradation*: for example, on Snapdragon 855 (Fig. 3a), the combination of a medium core and a small core results in worse performance (on average) than a medium core. As noted in previous work [61], we attribute this performance degradation to the overhead of inter-cluster communication; after inspecting the source code of TFLite [21] and of its matrix multiplication library Ruy [20], we also observe that the work of an operation is split *equally* among threads, which is suboptimal for heterogeneous cores.[6]

In Fig. 3, we observe a sublinear end-to-end speedup with respect to the number of cores for multithreading with homogeneous cores. Fig. 4 shows the speedup of different operation types with respect to the number of homogeneous cores. We observe that convolution, depthwise convolution (DW-Conv) and fully-connected (FC) operations achieve sublinear speedups as the number of threads increases. However, performance improvements on the remaining operations are negligible, due to the lack of support for parallel execution of these operations in the current TFLite implementation.

Insight 1. On mobile CPUs, multithreading has a significant impact on the performance of inference tasks. On homogeneous cores, multithreading leads to *sublinear* reduction of latency for convolution, depthwise convolution and fully-connected operations in TFLite; however, on heterogeneous cores, multithreading can result in *performance degradation* due to the overhead of inter-cluster communication.

[3]The TensorFlow implementations of these NAs are from [1], which also provides pre-trained parameters and Top-1/Top-5 test errors on the ImageNet-1K dataset.
[4]In the paper, boxplots indicate 1st quartile, median, and 3rd quartile of the data; whiskers extend for 1.5x the interquartile range.
[5]For clarity of presentation, we omit some outliers with substantially higher latency in Fig. 3 (<4% of the data per configuration) and report the complete data in [41].
[6]The work in [61] also proposes solutions to improve the throughput over heterogeneous cores. In our paper, we focus on the performance characteristics of ML workloads and use the current implementation of TFLite.

3.1.2 Effects of Quantization. On mobile devices with limited power and computing resources, neural architectures can be converted into lower-precision representations (e.g., 16-bit floating-point or 8-bit integers) to reduce memory utilization and computational demand, without substantial loss in accuracy. We focus on the approach of integer-arithmetic-only inference [34] available in TFLite, where both weights and activations are represented as 8-bit integers during inference.[7] Fig. 5 compares end-to-end inference latency using an 8-bit integer representation and a floating-point representation.[8] As can be seen, quantization shows a distinct speedup on various combinations of cores on all devices.

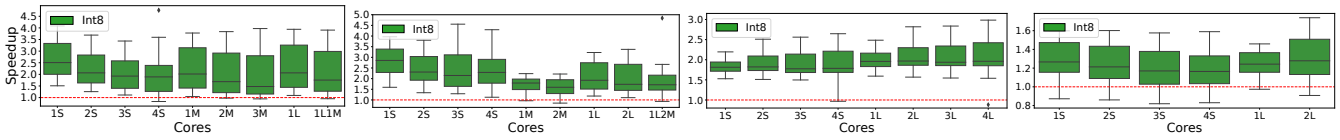
Fig. 6 depicts the latency improvement of each type of operation after quantization. On all devices, most operations achieve significant speedup when using 8-bit integers; however, padding and element-wise operations show *performance degradation* after quantization: the average latency of element-wise operations is increased by 2.55x and 2.60x on Snapdragon 855 and Exynos 9820, respectively. Previous work [34, 46] suggests that this degradation is due to the overhead of matching quantization ranges (i.e., the scale) of all inputs of quantized operations (e.g., element-wise addition).

Insight 2. Quantization can reduce latency and memory utilization of a model, significantly improving the performance of inference tasks. However, quantization can cause *performance degradation* for some operations due to the cost of scaling inputs.

3.2 Mobile GPUs

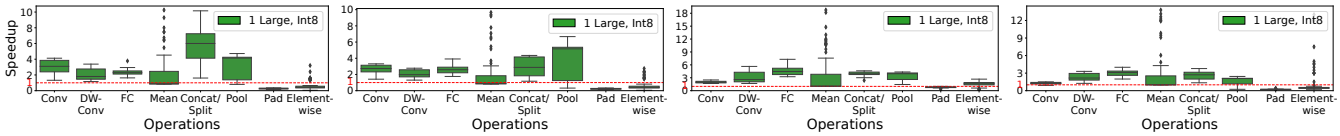
3.2.1 Effects of Kernel Fusion. Kernel fusion has been broadly adopted to reduce the overhead of dispatching kernels [68]. We analyzed the implementation of kernel fusion available in TFLite [18], which we report in Algorithm 3.1. Two consecutive operations of the computational graph are fused when the first operation has only one output tensor (Line 5) and the second operation: (1) is the only operation in the graph using this output tensor (Line 14),

[7]We study the effects of integer quantization only on mobile CPUs, because using 8-bit integers can cause significant overhead in the current implementation of the TFLite GPU delegate when extra GPU kernels for quantization and dequantization are invoked.
[8]Similarly to Fig. 3, we omit outliers (only of a couple of points) for better visualization and report complete data in [41].



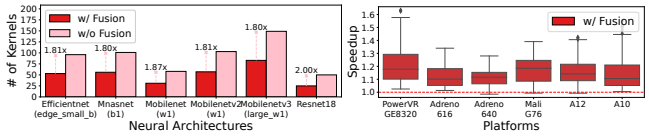
(a) Snapdragon 855 (b) Exynos 9820 (c) Helio P35 (d) A12 Bionic

Figure 5: Effects of Quantization on End-to-end Latency



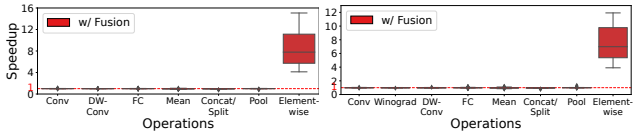
(a) Snapdragon 855 (b) Exynos 9820 (c) Helio P35 (d) A12 Bionic

Figure 6: Effects of Quantization on Operation-wise Latency



(a) Number of Kernels (b) End-to-end Latency

Figure 7: Effects of Kernel Fusion



(a) Snapdragon 855 (Adreno 640) (b) A12 Bionic

Figure 8: Effects of Kernel Fusion on Operation-wise Latency

(2) uses this output tensor as its first input to produce a single output (Line 22), and (3) has a compatible type (Line 24).

Fig. 7a illustrates that kernel fusion leads to a reduction in the number of OpenCL/Metal kernels of over 45% for real-world NAs. Fig. 7b shows the performance improvements from kernel fusion on different mobile devices.[9] We observe up to 1.22x speedup of the average end-to-end latency over all the neural architectures, due to a reduction in the cost of kernel dispatching. As shown in Fig. 8,[10] kernel fusion can significantly reduce the latency of element-wise operations by merging multiple kernels; at the same time, there is no substantial latency increase for other operations. This observation is in line in Algorithm 3.1: the operations fused into other operations are mainly element-wise operations (Line 24).

Insight 3. By substantially reducing the number of operation kernels, kernel fusion can improve the performance of inference tasks on mobile GPUs. However, only *element-wise operations* obtain substantial performance improvements; the effect on other operations is negligible.

[9]The outliers (only a couple of data points) are removed to improve visualization.
[10]A few outliers with large speedups on element-wise operations are reported in [41].

Algorithm 3.1: Kernel Fusion in TFLite GPU Delegate

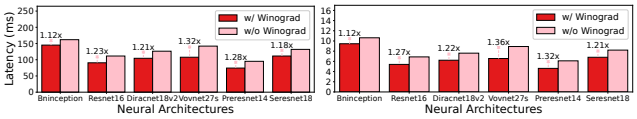
```
MergeNodes(nodes)
1  ready_tensors = []
2  for cur_node in nodes
3      for dst_tensor in cur_node.dst_tensors
4          ready_tensors.insert(dst_tensor)
5      if cur_node.dst_tensors.size() ≠ 1
6          continue
7      candidate_nodes = []
8      candidate_tensor_index = 0
9      for next_node in nodes
10         for k = 0 to next_node.src_tensors.size() - 1
11             if next_node.src_tensors[k] == cur_node.dst_tensors[0]
12                 candidate_tensor_index = k
13                 candidate_nodes.insert(next_node)
14     if candidate_nodes.size() ≠ 1 or candidate_tensor_index ≠ 0
15         continue
16     next_node = candidate_nodes[0]
17     if next_node.src_tensors[0] ∈ ready_tensors
18             and IsLinkable(next_node)
19         Merge(cur_node, next_node)
20         nodes.remove(cur_node)
21 return nodes

IsLinkable(node)
22 if node.output_tensors.size() ≠ 1
23     return False
24 if node.type ∈ [ACTIVATION, COPY, ADD, SUB, MUL, DIV, EXP,
   LOG, SQRT, SQUARE, ABS, NEG, POW, EQUAL, GREATER, LESS,
   MAXIMUM, MINIMUM]
25     return True
26 return False
```

3.2.2 Effects of Kernel Selection. Machine learning frameworks use different optimized implementations for the operations of neural architectures. Algorithm 3.2 summarizes the criteria used by TFLite to select the Winograd algorithm for convolution operations: when the input tensor and kernel size of a convolution operation satisfy

Index	Configurations			Conditions in Algorithm 3.2			Choosing Winograd Kernels	
	Input channels	Output channels	Output height	src_depth	dst_depth	total_tiles	Adreno	Mali
(1)	64	64	56	16	16	196	No	Yes
(2)	128	128	28	32	32	49	No	Yes
(3)	256	256	14	64	64	16	No	No

Table 2: Applicability of Winograd Kernels to Convolutions in ResNet16 (1 group, 3x3 kernel, stride 1)



(a) Helio P35 (PowerVR GE8320) **(b) A12 Bionic**

Figure 9: Effects of Winograd Kernels on End-to-end Latency

the criteria defined by the CheckWinograd function, a Winograd kernel will be selected. Fig. 9 shows the performance improvement from using Winograd kernels in real-world NAs; we observe performance improvements of up to 1.32x for PowerVR GE8320 and up to 1.36x for A12 Bionic.

Notably, kernel selection is hardware-dependent: none of the NAs obtain performance improvements on Adreno 640 or 616, because the requirements for applying the Winograd algorithm on these GPUs are stricter than Mali and PowerVR GPUs in the current TFLite implementation. For example, Table 2 presents the kernels selected for three convolution operations of ResNet16; all the operations have only one convolution group, kernel size 3x3, and stride 1. For convolution (1), src_depth and dst_depth fail to satisfy the conditions for Adreno GPUs (Line 17 of Algorithm 3.2), but meet the requirements for Mali and PowerVR GPUs (Line 21). For convolution (2), total_tiles is too small for Adreno 6xx GPUs (Line 24), but large enough for Mali and PowerVR GPUs (Line 28). Convolution (3) cannot be implemented using the Winograd algorithm in either GPU because of the small total_tiles (Line 28).

Insight 4. Framework-dependent optimizations have significant impact on the performance of inference tasks. In TFLite, convolution operations with certain shapes of input tensors and kernel sizes can use the Winograd algorithm to accelerate the execution. Therefore, an accurate performance prediction model needs to accurately capture which kernels are executed during inference.

Algorithm 3.2: Conv Kernel Selection in TFLite GPU Delegate

```
SelectConv2DKernel(gpu_info, op_info)
 1  if CheckGroupedConv2D(gpu_info, op_info)
 2      return Kernel(GroupedConv2D, gpu_info, op_info)
 3  else if CheckWinograd(gpu_info, op_info)
 4      return Kernel(Winograd, gpu_info, op_info)
 5  else return Kernel(Conv2D, gpu_info, op_info)

CheckGroupedConv2D(gpu_info, op_info)
 6  src_group_size = op_info.input_channel
 7  dst_group_size = op_info.output_channel / op_info.group
 8  if op_info.group ≠ 1 and src_group_size % 4 == 0
 9          and dst_group_size % 4 == 0
10      return True
11  return False

CheckWinograd(gpu_info, op_info)
12  if op_info.group ≠ 1 or op_info.kernel_shape ≠ 3x3
13          or op_info.stride ≠ 1
14      return False
15  src_depth = ⌈op_info.input_channel/4⌉
16  dst_depth = ⌈op_info.output_channel/4⌉
17  if gpu_info.type == Adreno and (src_depth < 32 or dst_depth < 32)
18      return False
19  else if gpu_info.type == AMD and (src_depth < 16 or dst_depth < 8)
20      return False
21  else if src_depth < 16 or dst_depth < 16
22      return False
23  total_tiles = ⌈op_info.output_height/4⌉ * ⌈op_info.output_width/4⌉
24  if gpu_info.type == Adreno6xx and total_tiles < 128
25      return False
26  else if gpu_info.type == Adreno and total_tiles < 64
27      return False
28  else if total_tiles < 32
29      return False
30  return True
```

4 METHODOLOGY

Given an input model file (e.g., a `.tflite` file) generated on a cloud server (e.g., during NAS), we aim at accurately predicting its end-to-end latency on different mobile CPUs and GPUs without deploying it to actual devices. Our approach includes the following steps: (1) from the model file, we first extract information from the operations (i.e., the execution units on mobile CPUs) of the computational graph; (2) for mobile GPUs, we deduce (without deploying to the mobile device) the actual kernels executed after kernel fusion and kernel selection (Section 4.1); (3) we use ML models to predict inference latency of each operation from its parameters (e.g., input shape and number of channels; Section 4.2); (4) end-to-end latency is estimated as the sum of predicted operation latencies plus the additional latency due to ML framework overhead. To train the ML models and to evaluate our approach, we collect latency measurements on a synthetic dataset including 1000 neural architectures from a NAS space (Section 4.3), which we plan to make publicly available to help further research on mobile inference performance.

4.1 Kernel Deduction

From the model file, we are able to extract the configurations of all operations of the computational graph. As discussed in Section 3.1.1,

these operations are executed sequentially on mobile CPUs; multiple threads collaborate only on the execution of each operation. For each type of operation, platform, and CPU core combination, we train a machine learning model to predict inference latency.

However, when using mobile GPUs, operations of the computational graph can be fused (Section 3.2.1) or implemented with optimized algorithms (Section 3.2.2), which have substantial effects on performance (as illustrated by our measurements); consequently, identifying which kernels are actually executed on the target device is critical to obtaining accurate latency predictions. To avoid the cost of deploying the neural architectures to physical devices (which is impractical for NAS, given the huge number of candidate NAs), we deduce the kernels executed on a device by simulating the process of kernel fusion and kernel selection, according to the principles elicited from the implementation of TFLite. Specifically, to predict latency on mobile GPUs, we first fuse kernels according to the rules in Algorithm 3.1; then, we use the rules in Algorithm 3.2 to select a kernel among {Conv2D, Winograd, GroupedConv2D} based on the parameters of each convolution operation (e.g., input size, output size, kernel size) and on the specific target device.

4.2 Prediction Models

To predict the latency of an operation, we use features associated with both memory access cost (e.g., size of input, output and parameters) and computational cost (e.g., FLOPs). Table 3 reports the features affecting latency of each operation type.

Formally, for each operation, given the feature vectors $\boldsymbol{x_i} \in X$ and latencies $y_i \in Y$ measured on a specific device, $i = 1 \ldots, N$ (where N is the size of the training dataset of the operation), we train a prediction model $f^* = \arg\min_f \frac{1}{N} \sum_{i=1}^{N} |(f(\hat{\boldsymbol{x}}_i) - y_i)/y_i|^2$ where each feature $x_{i,j}$ is standardized as $\hat{x}_{i,j} = (x_{i,j} - \mu_j)/\sigma_j$ based on its training set mean $\mu_j = (\sum_{i=1}^{N} x_{i,j})/N$ and standard deviation $\sigma_j = \sqrt{\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2/N}$. Note that we minimize the mean squared *percentage* error; during testing, we evaluate the mean absolute percentage error (MAPE) $\frac{1}{N} \sum_{i=1}^{N} |(f^*(\hat{\boldsymbol{x}}_i) - y_i)/y_i|$. For prediction model, we consider the following representative ML approaches [45] adopted in the literature [5, 6, 17, 35, 68].

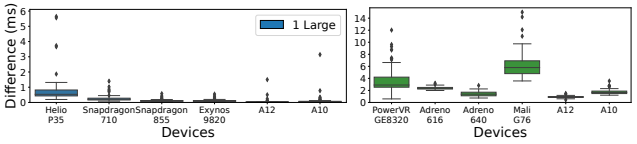
Lasso. We first consider a linear model $f(\boldsymbol{x}) = \boldsymbol{w}^T \boldsymbol{x}$ and estimate the optimal weights $\boldsymbol{w}^*$ as

$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \frac{1}{N} \sum_{i=1}^{N} \left| \frac{\boldsymbol{w}^T \hat{\boldsymbol{x}}_i - y_i}{y_i} \right|^2 + \alpha \|\boldsymbol{w}\|_1 \quad \text{s.t.} \quad \boldsymbol{w} \geq 0\,. \tag{1}$$

An L1 regularization term with hyperparameter α is included to control model complexity and to favor a sparse solution. We use grid search in $[10^{-5}, 10^2]$ to find the best α. Since each input feature $\hat{x}_{i,j}$ is positively correlated with latency, we constrain weights w_j to be nonnegative in Eq. (1).

Random Forests (RF). An RF model includes multiple decision trees to reduce the overfitting of a single decision tree. We tune hyperparameters including the number of decision trees (1 to 10) and the minimum number of samples to split an internal node (2 to 50) using 5-fold cross-validation.

Gradient-Boosted Decision Trees (GBDT). GBDT generates decision trees with gradient boosting on multiple stages. We tune hyperparameters including the number of gradient boosting stages (1 to 200) and the number of examples required to split a node (2 to 7) using 5-fold cross-validation.



(a) CPUs **(b) GPUs**

Figure 10: Difference between End-to-end Latency and Sum of Operation-wise Latency for Real-World NAs

Multi-Layer Perceptron (MLP). An MLP consists of multiple layers of fully-connected neurons. We tune the hyperparameters for the number of layers from 1 to 6 and select the number of neurons in each layer from {64, 128, 256, 512}. Similarly to previous work [17], we use ReLU activations after each layer and the Adam optimizer with learning rate selected from $\{5 \times 10^{-3}, 5 \times 10^{-4}, 5 \times 10^{-5}\}$, and weight decay selected from $\{10^{-3}, 10^{-4}, 10^{-5}\}$. We use 20% of training data as the validation set, and stop training when there is no improvement in the validation error over 50 epochs.

To obtain end-to-end latency predictions, we add up latencies predicted for all operations of the NA, since CPU operations and GPU kernels are executed sequentially by TFLite (in a topological order determined by their dependencies). We also account for the additional latency due to overhead and data transfers in TFLite; as shown in Fig. 10, the sum of the latencies measured for all operations is consistently lower than the measured end-to-end latency, especially on GPUs. Since the difference fluctuates around a constant value for all NAs on a specific GPU, we use the average difference between end-to-end latencies and the total operation-wise latencies in the training dataset to estimate this additional latency T_{overhead}. Formally, for a neural architecture with set of operations C, we predict end-to-end latency as $T_{\text{overhead}} + \sum_{c \in C} f_c^*(\hat{\boldsymbol{x}}_c)$, where f_c^* is the latency predictor trained from measurements of operations with the same type as c, and T_{overhead} is the estimated overhead.

4.3 Synthetic Dataset

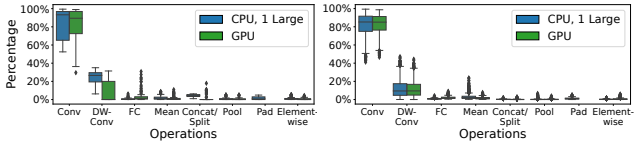
Next, we present our synthetic dataset of NAs sampled from a NAS space including operations and building blocks proposed in recent works. We first introduce the approach to collect latency measurements, and then describe the design of the NAS space.

4.3.1 Kernel Latency Profiling. We use the *TFLite Model Benchmark Tool* [23] to benchmark the performance of neural architectures. Since the tool supports latency measurements of elementary operations only on mobile CPUs, we record start/stop timestamps of GPU kernels by collecting profiling information at the OpenCL command queue (on Android) or Metal command buffer (on iOS), respectively. To reduce the overhead of timestamp recording, we dispatch the same kernel 256 times.[11] For iOS devices, we set the GPU Performance State [4] to *high* to acquire stable measurements over time. We also adopt the default precision settings of the TFLite Model Benchmark tool, which uses 32-bit floating-point variables on mobile CPUs and 16-bit floating-point variables on mobile GPUs.

[11] Here, we follow the TFLite implementation [22], which allows no more than 256 dispatches for Mali GPUs; we found that using a lower number of dispatches does not sufficiently reduce the overhead of timestamp recording.

Operation / Kernel	Features
Conv2D, Winograd, DepthwiseConv2D	Input height (width), input channel, output height (width), stride, kernel height (width), filters, input size, output size, kernel size, FLOPs
GroupedConv2D	Input height (width), input channel, output height (width), stride, kernel height (width), filters, input size, output size, kernel size, group number, FLOPs
FullyConnected	Input channel, filters, parameter size, FLOPs
Mean	Input height (width), input channel, kernel height (width), input size, FLOPs
Concat, Split	Input height (width), input channel, kernel height (width), output channel, input size, output size
Pooling	Input height (width), input channel, output height (width), stride, kernel height (width), input size, output size, FLOPs
Padding	Input height (width), input channel, output height (width), padding size, output size
Element-wise	Input height (width), input channel, input size

Table 3: Feature Space for Each Category of Operations



(a) Real-world NAs **(b) Synthetic NAs**

Figure 11: Latency Breakdown on Snapdragon 855

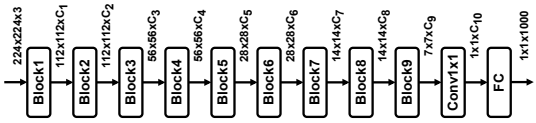


Figure 12: Design of the NAS Space for Synthetic Dataset

Fig. 11a shows the average latency breakdown for 102 real-world neural architectures[12] on Snapdragon 855. Notably, convolution and depthwise convolution operations account for most of the end-to-end latency. We observe that, for the same NAs, Winograd kernels are selected on Mali G76 but not on Adreno 640; as discussed in Section 3.2.2, kernel selection is different on each platform.

4.3.2 NAS Space for Sampling Neural Architectures. Fig. 11a highlights the importance of convolution and depthwise convolution operations in end-to-end latency. Consequently, we design a search space to effectively sample various configurations of operations for the purpose of understanding their performance characteristics. As illustrated in Fig. 12, synthetic neural architectures of our NAS space use a sequence of 9 blocks with fixed input height and width, following the design of sequential connections of blocks in MobileNetV2 [52].[13] The type and parameters of each building block are sampled uniformly at random among:

(1) A convolution layer (with kernel size 3x3, 5x5 or 7x7, and optional group size $4k$, with $1 \leq k \leq 16$).
(2) Depthwise separable convolution [29] (with kernel size 3x3, 5x5 or 7x7).
(3) Linear bottleneck [52] (with kernel size 3x3, 5x5 or 7x7, expansion rate 1, 3 or 6, optionally using Squeeze-and-Excite [28]).
(4) Average or max pooling layer (with window size 1x1 or 3x3).
(5) A split layer (with 2, 3 or 4 splits), followed by element-wise operations performed on each output tensor, and a concatenation layer that merges all output tensors.

Due to the limited memory and computing resources on mobile devices, we uniformly sample the output channel sizes of these building blocks (identified as C_1 to C_9) with the following constraints: $\{C_1, ...C_5\} \in [8, 80]$, $\{C_6, ...C_9\} \in [80, 400]$, and $C_{10} \in [1200, 1800]$.

We built a synthetic dataset including *1000 neural architectures* sampled from this NAS space. For each neural architecture, we collected training measurements on *6 mobile platforms* (Table 1), for a total of *90 scenarios*, covering (1) combinations of homogeneous or heterogeneous cores, (2) floating-point and 8-bit integer representations, and (3) mobile GPUs from different manufacturers. Fig. 11b illustrates the latency breakdown for neural architectures in our synthetic dataset, where the latency distribution over operations is similar to real-world neural architectures.

5 RESULTS

This section presents a comprehensive evaluation of our latency prediction framework across a broad range of scenarios: first, we present results on the default setting of NAS (Section 5.1), and then we evaluate the impact of hardware heterogeneity (Section 5.2), neural architecture diversity (Section 5.3), and ML framework optimizations (Section 5.4). In addition, to address concerns regarding the cost of training data collection, we present results using a small number of training examples (Section 5.5). Lastly, we quantitatively compare both our predictions and the design of the synthetic dataset with existing literature (Section 5.6).

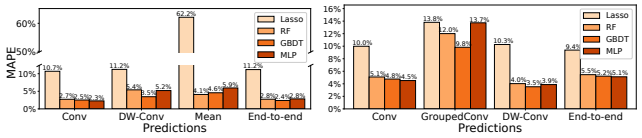
5.1 Default Setting: NAS Space

We first test our framework in the common scenario of predicting inference latency during NAS: we sample test data (the candidate architectures during NAS) and training data (the profiling architectures used to train our latency prediction model) uniformly at random from the same search space (Section 4.3.2). These sampled NAs constitute our synthetic dataset of 1000 samples; in this section, we use 900 of these for training and 100 for testing.

Fig. 13 presents the average MAPE across 6 platforms[14] under different ML approaches when predicting end-to-end latency, as well as the latency of the 3 operation types accounting for most of end-to-end latency. Based on the latency breakdown of synthetic neural architectures on CPUs and GPUs (Fig. 11b), convolution operations

[12] When presenting the percentage of end-to-end latency, we include the results of NAs that may not have all types of operations, e.g., depthwise convolution operations only appear in 44 NAs, so its median across 102 NAs is zero.
[13] In Section 5.3, we also evaluate our predictions on real-world NAs that consist of non-linearly connected building blocks.

[14] Due to lack of space, MAPE of each platform is reported in [41].



(a) CPU (One Large Core) **(b) GPU**

Figure 13: Predictions of ML Models (Synthetic NAs)

typically account for the largest proportion of end-to-end latency; consequently, the prediction error of convolution dominates the error of end-to-end latency prediction for all ML approaches on both CPUs and GPUs. For example, Lasso has a large MAPE (62.2%) for "mean" operations on CPU (Fig. 13a), while its MAPE for end-to-end latency is only 10.0%; that is because, as shown in Fig. 11b, for 75% of synthetic neural architectures, mean operations contribute to less than 3.6% of the CPU end-to-end latency.

As shown in Fig. 13, in our default setting, all nonlinear ML approaches (RF, GBDT, MLP) achieve comparable accuracy on end-to-end latency predictions, with average MAPE across six platforms below 2.8% for CPU predictions and below 5.5% for GPU predictions; Lasso achieves less accurate predictions (11.2% on CPUs and 9.4% on GPUs) because its linear model cannot represent non-linear relationships between latency and operation features, as identified by previous work [57, 68].

5.2 Case Study: Hardware Heterogeneity

Next, we evaluate our prediction framework under hardware heterogeneity, including scenarios with different CPU core combinations and with both floating-point and integer representations. We select GBDT as a representative ML approach in this section, since it shows comparable or slightly better predictions than RF and MLP in the case of a large CPU core (Fig. 13a).

Fig. 14 illustrates GBDT predictions of end-to-end latency over various core configurations.[15] We observe that more homogeneous cores typically lead to higher prediction errors. Using more cores can result in larger measurement variance, due to background jobs running on mobile devices (e.g., cameras, sensors, and networking services); measurement variance can impair the quality of profiling data and thus affect prediction accuracy.[16] For example, from the results on Exynos 9820 shown in Fig. 14b, the MAPE on 4 small cores (10.3% for floating-point and 10.5% for integer quantization) is higher than the MAPE on 1 small core (8.6% and 4.9%, respectively), due to the substantial interference of background jobs when an inference task attempts to make use of all the efficient cores on the device; in these situations, latency measurements have larger coefficient of variation, as shown in Fig. 15. Overall, GBDT achieves accurate predictions across all platforms: the worst MAPEs for homogeneous cores are 10.5% on Exynos 9820, 5.8% on Snapdragon 855, 6.0% on Helio P35, and 5.8% on A12 Bionic.

Note that using heterogeneous cores results in even higher variability of latency measurements due to inter-cluster communication [61]. In addition, as explained in Section 3.1.1, operations without multithreading implementations can be scheduled on arbitrary cores, complicating prediction; for example, when using 1 large and 1 medium core on Snapdragon 855, MAPEs (3.9% for floating-point and 5.5% for integer quantization) are higher with respect to using 2 medium cores (3.2% and 3.9%, respectively).

Fig. 16 presents predictions of GBDT for different GPUs. For convolution operations, we split the results of Conv2D and Winograd kernels in Fig. 16a because separate latency predictors are trained for each kernel; no Winograd kernel is used on Adreno 640 and 616 due to the rules of kernel selection presented in Section 3.2.2. Overall, GBDT achieves good end-to-end prediction across all six GPUs, with worst MAPE of 8.2% on Exynos 9820.

5.3 Case Study: Neural Architecture Diversity

Next, we evaluate our framework on diverse neural architectures: we consider a scenario where training data include candidates sampled *at the early stages of NAS*, while test data are highly accurate neural architectures generated *at the end of NAS*. In our evaluation, we use 1000 synthetic neural architectures as training data and 102 real-world neural architectures (from existing literature) as test data. The two sets of neural architectures have *different distributions* (i.e., we introduce a dataset shift): we observe that the latency of convolution operations in real-world neural architectures is generally lower than in synthetic neural architectures. Fig. 17a shows the percentage of end-to-end latency attributed to convolution operations (split by range) on Helio P35 (with a large core): convolutions greater than 50 ms dominate end-to-end latency in our synthetic neural architectures, while faster convolutions contribute more to real-world neural architectures.

Fig. 18a shows the average MAPE across six devices for the real-world neural architectures on CPUs. For most ML approaches trained on synthetic neural architectures, prediction errors are higher for real-world neural architectures than synthetic neural architectures (Fig. 13) that are generated from the same distribution as the training data. The only exception is Lasso, which achieves better predictions on real-world neural architectures, with end-to-end MAPE on CPUs (5.4%), similar to RF and GBDT. We attribute this anomaly to the better accuracy of Lasso predictions on fast operations (< 50 ms) due to higher weights assigned to these operations in Eq. (1), which we observe in both synthetic and real-world architectures (Fig. 17b). Since real-world architectures include a larger proportion of fast operations (in this specific dataset shift), average accuracy of Lasso is better on real-world architectures than synthetic neural architectures.

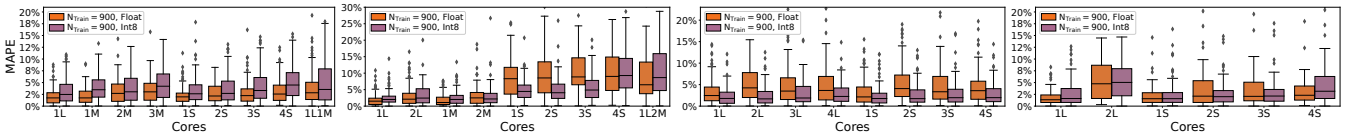
Fig. 18b presents predictions on mobile GPUs. We observe that, for some small real-world neural architectures, the overhead of TFLite is significant. Since the overhead has high runtime variability (in particular, on PowerVR GE8320 and Mali G76), it can affect the accuracy of end-to-end latency predictions, especially for neural architectures with low latency such as MobileNets.

5.4 Case Study: ML Framework Optimizations

Next, we illustrate the improvements of GPU predictions from accounting for ML framework optimizations such as kernel fusion and kernel selection.

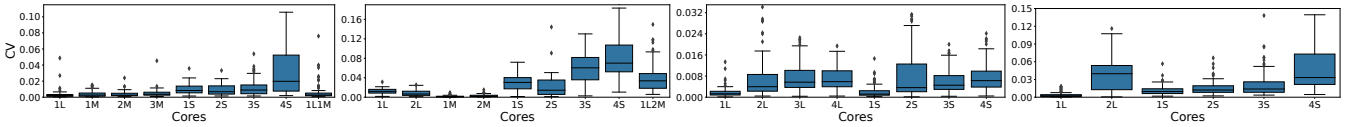
[15] For clarity of presentation, we omit some outliers (<9% data points for 1 large and 2 medium cores of Exynos 9820, and <4% data points for all other configurations), and report plots with all data points in [41].

[16] In our approach, we do not explicitly model background jobs; in practice, they depend on user activities and so are different at runtime.



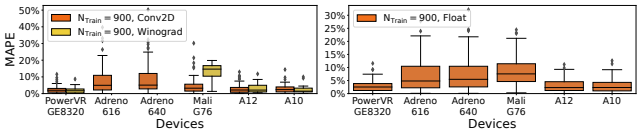
(a) Snapdragon 855 (b) Exynos 9820 (c) Helio P35 (d) A12 Bionic

Figure 14: Predictions of GBDT on Multicore CPUs (Synthetic NAs)



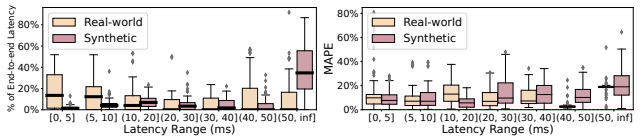
(a) Snapdragon 855 (b) Exynos 9820 (c) Helio P35 (d) A12 Bionic

Figure 15: Coefficient of Variation for Latency Measurements of Synthetic NAs on CPUs



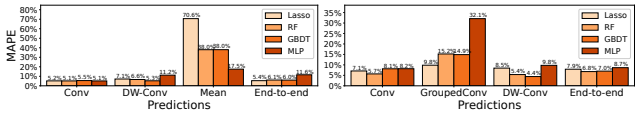
(a) Convolution (b) End-to-end

Figure 16: Predictions of GBDT on GPUs (Synthetic NAs)



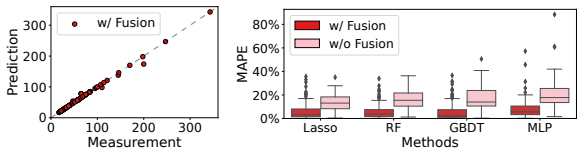
(a) Contribution to End-to-end (b) Predictions using Lasso

Figure 17: Convolution Operations with Different Latency Ranges (Helio P35 1 Large CPU Core)



(a) CPU (One Large Core) (b) GPU

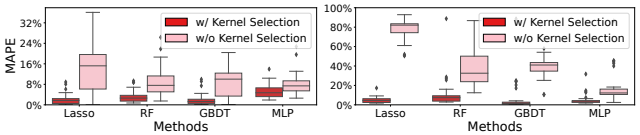
Figure 18: Predictions of ML Models (Real-World NAs)



(a) # of Kernels (b) MAPE on PowerVR GE8320

Figure 19: Effectiveness of Kernel Fusion on Predictions

Kernel Fusion. In Section 3.2.1, we show that kernel fusion considerably reduces the number of kernels and leads to improvements in end-to-end latency. Fig. 19a shows that, after following Algorithm 3.1 to estimate which kernels will be fused by TFLite (Section 3.2.1), we obtain a number of kernels close to actual measurements collected on 102 real-world neural architectures. Fig. 19b



(a) End-to-end Latency (b) Winograd Kernel

Figure 20: Prediction Error Reduction on PowerVR GE8320 by Accounting for Kernel Selection

illustrates that we obtain substantial error reduction in end-to-end latency prediction with respect to ML models which do not consider kernel fusion (labeled as "w/o Fusion").

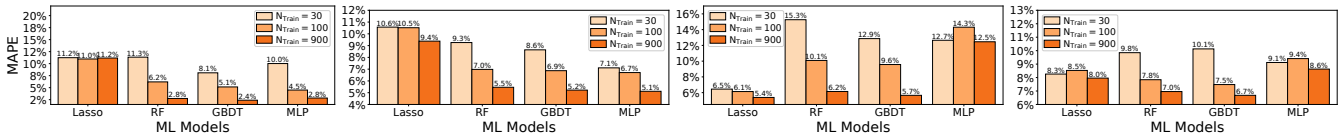
Kernel Selection. As introduced in Section 3.2.2, a convolution operation can be executed by TFLite using different kernel implementations compatible with the target device and convolution parameters. We deduce the actual kernels selected by TFLite for convolution operations (specifically, Conv2D and Winograd) and train separate predictors for each (since they exhibit different performance characteristics). Fig. 20a shows the considerable error reduction achieved by accounting for kernel selection on PowerVR GE8320, for real-world neural architectures that support Winograd kernels; Fig. 20b confirms that this reduction is due to more accurate predictions of the latency of Winograd kernels.

5.5 Case Study: Limited Training Data

The high cost of collecting sufficient training data is a common criticism of ML approaches to predict the latency of neural architectures during NAS [42]. In this section, we study the effects of training set size on different ML approaches, illustrating the benefits of a simple model when training data is limited.

5.5.1 Comparison of ML Approaches. Fig. 21 show prediction errors of different ML approaches for varying training set sizes N_{Train}, on synthetic neural architectures (presented in Section 5.1) and real-world neural architectures (presented in Section 5.3), respectively (errors are average MAPE across 6 platforms).[17] Predictions of Lasso are less sensitive to the size of training data, while other more complex approaches achieve higher errors when the training set size is

[17] MAPEs for each platform are reported in [41].



(a) CPU, Synthetic NAs **(b) GPU, Synthetic NAs** **(c) CPU, Real-World NAs** **(d) GPU, Real-World NAs**

Figure 21: Prediction Errors on Synthetic or Real-World NAs for Different Synthetic Training Set Sizes

decreased from 900 to 30. Consequently, when training data is limited, e.g., in Figs. 21a and 21b for training size of 30, a simple model such as Lasso achieves similar or better accuracy than complex models; Lasso is also more robust when test and training datasets have different distributions, even for large amounts of training data, e.g., when training on synthetic NAs but testing on real-world NAs in Figs. 21c and 21d. Complex models (GBDT, RF and MLP) are similarly accurate when sufficient training data is available and there is no dataset shift (i.e., training and testing datasets have similar distributions), e.g., in Figs. 21a and 21b for training set size of 900. In the case of data shift (Figs. 21c and 21d), MLP achieves the worst predictions with a training set of size 100. This is due to severe prediction errors on concatenation/split operations: on Pixel 4 (one large CPU core), MAPEs on concatenation/split operations are 56.7%, 1400.4% and 1068.7%, after training on 30, 100 and 900 neural architectures, respectively. This anomaly is due to the very small amount of training data (only 5, 25 and 312 concatenation/split operations from training data of 30, 100 and 900 neural architectures, respectively). Instead, for convolution operations with sufficient data, MLP prediction errors are 7.8%, 5.1% and 4.6% for training sets of size 30, 100 and 900, respectively, on the same platform.

Notably, for real-world neural architectures, using only 30 training examples, Lasso considerably outperforms other ML approaches on CPUs with a large core (Fig. 21c), with an average MAPE of 6.5% across six platforms. As pointed out by prior work [42], the cost of profiling only 30 neural architectures on each target device is negligible compared to measuring latencies of all candidate neural architectures (e.g., thousands) during NAS.

5.5.2 Lasso Predictions with Limited Training Data. Next, we thoroughly evaluate the predictions of Lasso with a limited training set size (i.e., 30 neural architectures) on real-world neural architectures, across a broad range of scenarios for hardware heterogeneity.

Fig. 22 shows the prediction error of Lasso on real-world neural architectures, across various combinations of cores and data representations.[18] Generally, the trend of prediction errors for homogeneous and heterogeneous clusters is similar to the results in Fig. 14. The maximum MAPE for combinations of homogeneous cores is 22.9% on Exynos 9820, 13.5% on Snapdragon 855, 9.6% on Helio P35, and 9.5% on A12 Bionic. We attribute the large prediction errors on Exynos 9820 to the noise of measurements collected with many small cores, which is due to background tasks and can affect the quality of training data for this limited dataset. By adding more training data, MAPEs can be reduced to less than 14.8%.

Fig. 23 shows the predictions of Lasso across multiple mobile GPUs. In general, the end-to-end predictions on the slower GPUs (MAPEs of 5.0% on PowerVR GE830 and 5.4% on Adreno 616) are better than on faster GPUs (MAPEs of 11.0% on Mali G76 and 10.7% on Adreno 640), since we observe smaller noise of measurements on slower GPUs over the longer execution time.

Since all the features are standardized, we use the magnitude of weights in the Lasso model to analyze the importance of different features. On all devices, using either CPUs or GPUs, we find the most critical features (those with largest weights) of convolution operations to be *FLOPs* and *kernel size*, which are strongly correlated with the costs of computation and memory access, respectively.[19] In contrast, the two most critical features of depthwise convolution operations are *FLOPs* and *input size*. Input size can dominate the cost of memory access for depthwise convolutions since their kernel sizes are substantially smaller than those of standard convolutions.

5.6 Comparison with Related Work

Lastly, we quantitatively compare our results with the state-of-the-art inference latency predictor nn-Meter [68] and conduct evaluations on the existing NAS benchmark dataset NATSBench [13].

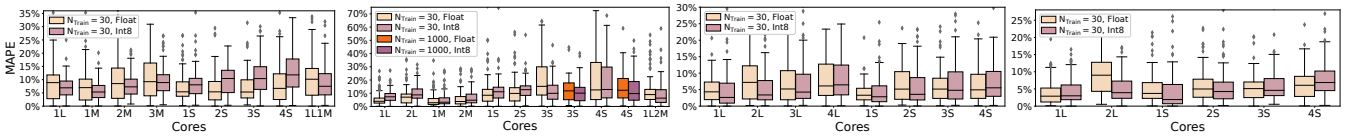
5.6.1 Predictors: nn-Meter. As noted in Section 1, nn-Meter [68] is a state-of-the-art technique for predicting inference latency on mobile devices; it uses a black-box model to estimate the rules of kernel fusion on a target device and predicts the latency of each kernel using Random Forest Regression. We first compared our results with those of nn-Meter using the pre-trained predictors provided by nn-Meter; however, nn-Meter's predictors failed to achieve accurate predictions on our dataset because they were trained on measurements collected using different compile options of TFLite (as detailed in [41]). Therefore, to achieve a fair comparison, we ran the source code of nn-Meter [48] to train a predictor with the same data used by our approach: (1) we first used nn-Meter to detect the rules of kernel fusion on four Android devices from Table 1;[20] (2) then, we trained kernel-level latency predictors on our synthetic dataset (including our latency measurements), but using the features and the hyperparameters of the Random Forest Regression model specified by nn-Meter.

Fig. 24 compares nn-Meter predictions (the average MAPE across four Android platforms) to those of our approach (using different ML models); as can be seen, our approach outperforms nn-Meter on both 102 real-world NAs and 100 NAs from our synthetic dataset across different sizes of training data. An important reason is that our approach considers a broader set of features for operations. For example, nn-Meter does not distinguish grouped convolutions from

[18] For clarity of presentation, we omit some outliers (<4% data points per configuration) and report plots with all data points in [41].

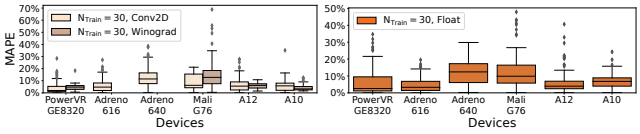
[19] However, as noted in Section 1, FLOPs alone is not an accurate proxy metric for the actual latency.

[20] In nn-Meter, rule detection requires benchmarking NAs on actual devices; nn-Meter does not support this for iOS devices.



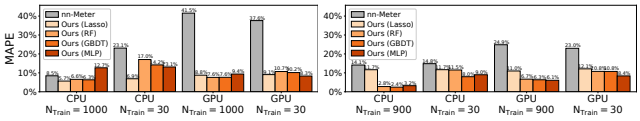
(a) Snapdragon 855 (b) Exynos 9820 (c) Helio P35 (d) A12 Bionic

Figure 22: Predictions of Lasso on Multicore CPUs (Real-World NAs)



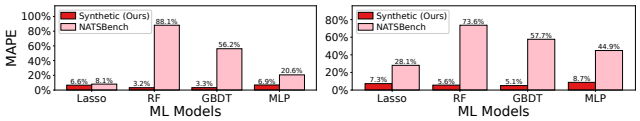
(a) Convolution (b) End-to-end

Figure 23: Predictions of Lasso on GPUs (Real-World NAs)



(a) Real-world NAs (b) Synthetic NAs

Figure 24: Comparison with nn-Meter



(a) CPU (b) GPU

Figure 25: Predictions on 44 Real-world NAs w/o DW-Conv

standard convolutions; a grouped convolution splits the input tensor into multiple groups of small tensors and conducts convolutions on each tensor, leading to a significant reduction in FLOPs. Consequently, nn-Meter mispredicts 14 real-world NAs with grouped convolutions (e.g., errors of 31.2% and 157.1% on the CPU and GPU of Helio P35, respectively). Notably, nn-Meter predictions are less accurate on GPUs for the following reasons: (1) nn-Meter does not account for kernel selection on GPUs, e.g., it neglects the fact that various kernels with distinct performance characteristics (such as Winograd) can be applied to the same convolution operation on different platforms (as evaluated in Section 5.4); (2) nn-Meter ignores the effects of ML framework overhead, which can be significant on GPUs (in particular, on PowerVR GE8320 and Mali G76, as shown in Section 5.3).

5.6.2 NAS Benchmark: NATSBench. Evaluated in related work [2, 68], the NATSBench [13] dataset includes NAs sampled from Topology Search Space (S_t) and Space Search Space (S_s). In S_t, each NA consists of operations with predefined configurations (e.g., number of channels) and different topology (i.e., interconnections between operations); in S_s, the topology is fixed and the number of channels is chosen from 8 candidates. For both datasets, we select 1000 NAs with the highest test accuracy on CIFAR-100 [36]; we observe that the diversity of operation configurations in these NAs is very limited. For example, there are only 11 and 239 unique configurations of convolution operations in the NAs from S_t and S_s respectively (compared to 6608 configurations in our synthetic dataset). In the NAS space of such limited configurations, building look-up tables by measuring the latencies of all possible configurations of each building block is sufficient to estimate the end-to-end latency of candidate NAs; in contrast, our NAS space covers over 2×10^7 configurations of convolution operations (i.e., with different number of input/output channels, kernel size, and group size), which makes look-up tables very costly to build and is better suited for inference latency prediction approaches during NAS.

In addition, the limited data diversity results in NATSBench being less representative of real-world NAs; for example, among 102 real-world NAs in our study, 58 contain depthwise convolutions, which are not present in NATSBench NAs (therefore, a prediction model cannot be trained for this type of operation). Fig. 25 compares prediction errors on the remaining 44 real-world NAs based on training with 1000 NAs from S_s (which includes a broader set of configurations than S_t) and from our synthetic dataset. As can be seen, more complex ML models are less accurate when trained with S_s, due to its limited diversity.

6 CONCLUSIONS

Using measurements collected on 6 mobile devices for a number of neural architectures (1000 synthetic NAS architectures and 102 real-world architectures), we showed the impact of different factors on inference latency, including optimizations applied by ML frameworks for mobile GPUs (kernel fusion and kernel selection), scheduling over heterogeneous subsets of CPU cores and integer representations after quantization, often neglected by related work. Based on this experimental evaluation, we proposed an approach to estimate end-to-end inference latency by training ML models to predict latency of each component type of neural architectures. Our approach can accurately predict latency of novel neural architectures on a given device using limited profiling data (e.g., from 30 architectures); notably, we achieve good accuracy also when the test dataset has different characteristics from training data, a common scenario in NAS. In future work, we plan to extend our evaluation and prediction approach to other efficiency metrics (e.g., power consumption), to different classes of specialized hardware accelerators for inference tasks (e.g., Apple Neural Engine), and to other machine learning framework optimizations (e.g., the OpenVINO ARM CPU plugin).

ACKNOWLEDGMENTS

This work was supported in part by the NSF CNS-1816887 and CCF-1763747 awards.

REFERENCES

[1] 2021. Sandbox for training deep learning networks. https://github.com/osmr/imgclsmob

[2] Saad Abbasi, Alexander Wong, and Mohammad Javad Shafiee. 2022. MAPLE: Microprocessor a Priori for Latency Estimation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

[3] Apple. 2016. Prioritize Work at the Task Level. https://developer.apple.com/library/archive/documentation/Performance/Conceptual/power_efficiency_guidelines_osx/PrioritizeWorkAtTheTaskLevel.html Accessed: 2022-10-10.

[4] Apple. 2021. Discover Metal debugging, profiling, and asset creation tools. https://developer.apple.com/videos/play/wwdc2021/10157 Accessed: 2022-10-06.

[5] Noureddine Bouhali, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. 2021. Execution Time Modeling for CNN Inference on Embedded GPUs. In *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*. 59–65.

[6] Halima Bouzidi, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. 2021. Performance prediction for convolutional neural networks on edge GPUs. In *Proceedings of the 18th ACM International Conference on Computing Frontiers*. 54–62.

[7] Wieland Brendel and Matthias Bethge. 2019. Approximating CNNs with Bag-of-local-Features models works surprisingly well on ImageNet. *arXiv preprint arXiv:1904.00760* (2019).

[8] Peter Bryzgalov, Toshiyuki Maeda, and Yutaro Shigeto. 2021. Predicting How CNN Training Time Changes on Various Mini-Batch Sizes by Considering Convolution Algorithms and Non-GPU Time. In *Proceedings of the 2021 on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn STrategy*. 11–18.

[9] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2019. Once-for-All: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations, ICLR*.

[10] 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*.

[11] Ping Chao, Chao-Yang Kao, Yu-Shan Ruan, Chien-Hsiang Huang, and Youn-Long Lin. 2019. HarDNet: A Low Memory Traffic Network. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[12] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, Peter Vajda, Matt Uyttendaele, and Niraj K. Jha. 2019. ChamNet: Towards Efficient Network Design Through Platform-Aware Model Adaptation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[13] Xuanyi Dong, Lu Liu, Katarzyna Musial, and Bogdan Gabrys. 2021. NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size. *IEEE transactions on pattern analysis and machine intelligence* 44, 7 (2021), 3634–3646.

[14] 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*, Vol. 33. 10480–10490.

[15] Jerome H. Friedman. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics* 29, 5 (2001), 1189–1232.

[16] Yanjie Gao, Xianyu Gu, Hongyu Zhang, Haoxiang Lin, and Mao Yang. 2021. *Runtime Performance Prediction for Deep Learning Models with Graph Neural Network*. Technical Report. Technical Report MSR-TR-2021-3. Microsoft.

[17] X Yu Geoffrey, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training. In *USENIX Annual Technical Conference*. 503–521.

[18] Google. 2022. Tensorflow Lite: Kernel Fusion Implementation. https://github.com/tensorflow/tensorflow/blob/v2.9.0/tensorflow/lite/delegates/gpu/common/gpu_model.cc#L393 Accessed: 2022-08-05.

[19] Google. 2022. Tensorflow Lite: ML for Mobile and edge devices. https://www.tensorflow.org/lite

[20] Google. 2022. TensorFlow Lite: Multithreading for Convolutions with the Ruy Library. https://github.com/google/ruy/blob/38a926/ruy/trmul.cc#L390 Accessed: 2022-08-05.

[21] Google. 2022. TensorFlow Lite: Multithreading for Depthwise Convolutions. https://github.com/tensorflow/tensorflow/blob/v2.9.0/tensorflow/lite/kernels/internal/optimized/depthwiseconv_multithread.h#L173 Accessed: 2022-08-05.

[22] Google. 2022. Tensorflow Lite: Profile Time for OpenCL Kernels. https://github.com/tensorflow/tensorflow/blob/v2.9.0/tensorflow/lite/delegates/gpu/cl/inference_context.cc#L792 Accessed: 2022-10-12.

[23] Google. 2022. TFLite Model Benchmark Tool. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/benchmark Accessed: 2022-07-12.

[24] Ubaid Ullah Hafeez and Anshul Gandhi. 2020. Empirical Analysis and Modeling of Compute Times of CNN Operations on AWS Cloud. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 181–192.

[25] Kai Han, Yunhe Wang, Qi Tian, Jianyuan Guo, Chunjing Xu, and Chang Xu. 2020. GhostNet: More Features From Cheap Operations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[27] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Identity mappings in deep residual networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*. Springer, 630–645.

[28] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. 2019. Searching for MobileNetV3. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[29] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861* (2017).

[30] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-Excitation Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[31] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[32] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[33] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on Machine Learning*, Vol. 37. PMLR, 448–456.

[34] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[35] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. 2018. Predicting the Computational Cost of Deep Learning Models. In *2018 IEEE International Conference on Big Data (Big Data)*. 3873–3882.

[36] Alex Krizhevsky and Geoffrey Hinton. 2009. Learning multiple layers of features from tiny images. (2009).

[37] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[38] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. 2019. On-Device Neural Net Inference with Mobile GPUs. *arXiv preprint arXiv:1907.01989* (2019).

[39] Youngwan Lee, Joong-won Hwang, Sangrok Lee, Yuseok Bae, and Jongyoul Park. 2019. An Energy and GPU-Computation Efficient Backbone Network for Real-Time Object Detection. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*.

[40] Jinyang Li, Runyu Ma, Vikram Sharma Mailthody, Colin Samplawski, Benjamin Marlin, Songqing Chen, Shuochao Yao, and Tarek Abdelzaher. 2021. Towards an Accurate Latency Model for Convolutional Neural Network Layers on GPUs. In *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, 904–909.

[41] Zhuojin Li, Marco Paolieri, and Leana Golubchik. 2022. Predicting Inference Latency of Neural Architectures on Mobile Devices. *arXiv preprint arXiv:2210.02620* (2022).

[42] Bingqian Lu, Jianyi Yang, Weiwen Jiang, Yiyu Shi, and Shaolei Ren. 2021. One proxy device is enough for hardware-aware neural architecture search. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 5, 3 (2021), 1–34.

[43] Sangkug Lym, Donghyuk Lee, Mike O'Connor, Niladrish Chatterjee, and Mattan Erez. 2019. DeLTA: GPU performance model for deep learning applications with in-depth memory system traffic analysis. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 293–303.

[44] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*. 116–131.

[45] Kevin P Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.

[46] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. 2021. A White Paper on Neural Network Quantization. *arXiv preprint arXiv:2106.08295* (2021).

[47] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[48] Microsoft Research nn Meter Team. 2021. nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices. https://github.com/microsoft/nn-Meter

[49] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *Proceedings of the International Conference on Learning Representations*.

[50] Zheng Qin, Zhaoning Zhang, Xiaotao Chen, Changjian Wang, and Yuxing Peng. 2018. Fd-mobilenet: Improved mobilenet with a fast downsampling strategy. In *2018 25th IEEE International Conference on Image Processing (ICIP)*. IEEE, 1363–1367.

[51] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollar. 2020. Designing Network Design Spaces. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[52] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[53] Dimitrios Stamoulis, Ruizhou Ding, Di Wang, Dimitrios Lymberopoulos, Bodhi Priyantha, Jie Liu, and Diana Marculescu. 2020. Single-Path NAS: Designing Hardware-Efficient ConvNets in Less Than 4 Hours. In *Machine Learning and Knowledge Discovery in Databases*. Springer, Cham, 481–497.

[54] Muhtadyuzzaman Syed and Arvind Akpuram Srinivasan. 2021. Generalized Latency Performance Estimation for Once-For-All Neural Architecture Search. *arXiv preprint arXiv:2101.00732* (2021).

[55] 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/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[56] Mingxing Tan and Quoc Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *International conference on machine learning*. PMLR, 6105–6114.

[57] Xiaohu Tang, Shihao Han, Li Lyna Zhang, Ting Cao, and Yunxin Liu. 2021. To bridge neural network design and real-world performance: A behaviour study for neural networks. *Proceedings of Machine Learning and Systems* 3 (2021), 21–37.

[58] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)* 58, 1 (1996), 267–288.

[59] Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, Wenyu Liu, and Bin Xiao. 2020. Deep high-resolution representation learning for visual recognition. *IEEE transactions on pattern analysis and machine intelligence* 43, 10 (2020), 3349–3364.

[60] Robert J Wang, Xiang Li, and Charles X Ling. 2018. Pelee: A real-time object detection system on mobile devices. *Advances in neural information processing systems* 31 (2018).

[61] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. 2019. High-throughput CNN inference on embedded ARM Big. LITTLE multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2254–2267.

[62] 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/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[63] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. 2019. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 331–344.

[64] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[65] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. 2018. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 285–300.

[66] Fisher Yu, Dequan Wang, Evan Shelhamer, and Trevor Darrell. 2018. Deep Layer Aggregation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[67] Sergey Zagoruyko and Nikos Komodakis. 2017. Diracnets: Training very deep neural networks without skip-connections. *arXiv preprint arXiv:1706.00388* (2017).

[68] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 81–93.

[69] 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)*.

[70] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *International Conference on Learning Representations, ICLR*.

[71] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.