Inference Latency Prediction for CNNs on Heterogeneous Mobile Devices and ML Frameworks

Zhuojin Li[a], Marco Paolieri[a], Leana Golubchik[a]

[a]*University of Southern California, Los Angeles, CA, 90089, USA*

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 (specifically, convolutional neural networks for image classification), ML frameworks, 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.

Keywords: Convolutional Neural Networks, NAS, Latency, Prediction, Mobile, GPU, CPU

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 [1, 2, 3] are typically designed through Neural Architecture Search (NAS) [4]. Recent work [5, 6, 7] proposes zero-shot NAS, which substantially reduces the search cost by utilizing proxy metrics to estimate the ML accuracy of each candidate NA without training. Consequently, the bottleneck of zero-shot NAS becomes latency measurement; for example, the evaluation of accuracy proxy metric in [5] takes less than a second for each candidate NA, while deploying and compiling each NA on a device for latency measurement typically requires a few minutes. In addition, NAs exhibit distinct performance characteristics across hardware platforms [8], and it is time-consuming to repeat the measurements on all platforms during NAS. As an alternative to measurements, existing approaches for evaluating the efficiency of NAs can be categorized as those using (1) proxy metrics [3, 9] (e.g., FLOPs), which are usually platform-independent and cannot accurately reflect the actual performance due to the diversity of platforms [10, 8]; (2) look-up tables [11, 12, 13] 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 [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24], but also including analytical performance models (e.g., accounting for computations [25] and memory access traffic of GEMM-based convolution [26, 27]). 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 [14, 17, 18, 19, 20] or Nvidia embedded GPUs [23, 24]; instead, the heterogeneity of CPUs and GPUs on mobile devices makes the

performance predictions more difficult. In particular, inference tasks are frequently performed on mobile devices using CPUs [28], due to the support of a broader set of available operations (e.g., Channel Shuffle [29] is currently unavailable on the TensorFlow Lite (TFLite) [30] GPU Delegate [31]; batch normalization is not implemented as Metal kernel for iOS GPUs in PyTorch Mobile [32]). 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 [33]; 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) on TFLite. Therefore, it is necessary to evaluate prediction approaches using heterogeneous hardware resources, in particular over multiple CPU cores; this is *not taken into consideration by existing work* on latency prediction for mobile CPUs [15, 21, 22].

(2) ML framework optimizations: Modern ML frameworks introduce optimizations that can significantly accelerate inference tasks. For example, operator fusion [34] reduces overhead in the invocation of kernels on GPUs: our tests show that kernel fusion can result in up to 22% and 48% performance improvements over 102 real-world NAs on TFLite and PyTorch Mobile, respectively (Section 3.2.1). Similarly, the choice of algorithms for implementing each operation can considerably affect inference performance: for example, TFLite uses the faster Winograd [35] algorithm for some (but not all) convolution layers on GPUs; PyTorch Mobile implements five types of kernels for convolution layers of different configurations on GPUs (Section 3.2.2). *Existing work* on latency estimation for GPUs [24, 23, 18, 20, 15] *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 [1, 10, 29], prompting the definition of new NAS search spaces. *Existing ML-based performance prediction models use training and test datasets with very similar NAs* [14, 24, 36], *or with a small set of popular NAs* [37, 18, 19]; 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, ML frameworks, 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 of convolutional neural networks for image classification that is *more representative of real-world NAs*, by including a broader set of NA blocks than current literature [38]; (ii) we measure and predict latency on *different combinations of heterogeneous CPU cores* and *different data representations* (i.e., floating-point or integer quantization [39]) over two ML frameworks, while related work [21, 22] uses only a single CPU core (unrealistic in practice) and floating-point calculations for a given ML framework. Notably, our solution explicitly accounts for optimizations applied by ML frameworks on each NA; in contrast, previous work nn-Meter [22] uses black-box models to estimate the effects of ML framework optimizations (and is limited to a single core for CPUs).

In this paper, we significantly extend our preliminary work [40] (which only considers TFLite) by collecting inference measurements and applying our prediction approach to PyTorch Mobile (for all NAs and on all mobile platforms, effectively doubling our experimental evaluation from 90 to 174 scenarios). The comparison of performance characteristics between different ML frameworks highlights not only the impact, but also the heterogeneity of ML framework optimizations. For example, many operations (including addition, mean, pooling) obtain significant speedups from multithreading execution in PyTorch Mobile, but no benefits in TFLite (Section 3.1.1); in contrast, model quantization can result in performance degradation on PyTorch (using large and medium cores on Snapdragon 855), instead of the speedups observed in TFLite (Section 3.1.2). These differences are due to the different implementations and optimization strategies of TFLite and PyTorch Mobile; to obtain accurate predictions, we analyze and model the optimizations applied by each ML framework. 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 using 2 ML frameworks (TFLite and PyTorch Mobile), 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). For mobile GPUs, we analyze two types of optimization strategies in ML frameworks: kernel fusion and kernel selection (Section 3.2). As a representative example, we present the principles of both strategies in TFLite and PyTorch Mobile, 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 by encoding the configurations of all operations as a single feature vector [14, 17, 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 precisely deduce the selected GPU kernels *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 [38] (studied in [14, 22]) 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 174 scenarios across 6 mainstream mobile platforms, including multicore combinations and use of integer quantization on both TFLite and PyTorch Mobile. 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 on different ML frameworks.

- 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 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 [21], 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 [41]) achieve very accurate predictions for NAs with similar characteristics (e.g., average errors 2.4% and 5.2% on CPUs and GPUs for TFLite; 2.6% and 4.8% on CPUs and GPUs for PyTorch Mobile); 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 [42]) are robust and still accurate (e.g., average errors of 5.4% and 8.0% on CPUs and GPUs for TFLite; 10.4% and 8.2% on CPUs and GPUs for PyTorch Mobile). When training data is very limited (e.g., 30 synthetic NAs), accuracy is lower (e.g., with GBDT, average errors of 8.1% and 8.6% on CPUs and GPUs for TFLite; 6.2% and 8.8% on CPUs and GPUs for PyTorch Mobile) 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 [21].

2. Background

2.1. Convolutional Neural Architectures for Image Classification

Convolutional Neural Networks (CNNs) play an important role in computer vision tasks. These models process an input image (e.g., 224x224 pixels and 3 channels) using a sequence of *convolutional layers*. Each layer uses a set of filters (e.g., moving windows of 3x3 pixels) that are applied at each image location (e.g., using a dot product across all input channels, followed by an element-wise activation function) to detect higher level features (e.g., shapes, colors, and textures); the outputs of these filters are collected into output channels (also known as *feature maps*), which are

[1] As in existing literature [14, 22] 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., [18] for cloud GPUs) due to the heterogeneity of mobile platforms.

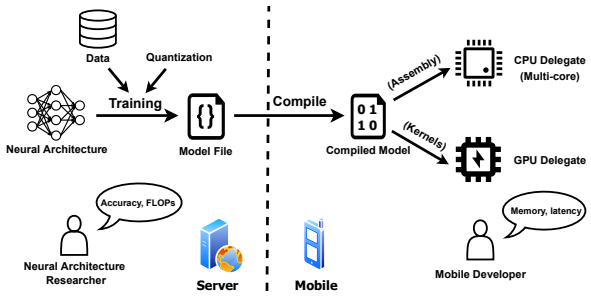


Figure 1: Lifecycle of Neural Architecture Development and Deployment on Mobile Devices

then provided as input to the next layer. Pooling layers are used to replace feature map values in a moving window (e.g., 3x3 pixels) with their maximum or mean, thus reducing the resolution of the map for efficiency. At the end of the CNN, fully connected layers are frequently used to obtain confidence scores of output classes as weighted sums of features extracted by previous layers.

CNNs are usually deep, to be able to learn complex and hierarchical features: earlier layers extract low-level local details (e.g., edges), while later layers capture high-level features as a combination of low-level ones (e.g., a specific shape). However, deeper CNNs are computationally expensive; in order to reduce complexity, *depthwise convolutions* [43] have been proposed to perform separate convolutions for each input channel, instead of using values from all input channels. *Grouped convolutions* are a variant where input channels are split into groups, each processed by a different set of filters. Feature maps calculated using different filters can also be *concatenated* or combined using *element-wise* addition. To improve efficiency, a sequence of operations (e.g., a convolutional filter and the following ReLU activation) can be combined as a single operation if a more efficient implementation is available (e.g., in *kernel fusion* of GPU operations).

The weights of the filters computing and combining feature maps are trained from a dataset of input/output examples; after training, CNNs are deployed to cloud servers or mobile devices for *inference tasks*, i.e., classification of new input images. When CNNs are deployed to mobile devices, memory and computation required for inference tasks are particularly critical, due to their limited resources and to responsiveness requirements for many applications (e.g., augmented reality or real-time user interaction). For this reason, we investigate methods to predict *end-to-end latency* of a given NA, i.e., the total time required to execute the operations of all layers, to obtain an input classification from an input example. Notably, inference latency does not depend on the input values or model weights, but on the input shape and the model architecture; this makes our prediction method useful for NAS.

2.2. Development and Deployment of Neural Architectures

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 [44, 43, 10] and NAS [45, 1, 2, 3]. 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* [46, 39] 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 intermediate results between operations; the complete computational graph is included in the `.tflite` model file; in PyTorch, the model is serialized and optimized as a TorchScript program, which contains the information of control and data flow during inference.

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

Table 1: Mobile Platforms in Our Study

A mobile device can be equipped with multiple hardware accelerators for 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" as a sequence of CPU operations and GPU kernels. Notably, different ML frameworks provide distinct implementations; for instance, the GPU kernels on Android are implemented in OpenCL for TFLite, while in Vulkan for PyTorch Mobile. Within a framework, 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, several consecutive operations/kernels can be "fused" into one to reduce the dispatching overhead (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.

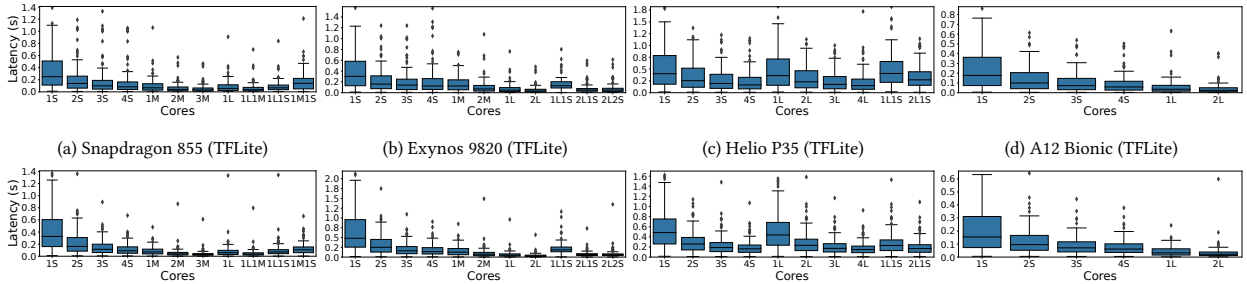
3.1. Mobile CPUs

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 mobile platforms 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 the CPU affinity of these threads to encourage their scheduling on the cores; for iOS devices, we specify the *quality-of-service (QoS) class* [47] of each thread to schedule on performance or efficiency cores. Considering the limited resources available on mobile devices, we select 102 real-world convolutional neural networks for image classification with up to 18 million parameters from 25 articles (with manual design or NAS) [48, 49, 50, 51, 52, 3, 13, 53, 54, 55, 56, 2, 43, 45, 1, 57, 58, 15, 59, 44, 60, 61, 62, 63, 64]. The TensorFlow and PyTorch implementations of these NAs are from [65], which also provides pre-trained parameters and Top-1/Top-5 test errors on the ImageNet-1K dataset. Similarly to related work, we observe in our experiments that inference latency depends on the shape of the input data but not on input values; therefore, we use a random 224x224 image (as in the ImageNet-1K dataset) for measurements of end-to-end inference latency.

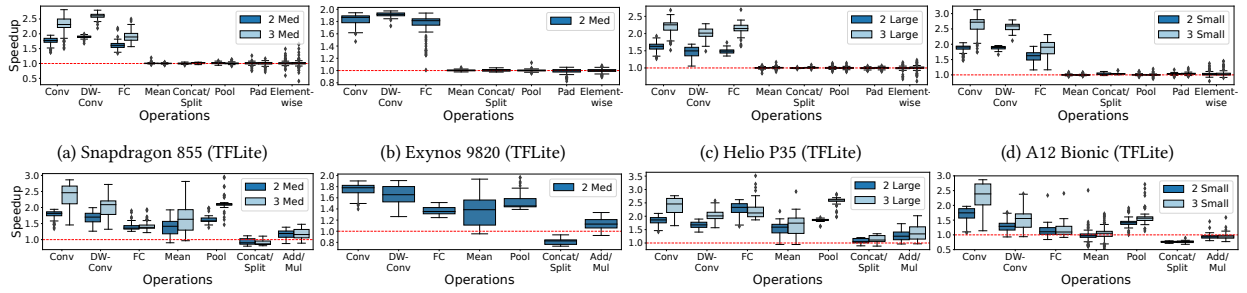
[2]Since the architectures of Snapdragon 710 and A10 are similar to Snapdragon 855 and A12, respectively, we report their measurements in Appendix A. The evaluation of our prediction method on Helio P35 and A10 Fusion using the GPU backend of PyTorch Mobile are not included in the paper due to their insufficient memory for the backend.



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

(e) Snapdragon 855 (PyTorch) (f) Exynos 9820 (PyTorch) (g) Helio P35 (PyTorch) (h) A12 Bionic (PyTorch)

Figure 2: Effects of Multicore on End-to-end Latency (L, M, S represent Large, Medium, Small cores, respectively)



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

(e) Snapdragon 855 (PyTorch) (f) Exynos 9820 (PyTorch) (g) Helio P35 (PyTorch) (h) A12 Bionic (PyTorch)

Figure 3: Effects of Homogeneous Multicore on Operation-wise Latency (Speedup over One Core)

Fig. 2 uses boxplots[3] to depict end-to-end latency of these real-world neural architectures with different multicore configurations.[4] Counterintuitively, *using multiple heterogeneous cores can result in performance degradation*: for example, on Snapdragon 855 (Figs. 2a and 2e), 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 [33], we attribute this performance degradation to the overhead of inter-cluster communication; also, we observe that the work of an operation is split *equally* among threads in modern ML framworks (e.g., TFLite Threadpool [66] with its matrix multiplication library Ruy [67], and PyTorch Mobile Threadpool [68]), which is suboptimal for heterogeneous cores.[5]

In Fig. 2, we observe a sublinear end-to-end speedup with respect to the number of homogeneous cores for multithreading. That is because, as shown in Fig. 3, convolution, depthwise convolution (DW-Conv) and fully-connected (FC) operations achieve sublinear speedups on both ML frameworks as the number of threads increases; the performance improvements of Mean and Pool operations are significant in PyTorch Mobile but negligible in TFLite, due to the lack of support for parallel execution of these operations in TFLite.

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

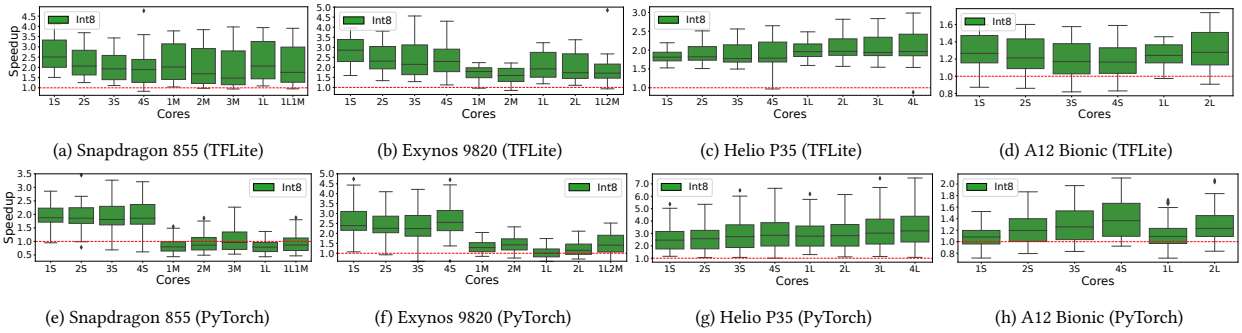
3.1.2. Effects of Quantization

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 com-

[3] In the paper, boxplots indicate 1st quartile, median, and 3rd quartile of the data; whiskers extend for 1.5x the interquartile range.

[4] For clarity of presentation, we omit some outliers with substantially higher latency in Fig. 2 (<4% of the data per configuration) and report the complete data in Fig. A.31 of the Appendix.

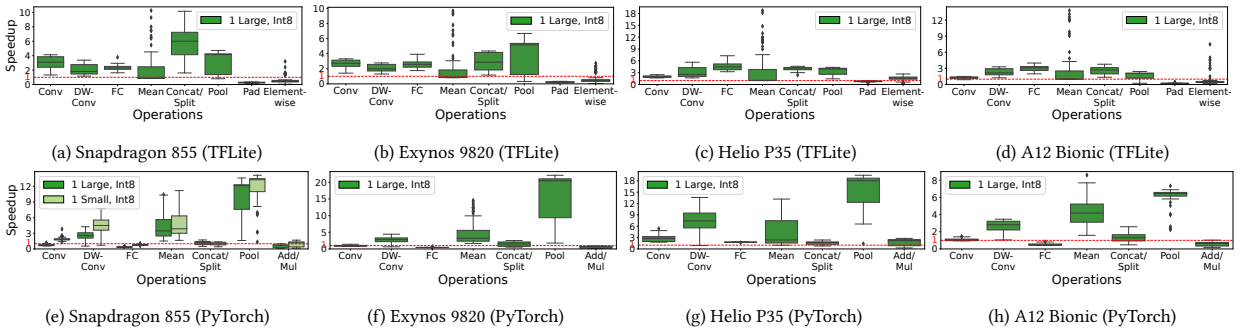
[5] The work in [33] also proposes solutions to improve the throughput over heterogeneous cores. In our paper, we focus on the performance characteristics of ML workloads and follow the current implementation of ML frameworks.



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

(e) Snapdragon 855 (PyTorch) (f) Exynos 9820 (PyTorch) (g) Helio P35 (PyTorch) (h) A12 Bionic (PyTorch)

Figure 4: Effects of Quantization on End-to-end Latency



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

(e) Snapdragon 855 (PyTorch) (f) Exynos 9820 (PyTorch) (g) Helio P35 (PyTorch) (h) A12 Bionic (PyTorch)

Figure 5: Effects of Quantization on Operation-wise Latency

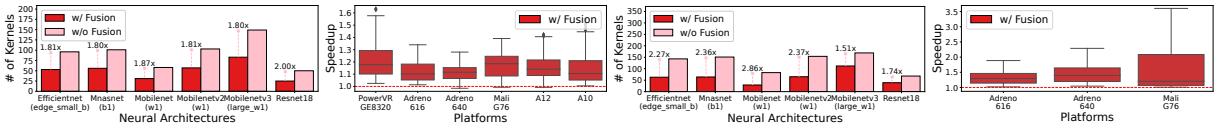
putational demand, without substantial loss in ML accuracy. We study the approach of integer-arithmetic-only inference [46] in TFLite (where both weights and activations are represented as 8-bit integers during inference[6]) and post-training static quantization in PyTorch[7], respectively. Fig. 4 compares end-to-end inference latency using 8-bit integer with 32-bit floating-point.[8] As can be seen, quantization shows a distinct speedup on various core combinations. Particularly, we observe performance degradation after quantization on large and medium cores on Snapdragon 855 in PyTorch Mobile (Fig. 4e); we attribute this degradation to the use of a different inference backend, QNNPACK [70], for quantized models in PyTorch, resulting in an average of 32% performance degradation for convolutions (for one large core) as compared to the inference backend XNNPACK [71] used for floating-point models, as shown in Fig. 5e. Notably, our prediction approach still achieves accurate prediction for this anomalous case, as shown in Section 5.2.

Fig. 5 depicts the latency improvement of each type of operation after quantization. In TFLite, 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 [46, 39] 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). In PyTorch Mobile, operations of DW-Conv, Mean, and Pool show significant performance improvements after quantization; however, the performance of the remaining operations varies on different cores of devices, due to the distinct implementations of the backends.

[6]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.

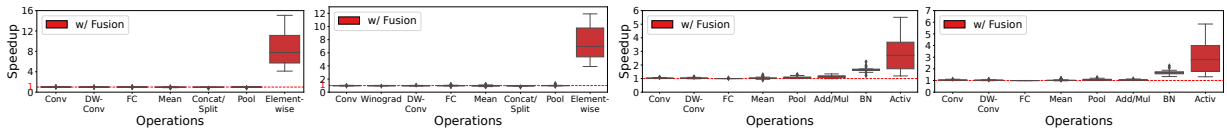
[7]We made minor modifications to the QNNPACK backend to solve a performance issue for DW-Conv with 7x7 kernels, as described in [69].

[8]Similarly to Fig. 2, we omit outliers (only of a couple of points) for better visualization.



(a) Number of Kernels (TFLite) (b) End-to-end Latency (TFLite) (c) Number of Kernels (PyTorch) (d) End-to-end Latency (PyTorch)

Figure 6: Effects of Kernel Fusion



(a) Adreno 640 (TFLite) (b) A12 Bionic (TFLite) (c) Adreno 640 (PyTorch) (d) Adreno 616 (PyTorch)

Figure 7: Effects of Kernel Fusion on Operation-wise Latency

Insight 2. While quantization reduces memory utilization and latency of inference tasks in most cases (as expected), there is *performance degradation* for some operations due to the cost of scaling inputs or inefficient backend implementations in ML frameworks.

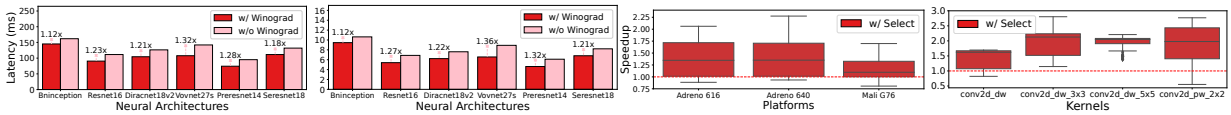
3.2. Mobile GPUs

3.2.1. Effects of Kernel Fusion

Kernel fusion has been broadly adopted to reduce the overhead of dispatching kernels [22]. In PyTorch Mobile, a sequence of optimizations are applied to models generated for mobile, including (1) fusion of the batch normalization (BN) with the convolution layers, (2) fusion of activation layers (as clamping) with the previous convolution and linear layers, and (3) fusion of ReLu layers into the previous addition operations. In TFLite, the implementation of kernel fusion is more sophisticated, as reported in Algorithm B.2: 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), (2) uses this output tensor as its first input to produce a single output (Line 22), and (3) has a compatible type (Line 24). To study the impact of kernel fusion, we modified the source code of TFLite [72] and PyTorch Mobile [73] to disable this feature.

Figs. 6a and 6c illustrate that kernel fusion leads to a reduction in the number of kernels of over 45% for real-world NAs on both frameworks. Figs. 6b and 6d show the performance improvements from kernel fusion on different mobile devices.[9] We observe up to 1.22x and 1.48x speedup of the average end-to-end latency over all the neural architectures for TFLite and PyTorch Mobile, respectively, due to a reduction in the cost of kernel dispatching. As shown in Fig. 7,[10] kernel fusion can significantly reduce the latency of certain operations (i.e., element-wise in TFLite; batch normalization and activation in PyTorch Mobile) by merging multiple kernels; at the same time, there is no substantial latency increase for other operations. This observation is aligned with our analysis: the operations fused into other operations are mainly element-wise operations in TFLite (Line 24 of Algorithm B.2), as well as batch normalization and activation layers in PyTorch Mobile.

Insight 3. By substantially reducing the number of operation kernels, kernel fusion can improve the performance of inference tasks on mobile GPUs. The fusion can substantially reduce the latency of *element-wise operations* in TFLite and *batch normalization* and *activation* in PyTorch Mobile by fusing them into the predecessor layer; the effect on other operations is negligible.



(a) PowerVR GE8320 (TFLite) (b) A12 Bionic (TFLite) (c) End-to-end Latency (PyTorch) (d) Adreno 640 (PyTorch)

Figure 8: Effects of Kernel Selection

3.2.2. *Effects of Kernel Selection*

Machine learning frameworks use different optimized implementations for the operations of neural architectures. Algorithm B.3 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 the criteria defined by the CheckWinograd function, a Winograd kernel will be selected. Figs. 8a and 8b 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 in TFLite: 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. Details are illustrated in Appendix B.

Similarly, PyTorch Mobile implements five types of Vulkan kernels for the convolution operations: conv2d, conv2d_pw, conv2d_dw, conv2d_dw_3x3, and conv2d_dw_5x5 based on the input/output size, kernel shape and number of groups of the convolution, as presented in Algorithm B.1. Fig. 8c shows that, in comparison with only using conv2d kernels, applying various optimized kernels leads to an average speedup of 1.33x on end-to-end latency; Fig. 8d further illustrates the improvement on each type of kernel (over conv2d) on Adreno 640.

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

4. Methodology

Given an input model 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 the configurations (e.g., input shape, channel size) of the operations (i.e., the execution units on mobile CPUs) in the neural architecture; (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 configurations (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 prediction models and to evaluate our approach, we collect latency measurements on both real-world NAs and a synthetic dataset including 1000 neural architectures from a NAS space (Section 4.3), which is available at [69] and was described in [74].

4.1. *Kernel Deduction*

From the model file, we are able to extract the configurations of all the operations of a neural architecture. As discussed in Section 3.1.1, these operations are executed sequentially on mobile CPUs; multiple threads collaborate within the execution of each operation. For each type of operation, platform, ML framework, and CPU core combination, we train a machine learning model to predict inference latency.

[9] The outliers (only a couple of data points) are removed to improve visualization. Disabling kernel fusion in PyTorch Mobile leads to failure on iOS GPUs, due to the lack of implementations for batch normalization in the Metal backend.

[10] A few outliers with large speedups on element-wise operations are reported in Fig. A.33 of the Appendix.

When using mobile GPUs, operations can be further 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 save the cost of deploying the neural architectures to physical devices, 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 ML framework. Specifically, to predict latency on mobile GPUs, we first fuse kernels according to the rules (e.g., Algorithm B.2 for TFLite); then, we use the rules (Algorithms B.1 and B.3) to select a kernel based on the configurations of each convolution operation and on the specific target device.

Operation / Kernel	Features
Conv, DW-Conv	Input height (width), input channel, output height (width), stride, kernel height (width), filters, input size, output size, kernel size, FLOPs
Grouped-Conv	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, BN, Activations	Input height (width), input channel, input size

Table 2: Feature Space for Each Category of Operations

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), as reported in Table 2. 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 [75] adopted in the literature [23, 24, 22, 18, 20].

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

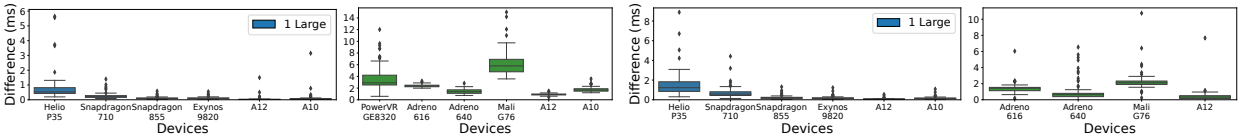
$$\boldsymbol{w}^* = \arg\min_{\boldsymbol{w}} \ \frac{1}{N}\sum_{i=1}^{N}\left|\frac{\boldsymbol{w}^T\hat{\boldsymbol{x}}_{\boldsymbol{i}} - y_i}{y_i}\right|^2 + \alpha\|\boldsymbol{w}\|_1 \quad \text{s.t.} \quad \boldsymbol{w} \geq 0\,. \tag{1}$$

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 non-negative 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.

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 the number of neurons in each layer from $\{64, 128, 256, 512\}$.



(a) CPUs (TFLite) (b) GPUs (TFLite) (c) CPUs (PyTorch) (d) GPUs (PyTorch)

Figure 9: Difference between End-to-end Latency and Sum of Operation-wise Latency for Real-World NAs

Similarly to previous work [18], we use ReLU activations after each layer and the Adam optimizer with learning rate from $\{5 \times 10^{-3}, 5 \times 10^{-4}, 5 \times 10^{-5}\}$, and weight decay 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 (in a topological order determined by their dependencies). We also account for the additional latency due to overhead and data transfers; as shown in Fig. 9, 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 $T_{\text{overhead}} + \sum_{c \in C} f_c^*(\hat{\boldsymbol{x}}_{\boldsymbol{c}})$, 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. We note that, if the same kernel (e.g., a 3x3 convolution) is executed multiple times within the NA, these are considered different operations, for which we collect different measurements.

In our experiments, we trained our prediction models (Lasso, Random Forests, GDBT, MLP) on an Intel i7-6800K CPU. Taking Pixel 4 (using one large CPU core) as an example, the process of comprehensive training and hyperparameter-tuning of all operations (convolutions, depthwise convolutions, etc.) from synthetic NAs required 1 minute for Lasso, 7 minutes for RF, 2.6 hours for GBDT, and 28.7 hours for MLP.

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 collecting latency measurements (Section 4.3.1) and then describe the design of the NAS space (Section 4.3.2).

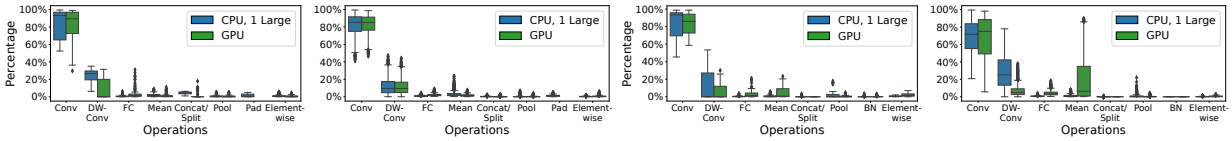
4.3.1. Kernel Latency Profiling

For TFLite, We use the *TFLite Model Benchmark Tool* [76] 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] and we set the GPU Performance State [78] to *high* for iOS devices, in order to acquire stable measurements over time.

In PyTorch Mobile, we utilize the Kineto profiler [79] to collect the duration of every CPU operation; for the Vulkan backend on Android GPUs, we measure the duration of GPU kernels by collecting GPU timestamps from the query pool in Vulkan; for Metal backend on iOS GPUs, similarly to TFLite, we dispatch each Metal command buffer for 256 times and record the GPU execution time.

We adopt the default precision settings of the TFLite Model Benchmark tool, which uses 32-bit floating-point on mobile CPUs and 16-bit floating-point on mobile GPUs. For PyTorch Mobile, we enable 16-bit floating-point

[11] Here, we follow the TFLite implementation [77], which allows no more than 256 dispatches for Mali GPUs; we found that using fewer dispatches does not sufficiently reduce the overhead of timestamp recording.



(a) Real-world NAs (TFLite) (b) Synthetic NAs (TFLite) (c) Real-world NAs (PyTorch) (d) Synthetic NAs (PyTorch)

Figure 10: Latency Breakdown on Snapdragon 855

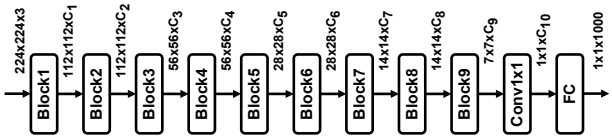


Figure 11: Design of the NAS Space for Synthetic Dataset

inference for Vulkan backend on Android GPUs. Figs. 10a and 10c show the average latency breakdown for 102 real-world neural architectures on Snapdragon 855 for TFLite and PyTorch Mobile, respectively[12]. Notably, convolution and depthwise convolution operations account for most of the end-to-end latency in both frameworks.

4.3.2. NAS Space for Sampling Neural Architectures

Figs. 10a and 10c highlights the importance of convolution and depthwise convolution operations in end-to-end latency. Consequently, we design a search space to sample diverse configurations of each operation type to learn their performance characteristics. As shown in Fig. 11, synthetic neural architectures from our NAS space contain a sequence of 9 blocks with fixed input height and width, following the design of sequential connections of blocks in MobileNetV2 [45].[13] The type and parameters of each building block are sampled uniformly at random among:

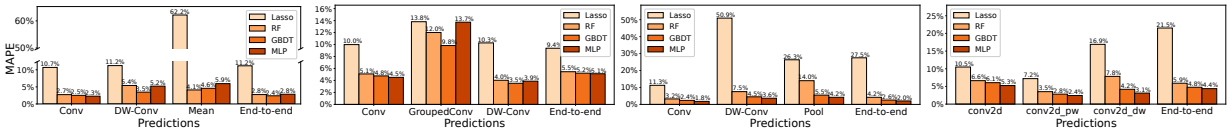
1. A convolution layer (with kernel size 3x3, 5x5 or 7x7, and optional group size $4k$, with $1 \leq k \leq 16$).
2. Depthwise separable convolution [43] (with kernel size 3x3, 5x5 or 7x7).
3. Linear bottleneck [45] (with kernel size 3x3, 5x5 or 7x7, expansion rate 1, 3 or 6, and optionally using Squeeze-and-Excite [1]).
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 $C_{10} \in [1200, 1800]$.

Similarly to real-world NAs, inference latency does not depend on the input values or model weights, but on the input shape and operations performed. For this reason, in our inference latency measurements for synthetic NAs, we used randomly initialized model weights and a random 224x224 input image. 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 *174 scenarios*, covering (1) combinations of homogeneous or heterogeneous cores, (2) floating-point and 8-bit integer representations, (3) mobile GPUs from different manufacturers, and (4) two mainstream ML frameworks. Figs. 10b and 10d illustrates the latency breakdown for NAs

[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.



(a) CPU, One Large Core (TFLite) (b) GPU (TFLite) (c) CPU, One Large Core (PyTorch) (d) GPU (PyTorch)

Figure 12: Predictions of ML Models (Synthetic NAs)

in our synthetic dataset, where the latency distribution over operations is similar to real-world NAs. In summary, our synthetic dataset includes latency measurement and corresponding parameters for 23275 operations under 84 scenarios for PyTorch Mobile, and 22182 operations under 90 scenarios for TFLite, resulting in 3951480 data points in total. This dataset is available at [69] and was described in [74].

5. Results

This section presents a comprehensive evaluation of our latency prediction framework across a broad range of scenarios: first, we show 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.

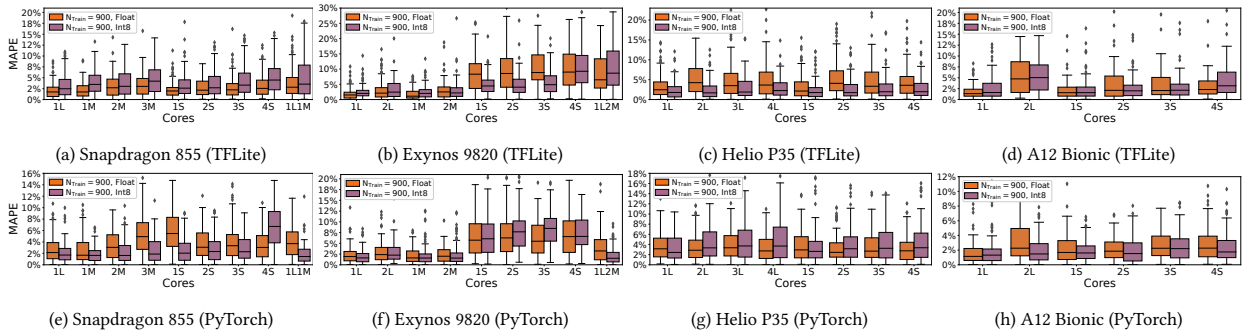
Fig. 12 presents the average MAPE across 6 platforms[14] on both TFLite and PyTorch Mobile under different ML approaches when predicting end-to-end latency, as well as the latency of the 3 operation types accounting for most of end-to-end latency. Based on the latency breakdown of synthetic neural architectures on CPUs and GPUs (Figs. 10b and 10d), convolution operations typically account for the most significant 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 in TFLite (Fig. 12a), while its MAPE for end-to-end latency is only 10.0%; that is because, as shown in Fig. 10b, 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. 12, in our default setting, all non-linear ML approaches (RF, GBDT, MLP) achieve comparable accuracy on end-to-end latency predictions, with average MAPE across six platforms: below 2.8% for CPUs and below 5.5% for GPUs on TFLite; below 4.2% for CPUs and below 5.9% for GPUs on PyTorch Mobile. Lasso achieves less accurate predictions (11.2% on CPUs and 9.4% on GPUs for TFLite; 27.5% on CPUs and 21.5% on GPUs for PyTorch Mobile) because its linear model cannot represent non-linear relationships between latency and operation features, as identified by previous work [8, 22].

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 (Figs. 12a and 12c).

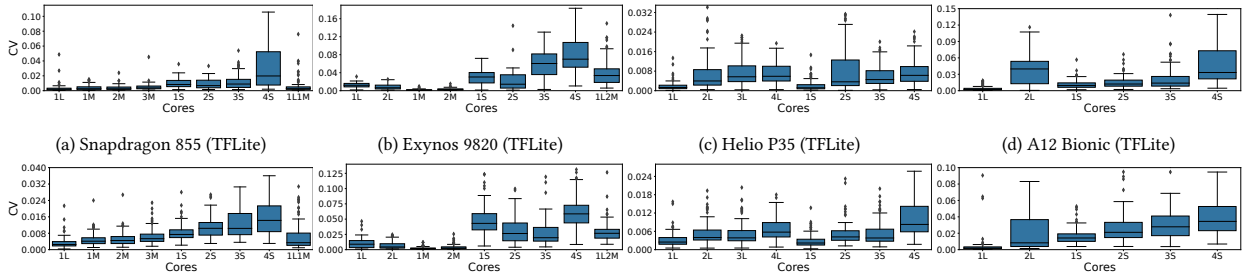
[14]Due to lack of space, MAPE of each platform is reported in Tables C.5 and C.7 of the Appendix.



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

(e) Snapdragon 855 (PyTorch) (f) Exynos 9820 (PyTorch) (g) Helio P35 (PyTorch) (h) A12 Bionic (PyTorch)

Figure 13: Predictions of GBDT on Multicore CPUs (Synthetic NAs)



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

(e) Snapdragon 855 (PyTorch) (f) Exynos 9820 (PyTorch) (g) Helio P35 (PyTorch) (h) A12 Bionic (PyTorch)

Figure 14: Coefficient of Variation for Latency Measurements of Synthetic NAs on CPUs

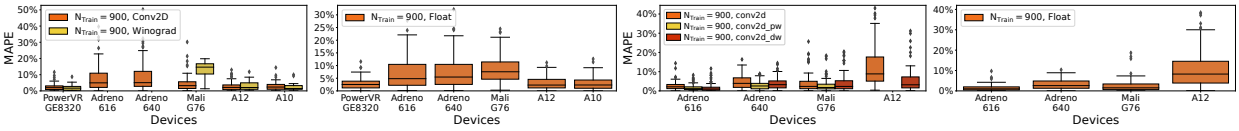
Fig. 13 illustrates GBDT predictions of end-to-end latency over various core configurations.[15] We observe that more homogeneous cores typically lead to higher prediction errors. Using more cores can result in larger measurement variance, due to background jobs running on mobile devices (e.g., cameras, 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. 13b, 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 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. 14. 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 for TFLite; 8.5% on Exynos 9820, 6.6% on Snapdragon 855, 5.1% on Helio P35, and 3.3% on A12 Bionic for PyTorch Mobile.

Note that using heterogeneous cores results in even higher variability of latency measurements due to inter-cluster communication [33]. 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 with TFLite (Fig. 13a), 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. 15 presents predictions of GBDT for different GPUs. For convolution operations, we split the results of different types of kernels (i.e., Winograd for TFLite; conv2d_pw and conv2d_dw for PyTorch Mobile) between Fig. 15a and Fig. 15c because separate latency predictors are trained for each kernel; no Winograd kernel is used on Adreno 640 and 616 with TFLite due to the rules of kernel selection presented in Section 3.2.2. Overall, GBDT achieves accurate end-to-end prediction across all six GPUs, with worst MAPE of 8.2% on Exynos 9820 for TFLite and 10.9% on A12 Bionic for PyTorch Mobile.

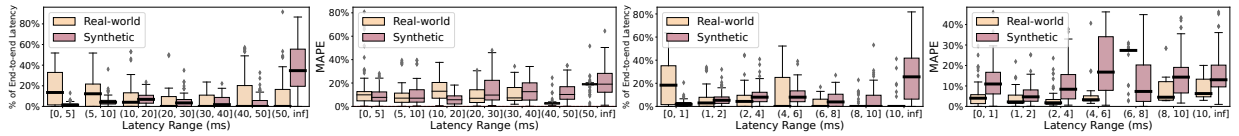
[15]For clarity of presentation, we omit some outliers (<9% data points for 1 large and 2 medium cores of Exynos 9820, and <4% data points for all other configurations), and report plots with all data points in Fig. A.34 of the Appendix.

[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) Convolution (TFLite) (b) End-to-end (TFLite) (c) Convolution (PyTorch) (d) End-to-end (PyTorch)

Figure 15: Predictions of GBDT on GPUs (Synthetic NAs)



(a) Contribution to End-to-end (TFLite) (b) Predictions using Lasso (TFLite) (c) Contribution to End-to-end (PyTorch) (d) Predictions using Lasso (PyTorch)

Figure 16: Convolution Operations with Different Latency Ranges (Helio P35 for TFLite, Snapdragon 855 for PyTorch Mobile)

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. Figs. 16a and 16c show the percentage of end-to-end latency attributed to convolution operations (split by range) on Helio P35 with TFLite, and on Snapdragon 855 with PyTorch Mobile: convolutions with higher latency dominate end-to-end latency in our synthetic neural architectures, while faster convolutions contribute more to real-world neural architectures.

Figs. 17a and 17c show 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. 12) 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% in TFLite and 10.4% in PyTorch Mobile). We attribute this anomaly to the better accuracy of Lasso predictions on fast operations (< 50 ms in Fig. 16a, and < 10 ms in Fig. 16c) due to higher weights assigned to these operations in Eq. (1), which we observe in both synthetic and real-world architectures (Figs. 16b and 16d). 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.

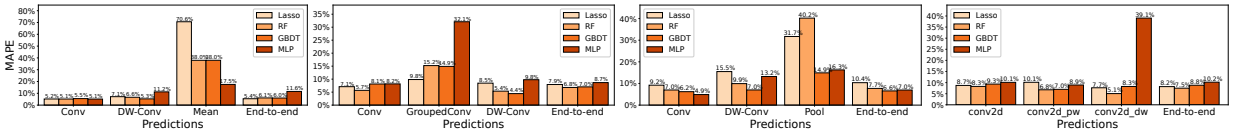
Figs. 17b and 17d presents predictions on mobile GPUs. We observe that, for some small real-world neural architectures, the overhead of TFLite is significant. Since the overhead has high runtime variability (in particular, on PowerVR GE8320 and Mali G76), it can affect the accuracy of end-to-end latency 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.

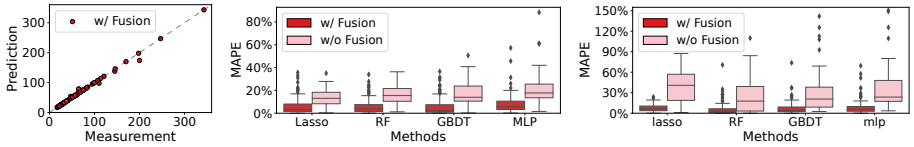
Kernel Fusion. In Section 3.2.1, we show that kernel fusion considerably reduces the number of kernels and leads to improvements in end-to-end latency. Fig. 18a shows that, after following Algorithm B.2 to estimate which kernels will be fused in TFLite (Section 3.2.1), we obtain a number of kernels close to actual measurements collected on 102 real-world neural architectures.[17] Figs. 18b and 18c illustrate that on both frameworks, 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").

[17]The fusion rules of PyTorch Mobile are very simple, as described in Section 3.2.1; thus, we omit the prediction results for the number of



(a) CPU, One Large Core (TFLite) (b) GPU (TFLite) (c) CPU, One Large Core (PyTorch) (d) GPU (PyTorch)

Figure 17: Predictions of ML Models (Real-World NAs)



(a) # of Kernels (TFLite) (b) MAPE on PowerVR GE8320 (TFLite) (c) MAPE on Adreno 640 (PyTorch)

Figure 18: Effectiveness of Kernel Fusion on Predictions

Kernel Selection. As introduced in Section 3.2.2, a convolution operation can be executed as different kernel implementations compatible with the its configuration and the target device. We deduce the kernels that ML frameworks select for convolution operations and train separate predictors for each (since they exhibit different performance characteristics). Fig. 19a shows the considerable error reduction achieved by accounting for kernel selection of TFLite on PowerVR GE8320, for real-world neural architectures that support Winograd kernels; Fig. 19b confirms that this reduction is due to more accurate predictions of the latency of Winograd kernels. Fig. 19c represents the effects of considering kernel fusion in PyTorch Mobile, where Lasso achieves more significant improvement than other ML approaches. We attribute this difference to the fact that the kernel selection in PyTorch Mobile is relatively simple: As shown in Algorithm B.1, the choice of kernel is only dependent on one or two features, which the non-linear ML models can capture. The linear model Lasso shows limited potential to represent such relations; for example, we observed that the depthwise convolutions with kernel size 7x7 are substantially slower than the ones with kernel size 3x3 or 5x5 due to lack of support for efficient implementations in PyTorch Mobile.

5.5. Case Study: Limited Training Data

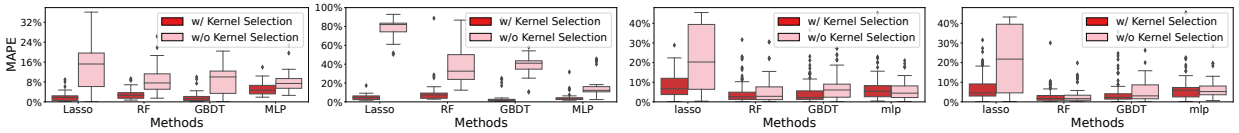
The high cost of collecting sufficient training data is a common criticism of ML approaches to predict the latency of neural architectures during NAS [21]. In this section, we study the effects of training set size on different ML approaches, illustrating the benefits of a simple model when training data is limited.

5.5.1. Comparison of ML Approaches

Fig. 20 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).[18] 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 decreased from 900 to 30. Consequently, when training data is limited, e.g., in Figs. 20a, 20b, 20e and 20f for training size of 30, a simple model such as Lasso achieves similar or better accuracy than some 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. 20c, 20d, 20g and 20h. 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. 20a, 20b, 20e and 20f for training set size of 900. In the case of data shift (Figs. 20c and 20d), MLP achieves the worst predictions on TFLite 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

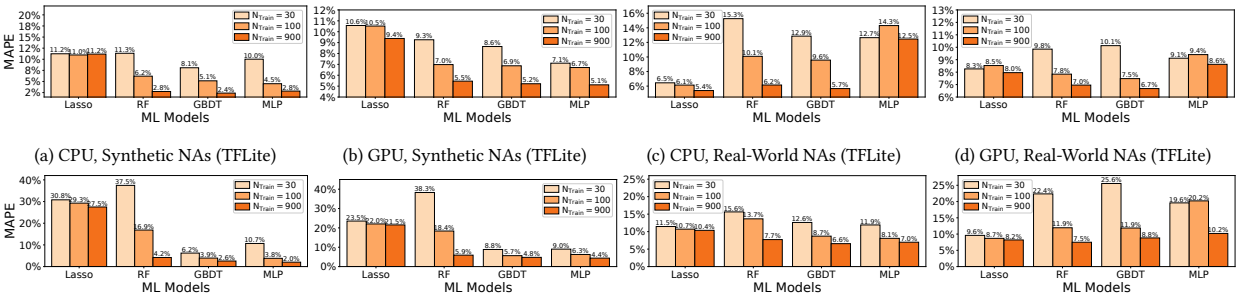
kernels.

[18] MAPEs for each platform are reported in Tables C.4 to C.7 of the Appendix.



(a) End-to-end Latency (TFLite) (b) Winograd Kernel (TFLite) (c) End-to-end Latency (PyTorch) (d) Convolution Kernels (PyTorch)

Figure 19: Prediction Error Reduction on PowerVR GE8320 by Accounting for Kernel Selection



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

(e) CPU, Synthetic NAs (PyTorch) (f) GPU, Synthetic NAs (PyTorch) (g) CPU, Real-World NAs (PyTorch) (h) GPU, Real-World NAs (PyTorch)

Figure 20: Prediction Errors on Synthetic or Real-World NAs for Different Synthetic Training Set Sizes

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, with an average MAPE across six platforms of 6.5% in TFLite (Fig. 20c) and 11.5% in PyTorch Mobile (Fig. 20g). As pointed out by prior work [21], 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

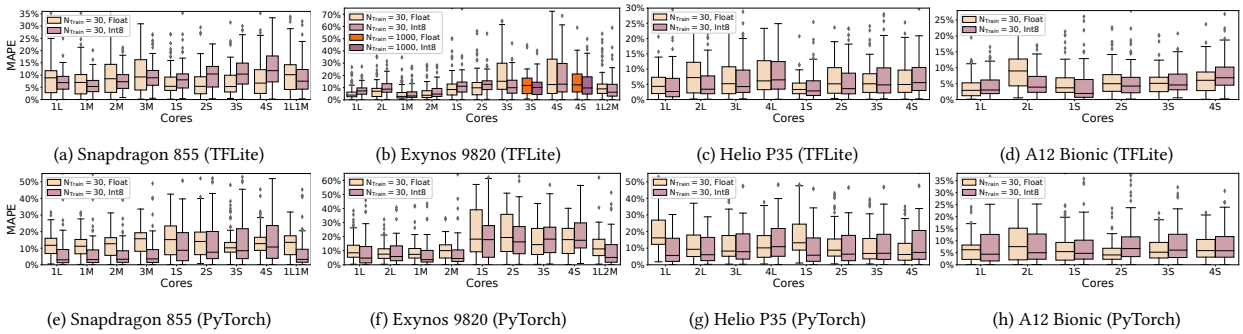
Next, we thoroughly evaluate the predictions of Lasso with a limited training set size (i.e., 30 neural architectures) on real-world neural architectures, across a broad range of scenarios for hardware heterogeneity.

Fig. 21 shows the prediction error of Lasso on real-world neural architectures, across various combinations of cores and data representations.[19] Generally, the trend of prediction errors for homogeneous and heterogeneous clusters is similar to the results in Fig. 13. 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 for TFLite; 22.3% on Exynos 9820, 16.2% on Snapdragon 855, 20.4% on Helio P35, and 9.9% on A12 Bionic for PyTorch Mobile. 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. For example, by adding more training data, MAPEs can be reduced to less than 14.8% in Fig. 21b. Fig. 22 shows the predictions of Lasso across multiple mobile GPUs. The maximum MAPE is 11.0% on Mali G76 for TFLite and 12.6% on A12 Bionic for PyTorch Mobile.

Since all the features are standardized, we use the magnitude of weights in the Lasso model to analyze the importance of different features. In general, on all devices of both frameworks, 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[20], except the following. For

[19]For clarity of presentation, we omit some outliers (<4% data points per configuration) and report plots with all data points in Fig. A.35 of the Appendix.

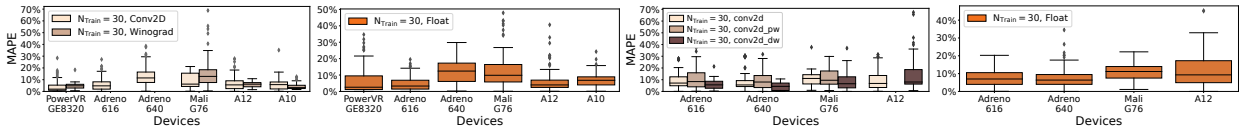
[20]However, as noted in Section 1, FLOPs alone is not an accurate proxy metric for the actual latency.



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

(e) Snapdragon 855 (PyTorch) (f) Exynos 9820 (PyTorch) (g) Helio P35 (PyTorch) (h) A12 Bionic (PyTorch)

Figure 21: Predictions of Lasso on Multicore CPUs (Real-World NAs)



(a) Convolution (TFLite) (b) End-to-end (TFLite) (c) Convolution (PyTorch) (d) End-to-end (PyTorch)

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

CPUs on PyTorch Mobile, the most critical features of convolutions are *FLOPs* and *output size*, as PyTorch Mobile implements a caching allocator for CPU memory [80] which saves the cost of rewriting kernel weights into memory over multiple inference runs; consequently, the feature correlated to memory access is the output size rather than the kernel size for CPUs on PyTorch Mobile. 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

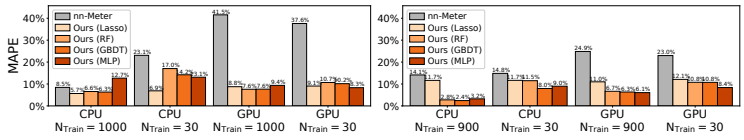
Lastly, we quantitatively compare our results with the state-of-the-art inference latency predictor nn-Meter [22] and conduct evaluations on the existing NAS benchmark dataset NATSBench [38].

5.6.1. Predictors: nn-Meter

As noted in Section 1, nn-Meter [22] 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 Appendix C. Therefore, to achieve a fair comparison, we ran the source code of nn-Meter [81] 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;[21] (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.

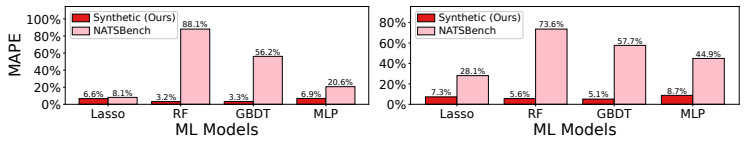
Fig. 23 compares nn-Meter predictions (the average MAPE across four Android platforms) to those of our approach (using different ML models); as can be seen, our approach outperforms nn-Meter on both 102 real-world NAs and 100 NAs from our synthetic dataset across different sizes of training data. An important reason is that our approach considers a broader set of features for operations. For example, nn-Meter does not distinguish grouped convolutions from standard convolutions; a grouped convolution splits the input tensor into multiple groups of

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



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

Figure 23: Comparison with nn-Meter



(a) CPU (b) GPU

Figure 24: Predictions on 44 Real-world NAs w/o DW-Conv

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 [14, 22], the NATSBench [38] 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 [82]; 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. 24 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.

5.7. Discussion and Threats to Validity

Hardware heterogeneity and ML frameworks optimizations are the main threats to validity for our work, as described next.

Due to hardware heterogeneity, ML frameworks may select entirely different kernel implementations on different devices (e.g., Winograd kernels as discussed in Section 3.2.2). As a result, using latency measurements collected on one device to obtain latency predictions on a different device may be inaccurate when different kernels and operations are executed. Even when the executed kernels are the same, the different hardware architectures (including

CPU/GPU frequency, number of cores, memory bandwidth, and cache sizes) may result in different performance characteristics. To make predictions for a new hardware platform without collecting latency measurements, a model would need to account for kernel selection and for the parameters of the hardware architecture. We are exploring the development of more general cross-device prediction models as future work.

In addition, changes to the implementation of kernels within ML frameworks may affect their performance, requiring new data collection and retraining of latency predictors. Such changes typically occur when optimized building blocks of NAs are proposed and integrated into ML frameworks. For example, recent advancements in Vision Transformer motivated the introduction of optimized implementations of multi-head attention layers in ML frameworks. Our approach requires new latency measurements to train predictors for these new operations.

6. Related Work

As observed in Section 1, related work has only limited consideration of the following challenges.

Limited consideration of hardware heterogeneity. Most existing work aims at latency predictions of training or inference tasks on cloud GPUs [14, 18, 20, 19, 17, 25, 26, 27] or embedded GPUs [23, 24], where Nvidia GPUs dominate the market for ML workloads. Instead, our paper studies multiple mainstream mobile platforms from different manufacturers, and tackles hardware heterogeneity across these platforms. Some recent works [21, 22, 15] focus on performance prediction on mobile CPUs, but are limited only to a single core with floating-point representation. Instead, our work evaluates inference latency of mobile CPUs across a broad range of realistic scenarios, including the utilization of *multiple heterogeneous CPU* cores, and both *floating-point and integer representations*.

Limited consideration of ML framework optimizations. The majority of existing work [18, 20, 15] proposes to predict latency based on the features extracted from neural architectures and hardware, but neglects the effects of ML framework optimizations. As identified by our results, accounting for these optimizations results in significant improvements of the predictions for real-world neural architectures across multiple ML approaches. Since ML framework optimizations cannot be analyzed on Nvidia cloud and edge GPUs (cuDNN is not open-source [83]), recent work [22] proposes a black-box approach to learn their policies (i.e., the algorithms for kernel fusion). In contrast, on mobile platforms, ML frameworks use open-source algorithms and custom kernels to support a broad range of heterogeneous GPUs; we highlight the optimizations in both TFLite and PyTorch Mobile, accurately inferring the actual kernels used after compilation without deploying and compiling NN models on actual devices.

Next, different approaches exist in the literature to model inference latency. Some works [14, 17, 16] adopt ML approaches to predict end-to-end latency of neural architectures by encoding *the entire neural architecture* as a single vector of input features; this approach, however, requires complicated ML techniques as well as large amounts of training data. In contrast, we make latency predictions for each component of the neural architecture, allowing simple ML algorithms that require less training data and are easier to interpret (e.g., in the case of Lasso) for understanding and development. Similarly to our work, component-wise approaches are used by [22, 15] and analytical performance models of computation and memory access also exist in the literature [25, 26, 27], but both lines of work have limited consideration of hardware heterogeneity and ML framework optimizations, as described above.

Finally, while our paper predicts latency of NAs that are *static* during inference (a common scenario in practice), *dynamic neural networks* (or *adaptive neural networks*) disable parts of the NA based on the complexity of the specific input example [84, 85], to ensure that inference latency is sufficiently low for a target application. In this case, performance bottlenecks may depend on the input data, requiring performance testing [86]. Our approach can, in principle, be extended to dynamic NAs by predicting latency of individual blocks selected at runtime. In this context, our approach of estimating latency for individual blocks is beneficial in contrast to end-to-end black-box models that would be difficult to adapt to this scenario. We plan to explore these settings in future work.

7. 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) and for different ML frameworks (TFLite and PyTorch Mobile). 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).

Acknowledgments

This work was supported in part by the NSF CNS-1816887, CCF-1763747, and IIS-1833137 awards.

References

[1] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, Q. V. Le, H. Adam, Searching for MobileNetV3, in: Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2019.

[2] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, Q. V. Le, MnasNet: Platform-Aware Neural Architecture Search for Mobile, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019.

[3] M. Tan, Q. Le, EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, in: International conference on machine learning, PMLR, 2019, pp. 6105–6114.

[4] B. Zoph, Q. V. Le, Neural Architecture Search with Reinforcement Learning, in: International Conference on Learning Representations, ICLR, 2017.

[5] M. Lin, P. Wang, Z. Sun, H. Chen, X. Sun, Q. Qian, H. Li, R. Jin, Zen-nas: A zero-shot nas for high-performance image recognition, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2021, pp. 347–356.

[6] J. Mellor, J. Turner, A. Storkey, E. J. Crowley, Neural architecture search without training, in: International Conference on Machine Learning, PMLR, 2021, pp. 7588–7598.

[7] H. Tanaka, D. Kunin, D. L. Yamins, S. Ganguli, Pruning neural networks without any data by iteratively conserving synaptic flow, Advances in neural information processing systems 33 (2020) 6377–6389.

[8] X. Tang, S. Han, L. L. Zhang, T. Cao, Y. Liu, 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.

[9] B. Zoph, V. Vasudevan, J. Shlens, Q. V. Le, Learning transferable architectures for scalable image recognition.

[10] N. Ma, X. Zhang, H.-T. Zheng, J. Sun, ShuffleNet v2: Practical guidelines for efficient cnn architecture design, in: Proceedings of the European conference on computer vision (ECCV), 2018, pp. 116–131.

[11] H. Cai, C. Gan, T. Wang, Z. Zhang, S. Han, Once-for-All: Train One Network and Specialize it for Efficient Deployment, in: International Conference on Learning Representations, ICLR, 2019.

[12] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, P. Vajda, M. Uyttendaele, N. K. Jha, ChamNet: Towards Efficient Network Design Through Platform-Aware Model Adaptation, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019.

[13] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, K. Keutzer, 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), 2019.

[14] S. Abbasi, A. Wong, M. J. Shafiee, MAPLE: Microprocessor a Priori for Latency Estimation, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops, 2022.

[15] H. Cai, L. Zhu, S. Han, ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware, in: International Conference on Learning Representations, ICLR, 2019.

[16] L. Dudziak, T. Chau, M. Abdelfattah, R. Lee, H. Kim, N. Lane, BRP-NAS: Prediction-based NAS using GCNs, in: Advances in Neural Information Processing Systems, Vol. 33, 2020, pp. 10480–10490.

[17] Y. Gao, X. Gu, H. Zhang, H. Lin, M. Yang, Runtime Performance Prediction for Deep Learning Models with Graph Neural Network, Tech. rep., Technical Report MSR-TR-2021-3. Microsoft (2021).

[18] X. Y. Geoffrey, Y. Gao, P. Golikov, G. Pekhimenko, Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training, in: USENIX Annual Technical Conference, 2021, pp. 503–521.

[19] U. U. Hafeez, A. Gandhi, Empirical Analysis and Modeling of Compute Times of CNN Operations on AWS Cloud, in: 2020 IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2020, pp. 181–192.

[20] D. Justus, J. Brennan, S. Bonner, A. S. McGough, Predicting the Computational Cost of Deep Learning Models, in: 2018 IEEE International Conference on Big Data (Big Data), 2018, pp. 3873–3882.

[21] B. Lu, J. Yang, W. Jiang, Y. Shi, S. Ren, 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.

[22] L. L. Zhang, S. Han, J. Wei, N. Zheng, T. Cao, Y. Yang, Y. Liu, 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, 2021, pp. 81–93.

[23] H. Bouzidi, H. Ouarnoughi, S. Niar, A. A. E. Cadi, Performance prediction for convolutional neural networks on edge GPUs, in: Proceedings of the 18th ACM International Conference on Computing Frontiers, 2021, pp. 54–62.
[24] N. Bouhali, H. Ouarnoughi, S. Niar, A. A. El Cadi, 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, 2021, pp. 59–65.
[25] H. Qi, E. R. Sparks, A. Talwalkar, Paleo: A Performance Model for Deep Neural Networks, in: Proceedings of the International Conference on Learning Representations, 2017.
[26] J. Li, R. Ma, V. S. Mailthody, C. Samplawski, B. Marlin, S. Chen, S. Yao, T. Abdelzaher, Towards an Accurate Latency Model for Convolutional Neural Network Layers on GPUs, in: MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM), IEEE, 2021, pp. 904–909.
[27] S. Lym, D. Lee, M. O'Connor, N. Chatterjee, M. Erez, 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, 2019, pp. 293–303.
[28] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, T. Leyvand, H. Lu, Y. Lu, L. Qiao, B. Reagen, J. Spisak, F. Sun, A. Tulloch, P. Vajda, X. Wang, Y. Wang, B. Wasti, Y. Wu, R. Xian, S. Yoo, P. Zhang, Machine learning at Facebook: Understanding inference at the edge, in: 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), IEEE, 2019, pp. 331–344.
[29] X. Zhang, X. Zhou, M. Lin, J. Sun, ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.
[30] Google, Tensorflow lite: Ml for mobile and edge devices, `https://www.tensorflow.org/lite` (2022).
[31] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, M. Grundmann, On-Device Neural Net Inference with Mobile GPUs, arXiv preprint arXiv:1907.01989 (2019).
[32] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al., PyTorch: An imperative style, high-performance deep learning library, Advances in neural information processing systems 32 (2019).
[33] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, T. Mitra, 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.
[34] W. Niu, J. Guan, Y. Wang, G. Agrawal, B. Ren, 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, 2021, pp. 883–898.
[35] A. Lavin, S. Gray, Fast Algorithms for Convolutional Neural Networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.
[36] M. Syed, A. A. Srinivasan, Generalized Latency Performance Estimation for Once-For-All Neural Architecture Search, arXiv preprint arXiv:2101.00732 (2021).
[37] P. Bryzgalov, T. Maeda, Y. Shigeto, 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, 2021, pp. 11–18.
[38] X. Dong, L. Liu, K. Musial, B. Gabrys, NATS-Bench: Benchmarking NAS Algorithms for Architecture Topology and Size, IEEE transactions on pattern analysis and machine intelligence 44 (7) (2021) 3634–3646.
[39] M. Nagel, M. Fournarakis, R. A. Amjad, Y. Bondarenko, M. Van Baalen, T. Blankevoort, A white paper on neural network quantization, arXiv preprint arXiv:2106.08295 (2021).
[40] Z. Li, M. Paolieri, L. Golubchik, Predicting Inference Latency of Neural Architectures on Mobile Devices, in: Proceedings of ICPE 2023, ACM, 2023, pp. 99–112. doi:10.1145/3578244.3583735.
[41] J. H. Friedman, Greedy Function Approximation: A Gradient Boosting Machine, The Annals of Statistics 29 (5) (2001) 1189–1232.
[42] R. Tibshirani, Regression shrinkage and selection via the lasso, Journal of the Royal Statistical Society: Series B (Methodological) 58 (1) (1996) 267–288.
[43] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, arXiv preprint arXiv:1704.04861 (2017).
[44] K. He, X. Zhang, S. Ren, J. Sun, Deep Residual Learning for Image Recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.
[45] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, L.-C. Chen, MobileNetV2: Inverted Residuals and Linear Bottlenecks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.
[46] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko, 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), 2018.
[47] Apple, 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 (2016).
[48] W. Brendel, M. Bethge, Approximating CNNs with Bag-of-local-Features models works surprisingly well on ImageNet, arXiv preprint arXiv:1904.00760 (2019).
[49] S. Ioffe, C. Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, in: Proceedings of the 32nd International Conference on Machine Learning, Vol. 37, PMLR, 2015, pp. 448–456.
[50] G. Huang, Z. Liu, L. van der Maaten, K. Q. Weinberger, Densely Connected Convolutional Networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.
[51] S. Zagoruyko, N. Komodakis, DiracNets: Training very deep neural networks without skip-connections, arXiv preprint arXiv:1706.00388 (2017).
[52] F. Yu, D. Wang, E. Shelhamer, T. Darrell, Deep Layer Aggregation, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.
[53] Z. Qin, Z. Zhang, X. Chen, C. Wang, Y. Peng, Fd-MobileNet: Improved mobilenet with a fast downsampling strategy, in: 2018 25th IEEE

International Conference on Image Processing (ICIP), IEEE, 2018, pp. 1363–1367.
[54] K. Han, Y. Wang, Q. Tian, J. Guo, C. Xu, C. Xu, Ghostnet: More features from cheap operations, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020.
[55] P. Chao, C.-Y. Kao, Y.-S. Ruan, C.-H. Huang, Y.-L. Lin, HarDNet: A Low Memory Traffic Network, in: Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV), 2019.
[56] J. Wang, K. Sun, T. Cheng, B. Jiang, C. Deng, Y. Zhao, D. Liu, Y. Mu, M. Tan, X. Wang, W. Liu, B. Xiao, Deep high-resolution representation learning for visual recognition, IEEE transactions on pattern analysis and machine intelligence 43 (10) (2020) 3349–3364.
[57] R. J. Wang, X. Li, C. X. Ling, Pelee: A real-time object detection system on mobile devices, Advances in neural information processing systems 31 (2018).
[58] K. He, X. Zhang, S. Ren, J. Sun, 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, 2016, pp. 630–645.
[59] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, P. Dollar, Designing Network Design Spaces, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2020.
[60] S. Xie, R. Girshick, P. Dollar, Z. Tu, K. He, Aggregated Residual Transformations for Deep Neural Networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.
[61] J. Hu, L. Shen, G. Sun, Squeeze-and-Excitation Networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.
[62] D. Stamoulis, R. Ding, D. Wang, D. Lymberopoulos, B. Priyantha, J. Liu, D. Marculescu, Single-Path NAS: Designing Hardware-Efficient ConvNets in Less Than 4 Hours, in: Machine Learning and Knowledge Discovery in Databases, Springer, Cham, 2020, pp. 481–497.
[63] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, K. Keutzer, SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size, arXiv preprint arXiv:1602.07360 (2016).
[64] Y. Lee, J.-w. Hwang, S. Lee, Y. Bae, J. Park, 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, 2019.
[65] Sandbox for training deep learning networks, https://github.com/osmr/imgclsmob (2021).
[66] Google, 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 (2022).
[67] Google, TensorFlow Lite: Multithreading for Convolutions with the Ruy Library, https://github.com/google/ruy/blob/38a926/ruy/trmul.cc#L390, accessed: 2022-08-05 (2022).
[68] PyTorch, Caffe2: ThreadPool Implementation, https://github.com/pytorch/pytorch/blob/v2.0.0/caffe2/utils/threadpool/ThreadPool.cc#L201, accessed: 2023-09-20 (2023).
[69] Mobile ML Benchmark, https://github.com/qed-usc/mobile-ml-benchmark/, accessed: 2024-03-08 (2024).
[70] QNNPACK: Quantized Neural Networks PACKage, https://github.com/pytorch/pytorch/tree/main/aten/src/ATen/native/quantized/cpu/qnnpack, accessed: 2023-09-20 (2023).
[71] Google, XNNPACK: High-efficiency floating-point neural network inference operators for mobile, server, and Web, https://github.com/google/XNNPACK, accessed: 2023-09-20 (2023).
[72] Google, TensorFlow Lite: Kernel Fusion Implementation, https://github.com/tensorflow/tensorflow/blob/v2.9.0/tensorflow/lite/delegates/gpu/common/gpu_model.cc#L421, accessed: 2022-08-05 (2022).
[73] Google, PyTorch Mobile: Vulkan Backend Optimization, https://github.com/pytorch/pytorch/blob/v2.0.0/torch/csrc/jit/passes/vulkan_rewrite.cpp#L344, accessed: 2024-04-29 (2023).
[74] Z. Li, M. Paolieri, L. Golubchik, A Benchmark for ML Inference Latency on Mobile Devices, in: Proceedings of EdgeSys 2024, ACM, 2024, pp. 31–36. doi:10.1145/3642968.3654818.
[75] K. P. Murphy, Machine learning: a probabilistic perspective, MIT press, 2012.
[76] Google, Tflite model benchmark tool, https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/benchmark, accessed: 2022-07-12 (2022).
[77] Google, 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 (2022).
[78] Apple, Discover metal debugging, profiling, and asset creation tools, https://developer.apple.com/videos/play/wwdc2021/10157, accessed: 2022-10-06 (2021).
[79] PyTorch, Kineto pytorch profiler, https://github.com/pytorch/kineto, accessed: 2023-09-26 (2023).
[80] PyTorch, Pytorch cpu caching allocator, https://github.com/pytorch/pytorch/blob/main/c10/mobile/CPUCachingAllocator.h, accessed: 2023-09-26 (2023).
[81] M. R. nn Meter Team, nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices, https://github.com/microsoft/nn-Meter (2021).
[82] A. Krizhevsky, G. Hinton, Learning multiple layers of features from tiny images (2009).
[83] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, E. Shelhamer, cudnn: Efficient primitives for deep learning, arXiv preprint arXiv:1410.0759 (2014).
[84] T. Bolukbasi, J. Wang, O. Dekel, V. Saligrama, Adaptive Neural Networks for Efficient Inference, in: Proceedings of ICML 2017, Vol. 70 of Proceedings of Machine Learning Research, PMLR, 2017, pp. 527–536.
[85] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, Y. Wang, Dynamic Neural Networks: A Survey, IEEE Trans. Pattern Anal. Mach. Intell. 44 (11) (2022) 7436–7456. doi:10.1109/TPAMI.2021.3117837.
[86] S. Chen, M. Haque, C. Liu, W. Yang, DeepPerform: An Efficient Approach for Performance Testing of Resource-Constrained Neural Networks, in: Proceedings of ASE 2022, ACM, 2022, pp. 31:1–31:13. doi:10.1145/3551349.3561158.

A. Supplementary Data

In this appendix, we include supplementary data from our measurements and prediction results. This data is provided here for completeness and includes measurements and predictions on two additional platforms (Snapdragon 710 and A10 Fusion) as well as the full set of outliers that were omitted in some of the figures of the main text for clarity of presentation.

Figs. A.25 to A.28 present the measurements on Snapdragon 710 and A10 Fusion, which are omitted in Figs. 2 to 5 (Section 3) for clarity of presentation; correspondingly, the predictions on these platforms are reported in Figs. A.29 and A.30, which are omitted in Figs. 13 and 21 (Section 5). Fig. A.31 depicts end-to-end latency of real-world NAs on six platforms for different multi-core configurations, including the outliers omitted in Fig. 2 for clarity of presentation (Section 3.1.1). Figs. A.32 and A.33 present the speedup of kernel fusion on end-to-end latency and on each type of operations, respectively, including the small set of outliers omitted in Figs. 6b and 7 (Section 3.2.1).

Tables C.4 to C.7 report the complete MAPEs of end-to-end latency predictions on each hardware platform, for synthetic and real-world neural architectures, on TFLite and PyTorch Mobile, respectively, across different ML approaches, with varying training set sizes. This detailed data corresponds to the results in Fig. 20 in the main text where these errors were averaged across hardware platforms. For predictions on different CPU core combinations and with both floating-point and integer representations, Fig. A.34 shows the end-to-end latency predictions of GBDT for synthetic neural architectures on various core combinations, including the small set of outliers omitted in Fig. 13 (Section 5.2); Fig. A.35 presents the end-to-end latency predictions of Lasso for real-world neural architectures on various core combinations, including the small set of outliers omitted in Fig. 21 (Section 5.5.2).

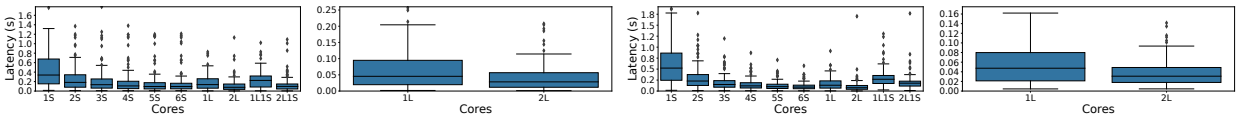
B. Details of Kernel Fusion and Kernel Selection

In this section, we elaborate the algorithms of kernel fusion (Section 3.2.1) and kernel selection (Section 3.2.2) in PyTorch Mobile and TFLite. Algorithm B.1 shows the implementation details of Vulkan kernel selection in PyTorch Mobile: when the convolution layer satisfies the criteria for depthwise convolutions (i.e., the group size is equal to input channels), a depthwise kernel can be applied based on kernel shape; also, a special pointwise kernel is implemented for convolutions with kernel shape 1x1.

Algorithm B.2 presents the implementation details of kernel fusion in TFLite GPU Delegate: two operations of the computational graph are fused when (1) the first operation has only one output tensor (Line 5), (2) the second operation is the only operation in the graph using this output tensor (Line 14), (3) the second operation uses this output tensor as its first input and produces a single output (Line 22), and (4) the next operation has a compatible type (Line 24).

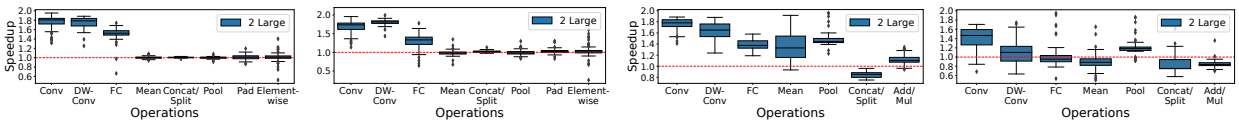
Algorithm B.3 summarizes the criteria used by TFlite to enable the use of the Winograd algorithm for convolution operations on GPUs: when the input tensor and kernel size of a convolution operation both satisfy certain *hardware-dependent* criteria (i.e., CheckWinograd), the kernel of Winograd is selected for the operation. For example, Table B.3 presents three convolution operations in ResNet16, which all 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), but meet the requirements for Mali and PowerVR GPUs (Line 21). For convolution (2), total_tiles is too small for Adreno 600-level 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).

Another operation allowing optimized implementations in TFLite is grouped convolution, which consists of three stages: (1) splitting the input tensor over channel size, (2) performing a convolution on each resulting tensor



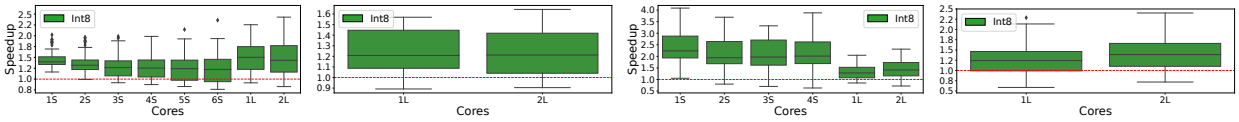
(a) Snapdragon 710 (TFLite) (b) A10 Fusion (TFLite) (c) Snapdragon 710 (PyTorch) (d) A10 Fusion (PyTorch)

Figure A.25: Effects of Multicore on End-to-end Latency



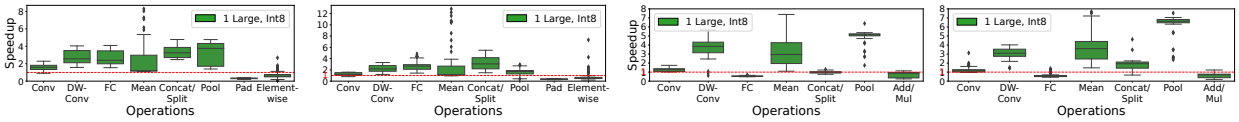
(a) Snapdragon 710 (TFLite) (b) A10 Fusion (TFLite) (c) Snapdragon 710 (PyTorch) (d) A10 Fusion (PyTorch)

Figure A.26: Effects of Homogeneous Multicore on Operation-wise Latency (Speedup over One Core)



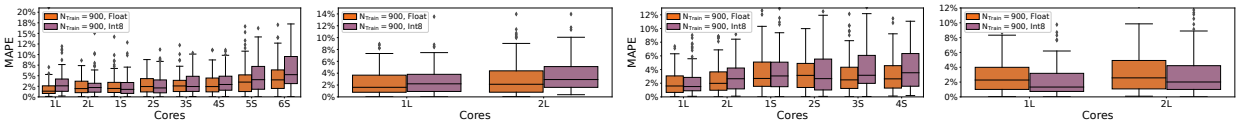
(a) Snapdragon 710 (TFLite) (b) A10 Fusion (TFLite) (c) Snapdragon 710 (PyTorch) (d) A10 Fusion (PyTorch)

Figure A.27: Effects of Quantization on End-to-end Latency



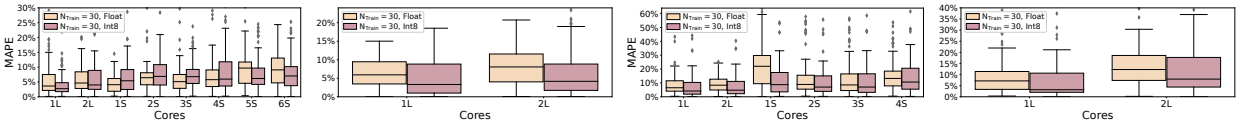
(a) Snapdragon 710 (TFLite) (b) A10 Fusion (TFLite) (c) Snapdragon 710 (PyTorch) (d) A10 Fusion (PyTorch)

Figure A.28: Effects of Quantization on Operation-wise Latency



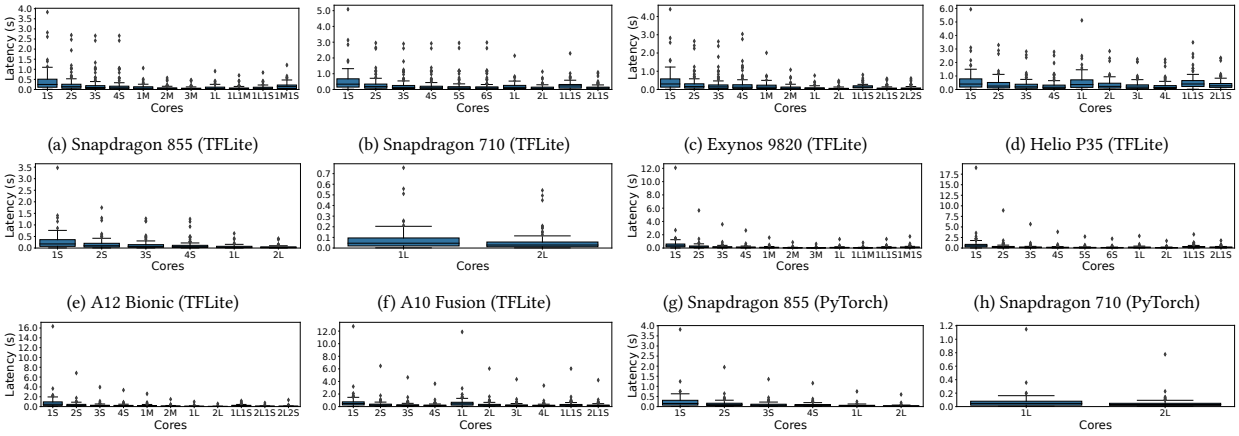
(a) Snapdragon 710 (TFLite) (b) A10 Fusion (TFLite) (c) Snapdragon 710 (PyTorch) (d) A10 Fusion (PyTorch)

Figure A.29: Predictions of GBDT on CPUs (Synthetic NAs)



(a) Snapdragon 710 (TFLite) (b) A10 Fusion (TFLite) (c) Snapdragon 710 (PyTorch) (d) A10 Fusion (PyTorch)

Figure A.30: Predictions of Lasso on CPUs (Real-world NAs)

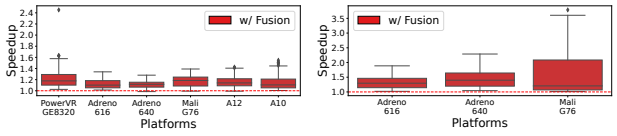


(a) Snapdragon 855 (TFLite) (b) Snapdragon 710 (TFLite) (c) Exynos 9820 (TFLite) (d) Helio P35 (TFLite)

(e) A12 Bionic (TFLite) (f) A10 Fusion (TFLite) (g) Snapdragon 855 (PyTorch) (h) Snapdragon 710 (PyTorch)

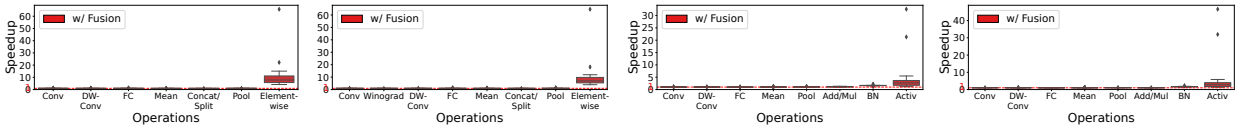
(i) Exynos 9820 (PyTorch) (j) Helio P35 (PyTorch) (k) A12 Bionic (PyTorch) (l) A10 Fusion (PyTorch)

Figure A.31: Effects of Multicore on End-to-end Latency



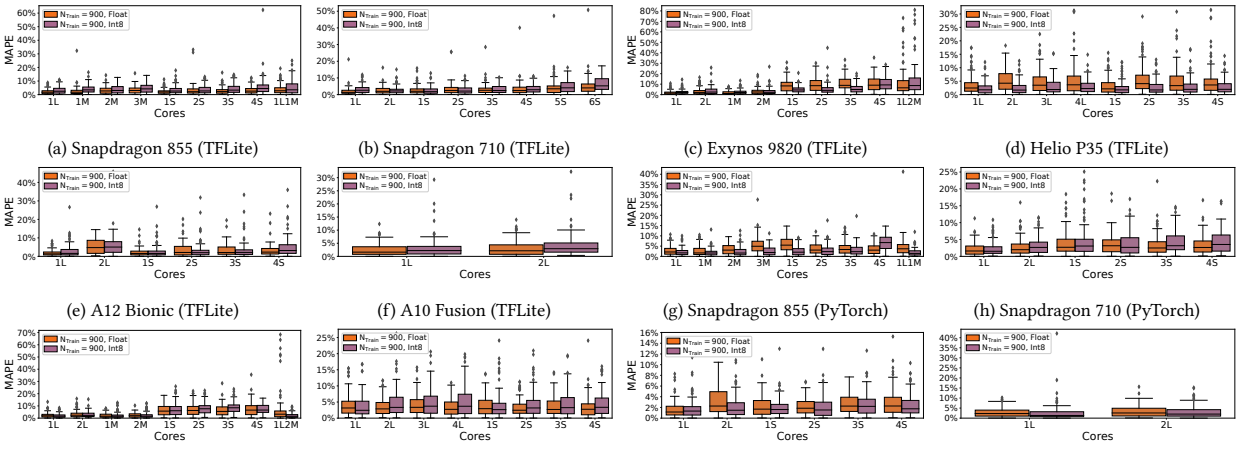
(a) TFLite (b) PyTorch

Figure A.32: Effects of Kernel Fusion on End-to-end Latency



(a) Adreno 640 (TFLite) (b) A12 Bionic (TFLite) (c) Adreno 640 (PyTorch) (d) Adreno 616 (PyTorch)

Figure A.33: Effects of Kernel Fusion on Operation-wise Latency



(a) Snapdragon 855 (TFLite) (b) Snapdragon 710 (TFLite) (c) Exynos 9820 (TFLite) (d) Helio P35 (TFLite)

(e) A12 Bionic (TFLite) (f) A10 Fusion (TFLite) (g) Snapdragon 855 (PyTorch) (h) Snapdragon 710 (PyTorch)

(i) Exynos 9820 (PyTorch) (j) Helio P35 (PyTorch) (k) A12 Bionic (PyTorch) (l) A10 Fusion (PyTorch)

Figure A.34: Predictions of GBDT on CPUs (Synthetic NAs)

Index	Configurations			Conditions in Algorithm B.3			If use Winograd	
	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 B.3: Applicability of TFLite Winograd Kernels to Convolutions in ResNet16 (1 group, 3x3 kernel, stride 1)

Algorithm B.1: Conv Kernel Selection in PyTorch Mobile Vulkan Backend

```
SelectConv2DKernel(op_info)
if op_info.is_transpose
    return Kernel(conv2d)
if op_info.groups == op_info.input_channel and op_info.output_channel ≠ 1
    if op_info.kernel_shape == 3x3
        return Kernel(conv2d_dw_3x3)
    else if op_info.kernel_shape == 5x5
        return Kernel(conv2d_dw_5x5)
    else
        return Kernel(conv2d_dw)
if op_info.kernel_shape == 1x1
    return Kernel(conv2d_pw)
return Kernel(conv2d)
```

(i.e., on each group), and (3) concatenating all output tensors. A naive implementation of grouped convolution uses an independent convolution kernel for each group, and two kernels for the split and concatenation operations. TFLite supports an optimized implementation of *grouped_convolution_2d* using only one kernel. Fig. B.36 illustrates the performance improvement of the optimized *grouped_convolution_2d* kernel over a naive implementation; we observe substantial improvements, e.g., 2.96x speedup for RegNetX004 on PowerVR GE8320.

C. Evaluations on nn-Meter

In this appendix, we report details of our quantitative comparison with related work, specifically nn-Meter. Recall that in the main text we provided a quantitative comparison with nn-Meter on *our dataset* (Section 5.6.1). Here, we provide details of why such a comparison was appropriate.

A natural approach to a quantitative comparison with nn-Meter would be to use nn-Meter's pre-trained predictors on Pixel4 Snapdragon 855 CPU (with a single thread) and Adreno 640 GPU (as provided in [81]), which is what we did initially. However, this did not work out (and hence a, what we believe to be a more fair comparison, using our data set in the main text) for the following reasons. Specifically, initially we evaluated our approach on the same experimental setup as nn-Meter by using TFLite v2.1. Due to the lack of support for operations (e.g., grouped convolutions) on GPU delegate of TFLite v2.1, 17 real-world NAs failed to be fully executed on GPUs. As a result, in our evaluations we selected only 85 real-world NAs and (re)generated 1000 synthetic NAs by removing grouped convolutions in our NAS space. As depicted in Fig. B.37, which compares nn-Meter predictions to those of our approach (with different ML models, all trained on 900 synthetic NAs), our approach achieves much better results (on Pixel 4) on both real-world and synthetic NAs. We believe that the main reason is that the ground truth measurements used by nn-Meter to train the predictors are collected from their customized TFLite benchmark tool (also provided in [81]), which was compiled without specifying the ARM64 architecture (because this argument was optional in

Algorithm B.2: Kernel Fusion in TFLite GPU Delegate

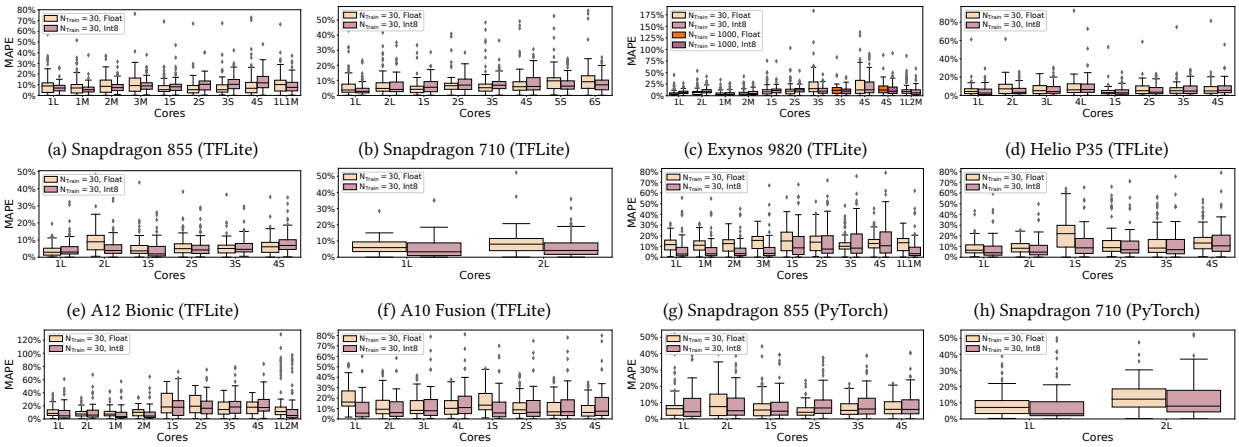
```
MergeNodes(nodes)
ready_tensors = []
for cur_node in nodes
    for dst_tensor in cur_node.dst_tensors
        ready_tensors.insert(dst_tensor)
    if cur_node.dst_tensors.size() ≠ 1
        continue
    candidate_nodes = []
    candidate_tensor_index = 0
    for next_node in nodes
        for k = 0 to next_node.src_tensors.size() - 1
            if next_node.src_tensors[k] == cur_node.dst_tensors[0]
                candidate_tensor_index = k
                candidate_nodes.insert(next_node)
    if candidate_nodes.size() ≠ 1 or candidate_tensor_index ≠ 0
        continue
    next_node = candidate_nodes[0]
    if next_node.src_tensors[0] ∈ ready_tensors
            and IsLinkable(next_node)
        Merge(cur_node, next_node)
        nodes.remove(cur_node)
return nodes

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

the earlier version of TFLite, but is now used by default)[22]. To better understand the poor predictions of nn-Meter in Fig. B.37, we compared the measurements from their customized benchmark tool with those of the standard TFLite benchmark tool (which we used to build our dataset, as noted in Section 4.3.1) and found that their customized benchmark tool consistently showed worse performance. This leads to nn-Meter's pre-trained predictors (trained on the data collected by their tool) giving much poorer predictions on our dataset.

Thus, as noted above, to achieve a fairer comparison, in the main text we took the route of reproducing their approach on our dataset (Section 5.6.1).

[22] We communicated with the authors of nn-Meter and we were given this information.



(a) Snapdragon 855 (TFLite) (b) Snapdragon 710 (TFLite) (c) Exynos 9820 (TFLite) (d) Helio P35 (TFLite)

(e) A12 Bionic (TFLite) (f) A10 Fusion (TFLite) (g) Snapdragon 855 (PyTorch) (h) Snapdragon 710 (PyTorch)

(i) Exynos 9820 (PyTorch) (j) Helio P35 (PyTorch) (k) A12 Bionic (PyTorch) (l) A10 Fusion (PyTorch)

Figure A.35: Predictions of Lasso on CPUs (Real-world NAs)

(a) Snapdragon 855 (Adreno 640) (b) Helio P35 (PowerVR GE8320)

Figure B.36: Effects of Using *grouped_convolution_2d* Kernels on End-to-end Latency in TFLite

(a) 85 Real-world NAs (b) 100 Synthetic NAs

Figure B.37: Comparison with the Pre-trained Predictors from nn-Meter on Pixel 4 (TFLite v2.1)

Approach	Training Size	Snapdragon 855		Exynos 9820		Snapdragon 710		Helio P35		A12 Bionic		A10 Fusion	
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Lasso	30	12.84%	17.95%	9.08%	10.29%	8.85%	14.15%	15.90%	6.05%	9.28%	8.23%	11.49%	6.72%
	100	12.93%	18.71%	8.87%	10.23%	8.72%	14.46%	14.88%	5.59%	8.93%	7.83%	11.48%	6.29%
	900	13.26%	16.36%	8.90%	9.63%	9.33%	12.67%	15.09%	5.31%	8.96%	6.90%	11.48%	5.41%
RF	30	10.71%	13.68%	13.52%	9.99%	11.83%	12.97%	9.98%	6.49%	8.69%	6.71%	13.38%	5.74%
	100	6.20%	9.43%	4.90%	8.58%	6.13%	11.47%	7.79%	3.83%	5.51%	4.77%	6.67%	3.78%
	900	2.83%	7.33%	2.82%	8.34%	2.29%	8.30%	3.09%	2.74%	2.46%	2.95%	2.99%	3.09%
GBDT	30	7.91%	12.52%	7.76%	9.59%	7.10%	15.97%	9.08%	4.93%	9.11%	4.57%	7.57%	4.28%
	100	3.97%	9.77%	4.36%	8.59%	4.73%	12.29%	5.45%	3.43%	5.79%	3.48%	6.61%	3.67%
	900	2.12%	7.60%	1.92%	8.41%	2.01%	6.56%	3.71%	2.77%	1.87%	2.96%	2.72%	2.99%
MLP	30	9.11%	10.02%	7.94%	8.55%	8.21%	10.12%	10.71%	4.84%	11.22%	4.70%	12.91%	4.41%
	100	4.03%	9.17%	3.84%	9.01%	3.07%	9.28%	6.61%	4.35%	4.72%	3.94%	4.71%	4.52%
	900	2.30%	6.37%	2.44%	8.19%	2.03%	6.35%	6.09%	3.35%	1.59%	3.13%	2.59%	3.30%

Table C.4: End-to-end Predictions on Synthetic Neural Architectures on TFLite (CPU Stands For a Large Core)

Algorithm B.3: 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
```

Approach	Training Size	Snapdragon 855		Exynos 9820		Snapdragon 710		Helio P35		A12 Bionic		A10 Fusion	
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Lasso	30	9.77%	12.04%	5.83%	12.68%	6.40%	4.78%	5.51%	6.79%	4.20%	6.51%	7.00%	6.73%
	100	8.23%	14.41%	4.85%	11.77%	7.08%	5.21%	4.87%	6.51%	4.36%	6.38%	7.46%	6.87%
	900	7.29%	12.10%	5.24%	12.28%	5.27%	4.59%	4.65%	6.06%	3.66%	6.07%	6.24%	6.67%
RF	30	14.79%	14.77%	20.15%	13.23%	14.37%	7.99%	18.86%	6.81%	11.95%	8.23%	11.44%	8.08%
	100	11.67%	9.94%	10.85%	11.24%	9.10%	5.72%	10.26%	7.19%	9.18%	7.00%	9.48%	5.95%
	900	7.43%	7.24%	8.01%	11.39%	5.02%	5.60%	5.71%	6.01%	5.61%	6.74%	5.12%	4.80%
GBDT	30	12.20%	12.13%	16.57%	12.50%	11.92%	9.03%	16.11%	6.92%	11.46%	11.03%	8.90%	9.14%
	100	12.32%	7.83%	10.28%	12.32%	7.38%	5.24%	10.19%	6.44%	8.72%	8.08%	8.53%	4.95%
	900	6.38%	6.68%	7.86%	11.87%	4.79%	4.15%	4.80%	5.86%	5.45%	6.23%	4.70%	5.28%
MLP	30	14.87%	7.79%	13.18%	9.94%	11.35%	8.52%	13.01%	7.03%	11.39%	12.43%	12.12%	8.98%
	100	18.31%	9.05%	16.61%	10.51%	12.35%	10.37%	12.25%	7.91%	15.09%	10.41%	11.22%	8.23%
	900	14.48%	7.59%	14.23%	11.06%	16.59%	11.06%	10.22%	7.08%	10.71%	8.72%	8.55%	6.24%

Table C.5: End-to-end Predictions on Real-world Neural Architectures on TFLite (CPU Stands For a Large Core)

Approach	Training Size	Snapdragon 855		Exynos 9820		Snapdragon 710		Helio P35		A12 Bionic		A10 Fusion	
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Lasso	30	27.46%	24.70%	29.15%	15.73%	43.22%	—	26.23%	23.86%	26.23%	29.82%	33.27%	—
	100	29.02%	24.31%	25.88%	14.22%	43.17%	—	23.13%	24.52%	24.80%	25.09%	30.44%	—
	900	27.52%	23.46%	24.51%	14.95%	38.25%	—	22.16%	23.00%	23.36%	24.72%	29.93%	—
RF	30	36.82%	44.52%	36.33%	33.06%	44.46%	—	34.64%	43.85%	33.83%	31.70%	39.28%	—
	100	18.15%	20.57%	15.02%	15.11%	22.88%	—	15.80%	19.33%	13.49%	18.51%	16.36%	—
	900	4.69%	5.90%	3.69%	4.36%	6.92%	—	3.54%	3.01%	2.71%	10.39%	4.02%	—
GBDT	30	6.29%	8.47%	5.08%	7.44%	9.06%	—	6.23%	5.77%	4.49%	13.67%	6.04%	—
	100	4.33%	5.38%	3.19%	3.67%	6.06%	—	3.76%	2.16%	2.52%	11.40%	3.82%	—
	900	2.79%	4.17%	2.33%	2.58%	4.02%	—	2.16%	1.44%	1.66%	10.89%	2.88%	—
MLP	30	6.97%	4.92%	3.76%	6.84%	35.74%	—	5.31%	9.54%	3.48%	14.79%	9.09%	—
	100	4.38%	5.18%	3.30%	3.51%	5.14%	—	3.11%	2.90%	3.08%	13.44%	4.25%	—
	900	2.27%	3.89%	1.81%	2.10%	2.49%	—	1.90%	1.38%	1.29%	10.14%	2.83%	—

Table C.6: End-to-end Predictions on Synthetic Neural Architectures on PyTorch Mobile (CPU Stands For a Large Core)

Approach	Training Size	Snapdragon 855		Exynos 9820		Snapdragon 710		Helio P35		A12 Bionic		A10 Fusion	
		CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU	CPU	GPU
Lasso	30	11.76%	7.43%	10.82%	10.97%	20.37%	—	8.45%	7.36%	7.69%	12.63%	9.38%	—
	100	11.68%	6.13%	9.09%	8.72%	19.17%	—	7.59%	7.71%	7.22%	12.19%	9.33%	—
	900	11.27%	5.70%	8.34%	8.71%	19.49%	—	7.07%	6.73%	6.77%	11.63%	9.40%	—
RF	30	16.12%	21.68%	15.45%	29.49%	23.20%	—	12.51%	23.28%	11.50%	14.98%	13.81%	—
	100	14.38%	10.65%	13.94%	11.66%	19.84%	—	12.04%	11.88%	10.55%	13.57%	9.00%	—
	900	8.25%	6.87%	6.57%	7.21%	11.09%	—	6.46%	5.23%	4.94%	10.65%	7.15%	—
GBDT	30	11.64%	29.62%	12.08%	26.29%	20.55%	—	10.75%	20.91%	7.23%	25.40%	7.39%	—
	100	8.44%	13.02%	8.24%	12.56%	11.35%	—	9.15%	8.60%	6.16%	13.26%	5.56%	—
	900	5.86%	10.29%	5.03%	9.04%	5.60%	—	4.73%	5.54%	4.65%	10.38%	5.55%	—
MLP	30	9.31%	19.74%	12.13%	16.19%	22.72%	—	11.03%	22.51%	7.59%	19.92%	7.93%	—
	100	8.86%	25.38%	5.43%	19.39%	10.75%	—	5.66%	22.49%	7.45%	13.57%	7.21%	—
	900	6.04%	12.12%	5.86%	7.51%	9.71%	—	6.25%	10.01%	4.57%	11.13%	6.04%	—

Table C.7: End-to-end Predictions on Real-world Neural Architectures on PyTorch Mobile (CPU Stands For a Large Core)