A Benchmark for ML Inference Latency on Mobile Devices

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

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

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

ABSTRACT

Inference latency prediction on mobile devices is essential for multiple applications, including collaborative inference and neural architecture search. Training accurate latency predictors using ML techniques requires sufficient and representative data; however, collection of such data is challenging. To overcome these challenges, in this work, we focus on constructing a comprehensive dataset that can be used to predict inference latency on mobile devices. Our dataset contains 102 real-world CNNs, 69 real-world ViTs and 1000 synthetic CNNs across 174 diverse experimental environments on mobile platforms, accounting for critical factors affecting inference latency, including hardware heterogeneity, data representations and ML frameworks. Our code is available at: https://github.com/qed-usc/mobile-ml-benchmark.git.

CCS CONCEPTS

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

KEYWORDS

Benchmark, Neural Networks, Inference, Latency, Mobile Devices

ACM Reference Format:
Zhuojin Li, Marco Paolieri, and Leana Golubchik. 2024. A Benchmark for ML Inference Latency on Mobile Devices. In *7th International Workshop on Edge Systems, Analytics and Networking (EdgeSys '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3642968.3654818

1 INTRODUCTION

Machine learning (ML) techniques, especially Deep Neural Networks, have been broadly adopted to mobile platforms and play a crucial role in performing inference tasks for applications such as facial recognition, text generation, and healthcare.

Consequently, predicting the inference latency of neural networks on resource-constrained mobile devices is an essential task across a broad range of applications, e.g.,: (1) collaborative inference [9, 11], where a deep neural network is partitioned between a mobile device and a cloud server for cooperative processing – accurate latency prediction for each component in the neural network facilitates finding optimal partitions, and (2) neural architecture search (NAS) [4, 23], which automates the exploration of neural architectures (NAs) with an appropriate trade-off between ML accuracy and efficiency – an accurate latency model ensures that the search process is directed towards architectures that satisfy stringent low inference latency constraints.

However, simple metrics such as FLOPs (the number of floating point operations) do not provide accurate proxies [15, 22] for real-world latency, which is influenced by diverse factors such as underlying optimization algorithms and hardware specifications. To bridge this gap, ML techniques have been increasingly applied to predict both end-to-end latency (e.g., by encoding the entire NA as a graph [6]) and latency of individual building blocks (e.g., represented as a tuple of configuration parameters [13]).

Notably, the availability of sufficient and representative data is essential for these ML-based approaches to achieve accurate latency predictions. However, construction of a comprehensive dataset for ML inference on mobile platforms is challenging due to the following reasons. (1) *Diverse experimental environment*: The substantial diversity of hardware configurations, operating systems and ML frameworks results in distinct characteristics of inference latency; designing a comprehensive dataset reflecting latency in diverse and representative environments requires careful consideration of these factors. (2) *Opacity of ML frameworks*: From our observations, current ML frameworks lack functionality for profiling the latency of each operation on mobile GPUs and hide the details of specific optimizations on target devices; lack of this information in existing datasets [10] presents challenges for constructing accurate block-wise predictors for inference latency. (3) *Evolving neural architecture designs*: State-of-the-art (SOTA) NAs continuously emerge and consist of novel building blocks, such as the recent advances in Vision Transformers (ViTs) achieving SOTA ML accuracy in vision tasks; however, there is a lack of existing public datasets benchmarking their real-world latency across diverse mobile platforms, since different NA designers provide their own model implementations and some are difficult to deploy on mobile platforms (e.g., due to use of operations unsupported on mobile GPUs).

Hence, in this work, we overcome these challenges by constructing a comprehensive dataset that can be used to predict inference latency on mobile devices. Specifically, our main contributions are:

- We present a methodology to generate, deploy and benchmark NAs across 174 scenarios, including mobile CPUs and GPUs on both Android and iOS devices, with float and integer data representations, on both TFLite and PyTorch Mobile frameworks; our methodology can be utilized for comprehensive performance evaluation of NAs.
- We release a dataset [2] collected by applying our methodology; the dataset includes 102 real-world and 1000 synthetic Convolutional Neural Networks (CNNs), covering a majority of representative configurations for typical building blocks. In

EdgeSys '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0539-7/24/04
https://doi.org/10.1145/3642968.3654818

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

Table 1: Mobile Platforms in Our Study

our previous work [13], latency prediction models trained on this dataset achieved high accuracy (MAPE errors of 2.4% and 5.2% for CPUs and GPUs, respectively).
- We systematically collect ViTs from HuggingFace [21] and test their compatibility with mobile platforms; based on this, we extend our dataset [13] with real-world latency measurements for 69 SOTA ViTs across multiple environments.
- We conduct evaluation of characteristics of end-to-end and operation-wise latency for the NAs in our dataset, providing insight for architecture design, including improvements in bottleneck blocks, as well as inference runtime implementations, such as efficient support of specific operations.

2 EXPERIMENTAL ENVIRONMENT

In this section, we describe the experimental environment, specifically ML frameworks (Section 2.1) and hardware (Section 2.2), that drive the development of our dataset.

2.1 ML Frameworks

Our dataset contains measurements from two mainstream ML frameworks for mobile platforms: TensorFlow Lite (TFLite) and PyTorch Mobile. We evaluate the platform-specific runtime implementations (i.e., TFLite *delegates* and PyTorch *backends*) that exhibit distinct performance characteristics (shown in Section 4.1).

2.1.1 TFLite. TFLite implements a variety of delegates to support execution on diverse hardware accelerators. For mobile CPUs, TFLite offers native CPU kernels (i.e., executable programs for ML operations) optimized for ARM Neon instructions, along with support for the XNNPACK delegate [8], a high-performance library for inference on ARM CPUs. For mobile GPUs, the TFLite GPU Delegate [12] builds OpenCL/OpenGL kernels for Android platforms and Metal kernels for iOS platforms. These delegates improve the flexibility of TFLite in choosing hardware for mobile deployment. TFLite models are interpreted as a computational graph, in which nodes represent ML operators and edges depict data dependencies. During inference, a designated delegate for a particular platform analyzes the graph and performs a set of optimizations: for example, (1) merging a sequence of nodes to reduce dispatch overhead, or (2) applying optimization algorithms (e.g., Winograd for convolution) to accelerate the execution on specific hardware.

2.1.2 PyTorch Mobile. Similarly to TFLite, PyTorch Mobile implements diverse hardware backends, including CPU backend for ARM CPUs, Vulkan backend for Android mobile GPUs, and Metal backend for iOS mobile GPUs. The CPU backend incorporates various computing libraries for inference: XNNPACK [8] for floating-point representations and QNNPACK [17] for integer representations. Models in PyTorch Mobile are represented as TorchScript Intermediate Representations (IRs), which consist of ML operations as well as flow control (e.g., loop and if-statements). This representation facilitates various optimizations, such as constant folding and operator fusion, for efficient execution and supports the deployment of models across diverse production environments.

2.2 Hardware

Modern mobile platforms commonly integrate cores with different computational capacities into the same CPU, known as the ARM big.LITTLE architecture; these cores are organized into multiple clusters, each consisting of homogeneous cores running at the same clock speed. For instance, the "big cores" with higher clock speeds are suitable for urgent and computationally intensive tasks, while the "LITTLE cores" with lower clock speeds are energy-efficient for less demanding tasks. In our dataset, we conduct measurements on both homogeneous cores within a cluster and heterogeneous cores across clusters. Existing work [20] shows that utilizing cores from different clusters does not necessarily lead to performance improvement because of the communication cost overhead due to synchronization between multiple clusters.

Mobile GPUs of different manufacturers exhibit heterogeneous hardware architectures which affect inference latency in different ways, e.g., (1) hardware support on Adreno GPUs for larger warp sizes can reduce computational time as compared to Mali Bifrost architectures or (2) Mali GPUs share global memory among compute units, while Adreno GPUs reserve on-chip local memory in each compute unit to speedup memory operations. Consequently, ML workloads exhibit distinct performance characteristics across hardware, which motivates comprehensive evaluations across diverse mobile platforms in our dataset, as summarized in Table 1.

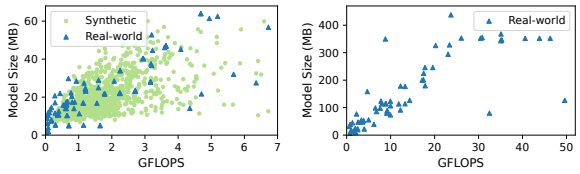
3 DATA COLLECTION

Construction of a comprehensive dataset involves the following steps: selecting neural architectures (Section 3.1), applying ML framework optimizations in model deployment (Section 3.2), properly configuring hardware (Section 3.3) and instrumenting ML frameworks (Section 3.4).

3.1 Neural Architecture Selection

In our dataset, we focus on efficient NAs for image classification; specifically, we evaluate (1) a broad range of real-world NAs developed in recent publications and (2) synthetic NAs covering a majority of representative configurations for typical building blocks.

3.1.1 Real-world. The real-world NAs include both CNNs and ViTs designed for image classification tasks. For CNNs, we adopt the



(a) CNNs **(b) ViTs**

Figure 1: FLOPs and Model Sizes for NAs in our dataset

implementations from [1], which provides pre-trained parameters as well as the Top-1 and Top-5 test errors on the ImageNet-1K dataset. We select 102 CNNs from 25 recent publications with small parameter sizes suitable for mobile devices: BagNet, BN-Inception, DenseNet, DiracNetV2, DLA, EfficientNet, FBNet, FD-MobileNet, Ghost-Net, HarDNet, HRNet, MnasNet, MobileNet, MobileNetV2, MobileNetV3, PeleeNet, PreResNet, ProxylessNAS, RegNet, ResNet, ResNeXt, SE-(Pre)ResNet, SPNASNet, Squeeze(Res)Net, VoVNet.

For ViTs, we utilize the implementations from HuggingFace [21] and select all available NAs, including 69 ViTs from 18 recent publications: BEiT, CvT, DeiT, DINOV2, EfficientFormer, FocalNet, LeViT, MobileViT, MobileViTV2, PoolFormer, PVT, SegFormer, SwiftFormer, Swin, SwinV2, VAN, ViT, ViTMSN. The repository offers PyTorch implementations for all the ViTs, but TensorFlow implementations are available for only 29 out of the 69 ViTs.

Fig. 1 depicts the FLOPs and model sizes for these NAs; as can be seen, ViTs tend to exhibit higher FLOPs and larger model sizes as compared to CNNs, which motivates the recent research trend in developing efficient ViTs for resource-constrained mobile platforms.

3.1.2 Synthetic. Training of an accurate latency predictor requires a broad coverage of block configurations; this is lacking in the real-world CNNs and existing benchmarks [5]. To this end, our dataset also contains synthetic CNNs, which is used in our earlier work [13] to train accurate latency predictors. This synthetic dataset contains vast configurations of typical CNN building blocks adopted in NAS; specifically, each NA contains a sequence of 9 blocks with output channels $\{C_1, ...C_9\}$ followed by a 1x1 convolution layer with output channel C_{10}; blocks {1, 3, 5, 7} perform downsampling to reduce the input height and width by half. The type and parameters of each building block are sampled uniformly at random from:

(1) A convolution layer (with 3x3, 5x5 or 7x7 kernel, and optional group size $4k$, with $1 \leq k \leq 16$).
(2) Depthwise separable convolution (with 3x3, 5x5 or 7x7 kernel).
(3) Linear bottleneck (with 3x3, 5x5 or 7x7 kernel, expansion rate 1, 3 or 6, and optionally using Squeeze-and-Excite).
(4) Average or max pooling layer (with 1x1 or 3x3 window).
(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.

Considering the limited computing resources on mobile devices, we place the following (additional) constraints on uniformly sampling the output channel sizes of these building blocks: $\{C_1, ...C_5\} \in [8, 80]$, $\{C_6, ...C_9\} \in [80, 400]$, and $C_{10} \in [1200, 1800]$. Based on this design, our search space can potentially cover over 2×10^7 possible convolution configurations, the dominant operations in CNNs. In our dataset, we sample 1000 synthetic NAs with 6608 configurations of convolution operations. Fig. 1a illustrates that the ranges of FLOPs and model sizes of these synthetic NAs (green dots) are consistent with real-world CNNs (blue triangles).

We note that, at this time, construction of a comprehensive search space of ViTs is challenging, because novel designs of ViTs are being rapidly proposed in current literature and many of these are not supported on mobile devices. Thus synthetic NAs for ViTs are currently not included in our dataset.

3.2 ML Framework Optimizations

In this section, we focus on the various optimizations applied by ML frameworks in the process of model deployment.

3.2.1 TFLite. In TFLite, a *.tflite* model is used across diverse hardware platforms by utilizing different delegates. At inference time, a range of optimizations, such as kernel fusion and specialized algorithm selection, are performed within each delegate; such implementation enables hardware-specific optimizations based on the capability of each accelerator. Quantization is a technique that converts weights and activations into low-precision representations, commonly used to reduce the model size and computational cost on mobile platforms. To make use of quantization, TensorFlow provides APIs, such as post-training quantization, to convert a 32-bit floating-point model into 8-bit integer representation before generating quantized *.tflite* models.

3.2.2 PyTorch Mobile. In PyTorch Mobile, a model in *torch.nn.Module* needs to be encoded as TorchScript IRs and then converted to backend-specific IRs with operations supported on the backend. During the conversion, a set of optimizations, such as operation fusion and folding of prepacked operations, are conducted to accelerate the inference. However, these IR-based optimizations occur during model generation phase, which lacks hardware information about the target device, as compared to TFLite. PyTorch also supports post-training static quantization, which, compared to TensorFlow, requires a sequence of manual modifications to the model source, such as substituting the implementations of certain operators with their quantized version, converting input tensors through *(De)QuantStub*, and specifying the structure for operation fusion.

Due to the limited support for various operations on PyTorch Mobile, some NAs cannot be converted into corresponding quantized versions (e.g., lack of quantization support for *roll* operation in Swin [14]) or deployed on mobile GPUs (e.g., failure of *as_strided* operation). We note that, in the current version of PyTorch (v2.1.2), 52 real-world ViTs are supported for quantization out of the 69 available on HuggingFace; however, execution on mobile GPUs is not supported. Thus, our dataset includes measurements on CPUs for (1) 69 real-world ViTs and (2) 52 supported quantized ViTs.

3.3 Hardware Configuration

In this section, we illustrate how to properly configure hardware in order to obtain consistent latency measurements. To collect measurements on a given set of CPU cores, we set the CPU affinity of the spawn computing threads to encourage their scheduling on these cores. Specifically, for Android devices, we use *taskset* command to launch the benchmark processor with the given CPU affinity; for iOS devices, we update the *quality-of-service (QoS) class*

for both the main thread and newly spawn threads to direct their scheduling on either performance or efficiency cores.

Additionally, in order to acquire stable measurements over time, we enforce the maximal CPU and GPU frequencies for Android devices and adjust the *GPU Performance State* to high for iOS devices through Xcode [3]. Moreover, to mitigate the impact of thermal throttling, we attach a physical cooling fan to the back of each device to lower the temperature during data collection.

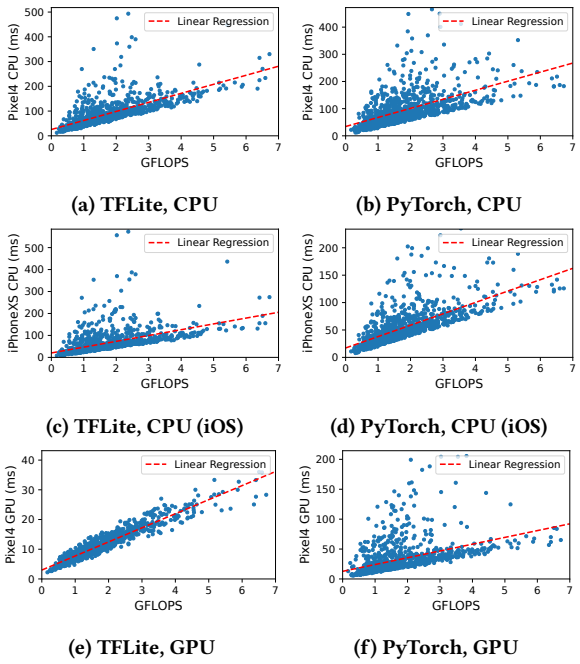
3.4 ML Framework Instrumentation

In this section, we describe our approach to instrumenting ML frameworks to collect operation-wise latency.

3.4.1 TFLite. We use the *TFLite Model Benchmark Tool* [7] in TensorFlow v2.15.0 to benchmark the inference latency of NAs. We adopt original CPU kernels implemented in TFLite for inference on mobile CPUs. For mobile GPUs, we enable 16-bit floating-point data representation to enhance performance. Since the benchmark tool supports latency measurements of operations only on mobile CPUs, we record start/stop timestamps of GPU kernels by collecting profiling information at both the OpenCL command queue in the GPU delegate for Android platforms and Metal command buffer in the Metal Delegate for iOS platforms. To reduce the overhead of timestamp recording, we dispatch the same kernel multiple times (e.g., 256) and record the average for that kernel. We also record the specific OpenCL kernel implementation selected by TFLite for each operation, e.g., whether convolution is executed using generic implementations or Winograd kernels (which give substantial acceleration when applicable [13]).

3.4.2 PyTorch Mobile. We also profile the latency of each architecture based on the *speed_benchmark_torch* script in PyTorch v2.0.0. For experiments on multi-core CPUs, we leverage the *pthreadpool* API, which allocates threads for CPU computation, to limit the thread counts to match the number of CPU cores. For Android GPUs, we enable 16-bit floating-point inference on Vulkan backend, aligned with our setup in TFLite GPU Delegate; to profile Vulkan kernels, we utilize the built-in GPU event collector in PyTorch by enabling the *USE_VULKAN_GPU_DIAGNOSTICS* macro. To profile Metal kernels for iOS GPUs, we follow the implementation in TFLite to dispatch each kernel multiple times and measure the GPU execution time through Metal command buffers.

We also made changes to the QNNPACK backend in PyTorch Mobile, addressing a performance issue in depthwise convolutions with kernel size 7x7. PyTorch Mobile implements a caching allocator for CPU memory [18], which keeps track of previously allocated memory blocks by mapping each *block size* to its most recent memory address. However, the current implementation can incur performance penalties when an inference task includes multiple weight tensors of the same size – this may reuse memory blocks during an inference task but requires repeated block initialization in subsequent inference tasks. While for other depthwise kernels weight initialization is negligible, it results in substantial cost for depthwise convolutions with kernel size 7x7. A number of different strategies are possible to resolve this performance issue: e.g., requests to the allocator could provide not only the size of a memory block but also its most recent address (which can be stored in the NA metadata). Since this problem can be easily fixed, we opted to include measurements that correspond to a fixed version of the system; specifically, we simply skipped repeated initialization of kernel weights as this gives us appropriate latency measurements, without substantial modifications to PyTorch Mobile.



(a) TFLite, CPU (b) PyTorch, CPU

(c) TFLite, CPU (iOS) (d) PyTorch, CPU (iOS)

(e) TFLite, GPU (f) PyTorch, GPU

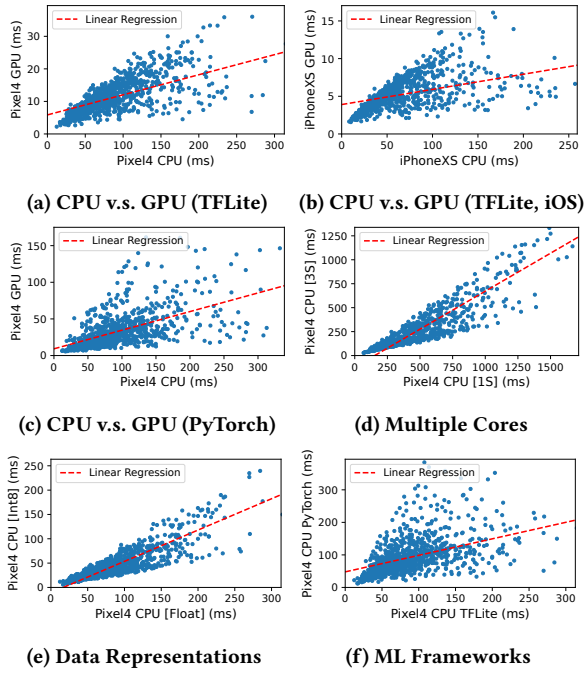
Figure 2: FLOPs and End-to-end Latency

4 DATA ANALYSIS

In this section, we analyze both end-to-end (Section 4.1) and operation-wise (Section 4.2) latency of NAs in our dataset.

4.1 End-to-end Latency

Fig. 2 presents the FLOPs and end-to-end latency for each synthetic NA, where latency shows a positive correlation with FLOPs but in a non-linear relationship: e.g., in all figures except Fig. 2e, the NAs with around 2 GFLOPs exhibit a large latency range. In addition, we fit linear regression model f^* to predict latency y_i based on FLOPs x_i of each NA (denoted by the red dashed line) and evaluate the mean absolute percentage error (MAPE): $\frac{1}{N}\sum_{i=1}^{N}|(f^*(x_i)-y_i)/y_i|$. We observe substantial MAPEs across hardware platforms and ML frameworks: for example, on Pixel 4 CPU, the MAPEs are 28.9% for TFLite (Fig. 2a) and 39.9% for PyTorch Mobile (Fig. 2b); on iPhone XS CPU, the MAPEs are 43.1% for TFLite (Fig. 2c) and 30.1% for PyTorch Mobile (Fig. 2f). In comparison, our earlier work [13] achieves MAPEs of 2.4% for CPUs and 5.2% for GPU for these synthetic NAs. We also notice that the MAPEs for TFLite GPUs are lower (e.g., 12.5% in Fig. 2e); the smaller scale of the y axis in this figure indicates that this stronger linearity is attributed to the highly optimized implementations of TFLite GPU Delegate which reduces the memory access cost across various hardware and effectively leverages the computing capacity of mobile GPUs.



(a) CPU v.s. GPU (TFLite) (b) CPU v.s. GPU (TFLite, iOS)

(c) CPU v.s. GPU (PyTorch) (d) Multiple Cores

(e) Data Representations (f) ML Frameworks

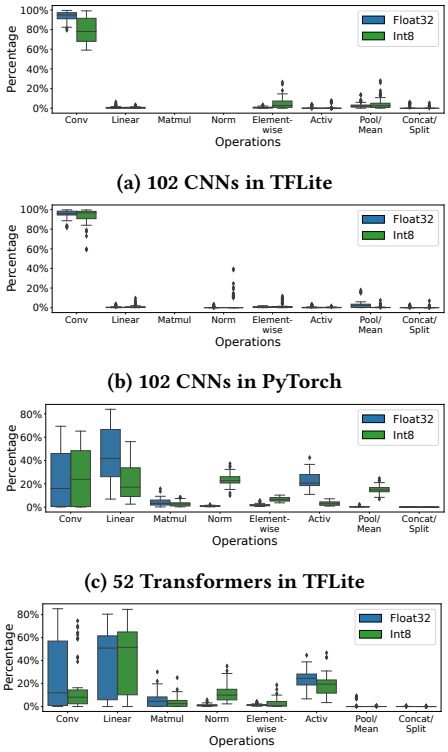
Figure 3: Latency Comparisons across Different Scenarios

Consistently with observations in recent studies [15, 22], FLOPs do not serve as an accurate proxy metric for evaluating efficiency of NAs. In fact, latency is affected by both hardware specifications and the underlying ML frameworks: for example, DRAM bandwidth as well as cache hierarchy can impact the cost of memory access; ML frameworks can leverage accelerated algorithms for kernel implementations (e.g., Winograd algorithm for convolution) based on hardware specifications. FLOPs measurements alone do not capture the effects of these factors.

We also compare latency across the experimental environments in our dataset to justify the necessity of measurements over a broad range of settings. The latency comparisons in Figs. 3a to 3c illustrate that the increase in CPU time does not consistently align with the increase in GPU time, regardless of ML frameworks or mobile platforms; consequently, linear regression (indicated by a red dash line) leads to high MAPE values in all cases: 30.8% for TFLite (Fig. 3a), 28.7% for iOS (Fig. 3b), and 48.1% for PyTorch Mobile (Fig. 3c). In addition, Figs. 3d to 3f compare the latency between (1) one small core and three small cores, (2) floating-point and integer representations, and (3) TFLite and PyTorch Mobile frameworks. In none of these cases is strong linearity observed and linear regression reveals high MAPEs in these comparisons: 40.0% for different number of cores, 29.0% between data representations, and 47.5% over ML frameworks. The distinct performance characteristics across these experimental environments highlight the importance of comprehensive evaluations in constructing our dataset.

4.2 Operation-wise Latency

An important component in our dataset is profiling of latency for each operation in NAs, leading to insight such as bottleneck operations, as illustrated in this section.



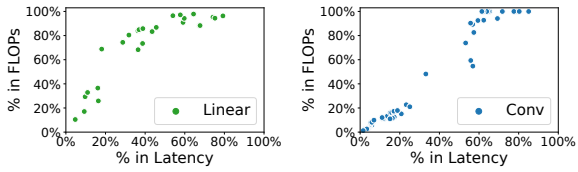
(a) 102 CNNs in TFLite

(b) 102 CNNs in PyTorch

(c) 52 Transformers in TFLite

(d) 69 Transformers in PyTorch

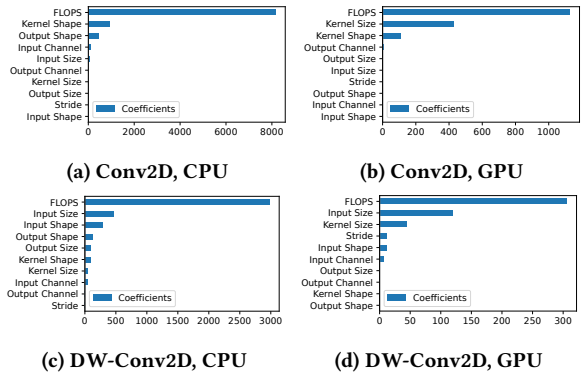
Figure 4: Real-world NAs Latency Breakdown (Pixel 4 CPU)



(a) TFLite, iPhone XS CPU (b) PyTorch, Pixel 4 CPU

Figure 5: Comparison of FLOPs and Latency for Transformers

4.2.1 Latency Breakdown. Figs. 4a and 4b depict the latency breakdown over multiple types of operations for 102 real-world CNNs in TFLite and PyTorch Mobile, respectively. As can be seen, convolution operations account for most of the end-to-end latency (i.e., a median of 91.2% on TFLite and 96.3% on PyTorch Mobile). In contrast, as depicted in Figs. 4c and 4d for ViTs, linear operations contribute significantly to the end-to-end latency, because the basic blocks (multi-head self-attention [19]) in transformers mainly consist of linear operations, while most linear operations in CNNs are fully-connected layers that produce final output classification. Notably, for ViTs, GELU activations occupy a significant amount of time in both frameworks (e.g., 42.4% for EfficientFormer-L1 in TFLite with *float32* representation), and layer normalization takes considerable time for integer representations (e.g., 37.2% for Swin-Tiny in TFLite with *int8* representation); both operations are typically overlooked when using FLOPs measurements, resulting in FLOPs being an inaccurate efficiency proxy. Figs. 5a and 5b further



(a) Conv2D, CPU **(b) Conv2D, GPU**

(c) DW-Conv2D, CPU **(d) DW-Conv2D, GPU**

Figure 6: Coefficients of Operation Features

compare the percentages of FLOPs and latency for linear operations in TFLite and convolution operations in PyTorch, respectively. As can be seen, NAs with a significant portion of FLOPs attributed to these layers spend disproportionate amount of time on these layers; in both figures, the points at the upper-middle positions correspond to nearly all FLOPs but contribute to only 60% of end-to-end latency. This observation motivates ViT researchers to measure actual latency rather than FLOPs in evaluations of model efficiency [16].

Quantization can also affect latency distribution across operations, since there is non-uniform speedup among operations due to quantization; for example, after quantization in TFLite, normalization operations consume a larger portion of the end-to-end latency as shown in Fig. 4c.

4.2.2 Effects of Operation Features. In this section, we analyze the effects of multiple features on latency. Specifically, we fit a linear regression model between various configurations of convolution and latency; we enforce positive coefficients in the linear model and utilize the magnitude of each coefficient to assess the positive contribution to latency for the corresponding feature. Fig. 6 depicts coefficients of features for convolution operations in TFLite derived from linear regression. As can be seen, FLOPs is the single most critical feature affecting latency of convolution and depthwise convolution. However, as previously noted, FLOPs alone do not serve as an accurate proxy for latency. There are also features that affect memory access cost, such as kernel size for convolution (37.8% of FLOPs' coefficient on GPU) and input size for depthwise convolution (38.8% of FLOPs' coefficient on GPU).

5 CONCLUSIONS AND FUTURE DIRECTIONS

We constructed a dataset of inference latency measurements across 174 diverse experimental environments, exhibiting diversity in hardware, data representation, and ML frameworks. We discussed generation, deployment and benchmarking of NAs in our dataset, arguing for use of latency measurements on mobile devices over FLOPs. We analyzed performance characteristics of end-to-end and operation-wise latency of NAs across diverse experimental environments, offering valuable insights for NA design and runtime implementation. Future directions include development of synthetic ViTs in our dataset (once ML frameworks provide needed support for their operations) and exploring energy consumption of inference, another important metric for mobile platforms.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2021. Sandbox for training deep learning networks. https://github.com/osmr/imgclsmob.
[2] 2024. A Benchmark for ML Inference Latency on Mobile Devices. https://github.com/qed-usc/mobile-ml-benchmark.git.
[3] Apple. 2024. Optimizing GPU performance. https://developer.apple.com/documentation/xcode/optimizing-gpu-performance.
[4] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once for All: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations*.
[5] Xuanyi Dong, Lu Liu, Katarzyna Musial, and Bogdan Gabrys. 2021. Nats-bench: Benchmarking nas algorithms for architecture topology and size. *IEEE transactions on pattern analysis and machine intelligence* 44, 7 (2021), 3634–3646.
[6] Lukasz Dudziak, Thomas Chau, Mohamed Abdelfattah, Royson Lee, Hyeji Kim, and Nicholas Lane. 2020. BRP-NAS: Prediction-based NAS using GCNs. In *Advances in Neural Information Processing Systems*, Vol. 33. 10480–10490.
[7] Google. 2022. TFLite Model Benchmark Tool. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/tools/benchmark.
[8] Google. 2023. XNNPACK: High-efficiency floating-point neural network inference operators for mobile, server, and Web. https://github.com/google/XNNPACK.
[9] Chuang Hu, Wei Bao, Dan Wang, and Fengming Liu. 2019. Dynamic adaptive DNN surgery for inference acceleration on the edge. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1423–1431.
[10] Vijay Janapa Reddi, David Kanter, Peter Mattson, Jared Duke, Thai Nguyen, Ramesh Chukka, Ken Shiring, Koan-Sin Tan, Mark Charlebois, William Chou, et al. 2022. MLPerf mobile inference benchmark: An industry-standard open-source machine learning benchmark for on-device AI. *Proceedings of Machine Learning and Systems* 4 (2022), 352–369.
[11] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 615–629.
[12] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. 2019. On-Device Neural Net Inference with Mobile GPUs. *arXiv preprint arXiv:1907.01989* (2019).
[13] Zhuojin Li, Marco Paolieri, and Leana Golubchik. 2023. Predicting Inference Latency of Neural Architectures on Mobile Devices. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 99–112.
[14] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*. 10012–10022.
[15] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European conference on computer vision (ECCV)*. 116–131.
[16] Sachin Mehta and Mohammad Rastegari. 2021. MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer. In *International Conference on Learning Representations*.
[17] Meta. 2023. QNNPACK: Quantized Neural Networks PACKage. https://github.com/pytorch/pytorch/tree/main/aten/src/ATen/native/quantized/cpu/qnnpack.
[18] PyTorch. 2023. PyTorch CPU Caching Allocator. https://github.com/pytorch/pytorch/blob/main/c10/mobile/CPUCachingAllocator.h.
[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
[20] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. 2019. High-throughput cnn inference on embedded arm big. little multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2019), 2254–2267.
[21] Thomas Wolf et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 38–45.
[22] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. 2018. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 285–300.
[23] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).