nnPerf: Demystifying DNN Runtime Inference Latency on Mobile Platforms

Paper #76, 12 pages plus references

Abstract

We present nnPerf, a real-time on-device profiler designed to collect and analyze the DNN model run-time inference latency on mobile platforms. nnPerf demystifies the hidden layers and metrics used for pursuing DNN optimizations and adaptations at the granularity of operators and kernels, ensuring every facet contributing to a DNN model's run-time efficiency is easily accessible to mobile developers via well-defined APIs. With nnPerf, the mobile developers can easily identify the bottleneck in model run-time efficiency and optimize the model architecture to meet system-level objectives (SLO). We implement nnPerf on TFLite framework and evaluate its e2e-, operator-, and kernel-latency profiling accuracy across four mobile platforms. The results show that nnPerf achieves consistently high latency profiling accuracy on both CPU (98.12%) and GPU (99.87%). Our benchmark studies demonstrate that running nnPerf on mobile devices introduces the minimum overhead to model inference, with 0.231% and 0.605% extra inference latency and power consumption. We further run a case study to show how we leverage nnPerf to migrate OFA, a SOTA NAS system, to kernel-oriented model optimization on GPUs.

1 INTRODUCTION

Mobile applications fueled by deep neural networks (DNNs) are becoming an indispensable part of our daily lives, creating unprecedented user experience spanning video [44, 74], audio [42, 68], and gaming [4, 8]. Despite their huge success in mobile market, running DNN-based applications on mobiles (*e.g.*, smartphones and smart tablets) remains a daunting task due to the limited onboard computing, memory, and power resources. This problem has raised a surge of research interest in optimizing model architecture.

To deal with resource scarcities on mobile devices, the current practice follows two general approaches – *i*) train a DNN model on a dedicated server, then fit it into the mobile device through model pruning [48, 49, 62] or quantization [78, 80]; and *ii*) search for an efficient neural architecture (*e.g.*, NAS [43, 54, 69]) to meet the specific system-level objectives (SLOs), *e.g.*, model inference latency. These model-tuning techniques have demonstrated their efficacy in a plethora of benchmark studies. However, their efficiency is vulnerable to *resources dynamics* of mobile devices and thus can hardly meet SLOs when deployed in the wild.

Mobile devices usually run multiple applications concurrently, *e.g.*, a foreground application with multiple tasks co-running in the background. Their resources at run-time are highly dynamic due to the change of application priorities and background activities [44]. Moreover, the dynamic voltage and frequency scaling (DVFS) module will adaptively update the processors' frequency based on the workload and temperature; hence the computing resources may change drastically over time [57]. Given these resource dynamics, the operators and kernels[1] tuned for one specific resource setting are unlikely to be optimal for other settings – an optimal kernel, for example, could become memory-bound or compute-bound alternatively due to resource contentions. Consequently, the model run-time efficiency will experience significant variations and thus can hardly meet SLOs (§2.3).

Table 1: Comparison of nnPerf with existing DNN model profilers designed for mobile platforms.

Systems	e2e latency	DNN Op		DNN Kernel		Process Timeline	On-device	Real-time results
		CPU	GPU	CPU	GPU			
nnPerf	✓	✓	✓	✓	✓	✓	✓	✓
AGI [5] & Perffeto [22]	✓	✓	×	×	×	✓	×	×
Snapdragon Profiler [25]	✓	✓	×	×	×	✓	×	✓
Android Studio profiler [2]	✓	✓	×	×	×	✓	×	✓
Adreno GPU Profiler [3]	✓	✓	×	×	×	✓	×	×
Tensorflow Benchmark [30]	✓	✓	×	×	×	×	×	×
Perfdog [29]	✓	×	×	×	×	✓	×	×
Xcode-Instrument [33]	✓	✓	✓	×	×	✓	×	✓
Soloπ [26] & Emmagee [10]	✓	×	×	×	×	×	✓	×

To deal with these resource dynamics, mobile developers should design resource-aware model adaptation or optimization by leveraging a performance analysis tool, *a.k.a.*, a profiler, that can monitor the model run-time latency accurately and unobtrusively, answering fundamental questions such as *where is the time spent on a DNN model inference? Would a well-tuned DNN model satisfy the specific SLO when deployed in a new device, and if not, why?* Developers can use such a profiler to identify the bottleneck of DNN model inference at various granularities, predict model performance in high fidelity, and adapt model architecture to the available resources promptly (*e.g.*, switching between different kernels adaptively to meet rigid SLOs in the presence of resource dynamics [47]), just like how back-end developers leverage the NVIDIA Nsight system/compute [18, 19] to optimize their cloud-native DNN models. In particular, such a profiler should satisfy the following requirements.

i) **Cross-stack design**. Like the NVIDIA Nsight compute [18] and Nsight system [19], the mobile profiler needs to track

[1]In the DNN system domain, the kernel refers to the atomic code segment that runs DNN operators on processors. It differs from the DNN kernel (essentially a filter) and GPU kernel (a code segment for image rendering).

the model activities at the granularity that a developer is interested in and desires to optimize. For instance, some mobile developers may be interested in the run-time latency at the level of operators, while others may want to know the latency breakdown at the granularity of kernels and make the kernel optimization accordingly. All existing profilers can only account for the latency down to the op-level at CPUs, leaving the efficacy of the atomic operation in DNN, the kernel, a black box. Moreover, due to the lack of kernel efficiency on GPU accelerators, existing hardware-aware NAS such as MNasNet [40] and OFA [37] are limited to mobile CPU accelerators, which wastes the powerful GPU resources.

ii) **Real-time and on-device profiling**. Beyond the Nsight compute and Nsight system, such a mobile profiler should monitor the model run-time latency at various granularities in an online fashion, reporting the results to developers timely through well-defined APIs. Existing mobile profilers such as Snapdragon Profiler [25], Android GPU Inspector (AGI) [5], and Perffeto [22] run offline on an auxiliary device (*e.g.*, a PC host). Hence they cannot be leveraged for real-time model architecture adaptation (*e.g.*, real-time NAS [40, 46, 60, 75]). Compared to the prediction-based methods, real-time on-device profiling provides more accurate, fine-grained latency feedback for training and pruning DNN models on mobile devices [41, 53], particularly in the presence of resource dynamics where latency predictions are highly inaccurate.

Table 1 compares the mainstream profilers adopted by developers. Nvidia Nisight series are tailored to Nvidia GPUs and thus do not apply to smartphones and smart tablets.

In this paper, we present nnPerf, a real-time on-device profiler designed to demystify the DNN model run-time inference latency on mobile platforms. nnPerf makes extensions to the TFLite framework [31] since TFLite dominates the mobile application markets[2]. It was packed as a TFLite Android ARchive (AAR) plug-in to mobile applications, providing developers with a holistic view of model run-time latency across the entire inference stack, including the e2e (end-to-end) model inference latency, operator-level latency, and kernel-level latency, on both mobile CPUs and GPUs. The design of nnPerf faces three technical challenges.

• **Unknown operators before model compiling.** Unlike e2e model latency profiling, where we can trace down to the C++ core implementation of the generic model invoke function and add timers before model compilation, the operators are generated and low-level optimized (*e.g.*, operator fusion [58, 72]) at the model compiling phase, and before that, we had no clue about which operator implementations will be executed. Besides, TFLite supports user-defined operators that differ drastically from each other and these customized operators remain unknown until the model enters run-time. To address this challenge, nnPerf leverages a unique opportunity in model registration where the TFLite interpreter sweeps all linked operators at the execution plan queue, exposing the operator IDs. nnPerf tracks this queue operation at run-time to capture the ID of all operators to be executed. It then automatically locates each operator's implementation on the C++ core with the ID and adds timers hereafter.

• **Closed-source kernel implementation.** Most mobile GPU SDKs are closed-sourced. Hence, we cannot follow the code-tracing mechanism to add timers at the proper positions of the source code to directly measure kernel latency. Even if we are allowed to access a few mobile GPU SDKs on mobile platforms through proprietorial APIs, we still have no clue about which kernels will be called before model inference because existing DNN model interpreters leverage kernel fusions [66] to improve model run-time efficiency and these fusion rules are unknown to us [77]. To address this issue, we take advantage of the event callback mechanism and make adaptions to track each kernel's lifecycle on run-time. nnPerf receives a notification from the callback event function when the kernel completes its execution. We then retrieve the crucial timestamps from the event object and track the kernel execution at high accuracy.

• **Unsychrinized timestamps between GPU and CPU.** To facilitate the model latency analysis, the profiler should draw the timeline of model execution at different levels of the model hierarchy by sorting the latency activities chronologically. However, a DNN model runs at both the mobile CPU (*e.g.*, scheduling, data preparation) and GPU (*e.g.*, kernel execution) during its entire lifecycle; these different processors have their own timebases (*i.e.*, clock source). The clock offset between these clock sources changes over time. To address this issue, we propose a lightweight time synchronization algorithm that exploits an observation that both CPU and GPU clocks will be used to record the kernel enqueue event, leaving us an opportunity to measure their time offset. We further propose a kernel injection mechanism to address the cold-start issue in time offset estimation and reduce the system overhead through batching.

Implementation. We implemented nnPerf on TFLite 2.6 framework and packed it as two AAR plug-in modules. Developers can replace the default TFLite AAR or the nightly-snapshot library in their android projects with our customized AARs for model latency profiling. Evaluations show that nnPerf achieves 0.367 ms, 0.014 ms, and 4.096 μs e2e-, operator-, and kernel-latency profiling errors, with 0.206% and 0.605% extra inference latency and power overhead. Moreover, we also conduct a case study to show how fine-grained profiling can improve the efficacy of hardware-aware NAS (§6).

[2]Over 86%(1436) of the most popular 1666 apps in Google Play Store are developed by TFLite [34].

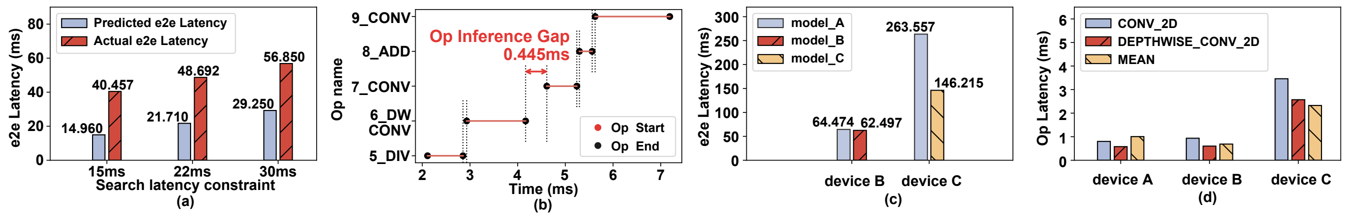


Figure 1: (a) Comparison of model e2e prediction result and the profiling result. (b) The operator inference timeline. (c) Comparison of model e2e latency on different mobile platforms. (d) The top-3 operator of MobilenetV3-large running on three mobile platforms.

Table 2: List of mobile platforms.

Device	SoC	CPU	GPU	RAM	OS Version
(A) Google Pixel 3XL	Snapdragon 845	Kryo 385	Adreno 630	4GB	Android 12 beta
(B) Samsung Galaxy Note10	Snapdragon 855	Kryo 485	Adreno 640	8GB	Android 11
(C) Xiaomi 10	Snapdragon 865	Kryo 585	Adreno 650	8GB	MIUI 13.0.7
(D) TB-RK3399ProD	Rockchip RK3399	Cortex-A72&A53	Mali-T860	6GB	Android 8.1

Contributions. To the best of our knowledge, nnPerf is the first real-time, on-device profiler that can track the model run-time latency at the granularity of kernels on both mobile CPUs and GPUs. We envision this profiler could enable developers to touch the software-hardware boundary of model inference on mobile platforms, prompting efficient hardware-aware and run-time model optimization to deal with resource dynamics on mobile platforms.

2 WHY NNPERF?

We justify the need for nnPerf from three perspectives. First, we show that the current model latency prediction methods are usually inaccurate. We then demonstrate that a model optimized for one specific hardware is usually not optimal for the others. Next, we show that the unique *resource dynamics* on mobile platforms lead to considerable uncertainty in model inference latency. Finally, we review existing profiler designs and set up the design target of nnPerf.

2.1 Latency Prediction is Inaccurate

Currently, no profilers support fine-grained (*e.g.*, kernel-level), on-device DNN execution profiling in real-time. The mainstreaming approach is to profile each operator's latency offline and then add them together to estimate the e2e latency for a given DNN model [36, 65]. However, this offline method is usually inaccurate.

We demonstrate the inaccuracy problem by profiling OFA [37], a state-of-the-art (SOTA) hardware-aware neural architecture search (NAS) system designed for mobile and embedded platforms, which can generate efficient model architecture based on given e2e model latency constraints. We benchmark OFA on a Samsung Galaxy Note 10 phone by searching for model architectures in three different delay constraint settings. The results are shown in Figure 1(a). We observe a huge gap between the predicted e2e model latency (blue bar) and the profiling result (termed as actual e2e latency in red). Below we elaborate on the reasons.

(i) First, the operator latency may change with the underlying hardware resources because the compiler may choose different kernels for an operator based on the availability of memory and computing resources.

(ii) Second, multiple operators could be fused together during model compiling, affecting model inference latency. Moreover, there are implicit glue operators [45] on the model execution framework that can distort model inference latency.

(iii) Third, the add-on latency prediction ignores the data preparation delay, which is the main cause of the latency gap in Figure 1(b) (*i.e.*, the latency between consecutive operators). The contribution of these data preparation delays to the model inference latency grows with the increasing depth of the model.

2.2 Hardware Heterogeneity Requires Fine-grained Profiling

Deploying DNN models on mobile platforms while retaining their model efficacy and efficiency is challenging. The current practice is to train models on the cloud or the server and then adapt the model architectures and parameters to mobile devices [37, 40]. However, such a tailored design lacks scalability because different mobile devices differ drastically in their computing and memory capacity. Hence a model customized for a device is unlikely to remain optimal for other types of mobile devices.

To demonstrate this issue, we again adopt OFA to obtain three MobilenetV3-large models customized for three different mobile devices *A*, *B*, and *C* listed in Table 2. We term these three device-dedicated models as *model_A*, *model_B*, and *model_C*. We first deploy these models on their dedicated devices and measure their individual e2e inference latency. We then deploy *model_A* on device *B*, and *C* and compare its e2e model inference latency against their dedicated models *model_B*, and *model_C*. Figure 1(c) shows the result. As expected, we observe up to 44.5% difference in the e2e inference latency against their dedicated models, which confirms our analysis.

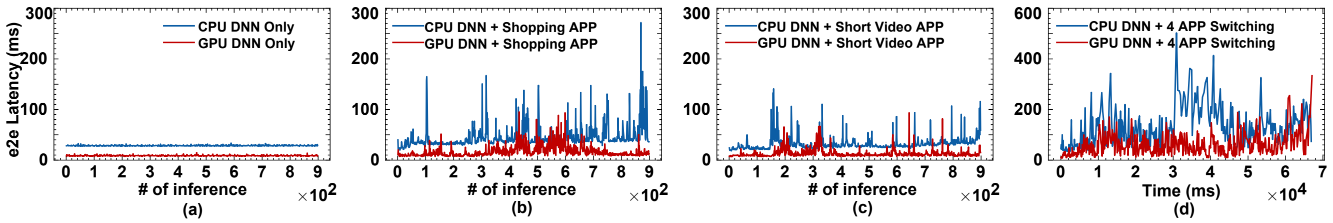


Figure 2: The impact of resource dynamics on DNN run-time latency. (a): the e2e latency of MobienetV3-large in the absence of a co-running APP. (b): the model e2e latency in the presence of a shopping APP. (c): the model e2e latency in the presence of a short video APP. (d): the model e2e latency in the presence of switching between four mobile applications.

Next, we run *model_A* on these three devices and pick up the top-3 heaviest operators on each device based on the operator latency. The results are shown in Figure 1(d). We observe the order of the top-3 operators of *model_A* changes with devices. For instance, the heaviest operator on device *A* (Samsung Note10) is MEAN. However, it changes to CONV_2D when we deploy the model on Xiaomi10 smartphone. These results reveal that the hardware heterogeneity among mobile devices will change the underlying operator executions. To close the gap between model design and deployment, it is thus essential to understand the model efficiency at fine-grained granularity.

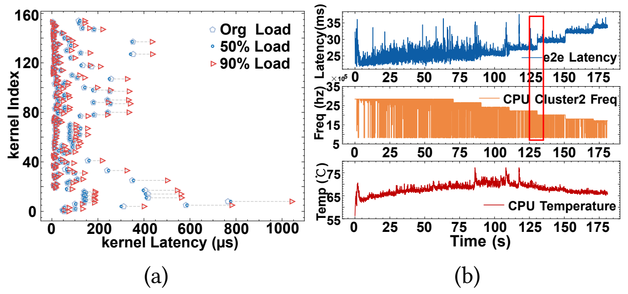


Figure 3: (a) Kernel latency in different memory load settings. (b) the e2e model inference latency varies with the frequency of the hardware accelerator.

2.3 Impact of Resource Dynamics

The gap between model design and deployment becomes even more severe when taking into account *resource dynamics* on mobile platforms. Mobile devices run multiple applications concurrently. The change of priority of these applications will alter the memory and computing resource availability, resulting in resource dynamics. Besides, to prevent hardware from overheating, mobile devices adopt DVFS to tune the frequency of their computing units, which brings dynamics to computing resources. Hence the model inference latency can hardly meet expectations when deployed on mobile platforms. Below we demonstrate these issues.

(i) Impact of APP activities. In this experiment, we measure the e2e inference latency of a MobienetV3-large-based image classification APP on a Xiaomi10 smartphone. The result is shown in Figure 2(a). We observe a constantly low e2e inference latency at about 12*ms* and 30*ms* when the model is running on GPU and CPU alone, respectively. However, as we launch a shopping APP and start browsing, we observe strong turbulence in the model e2e inference latency. As shown in Figure 2(b), the model e2e latency on GPU jumps from 12*ms* to around 30*ms*–50*ms* and peaks at around 100*ms* occasionally. This latency variation grows significantly when these two APPs are deployed on mobile CPUs. We observe a similar trend when the volunteer replaces the shopping APP with a short video APP, as shown in Figure 2(c).

Next, the volunteer is asked to switch between four mobile applications (including a shopping APP, a short video APP, a reading APP, and a MUSIC APP) running on the smartphone. We measure the impact of context switching on the model e2e inference latency. The result is shown in Figure 2(d). As expected, the foreground and background application switching also introduce significant e2e latency variation. For instance, the model e2e latency jumps up and down between 12*ms* and 300*ms*, and 33*ms* and 450 *ms* when running on the mobile GPU and CPU, respectively. We also measure the kernel latency of this model in different memory load settings. As shown in Figure 3(a), the kernel latency grows slightly when the memory workload increases to 50%. However, it then climbs drastically (up to 1.916×) when the memory workload grows from 50% to 90%.

(ii): Impact of thermal throttling. Unlike PCs or servers, most DVFS implementations on mobile devices suffer from overheating. If the temperature of mobile CPUs or GPUs goes beyond the safe range, *thermal throttling* happens – the frequency of CPUs and GPUs will be forced to a lower level to shed heat, degrading the application performance (*e.g.*, the FPS in games). In this experiment, we measure the e2e model latency in the presence of thermal throttling, as shown in Figure 3(b). When the temperature goes beyond the threshold, the OS triggers DVFS to reduce CPU/GPU frequency [56, 57], lowering their computing capability. As shown in Figure 3(b), the model inference latency strongly correlates with the frequency of the underlying hardware accelerator, which confirms our analysis.

2.4 The Need for a Cross-Stack Profiler

To address the above issues, the mobile developers have to understand the model run-time efficiency with a performance analysis tool – a *profiler*. Taking a step further, such a profiler should be accurate and versatile, working as an on-device service to monitor the model execution around the clock, answering questions such as where is the time spent on the model inference? What are the bottleneck operators and kernels? Would these bottlenecks change over time? Unfortunately, all existing profilers designed for mobile platforms cannot satisfy the requirements, as explained below.

• **Course-grained granularity**: the existing profilers can only track the DNN latency at the granularity of operator-level latency on mobile processors, leaving the kernel execution a black box.

• **Offline with an auxiliary device**: most existing profiles such as Tensorflow Benchmark [30], Android Studio Profiler [6], and Perffeto [22] run on an auxiliary device (*e.g.*, a PC host), making on-device, resource-aware optimization challenging. Only when inference is finished, the profiled data can be pulled in batch to an ADB-connected auxiliary device for analysis, making real-time feedback difficult.

To this end, we propose nnPerf, a real-time on-device profiler that collects DNN run-time latency at various granularities on both mobile CPUs and GPUs. nnPerf runs as a background monitor, with the minimum system overhead.

3 NNPERF DESIGN

We first present the cross-stack latency profiling that provides e2e, op-level and kernel-level latency, and then show how to align the GPU- and CPU-profiling results by a lightweight time synchronization mechanism.

3.1 Cross-Stack Latency Profiling

Similar to model execution on PCs or clouds, a DNN model running on mobile platforms is compiled into graphs (*i.e.*, operators) and executed in kernels. To better understand the run-time efficiency of a DNN model and locate the bottleneck in model execution[3], we design a cross-stack profiler that provides developers a holistic view of a DNN model's inference latency across the entire model stack, including *i*) end-to-end model inference latency (§3.1.1); *ii*) operator-level latency (§3.1.2); and *iii*) kernel-level latency (§3.1.3).

3.1.1 Accurate e2e model inference latency profiling. The e2e model inference latency refers to the time that a DNN model takes to complete an inference task. It accounts for the model execution on hardware accelerators. Current e2e latency is profiled either through the user-space API provided by model compiler framework [10, 22, 26, 71] or through offline code injection [2, 3, 25, 33]. Unfortunately, this reported latency is inaccurate because the user-space API measures the model execution on the high-level stack (*i.e.*, JAVA interface) where the time cost on library invokes and language conversion (*i.e.*, from JAVA JNI to TFLite C++ core) is not excluded.
Our solution. To avoid the impact of these upper-layer operations on e2e latency profiling, we embedded trace timers in ML framework low-level C++ library and built an asynchronous output module transparent to facilitate the high-level stack API. In the run-time, these two timers will be triggered right before and after the model execution respectively. The timestamps will be automatically pushed to a log file cross-stack integration which allows developers to retrieve e2e model execution latency in real-time.

To facilitate analysis of the operator latency (§3.1.2) and kernel latency (§3.1.3), this data output module also logs the operator's name and kernel merging information and further pushes this information together with the latencies for user access (§4). We leave trace timer design to §3.2.1.

3.1.2 On-demand timer injection design for operator-level profiling. Capturing the latency of each operator in a DNN model allows the developer to locate the bottleneck of model execution on the computing graph. The developer can replace the bottleneck operator with a lightweight alternative.

After compiling, the model will be translated into a computing graph containing a batch of operators. Meanwhile, the TFLite framework will build a program interpreter and link it to the computing graph. Shortly afterward, the TFLite runtime scheduler will prepare for the model inference by generating an execution plan execution_plan_queue. In the meantime, a function pointer OpInvoke will be generated to assist the runtime scheduler in issuing operators' execution plan. To profile the latency of each operator, one possible solution could be measuring how long this function pointer stays in each entry of execution_plan_queue. However, this solution is not accurate because it does not exclude the latency of data preparation (*e.g.*, tensor copy) that happens between two consecutive subgraphs or operators.

Measuring the precise latency of each operator through timer injection is challenging for the following reasons.

i) Different from e2e profiling where timers can be set up in advance,[4] the operators are generated (fused) at runtime and their implementations change drastically with different model architectures and hardware proccessors. Since we are not aware of which operators will be called before model compiling, we cannot follow the same idea to trace down to the C++ core and set timers at the corresponding operator's invoke function in advance.

[3]We use inference and execution interchangeably throughout the paper.

[4]DNN models with different architectures all share the same model execution function defined at the inference framework core (*i.e.* TFLite C++ core). So we can trace this function offline and set timers accordingly.

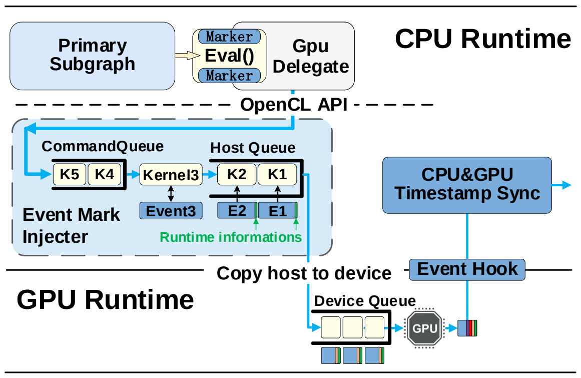


Figure 4: An illustration of kernel execution profiling. We register a customized event for each kernel during the enqueue-dequeue process.

ii) Blindly tagging all built-in operators readily available on the framework's C++ core is insufficient because the TFL-framework also supports user-defined operators custom_op that are unknown to us until the model enters run-time during which the framework collects custom_op from a separate custom library for registration.

Our solution. To address these challenges, we propose an adaptive on-demand timer injection mechanism to automate timer setups at the model's compiling stage. This design is based on an observation that during the operator registration, the TFLite runtime scheduler will go through the operators' execution queue execution_plan_queue with the point-FindOp. As a byproduct, we will obtain the ID of each operator. Hence, by tracking the object of this pointer in the registration phase, nnPerf will obtain the ID of each operator and further get the operator name by checking the mapping table <operator_name, operator_ID> provided by TFLite.

With the list of operator names, nnPerf then locates each operator's code segment in the source code through string matching and then adds timers right before and after each operator's core execution function automatically. At the inference phase, these timers will log the timestamps of each operator's execution and proactively send them (together with the operator ID) to the user space timely. This design excludes the data preparation delay and thus yields a more accurate operator latency profiling result.

3.1.3 Down to the kernel-level latency profiling. A kernel is a low-level code segment that runs on the hardware accelerator (*e.g.*, mobile GPU) to execute operators.

The kernel latency varies largely with the underlying hardware computing and memory resources. Given a model, an optimal kernel combination suited for device *A* is likely to be suboptimal for device *B* due to the difference in their hardware resources. Likewise, the best kernel(s) for an operator running on device *A* may change over time due to the variation in hardware resource availability. For instance, this kernel becomes memory-bound due to the surge of memory contention from another APP co-running at the mobiles.

Knowing the run-time latency of each kernel could reveal the real hurdle behind model efficiency, allowing developers to design hardware-aware model optimizations (*e.g.*, handcrafted kernel optimization [39, 55] and adaptive kernel generation [35, 79]). However, profiling run-time kernel latency on mobile platforms faces two grand challenges.

i) **Closed-source kernel implementation**. Most mobile GPU SDKs are closed-source [59, 77], especially those designed for smartphone SoCs. Hence we cannot follow the code tracing idea proposed for e2e- and operator-latency profiling to locate the function call of each kernel at the TFLite C++ core for timer injection.

ii) **Unknown kernel-fusion rules**. Existing DNN model interpreters (*e.g.*, TFLite [31], TVM [39], ncnn [17], paddle-lite [21]) adopt kernel fusion techniques [58, 66, 72] to improve model run-time efficiency. Accordingly, even if we can access mobile GPU SDKs on some mobile SoCs, we still have no clue about which kernels will be called before inference. Besides, the kernel fusion rules are device-dependent and are hidden from the developers [77]. Hence, the kernel information obtained by a model dry-run on a device would not apply to other devices.

Our solution. Motivated by the *event mechanism* cl_events defined in the OpenCL library, we propose a customized event registration mechanism nnPerf_event to track the life cycle of each kernel in model run-time without touching the kernel source code. Below we first describe how we leverage this mechanism for kernel latency profiling and then describe our event mechanism customization.

An event is a passive mechanism that allows the developers to be notified of an occurrence through a callback function. Upon the completion of model compiling, the interpreter generates a command queue CommandQueue, with each queue entry corresponding to a kernel or a data-transfer operation to be executed on the hardware accelerator. During the model inference, the interpreter copies each entry from the command queue (*i.e.*, CommandQueue) to the host queue (*i.e.*, Host_Queue) and further to the device queue (*i.e.*, Device_Queue) chronologically for kernel execution.

As shown in Figure 4, in each cycle of dequeue (from the command queue) and enqueue (to the host queue) operation, we associate the queue entry (*i.e.*, a kernel or a data-copy operation) with a customized event nnPerf_event. Once the host enqueues this entry to the Host_Queue, it has no control over how the kernel will be processed due to the black-box nature of kernel source code. However, it can receive a notification when the kernel completes its execution with the help of the associated nnPerf_event. To profile the precise kernel run-time latency, we retrieve the timestamp of

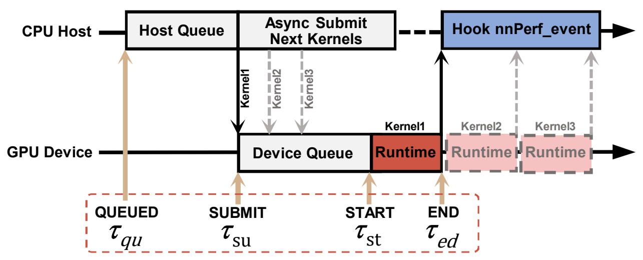


Figure 5: An illustration of kernel lifecycle.

each kernel's launching and completion time from the associated event notification (*i.e.*, τ_{st} and τ_{ed} shown in Figure 5). Moreover, to capture the entire lifecycle of each kernel (including the kernel launching overhead and kernel execution latency), the timestamps of queue operation (*i.e.*, τ_{qu} shown in Figure 5) are also logged in the event associated with each kernel. With all these timestamps, we can further draw the model execution timeline on kernel granularity.

nnPerf_event. The proposed event nnPerf_event is an object that inherits TFLite's standard class cl_event yet is customized for supporting kernel latency profiling, as explained below. First, in addition to caching key timestamps related to the kernel execution, in nnPerf_event, we define new variables and functions to store and further circulate host information and timestamps related to queue latency. In this way, mobile users could access all these critical timestamp information in a single data-retrieving operation. Second, to measure the latency of each phase (*e.g.*, kernel preparation, queue delay, kernel execution) in an entire kernel lifecycle, we also implement timestamp subtraction and precision conversion functions in nnPerf_event, alleviating the overhead posed to upper-layer developers. Last but not the least, we leverage caching technique to reduce the system overhead on repeat data-retrieving operations.

3.2 Time Synchronization

A DNN model will run alternately on both mobile CPU (*e.g.*, tensor and kernel preparation) and GPU (*e.g.*, kernel execution) in different stages of its lifecycle. Different underlying clock sources and timestamp APIs lead to potential non-monotonic chaos in profile timeline and timestamp results. It is therefore critical to synchronize GPU and CPU. But directly synchronizing them by message exchanging incurs extra overhead to the GPU execution and therefore leads to inaccurate profile. Below we elaborate on the unified time wrapper (§3.2.1) and time synchronization (§3.2.2).

3.2.1 Unified Time Wrapper. The timing API gettimeofday() used by existing tools (*e.g.*, Tensorflow Benchmark tools [30]) only supports microsecond precision and is thus not adequate for kernel latency profiling that desires nanosecond precision. In addition, gettimeofday() API has to access the

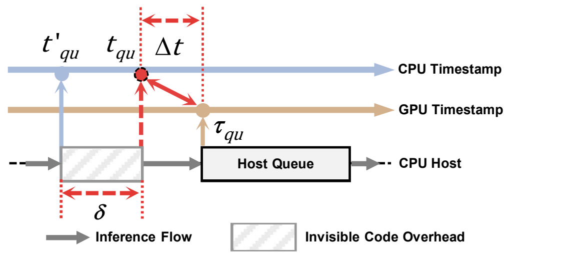


Figure 6: Time offset between CPU and GPU clock.

kernel space through a system call, while system calls usually come with delays that causes non-negligible errors on timestamp. Besides, some clock sources (*i.e.*, RDTSC and basic_TSC) will be affected by the out-of-order execution[5] and DVFS, leading to non-monotonic timestamps.

Our solution. To solve the above problems, we systematically study different time functions and the underlying clock sources across the hardware and software interface on Linux and finally pick a nanosecond-granularity time event, high-resolution_clock provided by the C++ Chrono library [7] as the time event in our customized time wrapper. Thanks to virtual dynamic shared object (vDSO) mechanism [32] that maps the timestamp from OS kernel to userspace without system calls, high_resolution_clock can directly retrieve the timestamp to minimize the timestamp error.

We pick arch_sys_counter, a highly efficient and monotonic clock, as the clock source of our time event function. Upon launching the model, nnPerf overwrites the default clock source of our time event with this monotonic clock arch_sys_counter. This clock then binds the code segment of interest with the nearby timers, ensuring their in-order execution. To facilitate the latency retrieval, we further pack all the timing-related functions into nnPerf customized timer class and incorporate this class into the TFLite framework c++ core.

3.2.2 Time synchronization. The timeline view [1, 13, 66] shows system events visually along a timeline that corresponds to the duration of a model execution at various granularity. However, obtaining a timeline view of model execution on mobile platforms faces multiple challenges, as elaborated below.

i) **Profile disordered timelines errors caused by dynamics clock offset**. The clock offset changes over time due to the dynamic usage of mobile SoC processors. Figure 7(a) shows the distribution of clock offset measurements over 114*s* (over 30,000 times kernel inference). We observe that the clock offset varies between -100 μs and 200 μs. Recall that kernel execution time is at a similar scale (hundreds of μs); ignoring this time offset would result in large errors

[5]Modern CPUs on mobile platforms adopt out-of-order execution technology to avoid the waste of instruction cycles [20]. Thus, the model inference code may not be sequentially executed.

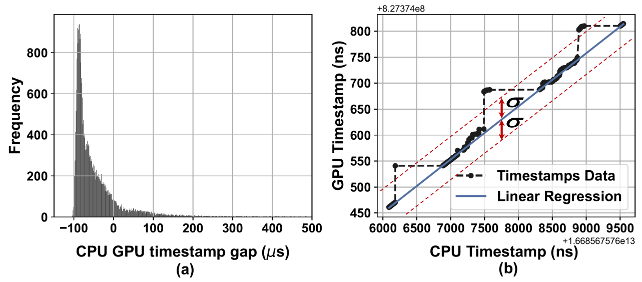


Figure 7: (a) Distribution of the measurement offset. (b) Leveraging linear regression to mitigate the offset.

in the timeline plotting. Moreover, the DVFS on mobile SoC will also change the frequency of the GPU and CPU, negatively affecting the clock offset.

ii) **Additional overhead in black-box GPUs impeded real-time profiling performance**. In practice, when we set synchronization point timers cross CPUs and GPUs, the hidden code between the CPU host side timer and the timer report by events in black box GPU, such as data copy and work-items assignment function, are wrapped [23, 51] in the proprietary, inaccessible binary dynamic link library libOpenCL.so. The hidden code random overhead requires efficient fitting to minimization in dynamic environments, and output in real-time is challenged.

Our solution. We propose a light-weight time synchronization mechanism that minimize the influence on GPU execution. It is based on a key observation that when the CPU pushes the kernel data from the Command Queue to the Host Queue, both CPU and GPU could record the timestamp of this enqueue event based on their own clock source. Let Δt be the clock offset; t_{qu} and τ_{qu} be the timestamp of the enqueue event based on the CPU clock and the GPU clock, respectively. Accordingly, we can estimate the clock offset using the equation $\Delta t = \tau_{qu} - t_{qu}$.

Taking a retrospect of Figure 6, we define δ as the measurement offset due to the hidden code execution. We have $\delta = \tau_{qu} - t'_{qu} - \Delta t$. Figure 7(a) shows the probability distribution function (PDF) of δ measured in our benchmark study. We observe that δ follows the normal distribution. This motivates us to leverage linear least squares (LLS) fitting to estimate δ. The LLS fitting can be formulated as follows:

$$min \sum_{i=0}^{n} (\tau_{qu_i} - t'_{qu_i})^2, i = 1, 2, ..., n \quad (1)$$

where the index i represents i^{th} event recorded based on the CPU and GPU clock. In the process of LLS fitting, we find outliers of reported timestamps always exist, as shown in Figure 7(b). As LLS fitting is sensitive to outliers, we set a threshold σ to exclude outliers. *i.e.*, if $| \delta_i | > \sigma$, the algorithm will reuse the previous fitting result to calculate synchronized timestamps. Although LLS is generally lightweight,

Algorithm 1 Dynamic Fitting Strategy

```
1:  Input :Pair<$t'_{qu}$,$\tau_{qu}$> $\mathcal{T}$, *sycn_q, *$n$, $\sigma$, $k$
2:  Output :Synchronized GPU timestamp
3:  function SYNC($\mathcal{T}$, &sycn_q ,$\sigma$, &$n$, $k$)
4:      $n$ ++; // Sliding window count
5:      if | $\mathcal{T}$.second - $\mathcal{T}.first$ |≤ $\sigma$ then
6:          sycn_q.pop(); sycn_q.push($\mathcal{T}$);
7:          if $n > k$ then // Re-fit with recent $\mathcal{T}$
8:              $n$ = 1;
9:              LR.Regression(sycn_q); // LLS fitting
10:             queue<pair<double, double» empty;
11:             swap (empty, sycn_q); // Clear sycn_q queue
12:         end if
13:         return LR.Predict($\mathcal{T}$.second);
14:     else if | $\mathcal{T}$.second - $\mathcal{T}.first$ |> $\sigma$ then
15:         // Reuse previous fitting results
16:         return LR.Predict($\mathcal{T}$.second)
17:     end if
18: end function
```

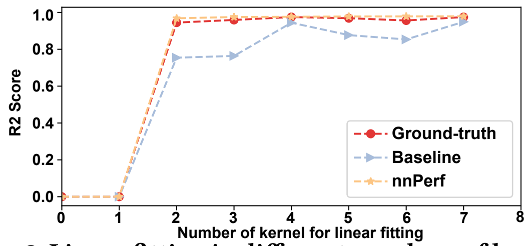


Figure 8: Linear fitting in different numbers of kernels. Averaging over 50 inferences on a Xiaomi10 phone.

running it per kernel execution is computationally intensive. Since kernels are submitted (*i.e.* clflush [51]) in group, We run the LLS fitting every k consecutive kernel in a group and apply the result to the next k kernels. k is hardware-specific, representing the number of kernels running at the hardware accelerator concurrently. This approach can effectively reduce the LLS overhead, allowing per-kernel time offset correction with a single multiply-add operation. Algorithm.1 shows the sketch of this algorithm.

Kernel injector. The LLS fitting relies on σ and k to proceed. However, at the beginning of model inference (*i.e.*, synchronize the first kernel in the first-time model inference), there is no historical data for determining σ and k. To address this cold-start issue, at the beginning of the first-time model inference, we proactively inject a group of empty kernels to the head of Command_queue. These injected kernels will run ahead of the real model kernels, providing timestamp readings for LLS fitting. Figure 8 compares the fitting accuracy of nnPerf with a baseline and ground truth in different numbers of injected kernel settings. nnPerf achieves consistently high R2 scores when injecting more than two

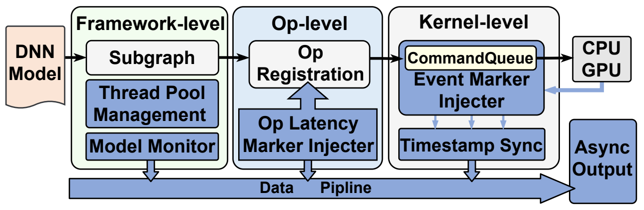


Figure 9: nnPerf implementation breakdown. Each design module in nnPerf is highlighted in blue.

kernels, outperforming the baseline (using the time offset estimated in the previous inference to the current inference) by 21.09%. Let Δt_1, Δt_2, and Δt_3 be the time offset estimated based on the three empty kernels, we estimate δ based on the two-sigma rule using the following equation:

$$\sigma = 2\sqrt[2]{\frac{1}{3}[(\Delta t_1 - m)^2 + (\Delta t_2 - m)^2 + (\Delta t_3 - m)^2]} \qquad (2)$$

where m is the average value of three time offset estimations, *i.e.*, $m = \frac{1}{3}(\Delta t_1 + \Delta t_2 + \Delta t_3)$. nnPerf keeps updating δ regularly as the algorithm proceeds. These three empty kernels introduce an ignorable delay to the model latency based on our benchmarks.

4 NNPERF IMPLEMENTATION

We implement nnPerf based on TFLite framework [31], with around 3,000 lines of C++ code. To facilitate system deployment, we compile nnPerf into a TFLite Android ARchive (AAR) file. Developers can lunch model profiling by simply replacing the default TFLite AAR or the nightly-snapshot library in their android projects with our customized AAR wrapper. nnPerf supports mapping the symbolic link of the result file to other paths by remounting partition in specified memory, allowing developers to build their own pipelines.

Figure 9 shows nnPerf implementation breakdown. On a high-level, the implementations are grouped into three modules. The first module is implemented on the framework level, including the thread pool management and functions related to model e2e latency profiling. The second module is implemented on the operator level, responsible for monitoring the operator latency. The third module is implemented on the kernel level, including the event marker injector and time synchronization. The last module is an efficient data transmission pipeline, where we amortize the IO operation overhead by proposing an asynchronized output queue.

Reducing system overhead. Following the Linux OS data monitoring solution for mobile devices (i.e., sysfs [63]), nnPstores profiling results on the user data partition (through IO writing) in real-time, allowing mobile users to access them without root permission. A DNN model usually contains tens of or even hundreds of kernels. Due to the asymmetry between kernel execution time (as small as 50 μs) and IO speed (around 100 μs), IO operation will become the bottleneck that blocks the subsequent operations. Caching the profiling results on RAM may alleviate IO operations. However, it will lead to a memory explosion, intensifying memory contentions between the DNN model and the profiler. For instance, the memory consumption due to data logs can easily go beyond 100 MB in around 30 s based on our benchmark study on a Xiaomi10 smartphone.

We leverage multi-threading to address this issue. Specifically, we build a thread pool, with each thread dedicated to a kernel for latency calculation and the follow-up IO operation (*i.e.*, storing the kernel latency to the user data partition). These threads can work asynchronously without blocking each other. To minimize the thread creation overhead, the thread pool will be built before the model compiling. At the run-time, we pair up each kernel and a thread chronologically in the event function registration phase. The multi-thread is set in a similar way for operators and the e2e model inference latency profiling. The pool size is set to 16 by default. Upon the completion of the first inference, the pool size grows automatically after knowing the number of kernels or operators of the current model.

5 EVALUATION

In this section, we first describe the experiment setup (§5.1), followed by the system performance (§5.2). The system overhead evaluation (§5.3) concludes.

5.1 Experiment Setup

We evaluate nnPerf on four mobile hardware platforms listed in Table 2. Unlike previous works [14] that disable small cores on mobiles to prevent scheduling interference on model inference, we deploy DNN models on these mobile platforms without making changes to their default hardware scheduling policies. We believe doing so can fairly reflect how mobile devices being used in real-world scenarios. Our evaluation is based on eight representative mobile DNN network architectures, including Mobilenetv3-small [16, 50], Squeeze- [27, 52], EfficientNets [9, 70], MNasNet1.0 [14, 69], ResNetV2 101 [24], MobileBert [15], SSDMobile [28] and Esrgan [11] , which are designed for image recognition, natural language processing, and style transfer tasks.

5.2 System Performance

Model-level and Op-level profiling accuracy. We evaluate the accuracy of e2e and op latency profiling. We connect Xiaomi10 to a PC with a USB cable and then use the AGI (*i.e.*, profiler trace mode) to lunch the DNN model on the phone. AGI will record the ground truth e2e model inference latency and send it back to the PC at the end of model inference. In the meantime, nnPerf embedded in the app will profile the DNN model e2e latency and ops latency in real-time. Since the e2e latency profiled by AGI is captured from

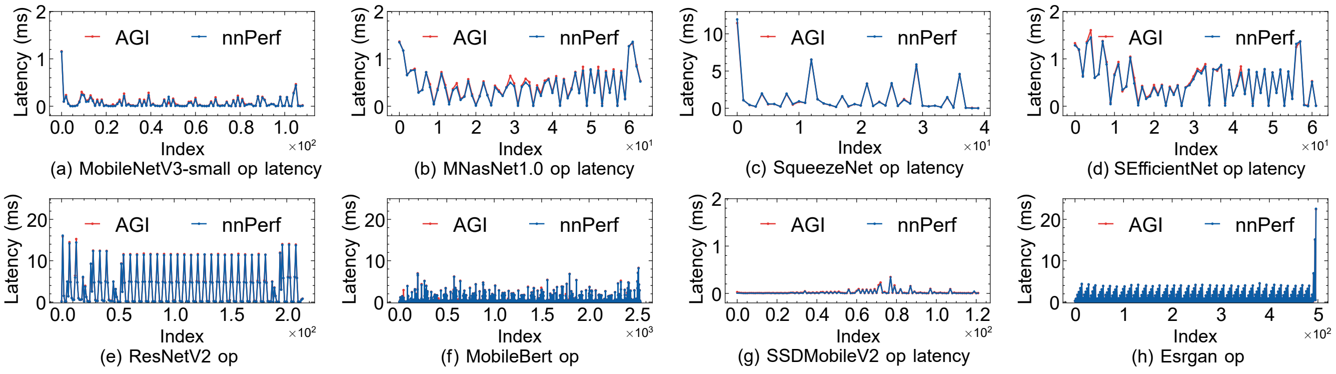


Figure 10: Comparison of real-time on-device nnPerf and offline with an auxiliary device AGI profiled op inference latency.

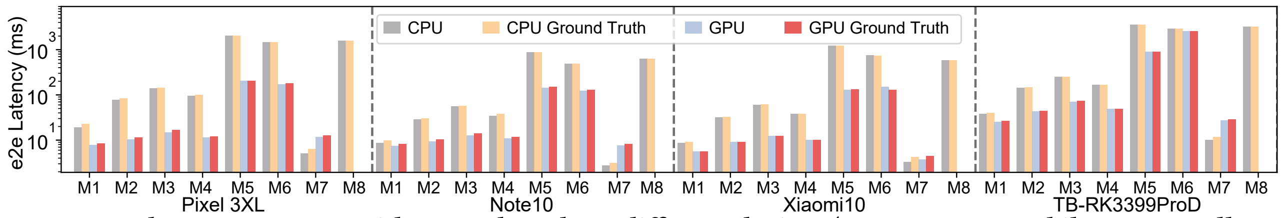


Figure 11: e2e latency compare with ground truth on different devices (M1 to M8 are mobilenetv3-small, mnasNet1.0, squeezeNet, efficientNet, resNetV2 101, mobileBert, ssd-mobileV2 and esrgan in sequence).

the TFLite API, the latency measured by AGI will be slightly higher than nnPerf.

Figure 10 compares the op latency between AGI and nnPe- across eight DNN models on Note10. We use Android GPU Inspector (AGI) [5], the android's official API trace tool, to obtain the op latency groundtruth. For the same reason as e2e results, the op latency of each model reported by AGI is slightly higher than the result reported by nnPerf. Compared to the groundtruth, nnPerf achieves an average error of 0.014 ms on the op latency. Figure 11 shows the e2e latency of four different DNN models (averaged over 4,000 inferences) running on four devices' CPU and GPU processors, respectively. As shown, nnPerf achieves consistently high e2e latency profiling accuracy on four different model architectures (with an average gap of 1.84% and 0.28% with respect to the ground truth on CPU and GPU, respectively).

Kenrel-level latency groundtruth. All existing real-time on-device profilers do not support kernel-level latency profiling. Besides, the benchmark tool random input way ignored other underlying expenses (*e.g.*, the impact of pipeline costs). We instead measure the kernel execution latency implicitly with AGI and use these latency measurements as the ground truth. Specifically, we profile the kernels of a given DNN model and then copy the first kernel and inject this copy to the position right after the first kernel in this DNN model. We then measure the e2e latency of this DNN model using AGI and subtract the e2e latency of the original DNN model to estimate the execution latency of the first kernel. We repeat this step for each kernel to get its

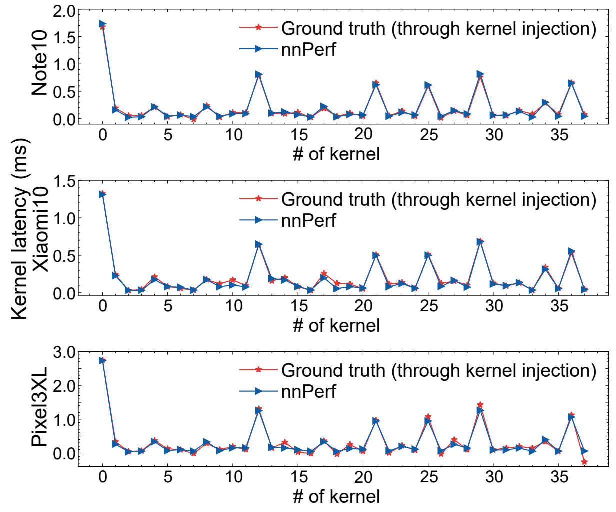


Figure 12: Comparison of nnPerf and groundtruth method profiled kernel inference latency. The results are averaged over 40,0000+ inferences.

latency and use these kernel latencies as the ground truth. We run this benchmark carefully to ensure no other foreground and background applications co-running on the mobiles that may disturb the kernel running latency. Moreover, we also fix the frequency of the CPU and GPU to avoid the impact of DVFS.

Kenrel-level profiling accuracy. Figure 12 shows the Squ- kernel latency profiled by nnPerf and the ground truth (thro- kernel injection) on three different mobile platforms. nnPerf

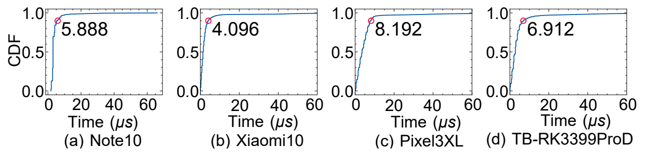


Figure 13: The CDF of per-kernel time offset after time synchronization on different platforms.

Table 3: Latency Overhead on different devices.

Device	Per Inference	Per Kernel	Percentage
(**A**) Google Pixel 3XL	0.020ms	0.201μs	**0.104**%
(**B**) Samsung Note10	0.020ms	0.227μs	**0.231**%
(**C**) Xiaomi 10	0.017ms	0.230μs	**0.116**%
(**D**) TB-RK3399ProD	0.037ms	0.241μs	**0.098**%

achieved a consistently high kernel latency profiling accuracy, with an average error of 0.001 ms, 0.012 ms, and 0.007 ms for the Samsung Note10, Xiaomi10, and Pixel 3XL platforms, respectively. These results demonstrate that our kernel profiling design can accurately measure kernel execution time.

Time synchronization accuracy. Since no profilers are currently available to report the kernel execution timeline in actual DNN runtime inference scenarios (*i.e.*, Run DNN in an app with ot), we build a test model using 100 repeated layers (*i.e.*, CONV2D + Hardwish). Each layer will execute using the same kernel after compiling. We measure the e2e inference latency of this test model using AGI and estimate the kernel execution latency by dividing the e2e latency by 100. Since these kernels are executed chronologically, we estimate the launching time and ending time of each kernel by adding the kernel latency gradually and further using these timestamps as the ground truth. We measure the difference between the ground truth and the timestamps reported by nnPerf and show the CDF of these time offsets in Figure 13. As shown, we observe that the 90^{th} percentile of time offset on three mobile devices are 5.888 μs, 4.096 μs, 8.192 μs, and 6.912 μs. Recall that the kernel latency is at a few hundred microseconds level, such a small time offset would not impact the accuracy of timeline plotting.

5.3 System Overhead

nnPerf runs on the mobile device to profile the model inference activity online. In this section, we evaluate the system overhead by measuring the extra model inference latency and power consumption due to the running of nnPerf.

Latency overhead. To evaluate the extra latency introduced by nnPerf, we measure the e2e model inference latency by co-running four versions of nnPerf on the mobile platform and subtracting the e2e model inference latency measured by TFLite (without nnPerf co-running on the mobile device). These three versions are nnPerf with preempt kernel, nnPerf

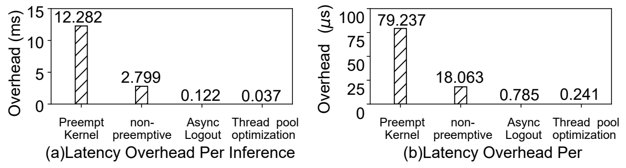


Figure 14: Extra inference latency due to nnPerf.

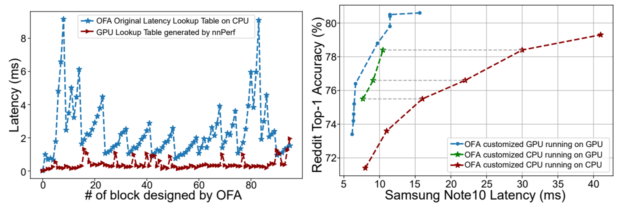
Table 4: 90-th percentile result of the time synchronization overhead under different CPU frequencies.

Device	825MHz	1056MHz	1497MHz	1804MHz
(**A**) Google Pixel 3XL	3.802μs	3.958μs	4.167μs	4.323μs
(**B**) Samsung Note10	3.646μs	3.802μs	3.542μs	3.177μs
(**C**) Xiaomi 10	4.323μs	4.167μs	3.281μs	3.698μs
(**D**) TB-RK3399ProD	3.802μs	3.958μs	4.167μs	4.323μs

with the non-preempt kernel, nnPerf with asynchronized logout, and nnPerf with thread pool optimization (the final version). Figure 14(a) shows the result. nnPerf introduces around 12.282 ms inference overhead when allowing kernel preempt. As we change it to non-preempt execution, the extra inference latency drops to 2.799 ms. It then goes down to 0.122 ms and further to 0.037 ms as we put on the asyncorinized logout and thread pool optimization module. The result clearly demonstrates that the final version of nnPerf introduces negligible delays to model inference. We repeat the above experiment on kernel-level profiling and show the result in Figure 14(b). As expected, the nnPerf with thread pool optimization introduces merely 0.241 μs latency to each kernel. Table 3 further shows the latency overhead of nnPerf on different devices. We observe a relatively stable performance across both kernel and e2e latency over all four types of mobile platforms.

Time synchronization overhead. To evaluate the overhead of time synchronization, we simulate the resource dynamic environment by switching between four levels of CPU frequencies. As shown in Table 4, the synchronization overhead in all scenarios is smaller than 4.4μs. Compared to the synchronization period in several milliseconds, the overhead is negligible. Besides, the thread pool used in nnPerf can further reduce the influence of time synchronization on overall latency.

Power overhead. As an on-device profiler, the power footprint of this tool is also crucial to its success, particularly on mobile platforms. To understand the power overhead of nnPerf, we run a DNN model on a mobile platform in the presence and absence of nnPerf. In each setting, we use the power monitor HVPM [12] to measure the real-time power consumption of this smartphone. We turn off the screen to avoid unnecessary power consumption during the measurement and ensure the initial temperature and the frequency of each experiment is the same. The mobile platform runs



(a) (b)

Figure 15: (a) Op-level latency table vs. kernel-level latency table. (b) Migrate OFA to kernel-granularity.

its default DVFS strategy throughout the experiment session. The result shows that nnPerf brings merely 0.605% extra power consumption to the mobile device, demonstrating its small power footprint.

6 CASE STUDY

In this section, we conduct a case study to show how we leverage nnPerf to migrate Neural Architecture Search (NAS) from CPU to mobile GPUs. To build an efficient neural network for mobile platforms, existing NAS algorithms leverage op-level latency information to guide the neural architecture search. Although a large body of work takes FLOPs or MACs as an indirect proxy to guide the neural architecture search [61, 67, 81], the resulting model usually fails to meet the system-level objectives (SLOs) because the reduction of FLOPs does not necessarily translate to the model latency reduction [69, 71, 75, 77].

Methodology. We demonstrate the feasibility of migrating OFA (once-for-all) NAS [37] from CPUs to GPUs. Given an e2e model inference latency requirement, the original OFA NAS takes advantage of operator-level latency lookup tables to search for a neural network architecture that satisfies this latency requirement. Similarly, we leverage nnPerf to build five kernel-level latency lookup tables [37] dedicated to the GPU accelerator on a Samsung Galaxy Note10 phone as shown in Figure 15(a). Each entry of the kernel latency is averaged over 200 inferences. Next, we write a wrapper to map the kernel-level latency reported by nnPerf to the same format as OFA designed latency lookup table for each block of DNN model structure. Finally, we feed these kernel latency lookup tables to OFA and search for an optimized neural network architecture dedicated to the mobile GPU on this Samsung Galaxy Note10 phone.

Result. Figure 15(b) shows the e2e latency and the ImageNet top-1 accuracy of these models. We use the models customized by OFA for Note10 CPU and running on CPU (*i.e.*, the original OFA) as the baseline. As shown, running this baseline on GPU (green line) directly brings about 52.07% model latency reduction, respectively. With a 1.19% top-1 accuracy increase, the e2e latency of our migrated GPU model (*i.e.*, kernel-oriented model running on GPU) is further reduced by 27.12%, demonstrating the efficacy of nnPerf.

7 RELATED WORKS

Model latency profiling. Model optimization is crucial to DNN applications. The industry has released a set of performance analysis tools that can measure DNN model efficiency at various granularities [2, 5, 18, 19, 22, 25, 29, 30, 33]. Among them, NVIDIA Nsight series [18, 19] are heavily used by model developers to investigate correlations, dependencies, activity, bottlenecks, and resource allocations. However, these tools are dedicated to NVIDIA GPUs which are rarely used by mobile platforms. There are also plenty of profilers designed for mobile platforms. Profiliers such as AGI [5], Tensorflow Benchmark [30], and Perffeto [22] allows developers to measure the model latency on mobile CPUs at the granularity of operators. Xcode-Instrument [33] further extends the operator profiling to mobile GPUs. However, all of these profiles do not support online profiling; hence they cannot be leveraged to characterize the model efficiency in the presence of resource dynamics over time. Besides, none of these solutions support kernel-level profiling on mobile GPUs, as summarized in Table 1.

Model latency estimation. Another trail of research is dedicated to model latency estimation. Early works estimate the latency based on hardware-agnostic FLOPs (*i.e.*, number of multiply-add operations) [64, 67, 81]. However, they are not accurate because the reduction of FLOPS does not consistently translate into model latency reduction [73, 76]. Later on, a group of works propose to store the latency of each layer in a look-up table and then estimate the e2e latency by summing up the latency of model layers [37, 38]. However, these designs are still not accurate because the operator fusion may happen in the model compiling. Notice that, a recent work named nnMeter [77] proposes to *estimate* the kernel latency through manually designed testing cases. The developers can sum up the kernel latency of a DNN model to estimate the e2e execution latency of their models. However, there are two major issues with this method. First, the kernel latency estimated by nnMeter is not accurate because it does not exclude the kernel waiting time. Second, the kernel latency is estimated in constant resource settings, and their efficacy is vulnerable to resource dynamics when deployed on mobile devices.

8 CONCLUSION

In this paper, we have presented the design, implementation, and evaluation of nnPerf, an online, on-device DNN model inference profiler. nnPerf provides μs-level profiling accuracy on the operator- and kernel-latency on both mobile CPUs and GPUs. We believe nnPerf could be further leveraged to understand the model inference activity at fine-grained granularity, identify bottlenecks in model run-time efficiency, and further assist run-time model optimization to deal with resource dynamics on mobile platforms.

REFERENCES

[1] Adreno GPU OpenCL Support. https://en.wikipedia.org/wiki/Adreno.
[2] Adreno GPU Profiler. https://developer.qualcomm.com/forums/software/adreno-gpu-profiler.
[3] Adreno GPU Profiler by Qualcomm. https://developer.qualcomm.com/forums/software/adreno-gpu-profiler?order=last_updated&sort=asc.
[4] AI creates new levels for Doom and Super Mario games. https://www.bbc.com/news/technology-44040007.
[5] Android GPU Inspector. https://gpuinspector.dev/.
[6] Android Studio Profiler. https://developer.android.com/studio/profile/android-profiler.
[7] Chrono. https://cplusplus.com/reference/chrono/.
[8] Deep Learning for Games. https://developer.nvidia.com/deep-learning-games.
[9] Efficientnet-b0 TFLite model TensorFLow HUb. https://tfhub.dev/tensorflow/lite-model/efficientnet/lite0/fp32/2.
[10] Emmagee - a practical, handy performance test tool for specified Android App. https://github.com/NetEase/Emmagee.
[11] Esrgan TFLite model TensorFLow HUb. https://tfhub.dev/captain-pool/esrgan-tf2/1.
[12] High Voltage Power Monitor. https://www.msoon.com/online-store/High-Voltage-Power-Monitor-p90002590.
[13] Mali GPU OpenCL Support. https://en.wikipedia.org/wiki/Mali_(GPU).
[14] MNasNet_1.0_224 TFLite model TensorFLow HUb. https://tfhub.dev/tensorflow/lite-model/mnasnet_1.0_224/1/default/1.
[15] MobileBert 101 TFLite model TensorFLow HUb. https://tfhub.dev/iree/lite-model/mobilebert/fp32/1.
[16] MobileNetV3-small-100-224 TFLite model TensorFLow HUb. https://tfhub.dev/iree/lite-model/mobilenet_v3_small_100_224/fp32/1.
[17] ncnn. https://github.com/Tencent/ncnn.
[18] NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.
[19] NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.
[20] Out-of-order execution. https://en.wikipedia.org/wiki/Out-of-order_execution.
[21] paddle-lite. https://github.com/PaddlePaddle/Paddle-Lite.
[22] Perfetto. https://ui.perfetto.dev/.
[23] Portable Computing Language (PoCL). http://portablecl.org/.
[24] ResnetV2 101 TFLite model TensorFLow HUb. https://tfhub.dev/tensorflow/lite-model/resnet_v2_101/1/default/1.
[25] Snapdragon Profiler. https://developer.qualcomm.com/software/snapdragon-profiler.
[26] SoloPi. https://github.com/alipay/SoloPi.
[27] Squeezenet TFLite model TensorFLow HUb. https://tfhub.dev/tensorflow/lite-model/squeezenet/1/default/1.
[28] SSDMobileV2 TFLite model TensorFLow HUb. https://tfhub.dev/iree/lite-model/ssd_mobilenet_v2_100/fp32/default/1.
[29] Tencent Perfdog. https://perfdog.qq.com/.
[30] Tensorflow Benchmark tools. https://www.tensorflow.org/lite/performance/measurement.
[31] TensorFlow Lite. https://www.tensorflow.org/lite/.
[32] vDSO. https://man7.org/linux/man-pages/man7/vdso.7.html.
[33] XCode Instrument . https://developer.apple.com/documentation/xcode-release-notes/xcode-14_1-release-notes.
[34] M. Almeida, S. Laskaridis, A. Mehrotra, L. Dudziak, I. Leontiadis, and N. D. Lane. Smart at what cost? characterising mobile deep neural networks in the wild. In *Proceedings of the 21st ACM Internet Measurement Conference*, page 658–672, New York, NY, USA, 2021. Association for Computing Machinery.
[35] S. G. Bhaskaracharya, J. Demouth, and V. Grover. Automatic kernel generation for volta tensor cores. *arXiv preprint arXiv:2006.12645*, 2020.
[36] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu. Neuralpower: Predict and deploy energy-efficient convolutional neural networks. In *Asian Conference on Machine Learning*, pages 622–637. PMLR, 2017.
[37] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.
[38] H. Cai, L. Zhu, and S. Han. Proxylessnas: Direct neural architecture search on target task and hardware. *arXiv preprint arXiv:1812.00332*, 2018.
[39] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.
[40] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11398–11407, 2019.
[41] S. Dhar, J. Guo, J. J. Liu, S. Tripathi, U. Kurup, and M. Shah. A survey of on-device machine learning: An algorithms and learning theory perspective. 2(3), 2021.
[42] H. Du, P. Li, H. Zhou, W. Gong, G. Luo, and P. Yang. Wordrecorder: Accurate acoustic-based handwriting recognition using deep learning. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1448–1456. IEEE, 2018.
[43] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *The Journal of Machine Learning Research*, 20(1):1997–2017, 2019.
[44] B. Fang, X. Zeng, and M. Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 115–127, 2018.
[45] J. Fromm, M. Cowan, M. Philipose, L. Ceze, and S. Patel. Riptide: Fast end-to-end binarized neural networks. *Proceedings of Machine Learning and Systems*, 2:379–389, 2020.
[46] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun. Single path one-shot neural architecture search with uniform sampling. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XVI 16*, pages 544–560. Springer, 2020.
[47] M. Han, H. Zhang, R. Chen, and H. Chen. Microsecond-scale preemption for concurrent {GPU-accelerated}{DNN} inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 539–558, 2022.
[48] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European conference on computer vision (ECCV)*, pages 784–800, 2018.
[49] Y. He, X. Zhang, and J. Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE international conference on computer vision*, pages 1389–1397, 2017.
[50] A. Howard, M. Sandler, G. Chu, L.-C. Chen, B. Chen, M. Tan, W. Wang, Y. Zhu, R. Pang, V. Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1314–1324, 2019.
[51] L. Howes and A. Munshi. *The OpenCL Specification 2.0.* Khronos OpenCL Working Group, 2015.

[52] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[53] W.-M. C. W.-C. W. C. G. S. H. Ji Lin, Ligeng Zhu. On-device training under 256kb memory. *arXiv preprint arXiv:2206.15472*, 2022.

[54] H. Jin, Q. Song, and X. Hu. Auto-keras: An efficient neural architecture search system. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1946–1956, 2019.

[55] W. Jung, T. T. Dao, and J. Lee. Deepcuts: a deep learning optimization framework for versatile gpu workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 190–205, 2021.

[56] J. M. Kim, Y. G. Kim, and S. W. Chung. Stabilizing cpu frequency and voltage for temperature-aware dvfs in mobile devices. volume 64, pages 286–292, 2015.

[57] S. Kim, K. Bin, S. Ha, K. Lee, and S. Chong. ztt: learning-based dvfs with zero thermal throttling for mobile devices. *GetMobile: Mobile Computing and Communications*, 25(4):30–34, 2022.

[58] J. Lee, Y. Liu, and Y. Lee. Parallelfusion: towards maximum utilization of mobile gpu for dnn inference. In *Proceedings of the 5th International Workshop on Embedded and Mobile Deep Learning*, pages 25–30, 2021.

[59] R. Liang, T. Cao, J. Wen, M. Wang, Y. Wang, J. Zou, and Y. Liu. Romou: Rapidly generate high-performance tensor kernels for mobile gpus. In *The 28th Annual International Conference On Mobile Computing And Networking (MobiCom 2022)*. ACM, February 2022.

[60] J. Lin, W.-M. Chen, Y. Lin, j. cohn, C. Gan, and S. Han. Mcunet: Tiny deep learning on iot devices. In H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 11711–11722. Curran Associates, Inc., 2020.

[61] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*, 2018.

[62] Z. Liu, H. Mu, X. Zhang, Z. Guo, X. Yang, K.-T. Cheng, and J. Sun. Metapruning: Meta learning for automatic neural network channel pruning. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 3296–3305, 2019.

[63] P. Mochel. The sysfs filesystem. In *Linux Symposium*, volume 1, pages 313–326. The Linux Foundation San Francisco, CA, USA, 2005.

[64] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz. Pruning convolutional neural networks for resource efficient inference. *arXiv preprint arXiv:1611.06440*, 2016.

[65] H. Qi, E. R. Sparks, and A. Talwalkar. Paleo: A performance model for deep neural networks. In *International Conference on Learning Representations*.

[66] Qualcomm. *Qualcomm Snapdragon Mobile Platform OpenCL General Programming and Optimization* . Khronos OpenCL Working Group, 2021.

[67] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *Proceedings of the aaai conference on artificial intelligence*, volume 33, pages 4780–4789, 2019.

[68] B. Sharma, C. Gupta, H. Li, and Y. Wang. Automatic lyrics-to-audio alignment on polyphonic music using singing-adapted acoustic models. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 396–400. IEEE, 2019.

[69] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.

[70] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[71] X. Tang, S. Han, L. L. Zhang, T. Cao, and Y. Liu. To bridge neural network design and real-world performance: A behaviour study for neural networks. *Proceedings of Machine Learning and Systems*, 3:21–37, 2021.

[72] H. Wang, J. Zhai, M. Gao, Z. Ma, S. Tang, L. Zheng, Y. Li, K. Rong, Y. Chen, and Z. Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021.

[73] Y. Wang, S. Ye, Z. He, X. Ma, L. Zhang, S. Lin, G. Yuan, S. H. Tan, Z. Li, D. Fan, et al. Non-structured dnn weight pruning considered harmful. *arXiv preprint arXiv:1907.02124*, 2, 2019.

[74] M. Xu, M. Zhu, Y. Liu, F. X. Lin, and X. Liu. Deepcache: Principled cache for mobile deep vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 129–144, 2018.

[75] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 285–300, 2018.

[76] F. Yu, Z. Xu, T. Shen, D. Stamoulis, L. Shangguan, D. Wang, R. Madhok, C. Zhao, X. Li, N. Karianakis, et al. Towards latency-aware dnn optimization with gpu runtime analysis and tail effect elimination. *arXiv preprint arXiv:2011.03897*, 2020.

[77] L. L. Zhang, S. Han, J. Wei, N. Zheng, T. Cao, Y. Yang, and 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*, page 81–93, New York, NY, USA, 2021. ACM.

[78] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang. Improving neural network quantization without retraining using outlier channel splitting. In *International conference on machine learning*, pages 7543–7552. PMLR, 2019.

[79] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[80] Y. Zhou, S.-M. Moosavi-Dezfooli, N.-M. Cheung, and P. Frossard. Adaptive quantization for deep neural network. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[81] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8697–8710, 2018.