DNNTune: Automatic Benchmarking DNN Models for Mobile-cloud Computing

CHUNWEI XIA, JIACHENG ZHAO, HUIMIN CUI, and XIAOBING FENG, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, China and School of Computer Science and Technology, University of Chinese Academy of Sciences, China
JINGLING XUE, School of Computer Science and Engineering University of New South Wales, Australia

Deep Neural Networks (DNNs) are now increasingly adopted in a variety of Artificial Intelligence (AI) applications. Meantime, more and more DNNs are moving from cloud to the mobile devices, as emerging AI chips are integrated into mobiles. Therefore, the DNN models can be deployed in the cloud, on the mobile devices, or even mobile-cloud coordinate processing, making it a big challenge to select an optimal deployment strategy under specific objectives.

This article proposes a DNN tuning framework, i.e., DNNTune, that can provide layer-wise behavior analysis across a number of platforms. Using DNNTune, this article further selects 13 representative DNN models, including CNN, LSTM, and MLP, and three mobile devices ranging from low-end to high-end, and two AI accelerator chips to characterize the DNN models on these devices to further assist users finding opportunities for mobile-cloud coordinate computing. Our experimental results demonstrate that DNNTune can find a coordinated deployment achieving up to 1.66× speedup and 15% energy saving comparing with mobile-only and cloud-only deployment.

CCS Concepts: • **Computer systems organization → Embedded software**;

Additional Key Words and Phrases: DNN, mobile-cloud computing, heterogeneous computing

ACM Reference format:
Chunwei Xia, Jiacheng Zhao, Huimin Cui, Xiaobing Feng, and Jingling Xue. 2019. DNNTune: Automatic Benchmarking DNN Models for Mobile-cloud Computing. *ACM Trans. Archit. Code Optim.* 16, 4, Article 49 (December 2019), 26 pages.
https://doi.org/10.1145/3368305

New article, not an extension of a conference paper. This work is supported in part by the National Key R&D Program of China (2016YFB1000402), the National Natural Science Foundation of China (61802368, 61521092, 61432016, 61432018, 61332009, 61702485, and 61872043), CCF-Tencent Open Research Fund, and an Australian Research Council grant (RG171010).

Authors' addresses: C. Xia, J. Zhao (corresponding author), H. Cui, and X. Feng, State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, No.6 Kexueyuan South Road Zhongguancun, Haidian District, Beijing, China, 100190, School of Computer Science and Technology, University of Chinese Academy of Sciences, No.19(A) Yuquan Road, Shijingshan District, Beijing, China, 100049; emails: {xiachunwei, zhaojiacheng, cuihm, fxb}@ict.ac.cn; J. Xue, School of Computer Science and Engineering University of New South Wales, Gate 14 on Barker Street, Sydney, Australia, NSW 2052; email: jingling@cse.unsw.edu.au.

1544-3566/2019/12-ART49
https://doi.org/10.1145/3368305

1 INTRODUCTION

In recent years, Deep Neural Networks (DNNs) have been widely adopted in a variety of applications, such as speech recognition, image classification, and natural language processing. Traditionally these applications are deployed as Request-Response services in the cloud; however, in the meantime, they are moving to the mobile devices, as the emerging AI chips are integrated into smartphones [3]. For example, Huawei Kirin 970 chipset has integrated a dedicated neural processing unit, thus enabling both cloud-based and on-device AI processing [4]. Gartner predicted that 80% of smartphones will have on-device AI capabilities by 2022 [8]. Therefore, the DNNs can be processed both by the cloud and mobiles, thus bringing rich computational resources to mobile users.

The cloud and mobile devices exhibit diverse performance and energy consumption, thus it is a big challenge to deploy DNNs between the cloud and mobile coordinately. In particular, when a DNN is completely processed in the cloud, the computation latency is minimized, but the raw input data need to be transferred to the cloud and introduces extra network transmission overhead. However, when a DNN is totally processed on mobile devices, the network transmission overhead is eliminated, but the computation latency will increase. Let us take two DNN examples on a representative Mobile ARM CPU, Qualcomm Snapdragon 625. For a small DNN model such as MLP that contains 5 layers, the mobile processing takes 136 ms, thus the latency is acceptable from the users' perspective. However, for a large DNN model such as ResNet-50 that contains 50 layers, the mobile processing takes 616 ms, thus the users' experience is significantly impacted. Therefore, the ResNet-50 is preferred to be processed in the cloud, reducing the latency to 217 ms (with the data transmission overhead counted). This challenge is discussed as "cloud-edge system" in the Berkeley view of system challenges for AI [54]. Meanwhile, researchers have proposed some technologies to support collaborative processing when the cloud is always available [29]. In this article, we assume an ideal connection with the cloud 100% available.

A representative research is Neurosurgeon [44], which automatically partitions DNN models between the mobile device and the cloud at the granularity of neural network layers. Neurosurgeon demonstrates the benefit of collaborative mobile-cloud processing. Mobile-cloud can both reduce the computation latency and energy consumption, which are both critical for user experience. Therefore, we take mobile-cloud collaborative processing as an important aspect for DNN inference. However, determining the optimal partitioning strategy is still a big challenge, since it significantly depends on the workload, the processing framework and the hardware platform, the amount of transferred data, the network bandwidth, and latency. Even worse, the mobile computing exhibits extremely significant diversity, including hardware capacity, vendor-provided libraries, and processing framework. The increasing diversity makes it more challenging to determine an optimal deployment strategy. Neurosurgeon designed the model partitioning mechanism into Caffe and studied how to select the partitioning point and process the model collaboratively, without considering auto-tuning. In comparison, DNNTune is decoupled with specific processing frameworks, and existing frameworks can be integrated into DNNTune. Furthermore, DNNTune focused on the configuration tuning, including the hardware platform and software processing framework. Moreover, DNNTune can provide architecture-oriented analysis results during the tuning process.

In this article, we propose a DNN tuning engine for mobiles, named DNNTune, assisting users to analyze the DNN model's performance and energy across a number of mobile platforms. DNNTune can support a variety of hardware and software frameworks, providing layer-wise execution characteristics for users, and further automatically seek for a partitioning point for mobile-cloud coordinate processing for the certain hardware/software combination.

The key challenge comes from the diversity of software frameworks, DNN model behaviors, and hardware platforms. First, layer-wise performance analysis requires to automatically instrument the various software frameworks, for monitoring and controlling the layer-wise execution. Second, some DNN models include layers running for a very short time, bringing obstacles to obtaining their precise execution time and profiling. Finally, different hardware platforms typically provide different libraries for deep learning; therefore, we need to integrate the various libraries into various software frameworks.

To meet the above three challenges, DNNTune enables users to specify three issues, i.e., how to generate the model for tuning, what the processing platform is (including hardware and software framework together with the DNN libraries), and what the objective is for tuning. First, with these configurations, our DNNTune framework generates the model for the user-selected framework, integrates user-specified libraries into the specified software framework, and instruments the specified software framework to monitor and control the layer-wise execution and profiling. Second, DNNTune leverages the specified software framework to run the DNN model on the mobile and the cloud, respectively, with layer-wise profiling collected. Finally, DNNTune theoretically partitions the DNN model at all possible points for execution, computes the performance and energy consumption for each partitioning point, and obtains the optimal deployment strategy under a certain objective. This article makes the following contributions:

- We propose the DNNTune, a DNN tuning framework, to provide layer-wise characteristics across a variety of software frameworks, DNN models, and hardware platforms. DNNTune can find the optimal mobile-cloud coordinate deployment strategy under a certain objective, with the DNN model being run only twice, one on the mobile and the other in the cloud.
- We propose an adaptive layer-wise profiling mechanism across a variety of software frameworks, DNN models, and hardware platforms. DNNTune can automatically integrate the specified libraries into the specified software framework, and further instrument the framework to monitor and control the layer-wise execution and profiling, automatically adapting to the execution time of the layers.
- We select a variety of DNN models (including CNN, LSTM, and MLP) and three mobile devices (including low-end to high-end smartphones, both CPUs and GPUs) and two AI accelerator chips to evaluate DNNTune. Our experimental results demonstrate that DNNTune can find a coordinated deployment achieving up to 1.66× speedup and 15% energy saving comparing with mobile-only deployment.

This article is organized as follows: Section 2 introduces the DNNTune framework. Section 3 describes our experimental setup and evaluation platforms. Section 4 presents how to find mobile-only optimal deployment strategy using DNNTune. Section 5 describes potential of collaborative mobile-cloud processing with detailed performance analysis. Section 6 analyzes all performance/energy related factors and provides detailed analysis for understanding the experimental results. Section 7 discusses related work. Section 8 concludes.

2 DNNTUNE FRAMEWORK

In this section, we first discuss the challenges of building a DNN tuning framework, then present the building blocks of our DNNTune framework.

2.1 Challenges

The challenge of building a DNN tuning framework for mobiles comes from two factors.

First, the mobile hardware and software exhibits significant diversity. As discussed in Facebook [60], the distribution of peak performance of smartphone SoCs exhibits a wide spread, and there is no standard mobile chipset to optimize for. Furthermore, there are a number of software frameworks and third-party libraries, which are not runnable on all hardware platforms.

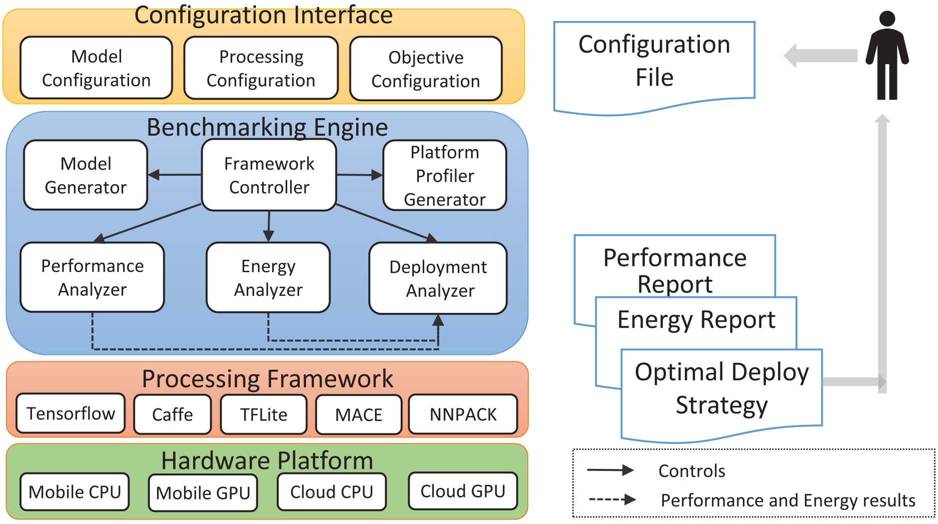


Fig. 1. The DNNTune framework overview.

Therefore, the tuning framework is required to automatically run the DNN models across various hardware/software configurations. **Specifically, given a hardware/software combination, the framework should be able to collect performance related hardware issues and translate them into corresponding software parameters to control the tuning process.**

Second, the DNN model diversity increases the challenge of layer-wise profiling. It is a critical issue to obtain layer-wise characteristics for mobile-cloud coordinate computing. However, the layer execution time exhibits a very large range, from several microseconds to hundreds of milliseconds. For a short term layer whose execution time is shorter than typical profiling intervals, the tuning framework is required to automatically insert glue codes to repeatedly execute the layer, for obtaining its profile.

2.2 DNNTune Framework

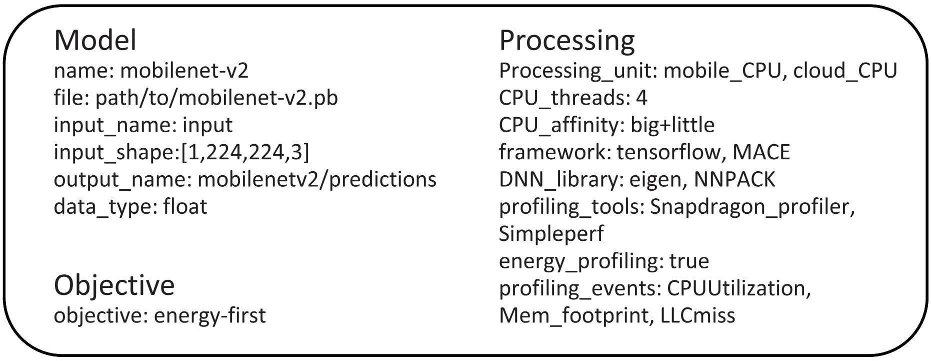
Figure 1 shows the overview of DNNTune, which reads the user's configurations and generates the reports on performance and energy, together with an optimal deployment strategy under certain objective. DNNTune contains two modules, i.e., a configuration interface and benchmarking engine.

The configuration interface provides users an approach to specify the DNN model for analyzing, the processing platform (including hardware platform, software framework, and DNN libraries), and the objective when deploying the model for mobile-cloud collaborative processing.

The benchmarking engine is the core module of DNNTune. As shown in Figure 1, the upper three modules serve to generate the DNN model with proper format ("Model Generator"), generate proper hardware profiler ("Platform Profiler Generator"), and generate the control parameters for the specified processing framework ("Framework Controller"). And the lower three analyzer modules analyze the performance/energy and seek to find an optimal deployment strategy.

2.3 Benchmarking Engine

2.3.1 Configuration Interface. The configuration interface enables users to configure the DNN model, the processing platform, and the analysis objective. Figure 2 shows an example. In particular, in the **Model** section, users can specify the DNN model that can be an existing Caffe or



```
Model
name: mobilenet-v2
file: path/to/mobilenet-v2.pb
input_name: input
input_shape:[1,224,224,3]
output_name: mobilenetv2/predictions
data_type: float

Objective
objective: energy-first

Processing
Processing_unit: mobile_CPU, cloud_CPU
CPU_threads: 4
CPU_affinity: big+little
framework: tensorflow, MACE
DNN_library: eigen, NNPACK
profiling_tools: Snapdragon_profiler,
Simpleperf
energy_profiling: true
profiling_events: CPUUtilization,
Mem_footprint, LLCmiss
```

Fig. 2. An example for the configuration interface.

TensorFlow input, together with the name and shape for input and output. Furthermore, data type can be specified for model quantization.

In the **Processing** section, users can specify the following issues:

- Hardware platform, including CPU, GPU, or AI accelerator for the mobile and cloud, the number of CPU threads, using big/little/big+little cores for mobile computing.
- Software configuration, including the processing framework and DNN library. For example, Figure 2 specifies two frameworks (TensorFlow and MACE) and two libraries (Eigen and NNPACK) for tuning.
- Profiling configuration, including which profiling tool will be used (by far Snapdragon profiler [18] and Simpleperf [17] are considered), whether enabling profiling for energy (performance profiling is always enabled), and the HW&SW events for profiling.

In the **Objective** section, users can specify the tuning objective, i.e., latency-first or energy-first.

2.3.2 Model Generator. The Model Generator serves to convert a Caffe or TensorFlow model to the format of user-specified framework. We select Caffe and TensorFlow format models as the input of DNNTune, since there have been a large number of existing models for them. A set of format translators is integrated in the model generator, which automatically selects the proper translator according to the user-specified input model format and the processing framework to be used. For example, when the user specifies the input model as TensorFlow format (.pb) and the processing framework as MACE, the DNNTune automatically invokes the TensorFlow-to-MACE translator to generate the input for MACE.

2.3.3 Framework Controller. The framework controller is the key component in DNNTune, to obtain compatibility across various software framework and hardware platforms. It serves to translate user configurations into the framework parameters, integrate specified libraries into the specified framework, and instrument the framework.

Configuration. First, the framework controller serves to translate user configurations into corresponding parameters according to specified processing framework. In particular, it reads the configuration file to extract framework-related configurations *on the mobile side*, including the processing unit, CPU thread number, CPU affinity. These configurations will be converted into framework parameters. For example, in MACE, these parameters are as follows: *device*, *omp_num_threads*, $cpu\_affinity\_policy$.

Libraries. Second, the framework controller provides support to switch between different libraries, which can be provided by the hardware vendor or third-party developers. For example,

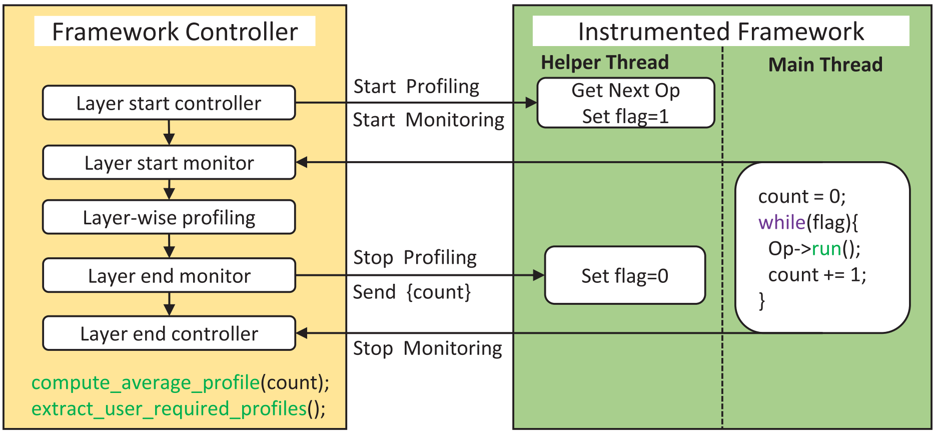


Fig. 3. The instrumentation for layer-wise profiling in DNNTune.

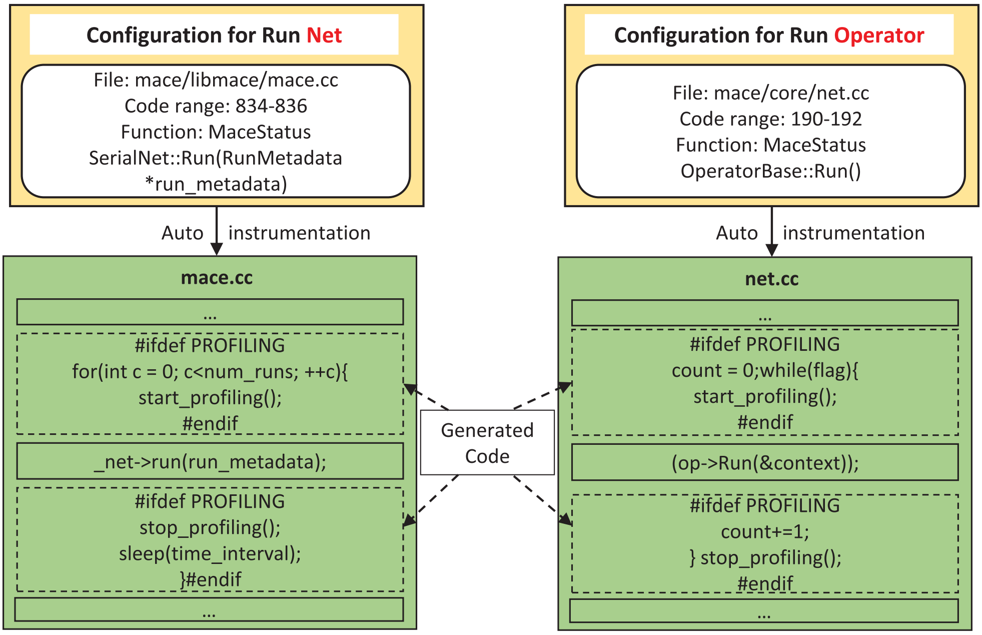
both TensorFlow and TFLite use Eigen as the default DNN library, but the performance of Eigen is not satisfactory. Therefore, these frameworks enable developers to select libraries at building time, such as MKLDNN, cuDNN, NNPACK. Therefore, we build multiple versions beforehand for each framework, with each library corresponding to one version. At tuning time, the framework controller selects the proper version according to the user's configuration.

Instrumentation. The framework controller inserts instrumentation codes into the framework, to monitor and control the layer-wise execution. The challenge comes from the short running layers, for which the execution time is even smaller than typical time interval of profiling tools.

For example, the Snapdragon Profiler samples the system power every 50 ms, while the computation latency of layer "bottleneck_3_2" in MobileNet-V2 is 5 ms. Therefore, to obtain the energy consumption of "bottleneck_3_2," the layer will to be executed iteratively as least 10 times to reach the time interval. We propose a layer-wise monitoring and controlling mechanism as shown in Figure 3. Our key insight is to adaptively control the execution time for each layer, ensuring that short running layers can run multiple times to reach the required time interval. Meanwhile, the long running layers will not be repeated unnecessarily.

As shown in Figure 3, the left side shows the framework controller, and the right side for the instrumented framework. In particular, before a layer is started for execution, the "Layer Start Controller" sends a signal to the instrumented framework. On the other side, we introduce a helper thread to wait for the signal from the layer start controller. When it receives the signal, the shared variable $nlayer\_flag$ is set to 1, indicating that the layer can be started. The main thread is blocked until the $nlayer\_flag$ is set. Afterwards, the instrumented framework sends a signal to the framework controller, notifying that the layer is actually started, then the layer is executed repeatedly. Meanwhile, the framework controller collects the profiling data for this layer. When the execution duration reaches the preset time interval, the "Time Interval Notifier" sends a signal to the instrumented framework, with the $nlayer\_flag$ being reset by the helper thread, indicating that the layer execution can be terminated. Finally, the layer end signal and the layer execution times ($count$) are sent to the framework controller, which invokes $compute\_average\_profile$ to compute the profiling for one execution of the layer.

Automatic Instrumentation. We observed that typical DNN programming frameworks leverage dataflow models and invoke DNN operations via *op->run* in the graph execution. Therefore, we enable users to specify the file name and function name of the graph execution, and our



```
...
#ifdef PROFILING
for(int c = 0; c<num_runs; ++c){
start_profiling();
#endif
_net->run(run_metadata);
#ifdef PROFILING
stop_profiling();
sleep(time_interval);
}#endif
...
```

```
...
#ifdef PROFILING
count = 0;while(flag){
start_profiling();
#endif
(op->Run(&context));
#ifdef PROFILING
count+=1;
} stop_profiling();
#endif
...
```

Fig. 4. Mechanism for auto instrumentation.

framework controller automatically seeks for the *op->run* in the function and inserts instrumentation codes as described above. Figure 4 uses MACE as an example to demonstrate how the automatic instrumentation works. The left part in white color shows the user specification, and the right side in green color shows the generated instrumentation.

2.3.4 Performance and Energy Analyzer. Performance and Energy Analyzer serves to analyze the profiling results and forwards the results to Deployment Analyzer. The Performance Analyzer associates the profiling results to the execution of operations and analyzes the performance of each operation such as GFLOPS, LLC misses, and Instructions Per Cycles(IPC). The Energy Analyzer computes the energy consumption of each operation based on the computation latency and mobile system power reported by the profiler.

2.3.5 Deployment Analyzer. The deployment analyzer serves to determine the optimal partitioning strategy using the following steps:

Step 1. Running and profiling. For a given DNN model d, DNNTune first runs it totally on the mobile and collects the layer-wise profiles d_m, then runs it totally in the cloud and collects the layer-wise profiles d_c.

Step 2. Computing for each partitioning point. Afterwards, DNNTune partitions the DNN model at all possible partitioning points, thus gets a set of partitioned models P, with each element corresponding to a partitioning point p. For each $m \in$ P, the analyzer automatically partitions the model at p, and we denote the first part as $m1$ and the second part as $m2$. $m1$ and $m2$ will be executed on the mobile and cloud, respectively, and their execution time and energy consumption can be computed from d_m and d_c, denoted as T_m, T_c, E_m, and E_c.

For wireless communication, we measure the end-to-end latency of 3G, LTE, and Wi-Fi on several mobile devices using TestMyNet [21] following Kang [44], computing the data transfer cost

Table 1. DNN Specifications

Name	Input	Output	Layers	# Params	FLOPs
rnn_ptb_small	[20, 200]	[20, 10K]	2	2.65M	14.8M
DeepSpeech	[16, 494]	[16, 29]	6	29M	464M
mnist_mlp	[784]	[10]	5	13.9M	13.9M
Transformer	[1,7]	[1, 14]	12	49M	-
Inception-V1	[224, 224, 3]	[1K]	22	6.79M	3.19 G
Inception-V3	[299, 299, 3]	[1K]	48	22.75M	6 G
ResNet-50	[224, 224, 3]	[1K]	50	25.6M	3.8 G
Vgg16	[224, 224, 3]	[1K]	16	138M	16 G
MobileNet-V1	[224, 224, 3]	[1K]	15	4.2M	576M
MobileNet V2	[224, 224, 3]	[1K]	20	3.4M	300M
SqueezeNet-V11	[227, 227, 3]	[1K]	15	1.2M	360M
ShuffleNet-V2 0.5×	[224, 224, 3]	[1K]	15	1.4M	41M
DeepLab-V3	[513, 513, 3]	[65, 65, 21]	20	2.1M	17.8 G

using the equation

$$T_t = T_{ttl} + v_p/b, \tag{1}$$

$$E_t = w * T_t, \tag{2}$$

where T_{ttl} is the network round-trip latency, v_p is the data volume transferred to the cloud that can be obtained from d_m, b is the network bandwidth, w is the power for the transfer module, e.g., Wi-Fi module.

Therefore, the end-to-end latency and energy consumption, i.e., T and E, can be computed using the following equation:

$$T^p = T_m + T_t + T_c, \tag{3}$$

$$E^p = E_m + E_t, \tag{4}$$

where we omit the energy consumption by the cloud, since we only consider the mobile side for battery saving.

Step 3. Traversing. Finally, DNNTune traverses the set P, to seeking for a partitioning point that is optimal to the user-specified objective.

2.3.6 Platform Profiler Generator. We leverage two tools for designing the platform profiler generator, i.e., Simpleperf and Snapdragon profiler. Simpleperf is a native profiling tool for Android, which provides a command-line interface supporting broadly the same options as the Linux-tool perf, with some Android-specific improvements. Snapdragon profiler is a profiling software developed by Qualcomm for Snapdragon series mobile SoCs, which provides a graphic interface to analyze CPU, GPU, DSP, memory, power, thermal, and network data.

3 EXPERIMENTAL SETUP

3.1 DNN Networks

In this article, we consider three kinds of DNN models, with their parameters listed in Table 1.

- **Convolutional Neural Networks (CNNs):**
  In this work, we select nine popular CNNs: four traditional CNNs (Inception-V1 [55], Inception-V3 [56], Vgg16 [53], and ResNet-50 [40]), four mobile-optimized CNNs

Table 2. Device Specifications for Edge Platforms

Device	OnePlus 5T		OnePlus 3		Redmi Note 4X	
SoC	Qualcomm Snapdragon 835		Qualcomm Snapdragon 820		Qualcomm Snapdragon 625	
Process	10 nm		14 nm		14 nm	
Notation	Mobile A-CPU	Mobile A-GPU	Mobile B-CPU	Mobile B-GPU	Mobile C-CPU	Mobile C-GPU
Platform	Kryo 280	Adreno 540	Kryo [2]	Adreno 530	Cortex A53	Adreno 506
Cores/Compute Units	4+4	256	2+2	256	8	96
L1 Cache	64 I+32 I KB | 64 D +32 D KB	-	32 I + 32 D KB	-	32 KB	-
L2 Chche/On Chip Mem	2+1 MB	1,024 KB	1 MB+512 KB	1,024 KB	1 MB	128 + 8 KB
Freq	4×2.45+4×1.85 GHz	710 MHz	2×2.15+2×1.59 GHz	624 MHz	2.0 GHz	650 MHz
RAM	8 GB LPDDR4X		6 GB LPDDR4		3 GB LPDDR3	
OS	Android 8.1		Android 8.1		Android 7.0	

(MobileNet-V1 [41], MobileNet-V2 [51], SqueezeNet-V11 [43], ShuffleNet-V2 0.5× [48]), and DeepLab-V3 [26] with MobileNet-V2 as network backbones. The first eight models are proposed for image classification and pre-trained on ImageNet dataset [31]. DeepLab-V3 is for image semantic segmentation and pre-trained on pascal-VOC dataset [32].

- **Recurrent Neural Network (RNNs):** RNNs are able to process arbitrary length of input data and gain extensive use in natural language processing. The most widely adopted RNN is Long Short Term Memory Networks (LSTMs). In this work, we use two LSTM models. One is rnn_ptb_small, a small LSTM described in Reference [62], which has two hidden layers with 200 hidden units per layer and is trained on the Penn Tree Bank (PTB) [49] dataset. The other is DeepSpeech [39], which has six layers: three FullyConnected layers, one LSTM layer with 4,096 hidden units, and two FullyConnected layers. The batch size is 16.
- **Multilayer Perceptrons (MLPs):** MLPs are proposed for classifications, which are composed of a chain of fully connected layers. In this article, we use a 5-layer small MLP network proposed in Reference [52], which is designed for hand-written digits recognition of MNIST dataset [47], and a big MLP network proposed in Reference [58], which is designed for language understanding, named as Transformer in this article.

3.2 Hardware Platforms

For the cloud side, we use Intel Xeon CPUs equipped with NVIDIA GPUs for DNN processing. In particular, the CPU processor is Intel Xeon E5-2620 v4 and the GPU is NVIDIA Titan V100.

For mobile devices, we select three distinct smartphones spanning from high end to low end, i.e., OnePlus 5T [15], OnePlus 3 [14], and Redmi Note 4X [16]. Both OnePlus 5T (referred to as `Mobile A`, high end) and OnePlus 3 (`Mobile B`, medium end) leverage ARM's big.LITTLE architecture, which is a heterogeneous computing architecture coupling relatively battery-saving and slow processor cores (LITTLE) with relatively more powerful and power-hungry ones (big). All the cores have access to the same memory regions, and workloads can be swapped between the big and little cores on the fly. Note that `Mobile B` used the Kryo CPU, which is Qualcomm's first custom 64-bit quad-core CPU based on the Arm Cortex-A series processor [2]. Redmi Note 4X (`Mobile C`, low end) consists of eight homogeneous cores. Table 2 depicts the detailed hardware specifications of the three mobile platforms. In our evaluation, we cleared all the background applications, turned off the screen, and switched the phone to fly-mode to avoid resource contention. For the embedded devices, we choose Jetson TX2 [23] as a representative AI platform at the edge.

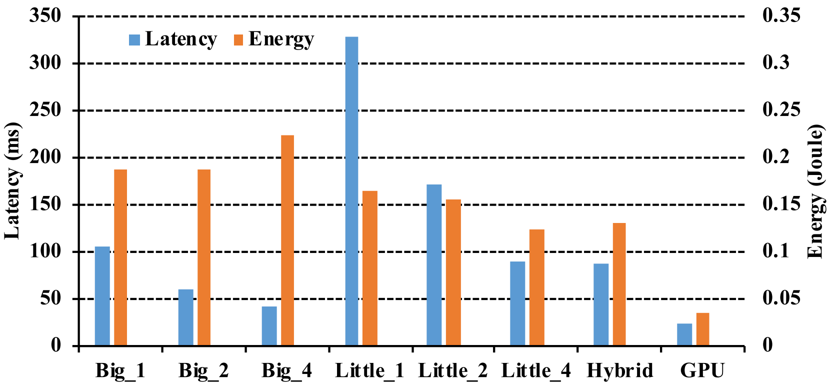


Fig. 5. Latency and energy consumption of MobileNet-V2 on `Mobile A` when using different resources.

It has a 256-core NVIDIA Pascal GPU and the typical power is 7.5W. More hardware specifications can be found at the web page of Reference [23].

3.3 Deep Learning Frameworks

We use three frameworks for evaluation, i.e., TensorFlow [25], MACE, and TFLite. All the three frameworks are compiled with Android NDK r14b (targeting *arm64-v8a*).

TensorFlow version r1.7 is used for evaluating CPUs, both for mobile and cloud processing, which uses Eigen3 [34] as its DNN library. As Tensorflow does not support mobile GPU, we use MACE [13] and TFLite [19] for evaluating GPU, which uses OpenCL to support mobile GPU computing.

4 MOBILE-ONLY OPTIMAL DEPLOYMENT

4.1 Latency-first

Figure 5 shows the latency and energy consumption of MobileNet-V2 on `Mobile A-CPU` when using only big cores (from 1–4 cores), only small cores (from 1–4 cores), all big+little cores, and using GPU, with the backend framework being MACE. In particular, the horizontal axis represents for different configurations, with the blue bars representing the latency against the left vertical axis, and the orange bars representing the energy consumption against the right vertical axis. From Figure 5, we can observe that, when using mobile CPU, 4 big cores provide the best performance, which is 42 ms. When mobile GPU is available, using GPU arrives at the best performance, which is 22 ms. We observed that comparing with little_4, using hybrid cores does not bring extra latency benefit or energy consumption; the reason is that MobileNet-V2 is a small CNN model and cannot fully utilize all the cores. When the workload is distributed across the 4 big cores and 4 little cores, each core works in a low-power mode. Therefore, more cores do not introduce more energy consumption. Detailed analysis will be discussed in Section 6.2.

4.2 Energy-first

In this article, we compute the energy consumption using the equation below:

$$E = T_m(e) * p_m(e), \tag{5}$$

where E is the energy consumption for computing a DNN model in units of Joule, e is the mobile platform, m is the DNN model, $T_m(e)$ is the DNN computation time on e in units of seconds, and $p_m(e)$ is the average system power of e when computing m. In particular, $p_m(e)$ is obtained by

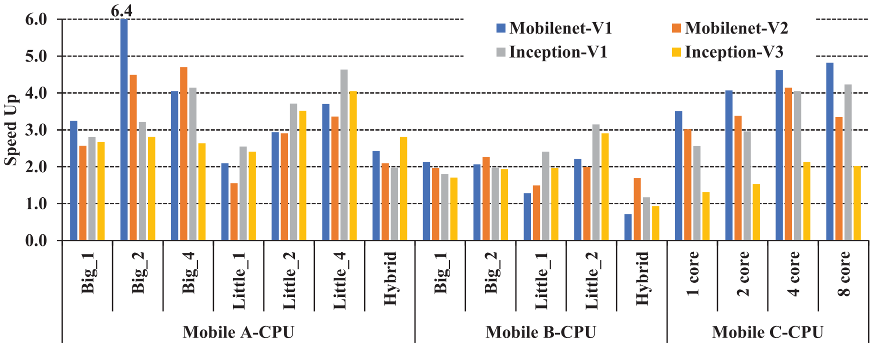


Fig. 6. Speedup of quantized models.

profiling. In particular, we first record and average the system power provided by Snapdragon_Profiler when there is no workload on platform e. We donate it as $p(e_{idle})$. Second, we start our workload m and record the system power as $p_m(e_{active})$. Third, we compute the system power consumed of running m using the equation $p_m(e) = p_m(e_{active}) - p(e_{idle})$. Figure 5 shows that using 4 little cores consumes the least energy (0.123 Joule) on CPU. While using GPU consumes the least (0.035 Joule) comparing with the optimal configurations of CPU.

4.3 Discussing Model Quantization

Representative CNN models are both computationally intensive and memory intensive, and researchers proposed “deep compression” [36], also known as model quantization, which converts the 32-bit float weights to using fewer bits, i.e., 8 bits, without significantly affecting the precision of DNN models.

In this section, we use TFLite [20] and MobileNet-V1, MobileNet-V2, Inception-V1, and Inception-V3 to evaluate the model quantization. The models are trained on ImageNet dataset [31] and we download them from the TFLite hosted models website [22].

The accuracy loss of the model quantization is beyond the scope of this article. In general, users need to re-train the model for quantization, and DNNTune can assist users to deploy the quantized model on the mobile and cloud. As reported in the TFLite official website [24], the accuracy loss of quantized models in inference is little, e.g., 1% accuracy degradation for MobileNet-V2.

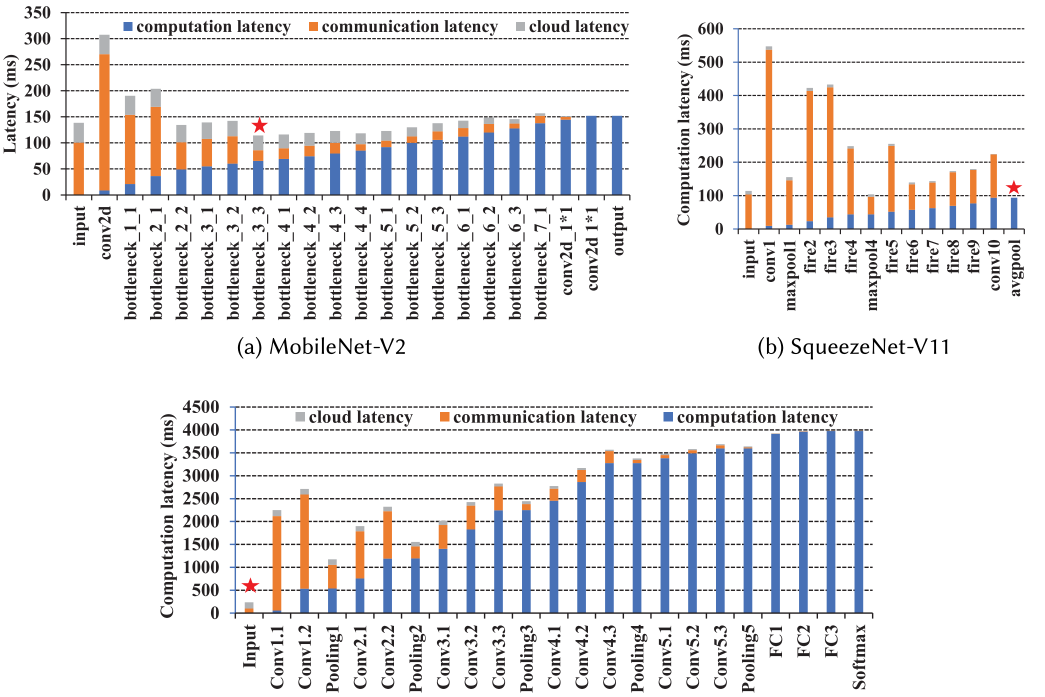
Figure 6 exhibits the significant speedup of quantization over non-quantization on three mobile CPUs. For example, MobileNet-V2 gains 4.7×, 2.0×, and 4.0× on the three mobile CPUs, respectively. The performance benefit comes from two issues. First, the 8-bit integer computation is much faster than the 32-bit floating-point computation. Second, the memory footprint of model weights and inputs is reduced by 75%, e.g., from 39 MB to 10.2 MB for MobileNet-V2 on `Mobile A-CPU`, and the cache miss count is significantly decreased, e.g., from 3.35M to 1.06M for MobileNet-V2 on `Mobile A-CPU` when using four big cores.

5 ANALYZING MODEL PARTITIONING

In this section, we investigate the potential collaborative mobile-cloud processing of DNN computation using DNNTune.

5.1 Evaluation Methodology

In our evaluation, we use CPUs and GPUs of `Mobile A`, `Mobile B`, and `Mobile C` as the mobile devices, and a server CPU as the cloud platform. In particular, the CPU processor is Intel Xeon



(a) MobileNet-V2

(b) SqueezeNet-V11

(c) Vgg16

Fig. 7. Mobile-cloud partitioning latency on mobile A-CPU via Wi-Fi.

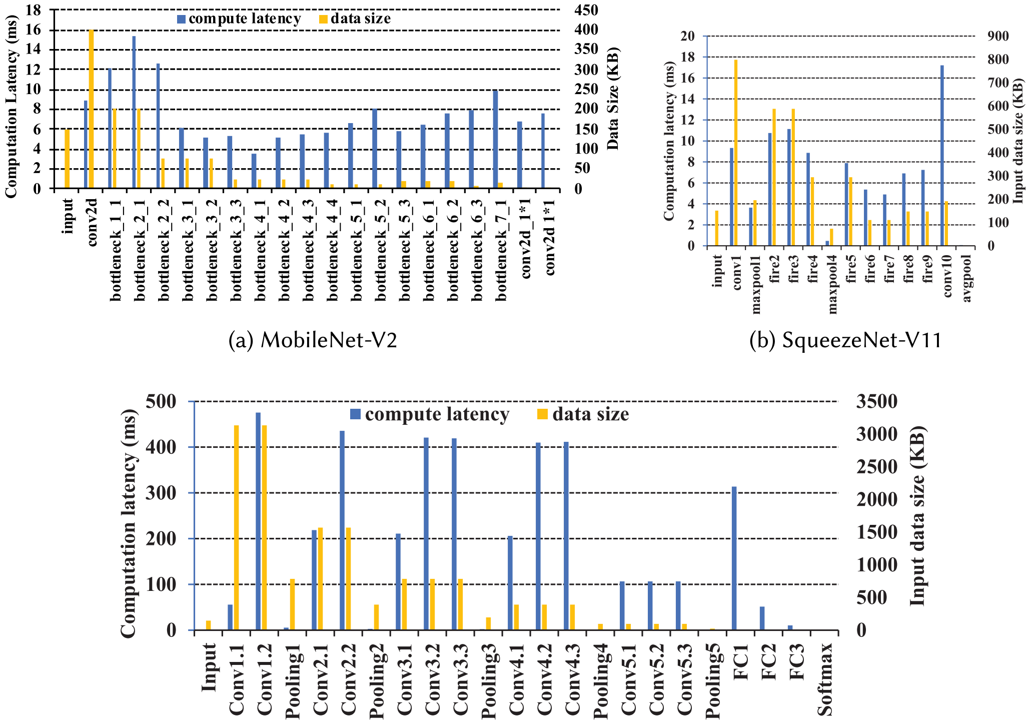
E5-2620 v4. We use TensorFlow for the evaluation of CPUs and MACE for mobile GPUs. The computation latency of executing MobileNet-V2, SqueezeNet-V11, Vgg16, and ShuffleNet-V2-0.5× is 38 ms, 10.7 ms, 90 ms, and 5 ms, respectively, on the cloud platform. For the mobile-cloud communication, we consider 3G, 4G, and Wi-Fi, with latency of transmitting the CNN raw input data to the cloud being 850 ms, 180 ms, and 100 ms [44].

5.2 Latency-first Model Partitioning

For mobile-cloud partitioning, the end-to-end latency is composed of three parts: the computation latency of executing the first part of DNN model on the mobile device, the communication latency of transferring the layer input data and results, and the computation latency of executing the left part of DNN model on the cloud platform. DNNTune traverses all the layers of each DNN model to determine the outperforming partitioning strategy to minimize the computation latency.

5.2.1 Analyzing CNN Behaviors for Model Partitioning. Whether model partitioning can benefit depends on two issues, i.e., the communication overhead at the partitioning point to transfer data from the mobile to cloud, and the benefit of processing the latter layers in cloud. In Figure 7, we take Mobile A-CPU as the platform and Wi-Fi as the network, and present the end-to-end latency when partitioning the model at all possible points, with each bar representing a partitioning point. For each bar, the blue, orange, and grey part represent the computation time on device, the communication time, and the computation time in cloud, respectively.

We use MobileNet-V2, SqueezeNet-V11, and Vgg16 to demonstrate why we can find collaborative processing opportunity for MobileNet-V2 but cannot for SqueezeNet-V11 and Vgg16. The key factor is the input data volume and computation time for each layer. Figure 8 shows the layer-wise



(a) MobileNet-V2

(b) SqueezeNet-V11

(c) Vgg16

Fig. 8. Layer-wise computation and layer input data size latency on Mobile A-CPU.

computation time and the input data volume for the three models, with the horizontal axis representing each layer, the blue bars representing the computation time (against the left vertical axis), and the yellow bars for the data volume (against the right vertical axis). For Figure 7 and Figure 8, we use one core for evaluation (big core for `Mobile A` and `Mobile B`).

- Mobile-cloud processing is optimal.
  As shown in Figure 7(a), for MobileNet-V2, `bottleneck_3_3` is the optimal partitioning point, and the end-to-end latency is 115 ms. In comparison, the mobile-only and cloud-only processing time are 154 ms and 138 ms, respectively. The layer-wise input data volume and computation time are shown in Figure 8(a). We can see that the layer of `bottleneck_3_3` has smaller input data volume, thus transferring the data to cloud would introduce slight overhead, i.e., 20.3 ms. In comparison, the overhead of transferring the input data of the first layer is 100.7 ms. Meanwhile, there is a large amount of computations after the layer, and the computations can achieve significant performance improvement when executing in cloud. It consumes 104.1 ms when executing the layers after the partitioning point of `bottleneck_3_3` on `Mobile A-CPU`, while only 33.2 ms in the cloud. Thus, partitioning at `bottleneck_3_3` brings significant performance improvement.
- Mobile-only is optimal. As shown in Figure 7(b), for SqueezeNet-V11, the mobile-only processing is optimal; the reason is that the computation time is very small on the mobile CPU. It consumes 99.7 ms on the `Mobile A-CPU` while 110.7 ms when executing in cloud (with transferring time counted in).
- Cloud-only is optimal.

Table 3. Mobile-cloud Partitioning on Three Mobile Platforms Using Wi-Fi

		MobileNet-V2					**ShuffleNet-V2 0.5×**				
	conf	mobile	cloud	mobile-cloud	speedup	partition point	mobile	cloud	mobile-cloud	speedup	partition point
Mobile A-CPU	1 big	154	138	115	1.2×	bottleneck_3_3	42	105	86	1.0×	output
	1 little	381	138	138	1.0×	input	112	105	86	1.23×	MaxPool
Mobile B-CPU	1 big	136	138	125	1.10×	bottleneck_5_2	33	105	33	1.0×	output
	1 little	181	138	138	1.09×	input	65	105	65	1.0×	output
Mobile C-CPU	1 core	358	138	138	1.0×	input	99	105	81	1.22×	Stage2
	8 core	190	138	138	1.0×	input	66	105	66	1.0×	output
Mobile C-GPU	GPU	152	138	83	1.66×	bottleneck_4_4	100	105	64.6	1.55×	MaxPool

As shown in Figure 7(c), for Vgg16, the cloud-only is optimal. There are two reasons. First, the data volume of the input data is small, as shown in Figure 8(c), thus it introduces slight data transfer overhead if the model is processed totally in the cloud. Second, the computation amount of the model is large, thus the cloud-only processing will achieve extremely significant performance improvement comparing with mobile processing.

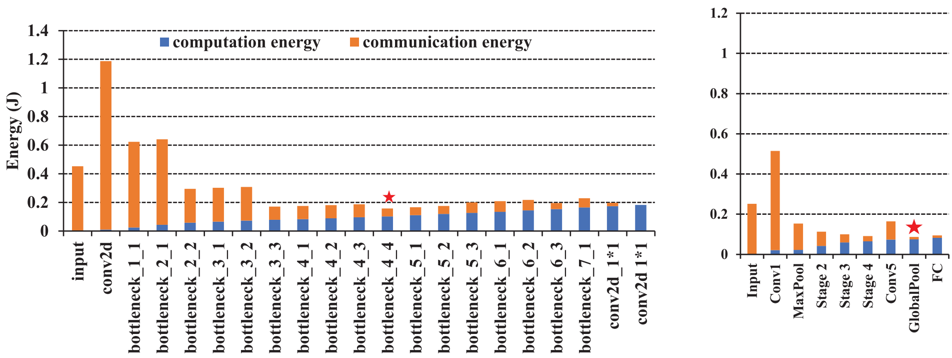
5.2.2 Results for More Models and Platforms. Besides MobileNet-V2, another model that has collaborative processing is ShuffleNet-V2, for which `MaxPool_2` is the optimal partitioning point, and the end-to-end latency is 86 ms, while the mobile-only and cloud-only processing times are 112 ms and 105 ms.

Similar results can be obtained for Mobile B and Mobile C. As Table 3 shows, `Mobile A-CPU` can achieve 1.2× speedup comparing to the mobile-only and cloud-only methods. For ShuffleNet-V2 0.5× on `Mobile B-CPU`, the mobile-only leads to the best latency, as the computation latency is lower than the communication latency.

And for ResNet-50, Inception-V3, DeepSpeech, and DeepLab-V3, cloud-only is optimal, as the computation latency of cloud is much lower than that of mobile device. While for small DNN models like SqueezeNet-V11, mobile-only is optimal, as the computation latency on mobile is lower than the communication latency. Intuitively, a model can benefit from model partitioning if its first few layers rapidly reduce the data volume, while its remaining layers still involve massive computations. Thus model partitioning can process the remaining layers in cloud with little communication overhead. For MLP and LSTM models such as Transformer and DeepSpeech, the cloud-only is optimal. There are two reasons. The first is that the network input data volume is small comparing with CNN models (less than 1 KB vs around 150 KB). The second is that computation latency is much lower on cloud than that of on mobile devices.

5.2.3 Behavior Under 4G/3G. When using 4G and 3G, the optimal partitioning strategy is mobile-only, as the communication overhead is extremely high for MobileNet-V2 and ShuffleNet-V2 0.5×. The communication overhead of transferring the input data of input layer is 180 ms for 4G and 890 ms for 3G, while the computation latency of MobileNet-V2 is 154 ms, 116 ms, and 190 ms for `Mobile A-CPU`, `Mobile B-CPU`, and `Mobile C-CPU`, respectively. Similar results can be found for ShuffleNet-V2 0.5×.

5.2.4 Model Partitioning Results for Using Mobile GPUs. When considering the model partitioning with mobile GPUs, we need to re-collect the computation time on GPUs in Figure 8. Here, we discuss the results and omit the figures due to space limit. For `Mobile A-GPU` and `Mobile B-GPU`, mobile-only is optimal for MobileNet-V2 regardless of using Wi-Fi, 4G, or 3G; the reason is that



(a) MobileNet-V2 (b) ShuffleNet-V2 0.5×

Fig. 9. Mobile-cloud partitioning energy consumption on mobile A-CPU via 4G.

the high-end GPU provides great computing power and reduces the computation latency to 22.1 and 28.3 ms, respectively, which is much lower than the cost of transferring the data to the cloud.

For `Mobile A-GPU` and `Mobile B-GPU`, the mobile-only is optimal for MobileNet-V2 and ShuffleNet-V2 0.5×. For `Mobile C-GPU`, the mobile-only and cloud-only latency are 152.3 ms and 138.7 ms, respectively, while the model partitioning approach is 83.8 ms and the optimal partitioning point is bottleneck_4_4 for MobileNet-V2 when using Wi-Fi. For ShuffleNet-V2 0.5×, the mobile-only and cloud-only latency are 100.9 ms and 105.7 ms, respectively, while the model partitioning approach is 64.6 ms and the optimal partitioning point is Stage2.

5.3 Energy-first Model Partitioning

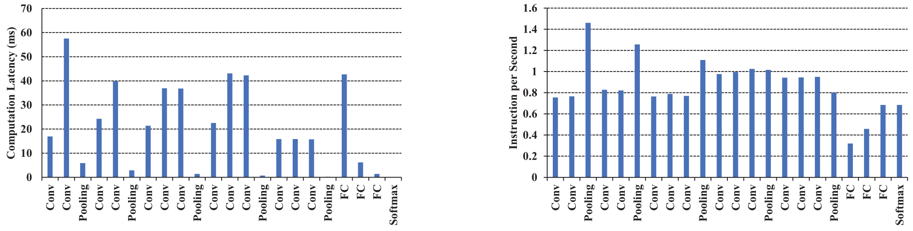
In this section, we switch our objective to energy consumption. We use TensorFlow as the backend framework and three mobile CPUs for evaluation. We choose to use one core in this section (for Mobile A and Mobile B, we use the big core). In particular, the active power of the 3G, 4G, and Wi-Fi module are 0.8, 2.5, and 1.2 watts, respectively. The energy consumption of transferring the raw input data (cloud-only) are 0.712, 0.450, and 0.121 Joule, respectively.

5.3.1 4G. Now, we examine the case when using 4G. Figure 9 shows the energy consumed by mobile A when partitioning the model at each possible point. It shows that the energy consumed for computation increases as the partitioning point moving from left to right, since more and more layers are computed on the device. The communication energy is determined by the volume of data transferred to the cloud. We found that `bottleneck_4_4` is the optimal partitioning point for MobileNet-V2, with the energy of 0.158 Joule, while the cloud-only and mobile-only are 0.453 Joule and 0.182 Joule, respectively. For ShuffleNet-V2 0.5×, `GlobalPool` is the optimal partitioning point, with the energy of 0.091 Joule, while the cloud-only and mobile-only are 0.252 and 0.099 Joule, respectively.

5.3.2 Wi-Fi. When using Wi-Fi, the cloud-only scheme would be optimal for MobileNet-V2, since the Wi-Fi latency is low enough to consume little energy to finish the communication (0.121 Joule), while the energy consumption of mobile-only approach is 0.182, 0.318, 0.268 Joule for `Mobile A-CPU`, `Mobile C-CPU`, and `Mobile C-CPU`, respectively. For ShuffleNet-V2 0.5×, mobile-cloud can save 5.5% to 15.1% energy on the three mobile platforms when compared to best of cloud-only and mobile-only approach, as Table 4 shows.

Table 4. Energy Consumption on Three Mobile CPUs Using Wi-Fi for ShuffleNet-V2 0.5×

	mobile-only	cloud-only	mobile-cloud	energy saving	partition point
Mobile A-CPU	0.088	0.121	0.077	12.5%	Stage2
Mobile B-CPU	0.099	0.121	0.084	15.1%	Stage2
Mobile C-CPU	0.055	0.121	0.052	5.5%	Stage4



(a) latency

(b) IPC

Fig. 10. The layer-wise computation latency and IPC for Vgg16 profiled by DNNTune.

5.3.3 3G. The energy consumption of transferring the input data of the input layer is 0.712 Joule when using 3G network, while the energy consumption of executing the MobileNet-V2 on the `Mobile A-CPU`, `Mobile C-CPU`, and `Mobile C-CPU` is 0.182, 0.318, and 0.268 Joule, respectively. For ShuffleNet-V2 0.5×, the mobile-only energy consumption is 0.088, 0.099, and 0.055 Joule, respectively. Therefore, the mobile-only is the optimal strategy when using 3G, since the communication energy overhead is higher than the computation energy for MobileNet-V2 and ShuffleNet-V2 0.5×.

5.4 Layer-wise Profiling

DNNTune can provide detailed layer-wise profiling information, assisting users to analyze the layer-wise behaviors for DNN models. In this section, we use Vgg16 and Mobile-A platform to demonstrate the profiling results of using Snapdragon_Profiler, for which the profiling interval is 50 ms. We use MACE as the backend software framework.

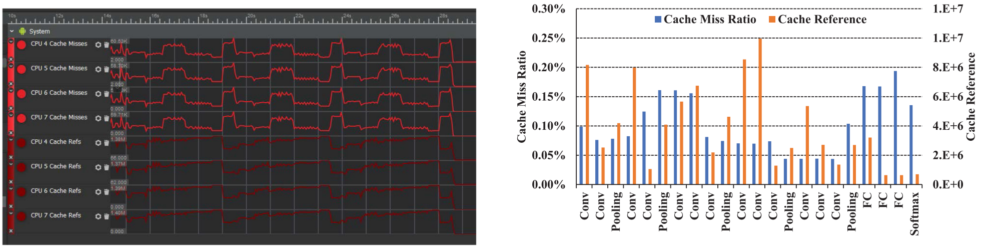
Figure 10(a) shows the profiling result of the layer-wise execution latency for Vgg16. The results show that the convolution layers occupied the most execution time. Using DNNTune, users can easily analyze the time distribution of a DNN model.

Figure 10(b) shows the profiling results of the layer-wise IPC (Instructions per Cycle) for Vgg16. The results show that the IPC of pooling layers is significantly higher than the convolution layers; the reason is there is a large number of integer instructions in pooling layers, while Kryo 280 has improved integer IPC, but lower floating-point IPC [11]. In particular, the integer score/MHz for Kryo 280 is 0.84, while the floating-point score/MHz is only 0.57 [6].

Snapdragon profiler can directly provide the whole-program profiling information for a DNN model, as Figure 11(a) shows the cache misses for Vgg16, which uses 50 ms as the sampling interval. To associate the whole-program cache behaviors to each layer, DNNTune can provide Figure 11. Figure 11 presents the layer-wise cache references and cache miss ratios, respectively, with the layer-wise profiling functionality of DNNTune.

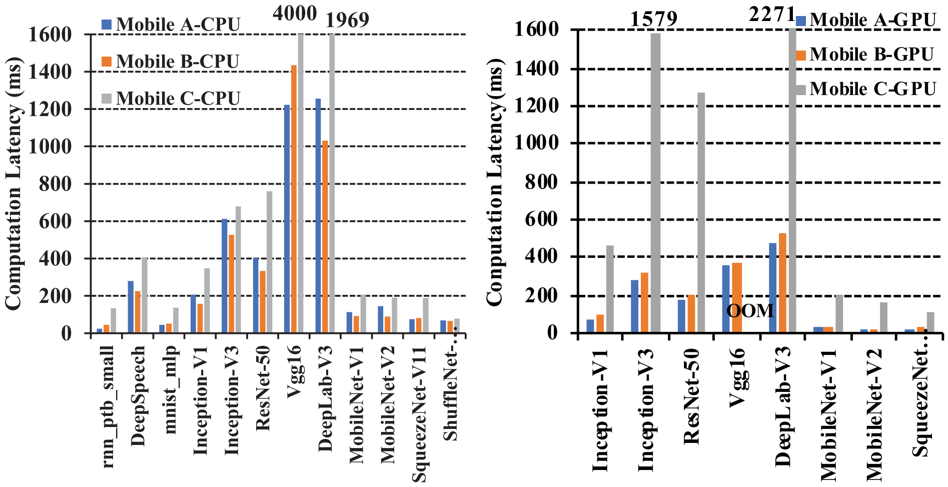
6 ANALYZING INFLUENCE FACTORS

In this section, we analyze five typical influence factors for DNN models on mobile platforms.



(a) Raw profiling data (b) cache references and miss ratio

Fig. 11. The layer-wise cache references and miss ratio for Vgg16 profiled by DNNTune.



(a) CPUs (b) GPUs

Fig. 12. Computation latency of DNN models on mobile CPUs and GPUs with optimal configuration.

6.1 Factor 1: Processing Unit

DNNTune can launch the DNN workload to CPU or GPU by specifying the *Processing_unit* in the configuration interface.

6.1.1 Latency. Figure 12 depicts the computation latency of DNN models on the three mobile CPUs(a) and GPUs(b). For each DNN model, we have three bars representing the results running on the three mobile CPUs, with the blue bars for high-end `Mobile A`, the orange bars for mid-end `Mobile B`, and the grey bars for low-end `Mobile C`. We use Tensorflow for the evaluation of CPU and MACE for the evaluation of GPU.

Figure 12(a) leads to our **Observation 1: Modern mobile CPUs are powerful enough to process small LSTM/MLP models, but cannot process large LSTM/MLP models and most CNNs unless they are specifically optimized for mobiles.**

As Figure 12(a) shows, the latency of small LSTM (rnn_ptb_small) and MLP(mnist_mlp) varies from 24 ms to 139 ms on the three mobile CPUs and is small enough to be directly deployed on mobile CPUs. But things are different for large LSTM/MLP models and CNNs. In particular, for the large LSTM model DeepSpeech, the computation latency varies from 215 ms to 400 ms, indicating that it cannot achieve real-time processing on mobile CPUs. For the large MLP model Transformer, the computation latency varies from 6,903 ms to 15,821 ms, leading to a similar conclusion with

DeepSpeech. Note that for Transformer, the computation latency value is too big to be plotted in Figure 12. Furthermore, the computation latency of traditional CNNs (Inception-V1, Inception-V3, Vgg16, ResNet-50, and DeepLab-V3), ranging from 157 ms to 4,000 ms, is too large to be deployed on mobile CPUs.

The computation of LSTM and MLP depends on the number of hidden nodes, the time steps, and the layers. Take the LSTM models for example. The small LSTM, e.g., rnn_ptb_small, has 20 steps and only 200 hidden nodes, while the large LSTM model, e.g., DeepSpeech, has 16 steps and 4,096 hidden nodes. Therefore, whether a LSTM model can be directly deployed on mobile CPUs depends on the model structure.

Since traditional CNNs are proposed to reach the highest accuracy for image classification, the models are computation-intensive and typically run in the cloud. Mobile-optimized CNN models, such as MobileNet-V1, MobileNet-V2, SqueezeNet-V11, ShuffleNet-V2 0.5×, are designed to effectively maximize accuracy while being mindful of the restricted resources for mobile applications.

On the mobile devices, mobile GPUs are integrated in the SoC typically for image display, and recently researchers are making efforts to leverage them for computation-intensive neural networks [42]. We present the experimental results of the DNN models running on three mobile GPUs in Figure 12(b), with the DNN models run in the framework of MACE. LSTM and MLP are not shown, since MACE does not support them.

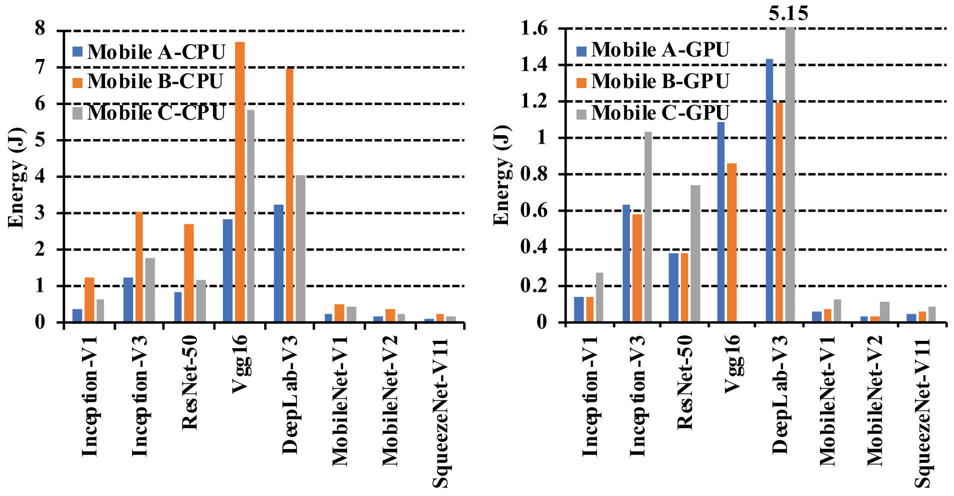
From Figure 12(b), we have **Observation 2: High-end and mid-end mobile GPUs can significantly reduce the computation latency than CPUs, while low-end cannot.**

As Figure 12(b) shows, the latency of MobileNet-V1 on `Mobile A-GPU` and `Mobile B-GPU` is 32 ms and 40 ms, respectively. Using mobile GPU gains 3.5× and 2.3× speedup comparing to the corresponding CPUs, and the average speedup across all the CNN models is 3.3× and 2.3×, respectively. The reason is that the high-end and mid-end GPUs have 256 compute units, while the corresponding CPUs have only 12 and 6 compute units, respectively (each little core has 1 ALU, and each big core has 2 ALUs).

However, things are different for low-end mobile GPUs. Note that we got out-of-memory (OOM) error when running float Vgg16 on `Mobile C-GPU` using MACE, as `Mobile C` only has 3 GB memory while `Mobile A` and `Mobile B` have 8 and 6 GB memory, respectively. The average speedup across all the CNN models is only 0.9× on `Mobile C`, which means that the low-end GPU is even slower than the CPU. This is because the low-end GPU has only 96 compute units. And the benchmarking result of OpenCL-Z [1] demonstrates that the performance of `Mobile C-GPU` is 32.55 GFLOPS, while the performance of `Mobile A-GPU` is 256 GFLOPS in comparison. As Facebook has found that in a median device, the GPU provides only as much as theoretical GFLOPS performance as CPU [60]. As `Mobile C` is the low-end device, the computation latency of GPU is even higher than that of GPU. For DeepLab-V3, `Mobile C-CPU` is better than `Mobile C-GPU`; the reason is that the DNN kernels for CPU are highly optimized, while the openCL for GPU is not highly optimized.

6.1.2 Energy. Figure 13 depicts the energy consumption of all the eight DNN models on the three mobile CPUs and GPUs, with the horizontal axis representing the DNN model names and the vertical axis representing the energy consumption. From Figure 13, we have **Observation 3: Mobile GPUs are more energy-efficient than Mobile CPUs, especially for high-end and mid-end mobile GPUs.**

As Figure 13(a) and Figure 13(b) show, the energy consumption of executing MobileNet-V1 is, respectively, 0.22, 0.51, and 0.35 Joule on the three mobile CPUs when the computation latency is minimized, while it is only 0.062, 0.063, and 0.117 Joule on the corresponding GPUs. On average, CPUs consume 2.9×, 7.5×, and 2.1× more energy than GPUs for `Mobile A`, `Mobile B`, and `Mobile C`, respectively.



(a) CPUs (b) GPUs

Fig. 13. Energy consumption of models on three mobiles.

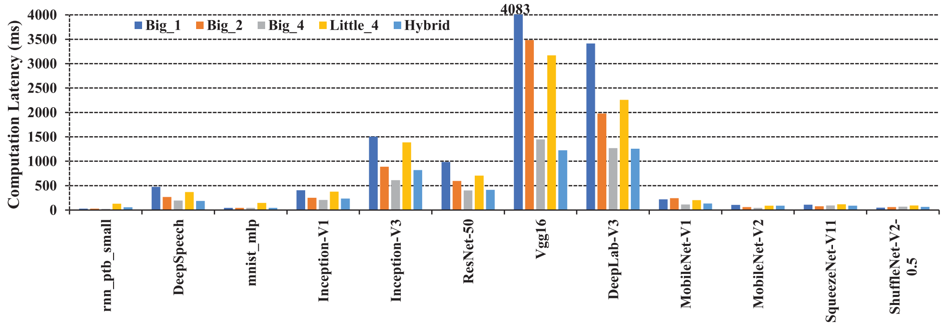


Fig. 14. Performance on `Mobile A-CPU` varying with CPU affinity and thread number.

6.2 Factor 2: CPU Affinity

In this section, we evaluate using big/little/hybrid cores, respectively, with TensorFlow and Eigen as the framework.

6.2.1 Latency. Figure 14 presents the computation latency on the high-end `Mobile A-CPU`. (a) shows the latency varying with the number of threads using only big cores, and (b) shows the computation latency varying with the CPU affinity using big/little/hybrid cores.

Figure 14 leads to **Observation 4: Simultaneously using big and little cores does not always benefit.** As Figure 14 shows, for Inception-V3, the hybrid execution is slower than only using big cores. And DeepLab-V3 exhibits the similar behavior. The reason is that when using big and little cores, the workload is automatically distributed by the Eigen library routine, which is unaware of the heterogenous architecture and evenly distributes the computation to the four big and four little cores. Therefore, DNNTune observed that CPU utilization of the big cores is up to 75% when using only big cores, while it decreases to 36% when using hybrid cores.

6.2.2 Energy. Figure 15 shows the power and energy consumption for MobileNet-V1 running on the three CPUs, with the blue bars representing the power (against the left vertical axis) and the red bars representing the energy (against the right vertical axis).

In particular, for big.LITTLE architectures, the big cores are much more energy-hungry than little cores. For example, the power of one little core is 0.5 watts while the one big core is 1.41 watts for the `Mobile A`, *using the power measuring approach in Section 4.2.* We find that using one little core saves only 18.3% energy comparing to using one big core. We explain the reason using Equation (5). Although the power of using one little core is much lower than that of using one big core (0.50 watt vs 1.41 watts), but the latency is much higher (483 ms vs 209 ms). Therefore, the energy consumption is 0.50 × 483 vs. 1.41 × 209, i.e, 18.3%. Note the high-end `Mobile A` has lower power than the mid-end `Mobile B`, and the reason is that the SoC of `Mobile A` is manufactured in 10 nm FinFET while the `Mobile B` is in 14 nm [12]. Another reason is that the micro-Architecture of `Mobile A-CPU` is up to 30% more energy-efficient than that of `Mobile B-CPU` [7].

6.3 Factor 3: CPU Thread Number

In this section, we evaluate the scalability of DNN models.

6.3.1 Latency. Figure 14 demonstrates that when configuring to use only big cores, CNNs exhibit good scalability with the number of threads. However, the scalability of rnn_ptb_small and mnist_mlp is worse, due to the fact that inference of rnn_ptb_small and mnist_mlp is dominated by the vector-matrix multiplication operation and the Eigen library is not optimized for vector-matrix multiplication to utilize the multi-cores. Similar results can be observed on the little cores.

6.3.2 Energy. Using Equation (5), we further depict the energy consumption of the three devices in Figure 15, and we can have **Observation 5: Using more cores does not always consume more energy.** The reason is that the energy consumption is the product of the power and execution time. For example, on `Mobile A-CPU`, comparing with using only one little core, using four little cores reduces the computation time from 483 ms to 183 ms, and thus reduces the energy consumption by 8%, even if it increases the power from 0.5 watt to 1.41 watts. Therefore, DNNTune automatically tunes the configuration to determine an optimal configuration for user-specified objective.

6.4 Factor 4: Processing Frameworks

In this section, we show how to use DNNTune to evaluate the end-to-end inference latency, energy consumption, and memory footprint of the three DNN frameworks, including Tensorflow, TFLite(float and quantized), and MACE.

Table 5 exhibits the experimental results of using `Mobile A-CPU` using four big cores on the three DNN frameworks. TFLite supports both float and int8 (quantization). "Lat," "En," and "M" are short for "Latency," "Energy," and "Memory," respectively, with the units being "ms," "Joule," and "MB."

6.4.1 Latency. For full precision(float) DNN models, MACE is the optimal framework for latency. The reason is that MACE is specially optimized for every common shape of convolution operations with SIMD optimization, while TFLite adopts the img2col+gemm manner for all convolution operations and uses the Eigen library, which is not highly optimized. However, when quantized DNN models are considered, TFLite quantized is the optimal. Furthermore, TFLite and MACE are much more memory-efficient than Tensorflow, as they are carefully optimized for mobile devices to minimize the memory consumption.

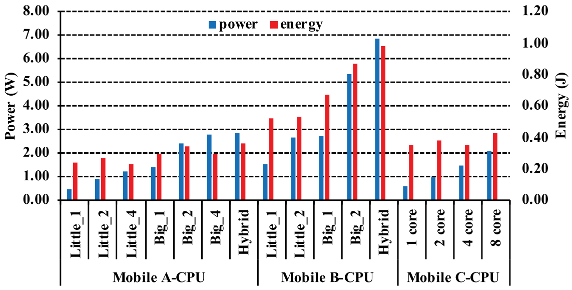


Fig. 15. System power and energy consumption when running MobileNet-V1.

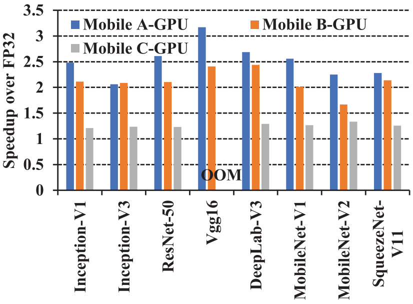


Fig. 16. Speedup of FP16 over FP32.

Table 5. Computation Latency, Energy Consumption, and Memory Usage of Seven CNN Models

	Inception-V1			Inception-V3			ResNet-50			Vgg16			DeepLab-V3			MobileNet-V1		
	Lat	En	M	Lat	En	M	Lat	En	M	Lat	En	M	Lat	En	M	Lat	En	M
TF	207	0.413	110	612	1.812	310	403	1.126	340	1,446	5.881	1,200	1,256	5.259	150	113	0.296	70
TFLite	216	0.663	50	865	2.876	120	479	1.805	120	1,959	8.365	630	1,547	5.047	80	124	0.204	30
MACE	87	0.459	90	574	2.958	100	196	1.023	150	432	2.169	400	843	4.276	100	45	0.237	20
Quant	50	0.194	10	216	0.963	30	122	0.570	30	446	2.202	160	720	3.146	20	24	0.102	10

6.4.2 Energy. There is no such framework that is optimal on energy consumption for all DNN models. Let us take two DNN models for illustration. For MobileNet-V1, TFLite is the optimal with 0.203 Joule energy consumption, while for ResNet-50, MACE is the optimal.

6.4.3 Memory. TFLite and MACE are more memory-efficient than Tensorflow, as they are specially designed for resource-constraint devices. Quantized models can save 74% memory occupation than full precision DNN models on average; the reason is that quantized models use int8 (one byte) data format for weights and input data while full precision models use float (four bytes).

6.5 Factor 5: Mobile GPU Half-precision

Typically, GPUs support half-precision floating point(FP16) operations for increasing performance with negligible precision loss [35]. In this section, we switch to using half-precision floating point number(FP16) for evaluation.

Figure 16 shows the speedup of FP16 over FP32 on three mobile GPUs. Note that the value for Vgg16 and Mobile C-GPU is missing, as when executing Vgg16 FP32 on `Mobile C-GPU`, the model size exceeded the memory size allocated for GPU. In particular, the speedup of Inception-V1 is 2.5× and 2.2× on `Mobile A-GPU` and `Mobile B-GPU`, respectively. The performance comes from two issues. First, high-end GPUs have native hardware that supports for FP16 data type [51]. Second, the half precision reduces the size of weights, thus reduce the off-chip memory traffic. Using `Snapdragon_Profiler`, we found that the max memory bandwidth of `Mobile A-GPU` drops from 2.9GB/s to 2.3GB/s. Thus, for high-end and mid-end mobiles, using DNN models with FP16 can both reduce the the computation latency and the memory consumption.

However, the low-end `Mobile C-GPU` benefits little from using FP16. As Figure 16 shows, `Mobile C-GPU` gains only 1.25× speedup using FP16; the reason is that the low-end Adreno GPU does not support native FP16.

Table 6. DNN Performance on Jetson TX2 and NPU

Platform	DNN Model	Inception-V1	Inception-V3	Resnet-50	MobileNet-V1	SqueezeNet-V11
NPU	Computation Latency (ms)	35	71	53	33	13
	Speedup to best mobile CPU	2.49×	8.08×	3.70×	1.76×	0.79×
	Speedup to best mobile GPU	2.27×	3.91×	3.25×	0.96×	1.81×
Jetson TX2	Computation Latency (ms)	17	91	59	15	12
	Speedup to best mobile CPU	5.12×	6.3×	3.70×	3.83×	0.86×
	Speedup to best mobile GPU	4.67×	3.05×	3.32×	2.11×	1.96×

6.6 Discussion: Evaluating AI Accelerators

In this section, we use two typical hardware platforms to evaluate DNNTune for AI accelerators, one is the NVIDIA Jetson TX2 with a 256-core Pascal GPU; the other is Honor 10 [10] with a Neural Processing Unit (NPU) integrated in Huawei's Kirin 970 SoC. To run DNN models on the NPU, we leverage the HiAI SDK [9] to convert the Caffe models (.caffemodel) and TensorFlow models (.pb) to the specific data format (.cambricon).

Table 6 shows the computation latency and speedup achieved when using the corresponding accelerators for five CNN models. When computing the speedup, we use the outperforming CPU and GPU configurations as the baselines, respectively. In particular, for `Mobile A-CPU`, the best CPU results are the computation latency when using four big cores and MACE as the framework, as discussed in Section 6.4.

Table 6 demonstrates that for most CNN models, AI accelerators can achieve significant performance improvement, except for MobileNet-V1 and SqueezeNet-V11. For traditional CNNs such as Inception and ResNet-50, NPU is 2.49× to 8.08× faster than the best CPU and 2.27× to 3.91× faster than the best GPU. Similar results can be observed for Jetson TX2. MobileNet-V1 and SqueezeNet-V11 are the exceptions, since these two models are optimized for mobile devices using the depthwise convolution, which significantly reduced the computations.

7 RELATED WORK

In this section, we discuss mobile and cloud DNN inference studies that relate to our work.

Mobile deep learning. Xu et al. [61] has presented that Android apps are increasingly adopting DL as core building blocks. The prior efforts to deploy the DNN models on mobile phones can be classified into two aspects: utilizing hardware (for example, DeepMon [42]) and optimizing models (Deep-compression [36]). Thus, our work is inspired by those extraordinary studies and try to characterize the behaviors of DNN models on mobile devices with optimization techniques.

DNN benchmarking. Turner et al. [57] implemented several common DNN compression techniques (weight pruning, channel pruning, and quantization) and evaluated the accuracy, execution time, and memory space on both CPU and GPU. They found that channel pruning can greatly reduce the execution time while weight pruning cannot. Their work provided us observations about model pruning and quantization. Our work not only considers the optimization techniques but also other configurations that affect the inference of DNN models and finds the opportunities for collaborative mobile-cloud computing. Wang et al. [59] proposed a parameterized benchmark suite, ParaDNN, to characterize TPU, GPU, and CPU for training DNN models (including FC, CNN, and RNN). They provided 14 major observations and insights and identify the strength and weakness of TPU, GPU, and CPU. Their work mainly focuses on the DNN training on the servers, while our work focuses on the DNN inference on mobile devices.

Mobile-cloud computing. Mobile-cloud processing has been researched by a number of researchers. CloneCloud [28] investigates how to seamlessly off-load arbitrary functionality code of an application to cloud. Comet [33] works at a more coarse-grained granularity to allow the off-loading of a thread. MAUI [30] leverages the benefits of managed language to profile an application and decide whether each function invocation should be off-loaded based on runtime statistics.

In recent years, a number of works focus on collaboratively processing DNNs using both cloud and edge devices [37, 44]. In particular, MCDNN [37] developed a compiler together with a runtime scheduler to tradeoff between the accuracy and resource consumption, by reasoning about on-device/cloud execution tradeoffs. Another relevant work is Reference [52], which focuses on the training phase of DNNs on cloud platforms, while our work focuses on inference on both cloud and mobiles.

Comparison to Neurosurgeon. Neurosurgeon [44] is the most related work to DNNTune, which proposed an approach to partition the computations of DNNs at the layer granularity and minimize inference latency and/or energy consumption, by uploading certain layers into the cloud. Neurosurgeon implemented the model partitioning mechanism into Caffe. In comparison, DNNTune is a general tuning infrastructure that is decoupled with any specific processing frameworks. For the model partitioning strategy, Neurosurgeon and DNNTune can get the same results, as provided in Section 5. However, as a tuning infrastructure, DNNTune can provide more results for tuning a number of factors, as the hardware platform (Section 6.1, Section 6.6), software processing configurations (Section 6.2, Section 6.3, Section 6.4), and DNN model optimizations (Section 4.3, Section 6.5). Moreover, DNNTune can provide architecture-oriented analysis results (Section 5.4).

DNN characterization and tuning. DNN characterization of mobile devices appeals to researchers. Facebook [60] looked at the mobile phones that Facebook runs on and introduced state-of-the-art DNN inference in the wild. Xu et al. [61] carried out the empirical study on Android apps and answered the questions of what kind of apps and how apps adopt DL. Hanhirova et al. [38] studied the latency and throughput of CNN models on both mobile and cloud platforms. We proposed detailed latency and energy characterization together with quantitative profiling analysis. Furthermore, our DNNTune framework can find the optimal configurations to minimize the execution time or energy consumption, in which those studies are deficient.

Researchers have also focused on the layer-wise profiling of DNN models. Kim et al. [46] analyze performance characteristics of different convolution algorithms for each layer, and they mainly evaluate the training and inference latency of DNN layers; while our framework can provide both software and hardware profiling event metrics of DNN layers. Karki et al. [45] propose a new DNN benchmark suite that can run on any platform that supports CUDA and OpenCL and evaluate five CNNs and two RNNs to analyze the in-depth architectural statistics of these networks. They present the latency and energy consumption breakdown and the instruction types of different layer types. However, using our framework, users can find opportunities for mobile-cloud coordinate computing to reduce latency and energy consumption.

TVM [27], an open-source deep learning software stack that bridges the gap between DNN frameworks and different hardware platforms, can automatically tune the code generation for **DNN operators**, i.e., including tile size, unrolling, and vectorization. Complementarily, our DNNTune framework automatically tunes how to construct a DNN network using the individual operators, which can be tuned using TVM or other tuners. Furthermore, DNNTune framework can also tune the partitioning point for mobile-cloud collaborative processing.

8 CONCLUSION AND FUTURE WORK

In this article, we present a DNN tuning framework, DNNTune, using it to analyze the behaviors for three representative DNN models (CNN, LSTM, and MLP), on three mobile platforms and two hardware platforms with AI accelerators. We evaluate five typical influence factors (including processing unit, CPU affinity, CPU thread number, processing frameworks, and mobile GPU half-precision) regarding the inference latency and energy consumption. DNNTune can report the profiling and tuning results and automatically find the optimal configuration when deploying DNN models on mobile devices. Furthermore, we collect the layer-wise execution time and energy consumption, and our DNNTune framework can find opportunities for mobile-cloud collaborative processing. Experimental results show that DNNTune can achieve 1.66× speedup on inference latency and reduce the energy consumption by 15.1% at most.

In future work, we will study how to support tuning for DNN training in DNNTune, study how to deal with the unreliable network and the performance variability in the wild, and support ONNX [5].

ACKNOWLEDGMENTS

The authors would like to thank all the anonymous reviewers for their valuable comments and helpful suggestions.

REFERENCES

[1] Guohui Wang. 2015. OpenCL-Z Android Official Webpage. Retrieved from http://web.guohuiwang.com/software/opencl_z_android.

[2] Qualcomm Technologies Inc. 2016. Snapdragon 820 Mobile Platform. Retrieved from https://www.qualcomm.com/products/snapdragon-820-mobile-platform.

[3] Apple Inc. 2017. The future is here: iPhone X. Retrieved from https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/.

[4] WikiChip. 2017. Kirin 970—HiSilicon. Retrieved from https://en.wikichip.org/wiki/hisilicon/kirin/970.

[5] Open Neural Network Exchange. 2017. Open Neural Network Exchange. Retrieved from https://github.com/onnx/onnx.

[6] Matt Humrick and Ryan Smith. 2017. The Qualcomm Snapdragon 835 Performance Preview. Retrieved from https://www.anandtech.com/show/11201/qualcomm-snapdragon-835-performance-preview/2.

[7] Michael Passingham. 2017. Snapdragon 835 Benchmarks Revealed: All you need to know about the new chip. Retrieved from https://www.trustedreviews.com/news/snapdragon-835-phones-processor-specs-speed-benchmark-chipset-cores-2944086.

[8] Inc. Gartner. 2018. Gartner Highlights 10 Uses for AI-Powered Smartphones. Retrieved from https://www.gartner.com/en/newsroom/press-releases/2018-03-20-gartner-highlights-10-uses-for-ai-powered-smartphones.

[9] HUAWEI Developer. 2018. HiAI Foundation. Retrieved from https://developer.huawei.com/consumer/cn/hiai#Foundation.

[10] Amazon.com Inc. 2018. Huawei Honor 10. Retrieved from https://www.amazon.com/Huawei-10-128GB-Factory-Unlocked-Smartphone/dp/B07D7GZBDW.

[11] WikiPedia. 2018. Kryo CPU. Retrieved from https://https://en.wikipedia.org/wiki/Kryo.

[12] Qualcomm Technologies Inc. and/or its affiliated companies. 2018. List of Qualcomm Snapdragon systems-on-chip. Retrieved from https://en.wikipedia.org/wiki/List_of_Qualcomm_Snapdragon_systems-on-chip.

[13] MACE Developers. 2018. Mobile AI Compute Engine Documentation. Retrieved from https://mace.readthedocs.io/en/latest/index.html.

[14] LineaseOS Wiki. 2018. OnePlus 3. Retrieved from https://wiki.lineageos.org/devices/oneplus3.

[15] LineaseOS Wiki. 2018. OnePlus 5t. Retrieved from https://wiki.lineageos.org/devices/dumpling.

[16] GSMArena.com. 2018. RedMi Note 4x. Retrieved from https://www.gsmarena.com/xiaomi_redmi_note_4x-8580.php.

[17] Google Developers. 2018. Simpleperf. Retrieved from https://developer.android.com/ndk/guides/simpleperf.

[18] Qualcomm Technologies Inc. 2018. Snapdragon Profiler. Retrieved from https://developer.qualcomm.com/software/snapdragon-profiler.

[19] Google Inc. 2018. Tensorflow Lite. Retrieved from https://www.tensorflow.org/mobile/tflite/.

[20] Google Inc. 2018. TensorFlow Lite is for mobile and embedded devices. Retrieved from https://www.tensorflow.org/lite/.

[21] TestMy.net. 2018. TestMyNet: Internet Speed Test. Retrieved from https://testmy.net/.

[22] Google Inc. 2019. Hosted models. Retrieved from https://www.tensorflow.org/lite/guide/hosted_models.

[23] NVIDIA Corporation. 2019. Jetson TX2 Module. Retrieved from https://developer.nvidia.com/embedded/jetson-tx2.

[24] Google Inc. 2019. Model optimization. Retrieved from https://www.tensorflow.org/lite/performance/model_optimization.

[25] Martín Abadi, Paul Barham, Jianmin Chen, et al. 2016. TensorFlow: A system for large-scale machine learning. In *Proceedings of the OSDI'16*. 265–283.

[26] Liang-Chieh Chen, Yukun Zhu, George Papandreou, et al. 2018. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the ECCV*.

[27] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the OSDI'18*. USENIX Association, 578–594. Retrieved from https://www.usenix.org/conference/osdi18/presentation/chen.

[28] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, et al. 2011. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the EuroSys'11*. ACM, 301–314.

[29] Eduardo Cuervo, Balasubramanian et al. 2010. MAUI: Making smartphones last longer with code offload. In *Proceedings of the MobiSys'10*. 49–62.

[30] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho et al. 2010. MAUI: Making smartphones last longer with code offload. In *Proceedings of the MobiSys'10*. ACM, 49–62.

[31] J. Deng, W. Dong, R. Socher et al. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the CVPR'09*.

[32] Mark Everingham, Luc Gool, Christopher K. Williams, et al. [n.d.]. The Pascal visual object classes (VOC) challenge. *Int. J. Comput. Vision* 88, 2 ([n.d.]), 303–338.

[33] Mark S. Gordon, Davoud Anoushe Jamshidi, Scott A. Mahlke et al. 2012. COMET: Code offload by migrating execution transparently. In *Proceedings of the OSDI*, Vol. 12. 93–106.

[34] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. Retrieved from http://eigen.tuxfamily.org.

[35] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.

[36] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural network with pruning, trained quantization, and Huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[37] Seungyeop Han, Haichen Shen, Matthai Philipose, et al. 2016. MCDNN: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the MobiSys'16*. 123–136.

[38] Jussi Hanhirova, Teemu Kämäräinen, Sipi Seppälä, et al. 2018. Latency and throughput characterization of convolutional neural networks for mobile computer vision. In *Proceedings of the 9th ACM Multimedia Systems Conference (MMSys '18)*. ACM, 204–215. https://doi.org/10.1145/3204949.3204975

[39] Awni Y. Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Y. Ng. 2014. Deep speech: Scaling up end-to-end speech recognition. *CoRR* abs/1412.5567 (2014). Retrieved from http://arxiv.org/abs/1412.5567.

[40] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the CVPR'16*. 770–778.

[41] Andrew G. Howard, Menglong Zhu, Bo Chen, et al. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[42] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. DeepMon: Mobile GPU-based deep learning framework for continuous vision applications. In *Proceedings of the MobiSys'17*.

[43] Forrest N. Iandola, Matthew W. Moskewicz, et al. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[44] Yiping Kang, Hauswald Johann, Cao Gao, et al. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the ASPLOS'17*. 615–629.

[45] Aajna Karki, Chethan Palangotu Keshava, Spoorthi Mysore Shivakumar, Joshua Skow, Goutam Madhukeshwar Hegde, and Hyeran Jeon. 2019. Tango: A deep neural network benchmark suite for various accelerators. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'19)*. IEEE, 137–138.

[46] Heehoon Kim, Hyoungwook Nam, Wookeun Jung, and Jaejin Lee. 2017. Performance analysis of CNN frameworks for GPUs. In *Proceedings of the ISPASS'17*.

[47] Yann LeCun and Corinna Cortes. 2010. MNIST handwritten digit database. Retrieved from http://yann.lecun.com/exdb/mnist/.

[48] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. ShuffleNet V2: Practical guidelines for efficient CNN architecture design. In *Proceedings of the ECCV'18*.

[49] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: The Penn treebank. *Comput. Linguist.* 19, 2 (June 1993), 313–330.

[50] Qualcomm Technologies Inc. 2017. *Qualcomm-SnapdragonTM Mobile Platform OpenCL General Programming and Optimization.*

[51] Mark Sandler, Andrew G. Howard, Menglong Zhu, et al. 2018. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection, and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'18)*.

[52] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. 2016. Benchmarking state-of-the-art deep learning software tools. In *Proceedings of the CCBD'16*. IEEE, 99–104.

[53] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR'15)*. http://arxiv.org/abs/1409.1556.

[54] Ion Stoica, Dawn Song, Raluca Ada Popa, et al. 2017. A Berkeley view of systems challenges for AI. *arXiv preprint arXiv:1712.05855* (2017).

[55] C. Szegedy, Wei Liu, Yangqing Jia, et al. 2015. Going deeper with convolutions. In *Proceedings of the CVPR'15*. 1–9.

[56] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, et al. 2015. Rethinking the inception architecture for computer vision. In *Proceedings of the CVPR'15*.

[57] J. Turner, J. Cano, V. Radu, E. J. Crowley, M. O'Boyle, and A. Storkey. 2018. Characterising across-stack optimisations for deep convolutional neural networks. In *Proceedings of the IISWC'18*. 101–110. DOI:https://doi.org/10.1109/IISWC.2018.8573503

[58] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *CoRR* abs/1706.03762 (2017). Retrieved from http://arxiv.org/abs/1706.03762.

[59] Yu Wang, Gu-Yeon Wei, and David Brooks. 2019. Benchmarking TPU, GPU, and CPU platforms for deep learning. *CoRR* abs/1907.10701 (2019). Retrieved from http://arxiv.org/abs/1907.10701.

[60] Carole-Jean Wu, David Brooks, Kevin Chen, et al. 2019. Machine learning at Facebook: Understanding inference at the edge. In *Proceedings of the HPCA'19*.

[61] Mengwei Xu, Jiawei Liu, Yuanqiang Liu, et al. 2018. When mobile apps going deep: An empirical study of mobile deep learning. *arXiv preprint arXiv:1812.05448* (2018).

[62] W. Zaremba, I. Sutskever, and O. Vinyals. 2014. Recurrent neural network regularization. *ArXiv e-prints* (Sept. 2014). Retrieved from arxiv:1409.2329

Received June 2019; revised October 2019; accepted October 2019