ONCE FOR ALL: TRAIN ONE NETWORK AND SPECIALIZE IT FOR EFFICIENT DEPLOYMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

We address the challenging problem of efficient deep learning model deployment, where the goal is to design neural network architectures that can fit different hardware platform constraints. Most of the traditional approaches either manually design or use Neural Architecture Search (NAS) to find a specialized neural network and train it from scratch for *each* case, which is computationally expensive and unscalable. Our key idea is to decouple model training from architecture search to save the cost. To this end, we propose to train a *once-for-all* network (OFA) that supports diverse architectural settings (depth, width, kernel size, and resolution). Given a deployment scenario, we can then quickly get a specialized sub-network by selecting from the OFA network without additional training. To prevent interference between many sub-networks during training, we also propose a novel *progressive shrinking* algorithm, which can train a surprisingly large number of sub-networks ($> 10^{19}$) simultaneously, while maintaining the same accuracy as independently trained networks. Extensive experiments on various hardware platforms (CPU, GPU, mCPU, mGPU, FPGA accelerator) show that OFA consistently achieves the same level (or better) ImageNet accuracy than SOTA NAS methods while reducing orders of magnitude GPU hours and CO_2 emission than NAS. In particular, OFA requires **16**× fewer GPU hours than ProxylessNAS, **19**× fewer GPU hours than FBNet and **1,300**× fewer GPU hours than MnasNet under 40 deployment scenarios.

1 INTRODUCTION

Deep Neural Networks (DNNs) deliver state-of-the-art accuracy in many machine learning applications. However, the explosive growth in model size and computation cost gives rise to new challenges on how to efficiently deploy these deep learning models on *diverse* hardware platforms, since they have to meet *different* hardware efficiency constraints (e.g., latency, energy). For instance, one mobile application on App Store has to support a diverse range of hardware devices, from a high-end iPhone-11 with a dedicated neural network accelerator to a 5-year-old iPhone-6 with a much slower processor. With different hardware resources (e.g., on-chip memory size, #arithmetic units), the optimal neural network architecture varies significantly. Even running on the same hardware, under different battery conditions or workloads, the best model architecture also differs a lot.

Given different hardware platforms and efficiency constraints (defined as deployment scenarios), researchers either design compact models specialized for mobile (Howard et al., 2017; Sandler et al., 2018; Zhang et al., 2018) or accelerate the existing models by compression (He et al., 2018) for efficient deployment. However, designing specialized DNNs for every scenario is engineer-expensive and computationally expensive, either with human-based methods or NAS. Since such methods need to *repeat* the network design process and *retrain* the designed network from scratch for *each* case. Their total cost grows linearly as the number of deployment scenarios increases, which will result in excessive energy consumption and CO_2 emission (Strubell et al., 2019). It makes them unable to handle the vast amount of hardware devices (23.14 billion IoT devices till 2018[1]) and highly dynamic deployment environments (different battery conditions, different latency requirements, etc.).

This paper introduces a new solution to tackle this challenge – designing a *once-for-all network* that can be directly deployed under diverse architectural configurations, amortizing the training cost. The

[1]https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

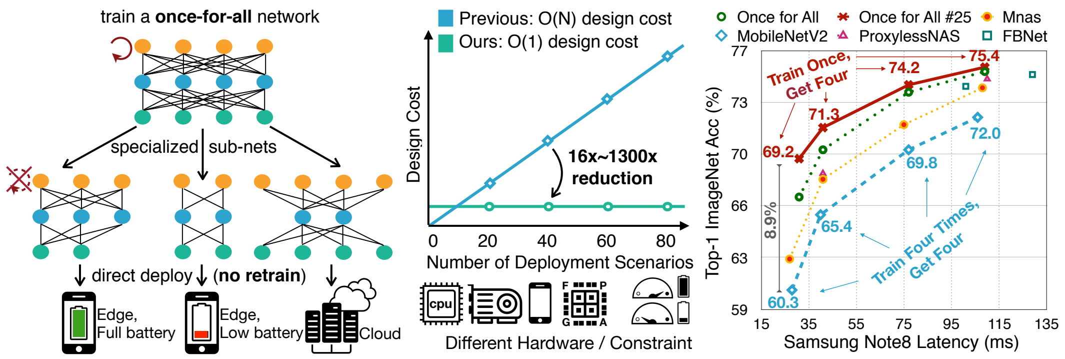


Figure 1: Left: a single once-for-all network is trained to support versatile architectural configurations including depth, width, kernel size, and resolution. Given a deployment scenario, a specialized sub-network is directly selected from the once-for-all network without training. Middle: this approach reduces the cost of specialized deep learning deployment from O(N) to O(1). Right: once-for-all network followed by model selection can derive many accuracy-latency trade-offs by training only once, compared to conventional methods that require repeated training.

inference is performed by selecting only part of the once-for-all network. It flexibly supports different depths, widths, kernel sizes, and resolutions without retraining. A simple example of *Once for All* (OFA) is illustrated in Figure 1 (left). Specifically, we decouple the model training stage and the model specialization stage. In the model training stage, we focus on improving the accuracy of all sub-networks that are derived by selecting different parts of the once-for-all network. In the model specialization stage, we sample a subset of sub-networks to train an accuracy predictor and latency predictors. Given the target hardware and constraint, a predictor-guided architecture search (Liu et al., 2018a) is conducted to get a specialized sub-network, and the cost is negligible. As such, we reduce the total cost of specialized neural network design from O(N) to O(1) (Figure 1 middle).

However, training the once-for-all network is a non-trivial task, since it requires joint optimization of the weights to maintain the accuracy of a large number of sub-networks (more than 10^{19} in our experiments). It is computationally prohibitive to enumerate all sub-networks to get the exact gradient in each update step, while randomly sampling a few sub-networks in each step will lead to significant accuracy drops. The challenge is that different sub-networks are interfering with each other, making the training process of the whole once-for-all network inefficient. To address this challenge, we propose a *progressive shrinking* algorithm for training the once-for-all network. Instead of directly optimizing the once-for-all network from scratch, we propose to first train the largest neural network with *maximum* depth, width, and kernel size, then progressively fine-tune the once-for-all network to support *smaller* sub-networks that share weights with larger ones. As such, it provides better initialization by selecting the most important weights of larger sub-networks, and the opportunity to distill smaller sub-networks, which greatly improves the training efficiency. It also prevents smaller sub-networks from hurting the accuracy of larger sub-networks by optimizing in the local space around well-trained larger sub-networks.

We extensively evaluated the effectiveness of OFA on ImageNet with many hardware platforms (CPU, GPU, mCPU, mGPU, FPGA accelerator) and efficiency constraints. Under all deployment scenarios, OFA consistently achieves the same level (or better) ImageNet accuracy than SOTA hardware-aware NAS methods while saving the GPU hours, dollars, and CO_2 emission by orders of magnitude when handling multiple diverse deployment scenarios.

2 Related Work

Efficient Deep Learning. Many efficient neural network architectures are proposed to improve the hardware efficiency, such as SqueezeNet (Iandola et al., 2016), MobileNets (Howard et al., 2017; Sandler et al., 2018), ShuffleNets (Ma et al., 2018; Zhang et al., 2018), etc. Orthogonal to architecting efficient neural networks, model compression (Han et al., 2016) is another very effective technique for efficient deep learning, including network pruning that removes redundant units (Han et al., 2015) or redundant channels (He et al., 2018; Liu et al., 2017), and quantization that reduces the bit width for the weights and activations (Han et al., 2016; Courbariaux et al., 2015; Zhu et al., 2017).

Neural Architecture Search. Neural architecture search (NAS) focuses on automating the architecture design process (Zoph & Le, 2017; Zoph et al., 2018; Real et al., 2019; Cai et al., 2018a; Liu et al., 2019). Early NAS methods (Zoph et al., 2018; Real et al., 2019; Cai et al., 2018b) search for high-accuracy architectures without taking hardware efficiency into consideration. Therefore, the produced architectures (e.g., NASNet, AmoebaNet) are not efficient for inference. Recent hardware-aware NAS methods (Tan et al., 2018; Cai et al., 2019; Wu et al., 2019) directly incorporate the hardware feedback into architecture search. As a result, they are able to improve the inference efficiency. However, given new inference hardware platforms, these methods need to repeat the architecture search process and retrain the model, leading to prohibitive GPU hours, dollars and CO_2 emission. They are not scalable to a large number of deployment scenarios. The individually trained models do not share any weight, leading to large total model size and high downloading bandwidth.

Dynamic Neural Networks. To improve the efficiency of a given neural network, some work explored skipping part of the model based on the input image. For example, Wu et al. (2018); Liu & Deng (2018); Wang et al. (2018) learn a controller or gating modules to adaptively drop layers; Huang et al. (2018) introduce early-exit branches in the computation graph; Lin et al. (2017) adaptively prune channels based on the input feature map. Recently, Slimmable Nets (Yu et al., 2019; Yu & Huang, 2019b) propose to train a model to support multiple width multipliers (e.g., 4 different global width multipliers), building upon existing human-designed neural networks (e.g., MobileNetV2 0.35, 0.5, 0.75, 1.0). Such methods can adaptively fit different efficiency constraints at runtime, however, still inherit a pre-designed neural network (e.g., MobileNet-v2), which limits the degree of flexibility (e.g., only width multiplier can adapt) and the ability in handling new deployment scenarios where the pre-designed neural network is not optimal. In this work, in contrast, we enable a much more diverse architecture space (depth, width, kernel size, and resolution) and a significantly larger number of architectural settings (10^{19} v.s. 4 (Yu et al., 2019)). Thanks to the diversity and the large design space, we can derive new specialized neural networks for many different deployment scenarios rather than working on top of an existing neural network that limit the optimization headroom. However, it is more challenging to train the network to achieve this flexibility, which motivates us to design the progressive shrinking algorithm to tackle this challenge.

3 Method

3.1 Problem Formalization

Assuming the weights of the once-for-all network as W_o and the architectural configurations as $\{arch_i\}$, we then can formalize the problem as

$$\min_{W_o} \sum_{arch_i} \mathcal{L}_{val}\big(C(W_o, arch_i)\big), \tag{1}$$

where $C(W_o, arch_i)$ denotes a selection scheme that selects part of the model from the once-for-all network W_o to form a sub-network with architectural configuration $arch_i$. The overall training objective is to optimize W_o to make each supported sub-network maintain the *same* level of accuracy as *independently* training a network with the same architectural configuration.

3.2 Architecture Space

Our once-for-all network provides one model but supports many sub-networks of different sizes, covering four important dimensions of the convolutional neural networks (CNNs) architectures, i.e., depth, width, kernel size, and resolution. Following the common practice of many CNN models (He et al., 2016; Sandler et al., 2018; Huang et al., 2017), we divide a CNN model into a sequence of units with gradually reduced feature map size and increased channel numbers. Each unit consists of a sequence of layers where only the first layer has stride 2 if the feature map size decreases (Sandler et al., 2018). All the other layers in the units have stride 1.

We allow each unit to use arbitrary numbers of layers (denoted as *elastic depth*); For each layer, we allow to use arbitrary numbers of channels (denoted as *elastic width*) and arbitrary kernel sizes (denoted as *elastic kernel size*). In addition, we also allow the CNN model to take arbitrary input image sizes (denoted as *elastic resolution*). For example, in our experiments[2], the input image size

[2]We use the same architecture space as ProxylessNAS [4], without SE [15] and Swish activation function [27] that are orthogonal to boost the accuracy [31].

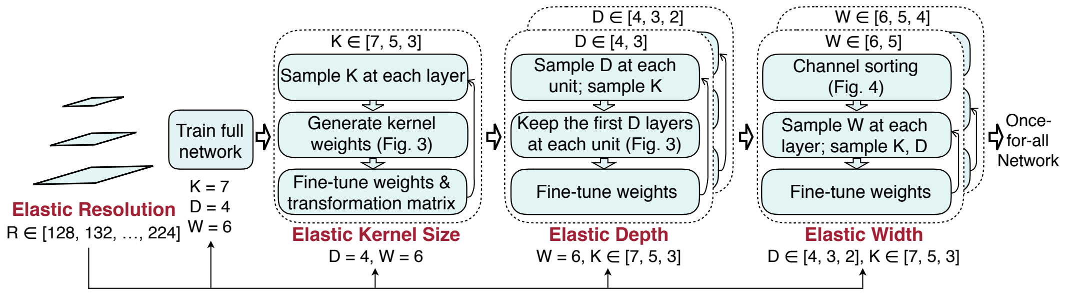


Figure 2: Illustration of the progressive shrinking process to support different depth D, width W, kernel size K and resolution R. It leads to a large space comprising diverse sub-networks ($> 10^{19}$).

ranges from 128 to 224 with a stride 4; the depth of each unit is chosen from $\{2, 3, 4\}$; the width expansion ratio in each layer is chosen from $\{4, 5, 6\}$; the kernel size is chosen from $\{3, 5, 7\}$. Therefore, with 5 units, we have roughly $((3 \times 3)^2 + (3 \times 3)^3 + (3 \times 3)^4)^5 \approx 2 \times 10^{19}$ different neural network architectures and each of them can be used under 25 different input resolutions. Since all of these sub-networks share the same weights (i.e., W_o) (Cheung et al., 2019), we only require 5.1M parameters to store all of them. Without sharing, the total model size will be prohibitive.

3.3 Training the Once-for-all Network

Naïve Approach. Training the once-for-all network can be cast as a multi-objective problem, where each objective corresponds to one sub-network. From this perspective, a naïve training approach is to directly optimize the once-for-all network from scratch using the exact gradient of the overall objective, which is derived by enumerating all sub-networks in each update step, as shown in Eq. (1). The cost of this approach is linear to the number of sub-networks. Therefore, it is only applicable to scenarios where a limited number of sub-networks are supported (Yu et al., 2019), while in our case, it is computationally prohibitive to adopt this approach.

Another naïve training approach is to sample a few sub-networks in each update step rather than enumerate all of them, which does not have the issue of prohibitive cost. However, with such a large number of sub-networks that share weights thus interfere with each other, we find it suffers from significant accuracy drop. In the following section, we introduce a solution to address this challenge by adding a *progressive shrinking training order* to the training process. Correspondingly, we refer to the naïve training approach as *random order*.

Progressive Shrinking. The once-for-all network comprises many sub-networks of different sizes where small sub-networks are nested in large sub-networks. To prevent interference between the sub-networks, we propose to enforce a training order from large sub-networks to small sub-networks in a progressive manner. We name this training order as *progressive shrinking* (PS). An example of the training process with PS is provided in Figure 2, where we start with training the largest neural network with the maximum kernel size (i.e., 7), depth (i.e., 4), and width (i.e., 6). Next, we progressively fine-tune the network to support smaller sub-networks by gradually adding them into the sampling space (larger sub-networks may also be sampled). Specifically, after training the largest network, we first support elastic kernel size which can choose from $\{3, 5, 7\}$ at each layer, while the depth and width remain the maximum values. Then, we support elastic depth and elastic width sequentially, as is shown in Figure 2. The resolution is elastic throughout the whole training process, which is implemented by sampling different image size for each batch of training data. We also use the knowledge distillation technique after training the largest neural network (Hinton et al., 2015; Ashok et al., 2018; Yu & Huang, 2019b). It combines two loss terms using both the soft labels given by the largest neural network and the real labels.

Compared to the naïve approach, PS prevents small sub-networks from interfering large sub-networks, since large sub-networks are already well-trained when the once-for-all network is fine-tuned to support small sub-networks. Additionally, during fine-tuning, the model is optimized in the local space around the well-trained large sub-networks by using a small learning rate and revisiting (i.e., sampling) well-trained large sub-networks. Regarding the small sub-networks, they share the weights with the large ones. Therefore, PS allows initializing small sub-networks with the most important weights of well-trained large sub-networks, which expedites the training process. We describe the details of the PS training flow as follows:

- **Elastic Kernel Size** (Figure 3 left). We let the center of a 7x7 convolution kernel also serve as a 5x5 kernel, the center of which can also be a 3x3 kernel. Therefore, the kernel size becomes

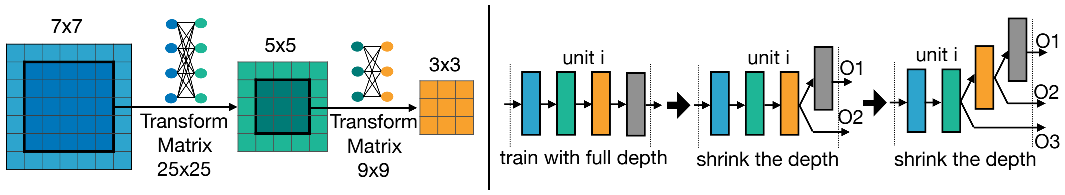


Figure 3: Left: Kernel transformation matrix for elastic kernel size. Right: Progressive shrinking for elastic depth. Instead of skipping each layer independently, we keep the first D layers and skip the last $(4 - D)$ layers. The weights of the early layers are shared.

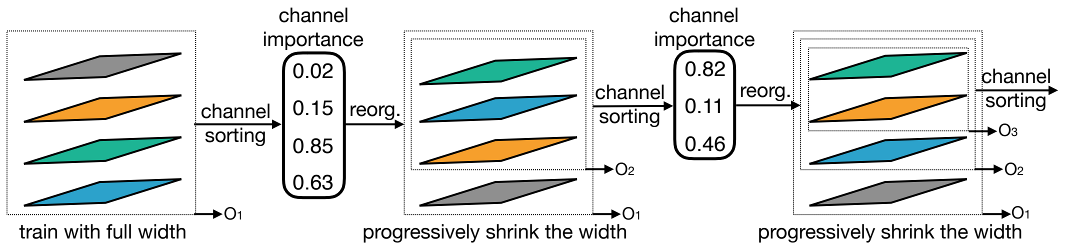


Figure 4: Progressive shrinking for elastic width. In this example, we progressively support 4, 3, and 2 channel settings. We perform channel sorting and pick the most important channels (with large L1 norm) to initialize the smaller channel settings. The important channels' weights are shared.

elastic. The challenge is that the centering sub-kernels (e.g., 3x3 and 5x5) are shared and need to play multiple roles (independent kernel and part of a large kernel). The weights of centered sub-kernels may need to have different distribution or magnitude as different roles. Forcing them to be the same degrades the performance of some sub-networks. Therefore, we introduce kernel transformation matrices when sharing the kernel weights. We use separate kernel transformation matrices for different layers. Within each layer, the kernel transformation matrices are shared among different channels. As such, we only need $25 \times 25 + 9 \times 9 = 706$ extra parameters to store the kernel transformation matrices in each layer, which is negligible.

- **Elastic Depth** (Figure 3 right). To derive a sub-network that has D layers in a unit that originally has N layers, we keep the *first* D layers and skip the last $N - D$ layers, rather than keeping *any* D layers as done in current NAS methods (Cai et al., 2019; Wu et al., 2019). As such, one depth setting only corresponds to one combination of layers. In the end, the weights of the first D layers are shared between large and small models.
- **Elastic Width** (Figure 4). Width means the number of channels. We give each layer the flexibility to choose different channel expansion ratios. Following the progressive shrinking scheme, we first train a full-width model. Then we introduce a channel sorting operation to support partial widths. It reorganizes the channels according to their importance, which is calculated based on the L1 norm of a channel's weight. Larger L1 norm means more important. For example, when shrinking from a 4-channel-layer to a 3-channel-layer, we select the largest 3 channels; whose weights are shared with the 4-channel-layer (Figure 4 left and middle). Thereby, smaller sub-networks are initialized with the most important channels on the once-for-all network which is already well trained. This channel sorting operation preserves the accuracy of larger sub-networks.

3.4 Specialized Model Deployment with Once-for-All Network

Having trained a once-for-all network, the next stage is to derive the specialized sub-network for a given deployment scenario. The goal is to search for a neural network that satisfies the efficiency (e.g., latency, energy) constraints on the target hardware while optimizing the accuracy. Since OFA decouples model training from architecture search, we do not need any training cost in this stage. To eliminate the repeated search cost, we use the predictor-guided architecture search process (Liu et al., 2018b) to find specialized sub-networks.

Specifically, we randomly sample 16K sub-networks with different architectures and input image sizes, then measure their accuracy on 10K validation images sampled from the original training set. These [architecture, accuracy] pairs are used to train an accuracy predictor to predict the accuracy of a model given its architecture and input image size[3]. Additionally, we build a latency lookup table (Cai et al., 2019) on each target hardware platform to predict the latency. Given a target hardware and latency constraint, we conduct evolutionary search (Real et al., 2019) based on the predictors to get a specialized sub-network. Since the cost of searching with predictors is negligible, we only need 40 GPU hours to collect the data pairs, and the cost stays constant regardless of #deployment scenarios.

[3]Details of the accuracy predictor is provided in Appendix A.

Sub-networks	D = 2				D = 4			
	W = 4		W = 6		W = 4		W = 6	
	K = 3	K = 7	K = 3	K = 7	K = 3	K = 7	K = 3	K = 7
Parameters	2.8M	2.9M	3.3M	3.5M	3.7M	4.0M	4.7M	5.1M
FLOPs	191M	233M	266M	328M	329M	419M	473M	607M
Independently Train	68.7	70.5	70.9	72.6	72.8	74.3	74.6	75.4
Random Order	67.3	69.8	69.2	71.3	71.4	72.7	72.6	73.5
Progressive Shrink (ours)	**68.7**	**71.2**	**71.0**	**73.3**	**73.3**	**74.8**	**74.8**	**75.9**
Δ Acc.	+1.4	+1.4	+1.8	+2.0	+1.9	+2.1	+2.2	+2.4

Table 1: ImageNet top1 accuracy (%) performances of sub-networks under resolution 224×224. "(D = d, W = w, K = k)" denotes a sub-network with d layers in each unit, and each layer has an width expansion ratio w and kernel size k.

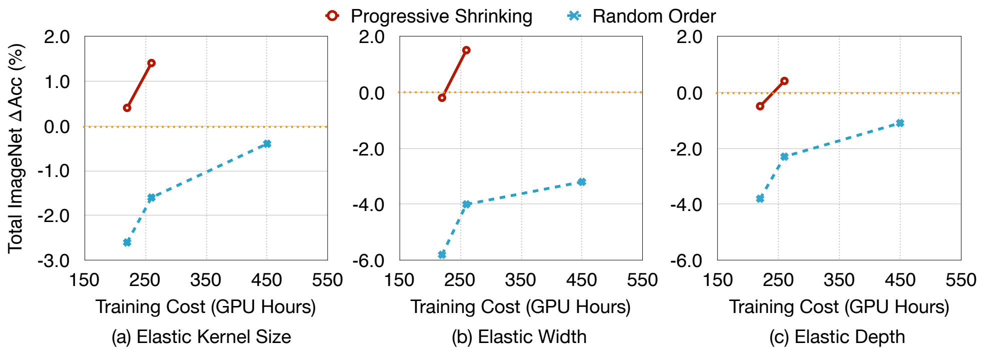


Figure 5: Progressive shrinking significantly improves the training efficiency and accuracy of the once-for-all network compared to random order.

4 Experiments

In this section, we first apply the progressive shrinking algorithm to train the once-for-all network on ImageNet (Deng et al., 2009). Then we demonstrate the effectiveness of our trained once-for-all network on various hardware platforms (Samsung Note8, Google Pixel1, Pixel2, NVIDIA 1080Ti, 2080Ti, V100 GPUs, Intel Xeon CPU, Jetson TX2, Xilinx ZU9EG and ZU3EG FPGAs) with different latency constraints. In total, we have 40 deployment scenarios with 10 hardware platforms and 4 latency constraints on each hardware platform.

4.1 Training the Once-for-all Network on ImageNet

Training Details. We use the standard SGD optimizer with Nesterov momentum 0.9 and weight decay $4e^{-5}$ to train models on ImageNet. The initial learning rate is 0.4, and we use the cosine schedule (Loshchilov & Hutter, 2016) for learning rate decay. The independent models are trained for 150 epochs with batch size 2048 on 32 GPUs. For training the once-for-all network, we use the same training setting with larger training cost (roughly 8×), taking around 1,200 GPU hours on V100 GPUs. This is one-time training cost which can be amortized by many deployment scenarios. Conventional models, even trained longer, can not achieve the same accuracy (2nd row of Table 2).

Results. The top1 accuracy of both independently trained models and the once-for-all networks under the same architectural settings are reported in Table 1. Due to space limits, we take 8 sub-networks for comparison, and each of them is denoted as "(D = d, W = w, K = k)". It represents a sub-network that has d layers for all units while the expansion ratio and kernel size are set to w and k for all layers. Compared to independently trained models, the once-for-all network trained by PS can maintain the same level (or better) accuracy under all architectural settings. We hypothesize that knowledge is transferred from larger sub-networks to smaller sub-networks, which enable them to learn better jointly. In contrast, without PS (i.e., using random order), the top-1 ImageNet accuracy drops significantly on all settings.

Progressive Shrinking is Effective. We further compare the progressive shrinking algorithm with random order under three settings corresponding to three dimensions (i.e., kernel size, width, and depth) respectively. In the first setting, only the kernel size is elastic. We use the total delta accuracy (ΔAcc) on ImageNet between the once-for-all network and independently trained models as the metric (y-axis, the higher the better). It is calculated with three architectural configurations (all layers use the same kernel size $k \in \{3, 5, 7\}$). Each data point corresponds to training a once-for-all network using PS or random order with a certain training cost (x-axis). In the second and third

Model	ImageNet Top1 (%)	FLOPs	Mobile latency	Search cost (GPU hours)	Training cost (GPU hours)	Total cost ($N=40$) GPU hours	CO_2e (lbs)	AWS cost
MobileNetV2 [29]	72.0	300M	106ms	0	150N	6k	1.7k	$18.4k
MobileNetV2 #1200	73.5	300M	106ms	0	1200N	48k	13.6k	$146.9k
NASNet-A [43]	74.0	564M	234ms	48,000N	-	1,920k	544.5k	$5875.2k
DARTS [22]	73.1	595M	-	96N	250N	14k	4.0k	$42.8k
MnasNet* [32]	74.0	317M	108ms	40,000N	-	1,600k	453.8k	$4896.0k
FBNet-C [35]	74.9	375M	129ms	216N	360N	23k	6.5k	$70.4k
ProxylessNAS [4]	74.6	320M	110ms	200N	300N	20k	5.7k	$61.2k
SinglePathNAS [8]	74.7	328M	-	288 + 24N	384N	17k	4.8k	$52.0k
AutoSlim [37]	74.2	305M	102ms	180	300N	12k	3.4k	$36.7k
Once for All w/o PS	72.5	323M	110ms	40	1200	1.2k	0.3k	$3.7k
Once for All w/ PS	**75.1**	332M	109ms	40	1200	1.2k	0.3k	$3.7k
Once for All w/ PS #25	**75.4**	332M	109ms	40	1200 + 25N	2.2k	0.6k	$6.7k

Table 2: Comparison with state-of-the-art hardware-aware NAS methods on Samsung Note8. OFA decouples model training from architecture search. The search cost and training cost both stay constant as the number of deployment scenarios grows ($N = 40$ in our experiments). "#25" denotes the specialized sub-networks are fine-tuned for 25 epochs after grabbing weights from the once-for-all network. *We cite the results of MnasNet without SE for a fair comparison of the search methodology. "CO_2e" denotes CO_2 emission which is calculated based on (Strubell et al., 2019). AWS cost is calculated based on the price of on-demand P3.16xlarge instances.

settings, the elastic dimension changes from kernel size to width and depth respectively, while all the other setups keep the same. The summarized results are shown in Figure 5. We find that (i) simply increasing the training cost cannot prevent random order from accuracy drop. The increasing trend quickly slows down in all three settings; (ii) In contrast, the PS algorithm quickly surpasses the accuracy of individually trained models.

4.2 Specialized Sub-networks for Different Hardware and Constraints

We apply our trained once-for-all network to get specialized sub-networks for different hardware platforms. For the GPU platforms, the latency is measured with batch size 64 on NVIDIA 1080Ti, 2080Ti and V100 with Pytorch 1.0+cuDNN. The CPU latency is measured with batch size 1 on Intel Xeon E5-2690 v4+MKL-DNN[4]. To measure the mobile latency, we use Samsung Note8, Google Pixel1 and Pixel2 using TF-Lite with batch size 1. On Jetson TX2, we use a batch size of 2 (single batch causes severe under-utilization). On Xilinx ZU9EG and ZU3EG FPGAs, we use DNNDK with batch size 1.

Comparison with NAS on Mobile. Table 2 reports the comparison between OFA and state-of-the-art hardware-aware NAS methods on the mobile platform (Samsung Note8). OFA is much more efficient than NAS when handling multiple deployment scenarios, since the cost of OFA is *constant* while others are *linear* to the number of deployment scenarios (N). With N = 40, the training time of OFA is 16× faster than ProxylessNAS, 19× faster than FBNet, and 1,300× faster than MnasNet. Without retraining, OFA achieves 75.1% top1 accuracy on ImageNet, which is 1.1% higher than MnasNet, 0.5% higher than ProxylessNAS, and 0.2% higher than FBNet while maintaining similar (or lower) mobile latency. By fine-tuning the specialized sub-network for 25 epochs, we can further improve the accuracy to 75.4%. Besides, we also observe that OFA with PS can achieve 2.6% better accuracy than without PS, showing the effectiveness of PS.

Results under Different Efficiency Constraints. Table 3 summarizes the results on the mobile platform under different latency constraints. OFA can re-design specialized neural networks for all scenarios without additional cost, while previous methods rescale an existing model using a width multiplier due to the high search and training cost (Sandler et al., 2018; Cai et al., 2019; Tan et al., 2018). As shown in Table 3, we can achieve much higher improvements over the baselines. With similar latency as MobileNetV2 0.35, we improve the ImageNet top1 accuracy from the MobileNetV2 baseline 60.3% to 66.6% (+6.3%) without retraining, and to 69.2% (+8.9%) after fine-tuning for 25 epochs. OFA is more effective than NAS + width multiplier adjustment.

Model	Latency	Top1 (%)
MobileNetV2 0.35	28ms	60.3
MnasNet 0.35	27ms	62.4 (+2.1)
Once for All (ours)	31ms	66.6 (+6.3)
Once for All #25	31ms	69.2 (+8.9)
MobileNetV2 0.5	40ms	65.4
MnasNet 0.5	41ms	67.8 (+2.4)
ProxylessNAS 0.5	41ms	68.2 (+2.8)
Once for All (ours)	41ms	69.8 (+4.4)
Once for All #25	41ms	71.3 (+5.9)
MobileNetV2 0.75	77ms	69.8
MnasNet 0.75	75ms	71.5 (+1.7)
Once for All (ours)	77ms	73.7 (+3.9)
Once for All #25	77ms	74.2 (+4.4)

Table 3: ImageNet accuracy under various latency constraints on Samsung Note8.

[4]https://github.com/intel/mkl-dnn

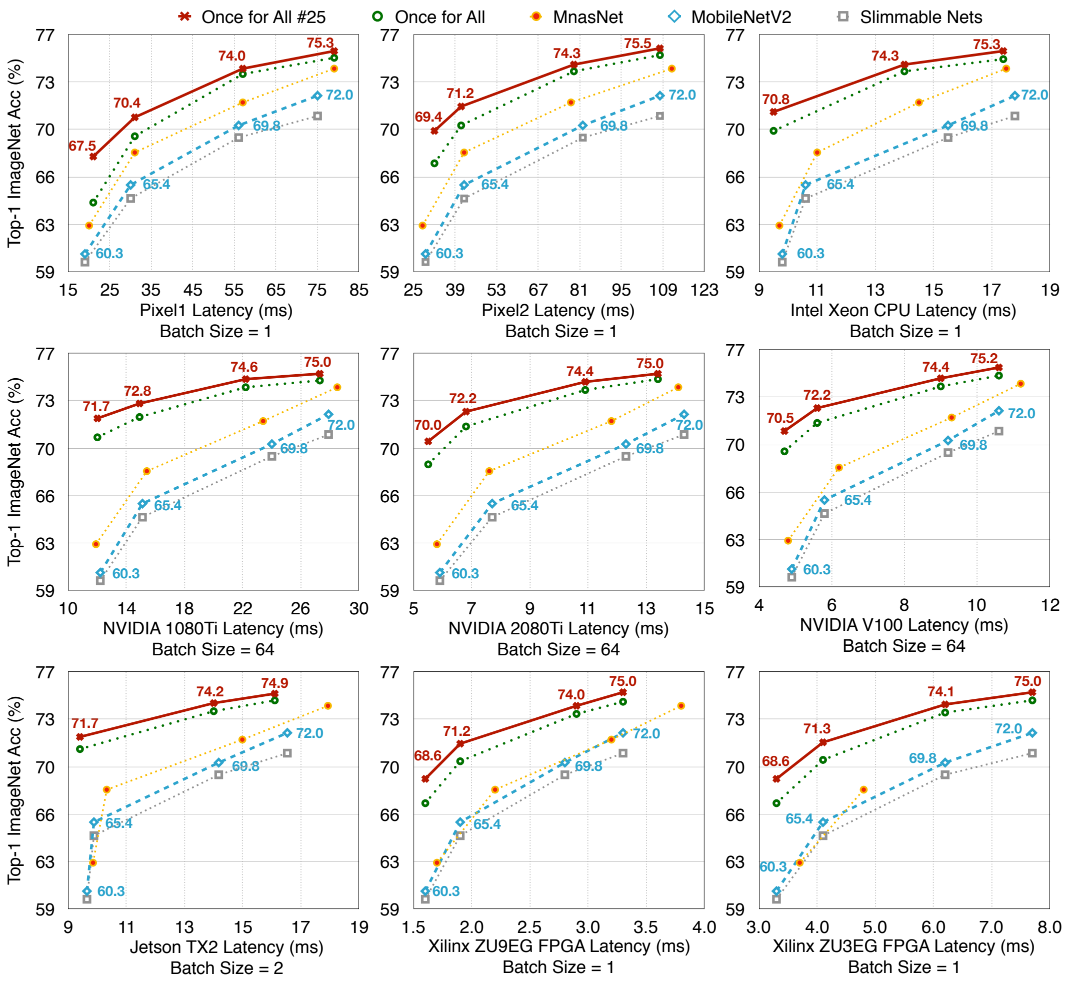


Figure 6: Specialized deployment results on CPU, GPU, mCPU, mGPU, and FPGA accelerator. Specialized models by OFA consistantly achieve significantly higher ImageNet accuracy with similar latency than non-specialized neural networks. More remarkably, specializing for a new hardware platform does not add training cost using OFA.

Results on Diverse Hardware Platforms. We extensively studied with effectiveness of OFA on 9 different hardware platforms (Figure 6). OFA consistently improve the trade-off between accuracy and latency by a significant margin, especially on GPUs which have more parallelism. With similar latency as MobileNetV2 0.35, "OFA #25" improves the ImageNet top1 accuracy from MobileNetV2's 60.3% to 71.7% (+11.4% improvement) on the 1080Ti GPU. It reveals the insight that using the *same* model for different deployment scenarios with *only* the width multiplier modified has limited impact on efficiency improvement: the accuracy drops quickly as the latency constraint gets tighter. On Xilinx ZU3EG, a low-resource FPGA, some width settings of MnasNets even fail to run due to "out of BRAM". We profiled the FPGA and drew its roofline model (see Appendix B): OFA models improved the arithmetic intensity by up to 48%/43% and GOPS/s by 70%/92% compared with the MobileNetV2/MnasNet family. OFA designs specialized model that better fits the hardware.

5 Conclusion

We proposed *Once for All* (OFA), a new methodology that decouples model training from architecture search for efficient deep learning deployment under a large number of deployment scenarios. Unlike previous approaches that design and train a neural network for *each* deployment scenario, we designed a *once-for-all network* that supports different architectural configurations, including elastic depth, width, kernel size, and resolution. It greatly reduces the training cost (GPU hours, energy consumption, and CO_2 emission) compared to conventional methods. To prevent sub-networks of different sizes from interference, we proposed a progressive shrinking algorithm that enable a large number of sub-network to achieve the same level of accuracy compared to training them independently. Experiments on a diverse range of hardware platforms and efficiency constraints demonstrated the effectiveness of our approach.

REFERENCES

Anubhav Ashok, Nicholas Rhinehart, Fares Beainy, and Kris M Kitani. N2n learning: Network to network compression via policy gradient reinforcement learning. In *ICLR*, 2018. 4

Han Cai, Tianyao Chen, Weinan Zhang, Yong Yu, and Jun Wang. Efficient architecture search by network transformation. In *AAAI*, 2018a. 2

Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. In *ICML*, 2018b. 3

Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019. URL `https://arxiv.org/pdf/1812.00332.pdf`. 3, 5, 7

Brian Cheung, Alex Terekhov, Yubei Chen, Pulkit Agrawal, and Bruno Olshausen. Superposition of many models into one. In *NeurIPS*, 2019. 4

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *NeurIPS*, 2015. 2

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009. 6

Zichao Guo, Xiangyu Zhang, Haoyuan Mu, Wen Heng, Zechun Liu, Yichen Wei, and Jian Sun. Single path one-shot neural architecture search with uniform sampling. *arXiv preprint arXiv:1904.00420*, 2019. 7

Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NeurIPS*, 2015. 2

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2016. 2

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 3

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *ECCV*, 2018. 1, 2

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 4

Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017. 1, 2

Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *CVPR*, 2018. 3

Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *CVPR*, 2017. 3

Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q Weinberger. Multi-scale dense networks for resource efficient image classification. In *ICLR*, 2018. 3

Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. 2

Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. In *NeurIPS*, 2017. 3

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *ECCV*, 2018a. 2

Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *ECCV*, 2018b. 5

Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search. In *ICLR*, 2019. 2, 7

Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In *AAAI*, 2018. 3

Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *ICCV*, 2017. 2

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016. 6

Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018. 2

Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. In *ICLR Workshop*, 2018. 3

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2019. 2, 3, 5

Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018. 1, 2, 3, 7

Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and policy considerations for deep learning in nlp. In *ACL*, 2019. 1, 7

Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, 2019. 3

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626v1*, 2018. 3, 7

Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *ECCV*, 2018. 3

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2009. 12

Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *CVPR*, 2019. 3, 5, 7

Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S Davis, Kristen Grauman, and Rogerio Feris. Blockdrop: Dynamic inference paths in residual networks. In *CVPR*, 2018. 3

Jiahui Yu and Thomas Huang. Autoslim: Towards one-shot architecture search for channel numbers. *arXiv preprint arXiv:1903.11728*, 2019a. 7

Jiahui Yu and Thomas Huang. Universally slimmable networks and improved training techniques. In *ICCV*, 2019b. 3, 4

Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas Huang. Slimmable neural networks. In *ICLR*, 2019. 3, 4

Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *CVPR*, 2018. 1, 2

Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. Trained ternary quantization. In *ICLR*, 2017. 2

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017. 2

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018. 2, 3, 7

A Details of the Accuracy Predictor

We use a three-layer feedforward neural network that has 400 hidden units in each layer as the accuracy predictor. Given a model, we encode each layer in the neural network into a one-hot vector based on its kernel size and expand ratio, and we assign zero vectors to layers that are skipped. Besides, we have an additional one-hot vector that represents the input image size. We concatenate these vectors into a large vector that represents the whole neural network architecture and input image size, which is then fed to the three-layer feedforward neural network to get the predicted accuracy.

B Roofline Model on FPGA

We analyze our specialized models on FPGA using DNNDK [5]. The profiling results are summarized in Figure 7, while the roofline models (Williams et al., 2009) on ZU9EG and ZU3EG are illustrated in Figure 8. Unlike high-end FPGAs that have DRAM modules which are directly connected to the programmable logic (PL), ZU9EG and ZU3EG share the same DRAM module with the processing system (includes the ARM CPU). Thus memory bandwidth is an expensive resource. Correspondingly, our specialized models reduce costly memory accesses by increasing arithmetic intensity (i.e., the number of arithmetic operations for each byte of memory access), which results in higher GOPS/s.

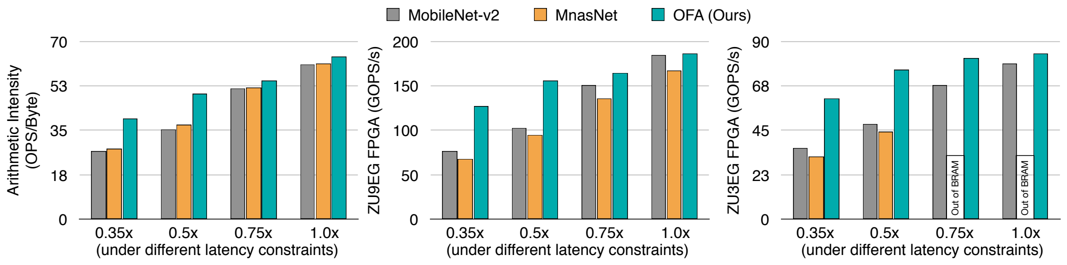


Figure 7: On FPGA, OFA models improve the arithmetic intensity and GOPS/s compared with the MobileNetV2/MnasNet family.

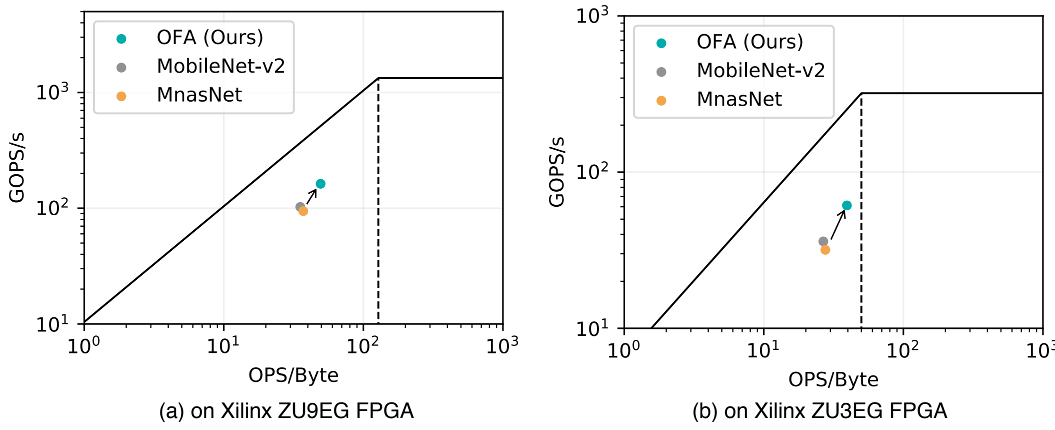
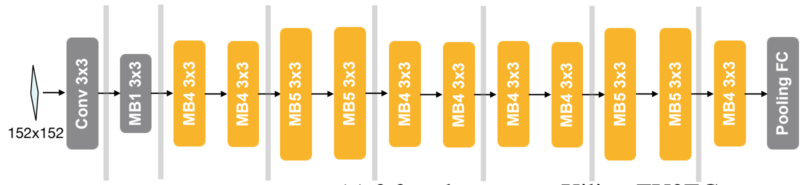


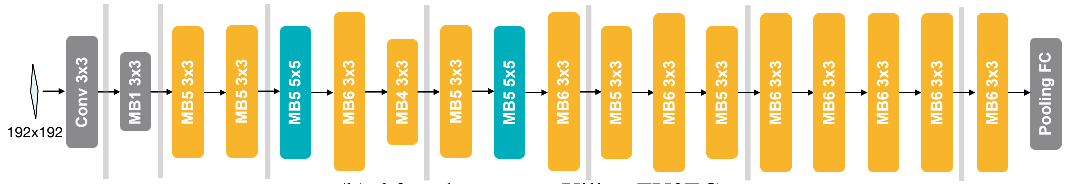
Figure 8: Roofline Models on Xilinx ZU9EG and ZU3EG FPGAs (log scale). Our OFA model increased the arithmetic intensity by 33%/43% and GOPS/s by 72%/92% on these two FPGAs compared with MnasNet.

[5]https://www.xilinx.com/products/design-tools/ai-inference/edge-ai-platform.html#dnndk

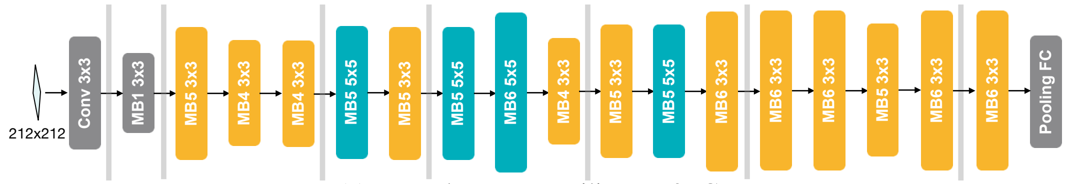
C Visualization of Our Specialized Models



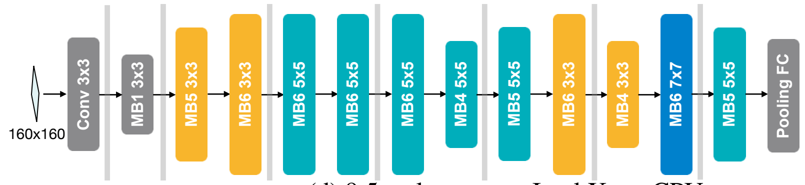
(a) 3.3ms latency on Xilinx ZU3EG.



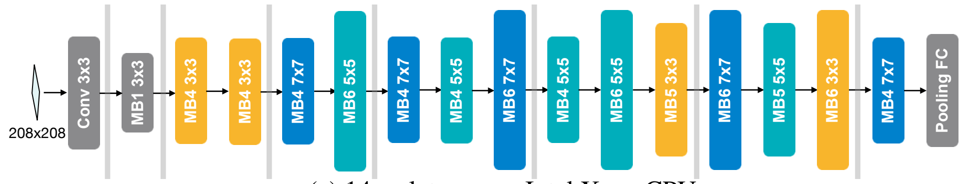
(b) 6.2ms latency on Xilinx ZU3EG.



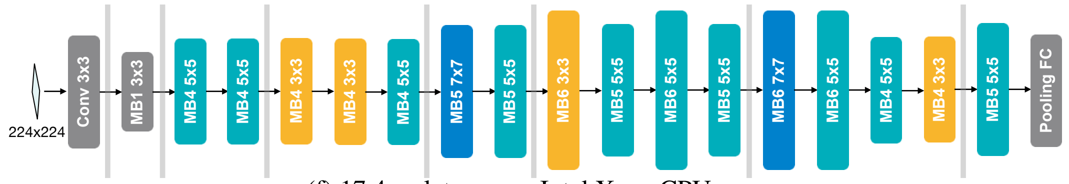
(c) 7.7ms latency on Xilinx ZU3EG.



(d) 9.5ms latency on Intel Xeon CPU.



(e) 14ms latency on Intel Xeon CPU.



(f) 17.4ms latency on Intel Xeon CPU.

Figure 9: Illustration of our specialized models on FPGA and Intel CPU. “MB4 3x3” means “mobile block with expansion ratio 4, kernel size 3x3”. We find that (i) small-kernel operations are preferred on FPGA. The smallest model on FPGA even only have 3x3 kernels. We hypothesize that small-kernel operations have better implementation on FPGA. Additionally, large-kernel operations may cause “out of BRAM” error on FPGA; (ii) while on Intel Xeon CPU with MKL-DNN, more than 50% operations are large-kernels. We also observed that the input resolution chosen by our framework is proportional to the latency.