HACScale: Hardware-Aware Compound Scaling for Resource-Efficient DNNs

Hao Kong[1,2], Di Liu[2], Xiangzhong Luo[1], Weichen Liu[1], and Ravi Subramaniam[3]

[1]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[2]HP-NTU Digital Manufacturing Corporate Lab, Nanyang Technological University, Singapore
[3]Innovations and Experiences – Business Personal Systems, HP Inc., Palo Alto, California, USA

Abstract— Model scaling is an effective way to improve the accuracy of deep neural networks (DNNs) by increasing the model capacity. However, existing approaches seldom consider the underlying hardware, causing inefficient utilization of hardware resources and consequently high inference latency. In this paper, we propose *HACScale*, a hardware-aware model scaling strategy to fully exploit hardware resources for higher accuracy. In *HACScale*, different dimensions of DNNs are jointly scaled with consideration of their contributions to hardware utilization and accuracy. To improve the efficiency of width scaling, we introduce importance-aware width scaling in *HACScale*, which computes the importance of each layer to the accuracy and scales each layer accordingly to optimize the trade-off between accuracy and model parameters. Experiments show that *HACScale* improves the hardware utilization by $1.92\times$ on ImageNet, as a result, it achieves 2.41% accuracy improvement with a negligible latency increase of 0.6%. On CIFAR-10, *HACScale* improves the accuracy by 2.23% with only 6.5% latency growth.

I. INTRODUCTION

Deep neural networks (DNNs) have demonstrated huge success in many applications, such as image classification, natural language processing, etc [1, 2]. DNNs involve huge amount of simple operations (addition and multiplication) which can be processed in parallel. To facilitate the deployment of DNNs, both academia and industry strive to design powerful accelerators to improve the performance of the underlying hardware to keep up with the rapid development of DNNs [3, 4]. Those Deep Learning Accelerators (DLAs) are equipped with larger memory capacity and more computing units, which empower them to host larger DNNs and perform more operations in parallel.

As deep learning devices evolve to be more powerful, the *hardware-agnostic* DNNs are likely to underutilize the underlying hardware. However, since the capacity of a DNN plays a pivotal role in the DNN's accuracy, i.e., the more complex the DNN is, the better accuracy the DNN can achieve, the underutilized resources may provide a great chance for a lightweight DNN to improve its predictive performance without compromising its execution performance. For many DNN applications, like robots, self-driving cars, etc, it will be highly desired to improve the accuracy while not affecting the execution latency. This empirical observation and practical demand in many emerging edge systems motivate us to improve the hardware utilization for both high accuracy and low latency.

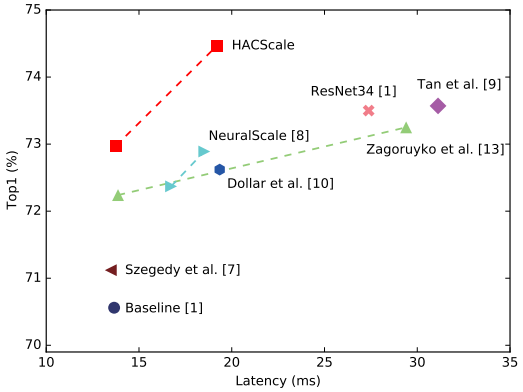

Fig. 1. The trade-off between accuracy and latency of different scaling methods on ImageNet. The baseline network is ResNet18 and the latency is measured on Nvidia Jetson Xavier.

Model scaling, as an effective way to improve the capacity of DNNs, has been coupled with neural architecture search (NAS) [5] and manually designed DNNs [6] to pursue higher accuracy. Some work exploits scaling one of the dimensions (*depth*, *width*, *resolution*) of a model to obtain higher accuracy [1, 7, 8, 9], while the others scale multiple dimensions in combination (i.e., compound scaling) to further explore the potential for improving accuracy [10, 11]. However, these approaches suffer from two problems: 1) Single-dimension scaling only brings limited accuracy improvements and the accuracy quickly saturates. 2) They do not consider the impact of scaling different dimensions on the hardware utilization. As a result, the scaled models still cannot utilize hardware resources efficiently, resulting in a significant increase in latency.

To address the above issues, we propose a hardware-aware compound scaling framework, dubbed *HACScale*, to fully exploit hardware resources to improve the accuracy without increasing the latency. We first investigate how scaling different dimensions affects the hardware utilization and the latency, and find that *width scaling* and *resolution scaling* can considerably improve the hardware utilization and maintain the original latency. Therefore, we design a scaling strategy to mainly scale the width and resolution of DNNs for better accuracy. In addition, as part of *HACScale*, we introduce importance-aware width scaling to separately scale the width of each layer according to their importance to the accuracy. By this means, the

accuracy of a DNN can be further improved within the same memory constraint compared with uniformly scaling all layers [6, 7]. As shown in Fig. 1, for systems that have a tight latency constraint, *HACScale* can significantly improve the accuracy without affecting the latency. Besides, for a relaxed latency constraint, *HACScale* still achieves a better trade-off between accuracy and latency compared with other methods. Our main contributions are summarized as follows:

1. We present *HACScale*, a hardware-aware compound scaling framework, to improve the hardware utilization of existing DNNs. By collaboratively scaling different dimensions based on their contributions to hardware utilization and prediction accuracy, *HACScale* empowers different DNNs to fully exploit the underlying hardware to improve their accuracy.
2. We propose a novel importance-aware width scaling method. During the scaling process, we quickly estimate the importance of each layer through an importance predictor and scale each layer individually, which enables us to control the width of each layer in a fine-grained manner and improves the efficiency of width scaling.
3. Experiments show that *HACScale* achieves 2.41% and 2.23% accuracy gains on ImageNet and CIFAR-10 with increasing the latency by only 0.6% and 6.5%, respectively.

The rest of this paper is organized as follows: Section II introduces some related work; Section III describes the proposed method; Section IV demonstrates the experimental results; And section V summarizes this paper.

II. Related work

Efforts have been made to implement neural networks on diverse hardware in order to pave the way for prevalent AI and artificial intelligence of things (AIoT). Neural network scaling and pruning are two promising and widely used techniques to adjust the complexity of neural networks.

Model Scaling: Traditional network design tends to construct deeper networks [1, 12, 13] to attain higher accuracy, while some work [6, 7, 14, 9] shows that increasing the width of networks can also achieve competitive accuracy. However, these approaches only consider scaling a single dimension of neural networks, which results in a very limited accuracy improvement. Recent work [10, 11] proposes to scale all dimensions of networks to achieve higher accuracy. However, they do not take into consideration the impact of scaling different dimensions on the inference performance, and the accuracy improvement is at the cost of much higher inference latency.

Channel Pruning: Channel pruning has been extensively studied over the past several years [15], where the foundation is laid on the fact that neural networks are overparameterized and thus have some redundant units. Different kinds of methods are proposed to identify and remove unimportant channels. Specifically, channel pruning contains two paradigms, one of which prunes channels according to their local importance within each layer [16, 17], while the other conducts pruning in a global manner [18, 19]. In general, local channel pruning requires more efforts to decide an individual pruning ratio for each layer, while global channel pruning just sorts all channels according to their contribution to prediction accuracy regardless of the position of each channel.

III. Proposed Method

A. Problem formulation

In this section, we give the formal formulation of our problem. Given a baseline convolutional neural network $\mathcal{N}$ with n layers, each convolutional layer can be defined as a function: $Y_i = \mathcal{F}_i(X_{<H_i,W_i,C_i>})$, where $X_{<H_i,W_i,C_i>}$ is the input tensor with a spatial dimension (H_i, W_i) and a channel dimension C_i. $\mathcal{F}_i$ and Y_i are the convolutional operator and the output tensor, respectively. A convolutional neural network is formed by cascading n convolutional layers. Consequently, the baseline neural network $\mathcal{N}$ can be formulated as:

$$\mathcal{N} = \bigodot_{i=1...n} \mathcal{F}_i(X_{<H_i,W_i,C_i>}) \tag{1}$$

Given a latency constraint $\mathcal{L}$, and a memory constraint $\mathcal{M}$, we strive to find a collection of scaling coefficients (d, r, Θ) to scale all dimensions of the neural network $\mathcal{N}$, where d is the depth scaling coefficient, r is the resolution scaling coefficient, and $\Theta = \{w_1, w_2, ..., w_n\}$ is the collection of each layer's width scaling coefficient. By this means, the scaled neural network can maximize its accuracy while still satisfying the latency constraint $\mathcal{L}$ and the memory constraint $\mathcal{M}$. The optimization objective can be formulated as:

$$\begin{aligned} \max_{d,r,\Theta} \quad & Accuracy(\mathcal{N}(d, r, \Theta)) \\ s.t. \quad & \mathcal{N} = \bigodot_{i=1...n\cdot d} \mathcal{F}_i(X_{<r\cdot H_i, r\cdot W_i, w_i\cdot C_i>}) \\ & Latency(\mathcal{N}(d, r, \Theta)) \le \mathcal{L} \\ & Memory(\mathcal{N}(d, r, \Theta)) \le \mathcal{M} \end{aligned} \tag{2}$$

However, exhaustively searching for the optimal combination of (d, r, Θ) suffers from a prohibitively expensive search cost, in the next section, we will present how *HACScale* can efficiently achieve the optimization goal.

B. Hardware-aware Compound Scaling

Compound scaling, instead of scaling an individual dimension, jointly scales multiple dimensions of a network to improve the accuracy. Compared with single-dimension scaling, compound scaling is capable of achieving higher accuracy.

To scale a network for higher accuracy without affecting the inference latency noticeably, we need to better exploit the hardware parallelism. Therefore, we selectively scale those dimensions which can improve the hardware utilization. To identify

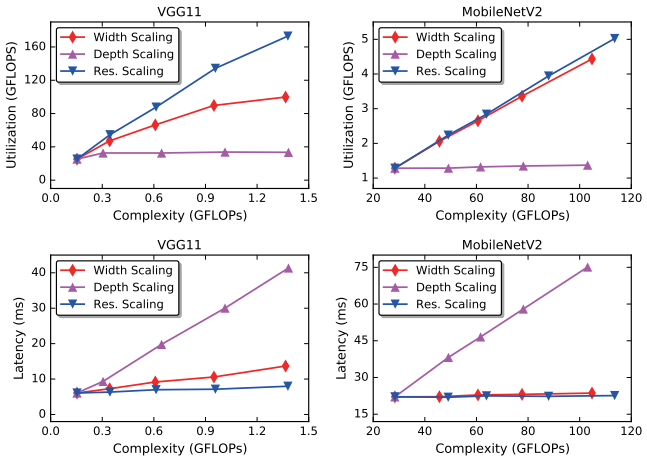

Fig. 2. (Upper): The improvement of hardware utilization by different scaling methods. The horizontal axis denotes the computational complexity of scaled models and the vertical axis is the hardware utilization while running those scaled models. (Lower): The impact of different scaling methods on the inference latency. The vertical axis represents the inference latency (ms) measured on Nvidia Jetson Xavier. Those figures show that *depth scaling* can hardly improve the hardware utilization and brings a steeper increase in the latency than *width scaling* and *resolution scaling*.

the impact of scaling different dimensions on the hardware utilization, we conduct a series of experiments, where different baseline models are scaled up to different computational complexity using three scaling methods: 1) *width scaling*, 2) *depth scaling*, and 3) *resolution scaling*. As shown in Fig. 2, for *resolution scaling* and *width scaling*, we find that the hardware utilization (GLOPS[1]), which is represented by the number of operations processed by the hardware per second, rises with the model complexity (GFLOPs[2]), while *depth scaling* does not change the hardware utilization obviously regardless of the model complexity. Consequently, scaling the depth of networks results in a sharp increase in the latency.

For many latency-critical systems, they expect to improve the accuracy without changing the latency. Then, *depth scaling* is inapplicable in this case since it will notably increase the latency. Considering our target (increasing accuracy without latency increase), we focus on the width and resolution scaling instead of jointly scaling all three dimensions. Our compound scaling strategy is formulated as:

$$d = 1, \quad r = \sqrt{s}^{\alpha}, \quad w = \sqrt{s}^{1-\alpha} \tag{3}$$

where s is a predefined scaling factor of the model's computational complexity, and $\alpha \in (0, 1)$ is a hyperparameter used to allocate the resources between *resolution scaling* and *width scaling*. To find the optimal value of α, we formulate the optimization problem as:

$$\begin{aligned} \max_{\alpha} \quad & Accuracy(\mathcal{N}(1, \sqrt{s}^{\alpha}, \sqrt{s}^{1-\alpha})) \\ s.t. \quad & \mathcal{N} = \bigodot_{i=1\ldots n\cdot 1} \mathcal{F}_i(X_{<\sqrt{s}^{\alpha}\cdot H_i, \sqrt{s}^{\alpha}\cdot W_i, \sqrt{s}^{1-\alpha}\cdot C_i>}) \\ & \alpha \in (0, 1) \end{aligned} \tag{4}$$

Different from the previous work directly searching for all three scaling factors [10], we only need to optimize a single hyperparameter α, which narrows the search space and greatly cut down the search cost. After obtaining the optimal value of α, the scaling coefficients for all dimensions can be computed with Equation 3. The predefined depth scaling coefficient and the searched resolution scaling coefficient will be applied directly, while the width scaling coefficient for each layer will be further optimized by the proposed importance-aware width scaling algorithm.

C. Importance-aware width scaling

Width scaling has been widely used to improve the representational power of DNNs by adding additional weight kernels to each layer. In practice, previous methods scale the width of networks in a uniform manner, where all layers will be scaled by a uniform scaling coefficient [6, 10, 7]. However, different layers in a network contribute differently to the accuracy [16, 17, 19], thus scaling all layers uniformly is inefficient in terms of improving the accuracy. To address this problem, we propose to scale each layer individually according to their importance to the accuracy instead of using the searched uniform width scaling coefficient.

Importance modeling: Inspired by [19], we define the importance of a layer as the loss increment of the network when removing a weight kernel from this layer. The loss increment can be approximated by adopting the first-order Taylor expansion on the removed weight kernel:

$$\begin{aligned} \mathcal{I}_l &= (\mathbb{E}(K) - \mathbb{E}(K|k_l = 0))^2 \\ &\approx \sum_{m \in k_l} (\frac{\partial \mathcal{L}}{\partial m} \cdot m)^2 = \sum_{m \in k_l} (g_m \cdot m)^2 \end{aligned} \tag{5}$$

where $\mathbb{E}$ denotes the loss function, K represents the weight kernels of the network, and k_l is a weight kernel in the l-th layer. m is a single parameter in k_l and $g_m = \frac{\partial \mathcal{L}}{\partial m}$ is the gradient of m, which can be computed through backpropagation on the network.

As a weight kernel is added to or removed from a layer, the importance of the layer will change and the previously computed importance score will no longer apply. However, it is computationally prohibitive to update the importance of each layer by training the network from scratch whenever a new weight kernel is added to the network. Therefore, we build an importance predictor for each layer to predict its importance. We first sample several networks with different layer width

[1]GFLOPS: Giga Floating-Point Operations Per Second
[2]GFLOPs: Giga Floating-Point Operations

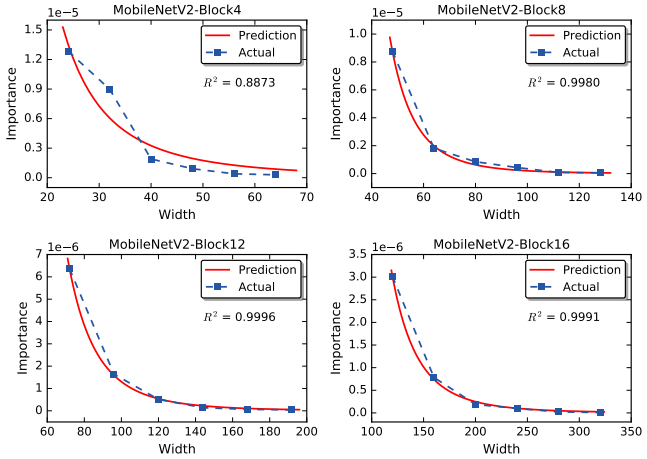

Fig. 3. The actual and predicted importance of different layers on CIFAR-10. $R^2 \in [0, 1]$ denotes the coefficient of the determination between the actual and predicted importance. A higher value of R^2 means a better prediction model.

configurations, and then we train these networks to obtain the importance of each layer under different width configurations.

As demonstrated in Fig. 3, the importance of a layer is highly related to the layer width. We model the relationship between the importance and the layer width as a power function:

$$\mathcal{I}_l = a_l \cdot C_l^{b_l} \tag{6}$$

where C_l is the number of channels of the l-th layer. a_l and b_l are the parameters of the l-th layer's prediction model, which are fitted by non-linear least squares (NLS). With these importance prediction models, we are able to quickly estimate the importance of each layer without conducting the time-consuming training process, thereby accelerating the scaling process.

Width scaling: The importance-aware width scaling algorithm is summarized in Algorithm 1. First, we initialize a baseline model and generate the importance score of each layer with its importance prediction model. Subsequently, we iteratively add weight kernels to the layer with the highest importance score. In each iteration, we will do four things: 1) Sorting all layers according to their importance and identify the most important layer (denoted as i). 2) Updating the width scaling coefficient of layer i with a scaling stride $s = 10$ (i.e., the number of weight kernels to add). 3) Updating the width of layer i by adding s weight kernels to this layer. 4) Updating the importance score of layer i using its prediction model according to the updated width. At the end of each iteration, we will check the latency and memory consumption of the scaled model to make sure the latency constraint and memory constraint are not violated. To eliminate the expensive deployment cost, the latency is estimated by a simple but effective latency predictor and the memory consumption is quantified by the network parameters. After the whole scaling process, we will obtain the scaling coefficient of each layer and we can directly scale the model as Equation 3.

Algorithm 1: Importance-aware width scaling

Require: The importance prediction model of each layer $\{\mathcal{I}_l = a_l \cdot C_l^{b_l}\}_{l=1}^n$, the resolution scaling coefficient r, the baseline width configuration $\{C_l\}_{l=1}^n$, the latency constraint $\mathcal{L}$, and the memory constraint $\mathcal{M}$.

Ensure : The width scaling coefficient of each layer $\Theta = \{w_l\}_{l=1}^n$

Initialize the network $\mathcal{N}$ with r and $\{C_l\}_{l=1}^n$, the scaling stride $s = 10$, and the width scaling coefficients $\{w_l = 1\}_{l=1}^n$;
for *Each layer l in the network $\mathcal{N}$* **do**
  Compute importance $\mathcal{I}_l = a_l \cdot C_l^{b_l}$;
end
while $Latency(\mathcal{N}) \leq \mathcal{L}$ *and* $Memory(\mathcal{N}) \leq \mathcal{M}$ **do**
  Find the most important layer $i = \underset{l}{\operatorname{argmax}}\ \mathcal{I}_l$;
  Update the scaling coefficient $w_i = \frac{C_i + s}{C_i} \cdot w_i$;
  Update the layer width $C_i = C_i + s$;
  Update the importance $\mathcal{I}_i = a_i \cdot C_i^{b_i}$;
end
return $\Theta = \{w_l\}_{l=1}^n$

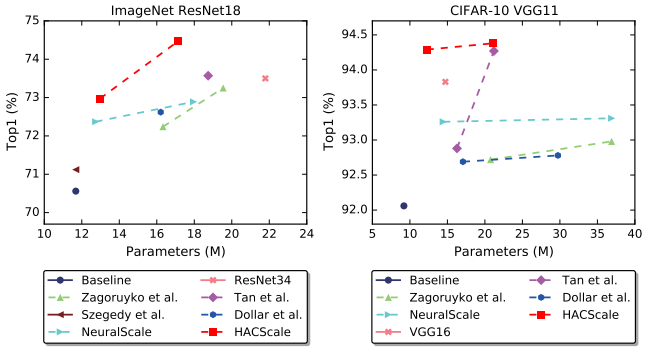

Fig. 4. The parameter efficiency of different scaling methods. The baseline models for ImageNet and CIFAR-10 are ResNet18 and VGG11, respectively.

IV. EXPERIMENTS

The proposed *HACScale* is implemented with PyTorch and evaluated on CIFAR-10 and ILSVRC2012 (i.e., ImageNet). CIFAR-10 contains 50,000 training images and 10,000 test images with a size of 32×32. ImageNet contains 1.28 million training images and 50,000 validation images with a size of 224×224. We scale networks with *HACScale* and other state-of-the-art approaches under two scenarios: 1) We set a tight latency constraint to compare the potential of different methods for improving the network accuracy under current latency. 2) We loosen the latency constraint to explore the efficacy of different methods in terms of the trade-off between accuracy and latency.

Training settings: All networks are trained using stochastic gradient descent (SGD) with a momentum of 0.9. On CIFAR-10, networks are trained for 300 epochs with a batch size of 128. The initial learning rate $\lambda = 0.1$ and divided by 10 at epoch 100, 200, and 250. On ImageNet, we train all networks for 100 epochs with a batch size of 1024 and 5 epochs

TABLE I
THE COMPARISON OF DIFFERENT MODEL SCALING APPROACHES ON IMAGENET AND CIFAR-10

Constraint	Method	Params (M)	FLOPs (M)	Latency (ms)†	Power (W)	↑ Utilization (GFLOPS)	↑ Power Efficiency (GFLOPs/J)	↑ Top1 (%)
ImageNet ResNet18								
	Baseline [1]	11.68	1735.22	13.66	14.87	127.03 (1x)	8.54 (1x)	70.56 (+0.0)
Tight	Zagoruyko et al. [6]	16.33	2365.95	13.87	22.34	170.58 (1.34x)	7.64 (0.89x)	72.24 (+1.68)
	Szegedy et al. [8]	11.68	3253.31	13.48	25.6	241.34 (1.9x)	9.43 (1.1x)	71.12 (+0.56)
	NeuralScale [9]	12.74	2988.33	16.73	20.68	178.62 (1.41x)	8.64 (1.01x)	72.37 (+1.81)
	HACScale (our)	12.96	3359.73	13.74	24.73	**244.52 (1.92x)**	**9.89 (1.16x)**	**72.97 (+2.41)**
Loose	Zagoruyko et al. [6]	19.54	2933.99	29.4	29.87	99.8 (0.79x)	3.34 (0.39x)	73.25 (+2.69)
	ResNet34 [1]	21.79	3587.41	27.38	18.54	131.02 (1.03x)	7.07 (0.83x)	73.5 (+2.94)
	Tan et al. [10]	18.74	2836.77	31.12	23.9	91.16 (0.72x)	3.81 (0.45x)	73.57 (+3.01)
	Dollar et al. [11]	16.21	2788.23	19.35	21.27	144.09 (1.13x)	6.77 (0.79x)	72.62 (+2.06)
	NeuralScale [9]	17.96	3983.63	18.52	29.3	215.1 (1.69x)	7.34 (0.85x)	72.89 (+2.33)
	HACScale (our)	17.14	4812.85	19.21	33.21	**250.54 (1.97x)**	**7.54 (0.88x)**	**74.47 (+3.91)**
CIFAR-10 VGG11								
	Baseline [12]	9.2	150	11.92	7.62	12.58 (1x)	1.65 (1x)	92.06 (+0.0)
Tight	Zagoruyko et al. [6]	20.76	343.08	12.88	12.23	26.63 (2.12x)	2.18 (1.32x)	92.72 (+0.66)
	VGG16 [12]	14.73	314.03	18.45	9.14	17.02 (1.35x)	1.86 (1.13x)	93.83 (+1.77)
	Tan et al. [10]	16.28	493.48	15.37	11.9	32.11 (2.55x)	2.69 (1.63x)	92.88 (+0.82)
	Dollar et al. [11]	17.07	328.83	13.25	11.74	24.82 (1.97x)	2.11 (1.28x)	92.69 (+0.63)
	NeuralScale [9]	14.44	604.81	12.56	11.94	48.15 (3.83x)	4.03 (2.44x)	93.26 (+1.20)
	HACScale (our)	12.24	1442.37	12.7	12.92	**113.57 (9.03x)**	**8.79 (5.33x)**	**94.29 (+2.23)**
Loose	Zagoruyko et al. [6]	36.89	608.44	28.63	12.98	21.25 (1.68x)	1.63 (0.99x)	92.98 (+0.92)
	VGG19 [12]	20.4	399	23.43	10.61	17.03 (1.35x)	1.64 (0.99x)	93.71 (+1.65)
	Tan et al. [10]	21.2	893.15	24.98	12.38	35.75 (2.84x)	2.89 (1.75x)	94.27 (+2.21)
	Dollar et al. [11]	29.74	575.07	24.58	13.64	23.4 (1.86x)	1.7 (1.03x)	92.78 (+0.72)
	NeuralScale [9]	36.9	1418.14	29.19	13.98	48.58 (3.86x)	3.48 (2.11x)	93.31 (+1.25)
	HACScale (our)	21.1	2547.08	25.5	14.15	**99.89 (7.94x)**	**7.06 (4.28x)**	**94.38 (+2.32)**

† The latency on ImageNet is measured on Nvidia Jetson Xavier, and the latency on CIFAR-10 is measured on Nvidia Jetson TX2.

of gradual warmup. We use exponential learning rate scheduling with the initial learning rate $\lambda = 2.0$ and the decay factor $\beta = 0.02$.[3] Besides, we also use label smoothing with $\epsilon = 0.1$, mixup with $\alpha = 0.2$, AutoAugment, stochastic weight averaging, and mixed precision training. For a fair comparison, the results of all methods are obtained under the same training settings.

Deployment: We deploy models for CIFAR-10 onto Nvidia Jetson TX2, and models for ImageNet onto Nvidia Jetson Xavier to evaluate the inference latency and the power consumption. The final latency is computed by averaging the latency of 50 inferences with a batch size of 1. The final power consumption is also the average power consumption measured during inference.

A. Results on ImageNet

We use ResNet18 [1] as the baseline model for different scaling methods on ImageNet. As demonstrated in TABLE I, under both the tight latency constraint and the loose constraint, *HACScale* achieves the highest accuracy among all approaches. Compared with the baseline model, *HACScale* improves the accuracy by 2.41% almost without increasing the latency, while obtaining 3.91% accuracy improvement by consuming additional 40% latency budget. By contrast, Tan et al. [10] attains an accuracy gain of 3.01% with an increase of 128% in latency. Meanwhile, as shown in Fig. 4, benefiting from the importance-aware width scaling strategy, *HACScale* achieves the best parameter efficiency, which improves the accuracy by 2.41% with an increase of only 11% in parameters, while NeuralScale increases the parameters by 53.77% for an accuracy improvement of 2.33%. As a consequence, the scaled model by *HACScale* can be deployed to more memory-constrained devices. In addition, *HACScale* improves the hardware utilization by a large margin. It achieves $1.92\times$ and $1.97\times$ improvements in the hardware utilization under the tight and loose latency constraint, respectively. Consequently, even though the scaled models by *HACScale* contain more FLOPs, they can still achieve lower latency due to more efficient utilization of hardware. The power efficiency is represented as the number of GFLOPs completed per Joule. As shown in Fig. 5, *HACScale* also achieves better power efficiency compared with other methods.

[3] $\lambda_t = \lambda\beta^{\frac{t}{T}}$, where λ_t is the learning rate of the current epoch, t is the current epoch, and T is the number of total epochs.

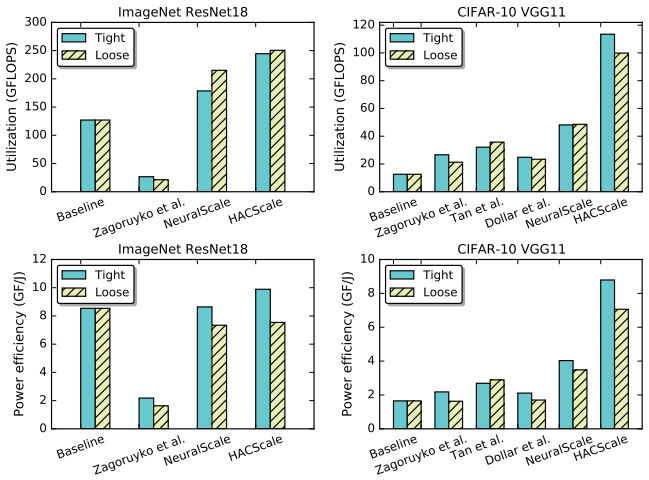

Fig. 5. The hardware utilization and power efficiency of different approaches under the tight latency constraint and the loose latency constraint. The hardware platforms for ImageNet and CIFAR-10 are Nvidia Jetson Xavier and Nvidia Jetson TX2, respectively.

B. Results on CIFAR-10

As demonstrated in TABLE I, under the tight latency constraint, *HACScale* improves the hardware utilization by $9.03\times$, which surpasses other approaches by a large margin. Consequently, *HACScale* obtains an accuracy improvement of 2.23% with a latency increase of only 6.5%. As a comparison, NeuralScale [9] only improves the accuracy by 1.2% with a similar latency. Besides, compared with the baseline model, *HACScale* achieves $5.33\times$ higher power efficiency, which is also the highest among all approaches for comparison. Due to the importance-aware width scaling, *HACScale* only improves the model parameters by 33.1% compared with the baseline model, while NeuralScale [9] has 56.96% more parameters than the baseline model.

On the other hand, under the loose latency constraint, we improve the hardware utilization by $7.94\times$. Consequently, *HACScale* gains a 2.32% accuracy improvement with a latency of $25.5ms$ on Nvidia Jetson TX2, while NeuralScale [9] takes $29.19ms$ but only improves the accuracy by 1.25%. Similarly, Zagoruyko et al. [6] spends $28.63ms$ on achieving a 0.92% accuracy improvement. In addition, the scaled model by *HACScale* only contains 21.1M parameters, which is similar to Tan et al. [10] and VGG19 but better than the rest.

V. Conclusion

In this paper we present a compound scaling framework to efficiently utilize the hardware to improve the accuracy of DNNs without affecting the latency. After analyzing the hardware utilization of different scaling strategies, we find that *width scaling* and *resolution scaling* can significantly improve the hardware utilization. By collaboratively scaling the width and the resolution of a DNN based on their contributions to the accuracy, *HACScale* yields networks that achieve notably higher accuracy without compromising the latency. By separately scaling each layer of a DNN according to their importance to the accuracy, we further improve the accuracy with fewer parameters. Experiments on different datasets and hardware platforms demonstrate that, compared with other approaches, *HACScale* is capable of achieving higher accuracy with fewer parameters and lower latency. In future, we plan to combine *HACScale* with other techniques like NAS to design hardware-tailored, competitive, and efficient DNNs.

Acknowledgement

This study is supported under the RIE2020 Industry Alignment Fund – Industry Collaboration Projects (IAF-ICP) Funding Initiative, as well as cash and in-kind contribution from the industry partner, HP Inc., through the HP-NTU Digital Manufacturing Corporate Lab. This work is also partially supported by NTU NAP M4082282 and SUG M4082087, Singapore.

References

[1] K. He *et al.*, "Deep residual learning for image recognition," in *CVPR*, 2016, pp. 770–778.

[2] A. Vaswani *et al.*, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[3] N. P. Jouppi, *et al.*, "A domain-specific architecture for deep neural networks," *Communications of the ACM*, vol. 61, no. 9, pp. 50–59, 2018.

[4] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, 2017, pp. 1–12.

[5] T. Elsken *et al.*, "Neural architecture search: A survey," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019.

[6] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv preprint arXiv:1605.07146*, 2016.

[7] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *CVPR*, 2018, pp. 4510–4520.

[8] C. Szegedy *et al.*, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proceedings of the AAAI*, vol. 31, no. 1, 2017.

[9] E. Lee and C.-Y. Lee, "Neuralscale: Efficient scaling of neurons for resource-constrained deep neural networks," in *CVPR*, 2020, pp. 1478–1487.

[10] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," *arXiv preprint arXiv:1905.11946*, 2019.

[11] P. Dollár *et al.*, "Fast and accurate model scaling," in *Proceedings of the CVPR*, 2021, pp. 924–932.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[13] S. Xie *et al.*, "Aggregated residual transformations for deep neural networks," in *Proceedings of the CVPR*, 2017, pp. 1492–1500.

[14] M. Tan *et al.*, "Mnasnet: Platform-aware neural architecture search for mobile," in *CVPR*, 2019, pp. 2820–2828.

[15] D. Liu *et al.*, "Bringing ai to edge: From deep learning's perspective," *arXiv preprint arXiv:2011.14808*, 2020.

[16] T.-J. Yang *et al.*, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *Proceedings of the ECCV*, 2018, pp. 285–300.

[17] M. Lin *et al.*, "Hrank: Filter pruning using high-rank feature map," in *CVPR*, 2020, pp. 1529–1538.

[18] T.-J. Yang *et al.*, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the CVPR*, 2017, pp. 5687–5695.

[19] P. Molchanov *et al.*, "Importance estimation for neural network pruning," in *Proceedings of the CVPR*, 2019, pp. 11 264–11 272.