On Hardware-Aware Design and Optimization of Edge Intelligence

Shuo Huai*†, Hao Kong*†, Xiangzhong Luo*, Di Liu‡, Ravi Subramaniam§,
Christian Makaya§, Qian Lin§, Weichen Liu*
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
†HP-NTU Digital Manufacturing Corporate Lab, Nanyang Technological University, Singapore
‡Department of Computer Science, Norwegian University of Science and Technology, Norway
§HP Inc., Palo Alto, California, USA

***Abstract*—Edge intelligence systems, the intersection of edge computing and artificial intelligence (AI), are pushing the frontier of AI applications. However, the complexity of deep learning models and heterogeneity of edge devices make the design of edge intelligence systems a challenging task. Hardware-agnostic methods face some limitations when implementing edge systems. Thus, hardware-aware methods are attracting more attention recently. In this paper, we present our recent endeavors in hardware-aware design and optimization for edge intelligence. We delve into techniques such as model compression and neural architecture search to achieve efficient and effective system designs. We also discuss some challenges in hardware-aware paradigm.**

I. Introduction

The rule of thumb when designing deep learning models is *the higher complexity, the better accuracy*. This rule almost applies to all deep learning models, such as convolutional neural networks (CNNs) and large language models (LLMs) which are widely exposed to public recently due to the breakthrough success of ChatGPT. Meanwhile, models are expected to be implemented at the *edge* close to data so that some computation can be processed locally and expensive communications can be avoided [1], [2]. Recently, new efforts even strive to execute LLMs offline on mobile devices without accessing powerful servers[1].

To expedite complex models on the edge, numerous edge accelerators were devised in past years and more are expected in the near future [3]. Due to high data parallelism and relatively simple operations, mainly addition and multiplication, emerging accelerators feature many simple processing units and deploy an advanced communication infrastructure, e.g., network on chips [4], to connect all processing units [5]. Model training and inference on accelerators involve loading a huge amount of model's weights to processing units and transferring intermediate activation layer by layer via communication infrastructure. Accelerators deploy diverse design paradigms with different processing units and communication infrastructure. As a result, this leads to significant performance variations of the same model on different platforms [6] and in turns propels to exploit hardware-aware design and optimization [7].

Huai, Kong, and Luo contribute equally and are ordered alphabetically.
[1]https://mlc.ai/mlc-llm/

To unleash the full potential of an edge intelligence system, a model should fully utilize the underlying hardware resources so that the accuracy may be maximized while the required performance is guaranteed. Hence, the model should be customized for the target hardware platform. However, there is a huge design space to explore when designing an edge intelligence system, diverse hardware accelerators, various intelligent applications with different datasets, and different performance requirements. There are two ways to achieve this goal: *design* or *optimization*.

- When an extant model is too complicated to implement on an edge platform, compressing the model which is designed in a hardware-agnostic fashion can tailor the compressed model for the edge.
- We also can design a new and lightweight model to fit an edge platform, where the optimal accuracy can be retained and the required performance can be guaranteed.

Both methods should be conducted in a hardware-aware fashion to guarantee that the optimized or newly designed model can achieve or satisfy the expected performance requirement. In this paper, we discuss our recent endeavors to design and optimize edge intelligence systems in a hardware-aware manner and we also outline some unaddressed challenges. In our works, we take CNNs as the main target.

The remainder of this paper is organized as follows: Section II discusses the modelling techniques needed for hardware-aware methods. Section III briefly presents three efforts we recently made in designing and optimizing edge intelligence systems. Section IV further discusses some challenges that can be addressed in this area. Section V concludes this paper.

II. Hardware and CNN Modelling

Designing and optimizing edge intelligence is a non-trivial task due to a huge design space, spanning from model design or selection to training configurations, optimization methods, and hyper-parameters tuning. It is prohibitively expensive to train each design point from scratch and evaluate it on the target edge platform. This not only incurs a significant training expenditure but also environmental sustainability issue caused by high power consumption of servers. Thus, a more common way to evaluate the quality of a specific design is to model the target hardware, so that the performance or energy

consumption of one design can be quickly and approximately obtained using a hardware model. In some cases, a model that can predict accuracy of an input CNN model with different configurations is also desired, e.g., model compression and scaling, where such accuracy model can guide the optimization. Hardware and accuracy models significantly narrow the design space and boost the search procedure. In this section, we discuss the hardware modelling technique as well as the accuracy modelling technique used in our methods.

A. Hardware Modelling

The needs of hardware modelling are twofold: *The inaccurate proxy metrics*–The direct performance metrics of a model are latency, throughput, or energy consumption. These metrics can be obtained by evaluating models on the target platform. It is difficult to always evaluate models on the target hardware, given a huge design space. Hence, many works tend to use some *proxy* metrics, like FLOPs and MACs, to represent real performance of a CNN model. The application of proxy metrics is based on an assumption that there is a straight mapping from real metrics to proxy metrics. Unfortunately, this assumption does not always hold, and reduction on proxy metrics cannot translate into real reduction in terms of latency or energy [8], [4]. The second reason to have a hardware model is that some literature reveals that hardware architectures have considerable impact on a model's performance [5]. Directly measuring the performance on the target device is favorable but infeasible due to the aforementioned design space and diverse ecosystems used by different edge hardware. Such diversity usually requires a conversion from a general framework, like PyTorch, to a specific format only supported by a specific hardware. As a result, A large design space accumulates to a non-negligible conversion overhead [9]. The two reasons together propel the development of hardware modelling.

Since CNNs have relatively simple operations, mainly addtion and multiplication, some strive to use look-up tables (LUTs) to model performance of CNNs. However, LUTs are unable to capture the complex data transmission occurred within the target hardware, thereby leading to imprecision when a model changes [8]. To address the inferiority of LUTs, others, including our works, start to use machine learning (ML) based methods to model performance of CNNs on a hardware. ML-based methods require to collect some operational data and train an ML model with the data. ML models used to predict results can be simple multi-layer perceptron (MLP) or more complex graph neural networks. We found from our experiments, a simple MLP is already able to achieve a relatively good result in terms of prediction accuracy. Hence, in our methods, we mainly use MLP as means to model hardware performance. Figure 1 shows one result we obtained for Nvidia Jetson AGX Xavier, where the left figure shows the results of our MLP-based modelling and the right figure shows the comparison between our MLP modelling ($\text{RSME}_2 = 0.41ms$) and a LUT-based modelling ($\text{RSME}_1 = 11.49ms$).

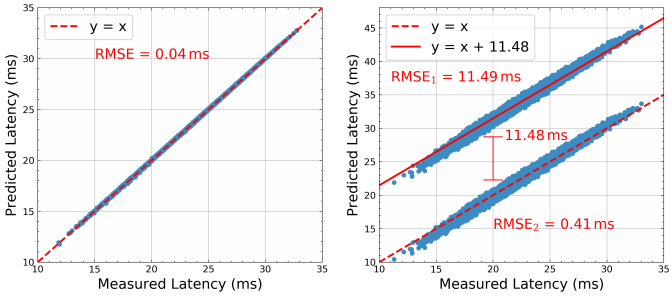

Fig. 1. The comparison between MLP and LUT [8].

B. CNN Modelling

CNN modelling aims at modelling a model's accuracy. It is required, because the accuracy of a model can only be evaluated after the model has been fully trained. Training is known to be computation-intensive, especially for some complex dataset. Some model-level modifications, like pruning and compression, need to re-train the model to retain its accuracy, and this tedious procedure and high cost prohibits the possibility of exploring a large design space. As a result, similar to hardware modelling, some works propose to model the accuracy of different configurations of a CNN architecture in order to shrink the large design space and speed up the exploration procedure. Thanks to some interesting observations from the network design space exploration [10], CNN modelling only needs to sample a small amount of models with different configurations and train them with a small number of epochs. We only use a CNN modelling in multi-dimensional compression discussed in Section III-B, where it facilitates the quick exploration of a large design space. However, we envision that other design and optimization methods may also benefit from a good CNN modelling.

III. Hardware-Aware Design and Optimization

In this section, we present three of our recent works in hardware-aware design and optimization. We start with the widely-studied model compression.

A. ZeroBN

The first work presented in this section is a pruning method, namely *ZeroBN*. *ZeroBN* is motivated by two flaws of existing pruning methods. Similar to what we discuss in Section II-A, existing methods deploy proxy metrics to guide their pruning procedures. This is difficult for a pruning method to achieve a target latency which is paramount for many latency-sensitive applications. To this end, these pruning methods have to repeat a three-stage procedure, *pre-training*, *pruning*, and *fine-tuning*, to guarantee the required latency, while maximizing the accuracy of the pruned model. This drives us to think about how we can prune a model to satisfy its latency requirement in an efficient way.

ZeroBN is our answer for this question [11]. *ZeroBN* is a learning-based pruning method to directly learn a compact model that can satisfy the latency requirement. The main

advantage of ZeroBN is that it is a *one-shot learning process*, i.e., by having a normal training-like process, it can derive a pruned model with a competitive accuracy and latency guarantee. The overview of *ZeroBN* is shown in Fig. 2, where it takes as input a large redundant CNN without training and outputs a compact and well-trained model that satisfies its latency constraint.

ZeroBN divides a traditional training period into three phases to implement the efficient pruning method, where the three phases are 1) **Initial Training** (IT); 2) **Zero Training** (ZT); and 3) **Recovery Training** (RT). The three phases are similar to the three stages in normal pruning methods, pre-training, pruning, and retraining. However, *ZeroBN* combines all them into one training period.

IT phase is equivalent to the pre-training in normal pruning methods, but **IT** is only conducted for several epochs at the beginning of the whole procedure. **IT** aims to obtain some initially trained weights for the input model so that we can evaluate the importance of different channels in later phases. One significant difference between *ZeroBN* and others in terms of training is that we adopt sparsity training in the whole *ZeroBN* process to impose sparsity regularization on scaling factors which are used to identify redundant channels and prune the model.

After several epochs of **IT**, *ZeroBN* proceeds to two iterative phases: **ZT** and **RT**. During these two phases, **ZT** will *"soft-prune"* the input model by temporarily excluding some redundant channels based on their scaling factors and a compression ratio which is determined according to the latency constraint and a latency prediction model. The purpose of **ZT** is to derive a compressed model, train and evaluate it. During **ZT**, a latency predictor is integrated to efficiently calculate a compression ratio which can guarantee the compressed model meet its latency requirement. The latency predictor is the same to what we introduced in Section II-A. *ZeroBN* exploits a global pruning method, i.e., we rank all channels within the model and determine redundant channels based on the global ranking and the computed compression ratio.

ZT trains the "compressed" model for a certain number of epochs, and then **RT** restores the "compressed" model to the full model and trains it for a number of epochs. The rationale behind **RT** is that although the compressed model generated by a **ZT** phase can satisfy its latency constraint, it may be not the optimal model in terms of accuracy due to insufficient training and different training batches. Thus, after a **ZT** phase, we introduce a **RT** phase that restores the compressed model back to the full scale. **RT** can avoid a **ZT** phase from ending up with a suboptimal model, thereby giving *ZeroBN* a chance to learn a better compressed model. **ZT** and **RT** are interleaved after the **IT** phase, and the whole procedure ends with a **ZT** phase that generates the final compact model with latency guarantee.

Figure 3 shows the changes in the importance of all channels during the *ZeroBN* process on an example model, where the number of epochs for **IT** is 6, the compression ratio δ is set to a constant 0.3, and **ZT** and **RT** both take 2 epochs for training. From this figure, we can see how channels' importance is changed over the training procedure, where some channels exhibit consistent importance and others vary significantly. This justifies the necessity of **RT**.

ZeroBN has the same number of epochs as a normal training, where we just split the training epochs into the three parts introduced above. Thus, it does not add any extra overhead in terms of training and pruning. From the experiments [11], we found that *ZeroBN* is an efficient method to design compact models for diverse hardware. It has been open-sourced at https://github.com/HPInc/ZeroBN. We also have explored the potential of such learning-based pruning method in collaborated learning paradigm [12].

B. SmartScissor

ZeroBN provides a means to compress a redundant model for edge systems, but it only considers one dimension of CNN models, i.e., the width (the number of channels per layer). Besides the width, there are two other dimensions which can affect the complexity of a CNN model, i.e., the depth (the number of layers) and the resolution (the size of inputs). When it comes to edge systems, we have to consider two factors, *model complexity* and *computational cost*. Model complexity reflects the number of parameters and weights, or MACs/FLOPs in a model, while computational cost shows the execution complexity, mainly intermediate activations generated by a model during its inference. The implementation of complex models on memory-limited edge systems is hindered by high model complexity and computational cost. Compressing width or depth is able to reduce both model complexity and computational cost, while compressing input size can significantly reduce computational cost. For instance, as the MACs of a CNN reduce quadratically with respect to the image input size, many works resize the input images to a smaller resolution (e.g., 112×112) to reduce the computational cost [13].

The three dimensions of CNN models present an opportunity to compress a complex model in a joint way, instead of only compressing width as *ZeroBN*. EfficientNet [14] is the pioneer work to explore the joint compression and finds such joint compression can achieve better accuracy with lower model complexity. However, there is no free lunch, and this opportunity also poses some challenges. Since three dimensions can be tuned, it exhibits a huge design space. As discussed before, once a small modification is applied to a model, the modified model has to be re-trained to retain accuracy, thereby resulting in high overhead. Thus, a more feasible way as did in EfficientNet [14] is to have a compound shrinking[2], i.e., finding one coefficient for all three dimensions and using it to scale the three dimensions simultaneously. Nevertheless, finding the optimal compound coefficient is a challenging task that still needs significant effort.

When searching for the optimal compound coefficient, some additional attention must be paid to resolution. Different

[2]We use shrinking to refer to compression throughout this session for *SmartScissor*, because we have used shrinking in the original framework.

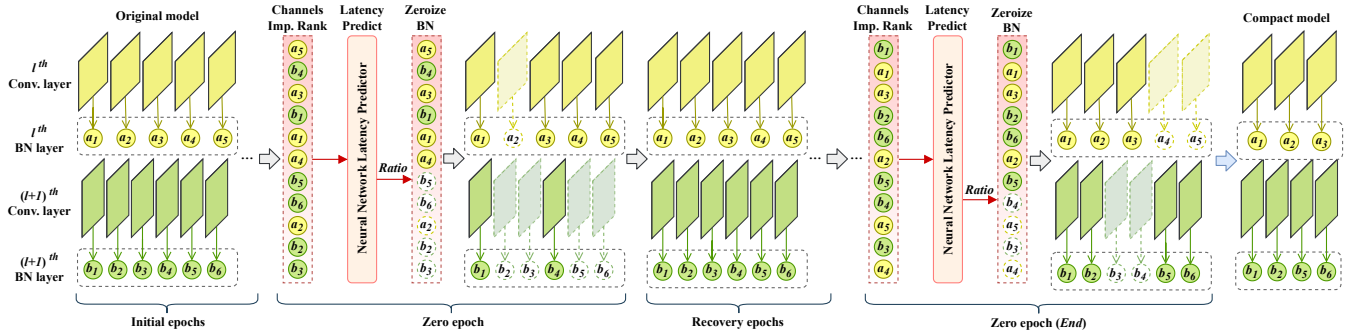

Fig. 2. The process of ZeroBN [11].

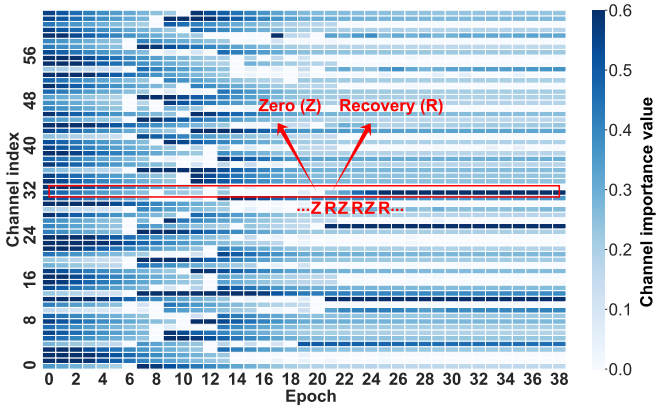

Fig. 3. Channel importance changes during ZeroBN [11].

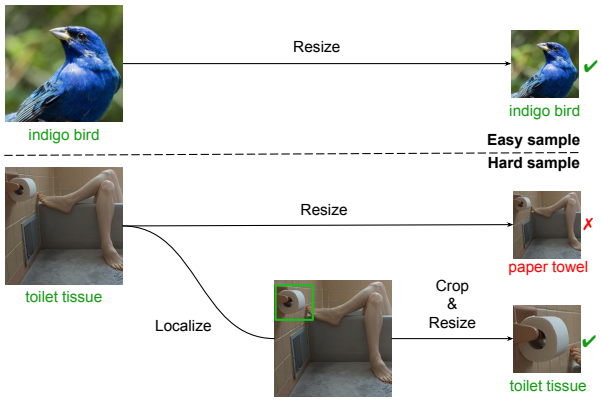

Fig. 4. The prediction results of our pretrained ResNet-50 model. For easy samples, the network can still generate correct predictions at a small resolution (e.g. 112 × 112 for ImageNet). For hard samples, simply resizing the images to a small resolution can lead to misclassification, while the dynamic cropping strategy can correctly classify hard samples at the small resolution [13].

images demonstrate varying levels of classification difficulties. As demonstrated in Fig. 4, easy samples with clear foreground can be correctly recognized even at a small resolution. For hard samples, as the foreground object only occupies a small portion of the whole image, directly shrinking the image to a small resolution will lose the details of the object, leading to a misprediction. Nevertheless, if we can crop the foreground object for inference, even hard samples can be correctly classified at a lower resolution. However, existing image preprocessing methods, e.g., ResizedCenterCrop (RCC), crop all images in a static manner and cannot achieve such instance-aware fine cropping. The efficiency of compounding shrinking and the observation we obtained from image resizing shown in Fig. 4 motivate our work, namely *SmartScissor*.

The overview of *SmartScissor* is plotted in Fig. 5. The general idea behind *SmartScissor* is to propose a method that can effectively and efficiently identify objects in input images such that it can facilitate the inference with low-resolution images, and then a more effective compound shrinking (CS) method can be determined under a complexity budget. In *SmartScissor*, we propose a dynamic image cropping (DIC) component to find the object of interest in an input image. DIC first efficiently localizes the most discriminative foreground of the input image with a lightweight foreground predictor, then the detected foreground region will be preserved and the redundant background will be discarded. DIC is capable of generating fine-cropped images with less spatial redundancy, i.e., low resolution, thereby improving the inference accuracy even under low-resolution settings. The success of DIC attributes to the following two factors.

- **Data**: for classification datasets like ImageNet, there is no out-of-the-box position annotation for the foreground object. Moreover, the position of the foreground object varies in different images, which makes it difficult to efficiently localize the foreground object. To address this limitation, we have a bounding box generation before DIC, where we use Grad-CAM [15] to automatically generate the position annotations. Grad-CAM is deployed to generate a salience map for each image, where a well-trained CNN (e.g. ResNet50) is applied. Then, the bounding box is gradually shrunk and determined based on a threshold t. Figure 6 shows the cropped images under different t, while Table I shows the different accuracy under different t.
- **Light-weight predictor**: Although the foreground predictor we want is similar to object detectors and dozens of object detectors have been proposed, such as SSD and Faster R-CNN, these detector architectures are com-

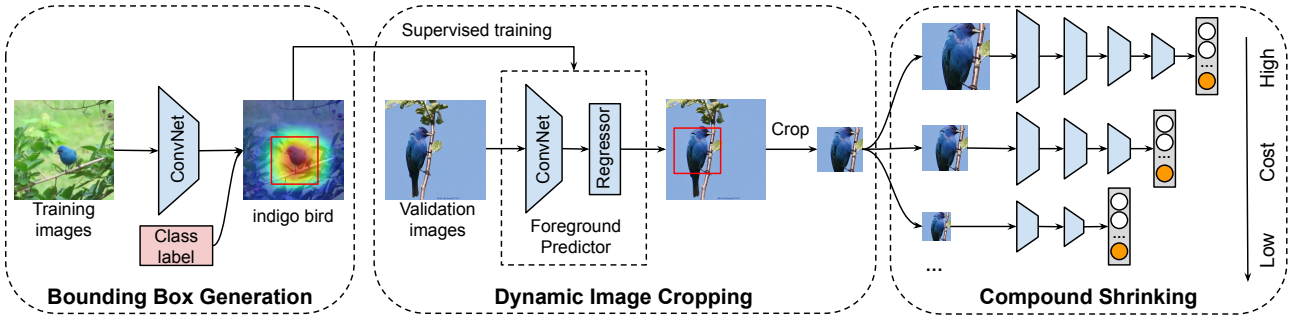

Fig. 5. The overview of *SmartScissor* [13].

Fig. 6. By applying different values of the salience threshold t, we can obtain different cropped images. The larger the threshold value, the more radical the cropping [13].

Model	#Params	#MACs	t	Acc@1
ResNet50	25.6 M	4.1 B	0 (Baseline)	76.02 %
			0.25	76.45 %
			0.5	**76.88** %
			0.75	76.32 %

TABLE I
THE IMPACT OF USING DIFFERENT SALIENCE THRESHOLDS ON PREDICTION ACCURACY. THE MODEL IS TRAINED AND EVALUATED ON IMAGENET-1K. $t = 0$ MEANS USING THE ORIGINAL IMAGES WITHOUT GRAD-CAM CROPPING [13].

pletely inapplicable to our task due to their high complexity. Applying the existing object detector will undermine the benefit of model compression, and an object detector may be more complex than a CNN model itself. We thus design a lightweight foreground predictor, whose detailed architecture is summarized in Table II. It consists of several residual bottleneck blocks [16] and a fully connected layer as the single-box regressor. A residual bottleneck contains two convolutional layers with 1×1 kernels and one convolutional layer with 3×3 kernels in the middle. The computational cost mainly results from the 3×3 convolutional layer. Therefore, to reduce the cost and accelerate the predictor, we only stack two residual bottleneck blocks in each stage, and each block is only equipped with a small number of channels. The proposed predictor only contains 0.27M parameters and 0.09B MACs, which are negligible compared to popular object detectors (e.g., Faster R-CNN with 134.7M (499×) parameters and 15.1B (167.8×) MACs) in terms of model complexity. Moreover, the small overhead of the foreground predictor will be mitigated by CS.

Once the predictor is trained, it can be directly applied to different classification backbones without any extra training overhead. During inference, the trained predictor will quickly localize the foreground object of an input image and generate a finely cropped image, which allows CNN models to predict the input image at lower resolution, thus significantly reducing computational cost.

Fig. 7. The bounding boxes generated with the salience threshold $t = 0.5$, which accurately localize the key object in each image [13].

Stage	Block	Resolution	#C	#L
1	Conv 3×3	224 × 224	16	1
2	Residual Bottleneck	112 × 112	16	2
3	Residual Bottleneck	56 × 56	32	2
4	Residual Bottleneck	28 × 28	32	2
5	Residual Bottleneck	14 × 14	64	2
6	Pooling & Linear	7 × 7	4	1
#Params: 0.27M				
#MACs: 0.09B				

TABLE II
THE ARCHITECTURE OF THE PROPOSED BOX PREDICTOR. #C DENOTES THE NUMBER OF CHANNELS AND #L DENOTES THE NUMBER OF LAYERS [13].

The proposed DIC underpins the low-resolution prediction. Now, we can look at the model complexity, width and depth. As shown in EfficientNet [14], CS that jointly compresses the three dimensions of a model promises higher accuracy over single-dimension compression. The key of CS is to calculate a shrinking coefficient for all dimensions according to their trade-off between accuracy and model complexity. Given the rule of thumb in CNN design, the more complex a model is, the more likely it is to achieve higher accuracy. With a

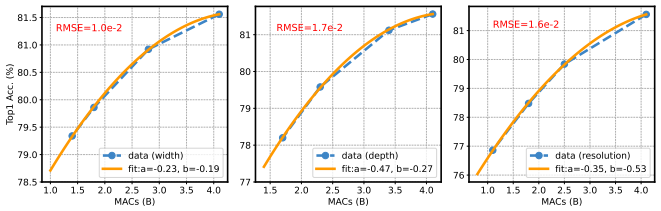

Fig. 8. The actual accuracy (blue dotted line) and the estimated accuracy (yellow line) over MACs by separately shrinking the three dimensions. The low root mean square error (RMSE) indicates that the accuracy estimator can well fit existing data [13].

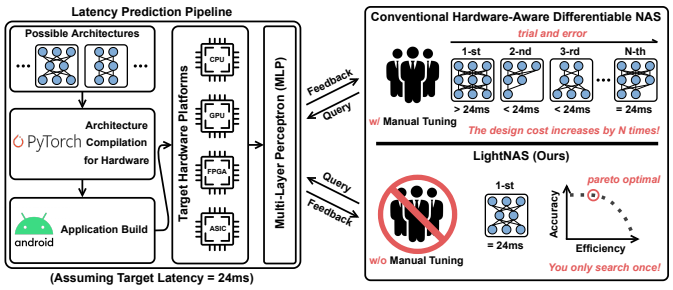

Fig. 9. An intuitive illustration of the proposed LightNAS framework [8].

number of MACs as the constraint or target in *SmartSissor*, it is favorable to have a *just-enough* shrinking, i.e., the number of MACs of the model compressed by CS can be approximately equal to the target number. Intuitively, shrinking different dimensions has different impacts on accuracy and model complexity. The shrinking coefficient needs to strike a good balance among the three dimensions.

EfficientNet exploits a time-consuming and computationally expensive grid search to determine the coefficient. To calculate the shrinking coefficient efficiently, we first quantify the trade-off of each dimension between accuracy and model complexity. Here we use MACs as the metric to measure the cost of models, because all three dimensions are related to the MACs of a model while only the depth and width can affect the model parameters. Given a MACs budget $\mathcal{M}$, we first obtain the accuracy drops resulting from separately shrinking different dimensions, which can be represented as:

$$\Delta A_s(\mathcal{M}) = A_0 - A_s(\mathcal{M}) \tag{1}$$

where $s \in \{d, w, r\}$ represents the shrunk dimension, $A_s(\mathcal{M})$ denotes the accuracy of the shrunk model, and A_0 denotes the accuracy of the original model. Based on our empirical analysis, we design the following equation to determine the shrinking coefficient for each dimension:

$$\mathcal{C}_s(\mathcal{M}) = \frac{\sqrt[3]{\Delta A_d(\mathcal{M}) \cdot \Delta A_w(\mathcal{M}) \cdot \Delta A_r(\mathcal{M})}}{\Delta A_s(\mathcal{M})} \tag{2}$$

where $\mathcal{C}_s(\mathcal{M})$ denotes the shrinking coefficient of the dimension s ($s \in \{d, w, r\}$). Through Eq. (1) and Eq. (2), we are able to efficiently calculate the coefficients once we obtain the accuracy degradation of the three dimensions in the given MACs regime.

However, the training cost of calculating the accuracy drop is still non-negligible. To mitigate the training overhead, we propose a dimension-wise accuracy estimator to quickly estimate the accuracy of the compressed models and calculate the accuracy degradation resulting from shrinking different dimensions in the given MACs regime. First, we sample a couple of models with different MACs by separately shrinking the three dimensions. As demonstrated in Fig. 8, the accuracy distribution of the three dimensions along MACs can be well fitted by a quadratic polynomial. Therefore, we design a simple yet effective polynomial estimator to predict the accuracy with respect to the target MACs $\mathcal{M}$. The estimator can be formulated as follows:

$$A_s(\mathcal{M}) = a_s(\mathcal{M} - \mathcal{M}_0)^2 + b_s(\mathcal{M} - \mathcal{M}_0) + A_0 \tag{3}$$

where $\mathcal{M}_0$ is the MACs of the original model. a_s and b_s are the hyperparameters to fit for dimension s ($s \in \{d, w, r\}$). Figure 8 shows that the proposed estimator can well fit the existing data. Due to the simple and intuitive design of the estimator, we only need to sample and train very few models to train the estimator, and this cost is a one-time cost. With the accuracy estimator established, we are capable of directly calculating the accuracy drop and subsequently the shrinking coefficient across a wide range of MACs regimes. As a result, the cost of determining the coefficients is significantly reduced compared to directly training models to obtain the coefficients. We have conducted extensive experiments on different datasets in comparison with several state of the arts, where *SmartSissor* can achieve higher accuracy with lower computational complexity [13].

C. LightNAS

So far, we have discussed two frameworks that are able to optimize complex and redundant CNN models for edge systems. Another method to generate effective and efficient models for edge systems is to design a model from scratch, like MobileNet, ShuffleNet, etc, which best fit edge platforms. These models are hand-crafted by experienced ML practitioners based on their rich skills and knowledge. Recently, neural architecture search (NAS) has become an emerging technique to replace time-consuming hand-crafted design and automate the design of competitive CNNs. The well-established NAS methods can be divided into reinforcement learning methods, evolutionary methods, and gradient-based (a.k.a., differentiable) methods. Among them, differentiable NAS has started to take the stage of NAS research, thanks to its promising search efficiency. In this section, when referring to NAS, we mean differentiable NAS.

To design hardware-efficient models for resource-limited edge systems, several hardware-aware NAS methods [2] have been developed, which typically leverage the hardware performance metrics (e.g., latency and energy) to guide the

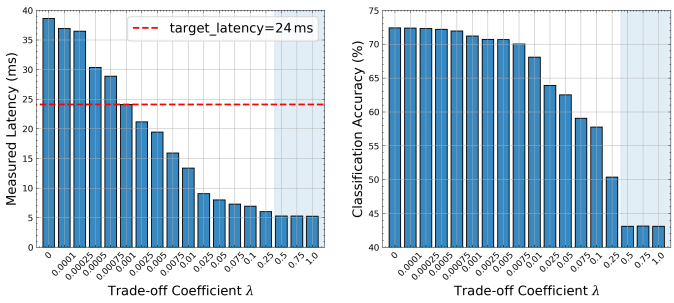

Fig. 10. Illustration of the architecture search results under $\lambda \in [0, 1]$ [8].

search process for hardware-efficient models. The optimization objective can be formulated as follows:

$$\underset{\alpha}{\text{minimize}} \ \mathcal{L}_{valid}(w^*(\alpha), \alpha) + \lambda \cdot LAT(\alpha) \tag{4}$$

where α represents a specific architecture generated by NAS, and w^* denotes the weights of architecture α. $LAT(\cdot)$ is the latency, and $\lambda \geq 0$ is a constant to control the trade-off magnitude between accuracy and latency. Generally, NAS is to find an architecture α that can minimize the loss function of $\mathcal{L}$ and latency regularization.

λ in Eq. (4) plays an unnoticed role in hardware-aware NAS. Similar to *ZeroBN* and *SmartScissor*, NAS expects to find the most complex model that can meet our performance requirement. To search an optimal model for an edge platform satisfying a given latency constraint, λ may need to be tuned for several or many rounds so that the searched model satisfies the specified hardware performance constraint. To show this, we leverage FBNet [17] to repeat a plethora of architecture search experiments under different settings of $\lambda \in [0, 1]$. As shown in Fig. 10, λ is able to trade off between accuracy and latency, which, unfortunately, is quite challenging to tune. As a result, to find the required architecture around the specified latency, the procedure has to repeat multiple search experiments (empirically 10), significantly increasing the search cost by $10\times$ times.

To overcome such limitations, we propose a novel hardware-aware NAS framework, dubbed *LightNAS* [8]. *LightNAS* aims to find the required architecture that satisfies a specified latency within one single search as shown in Fig. 9 (i.e., *you only search once*). The optimization objective of *LightNAS* is similar to Eq. (4) with a small modification as follows:

$$\underset{\alpha}{\text{minimize}} \ \mathcal{L}_{valid}(w^*(\alpha), \alpha) + \lambda \cdot \left(\frac{LAT(\alpha)}{T} - 1 \right) \tag{5}$$

where T is the specified latency constraint. In *LightNAS*, we use the hardware model discussed in Section II-A to quickly evaluate the latency of architecture α. **Different from previous NAS methods, we set λ in Eq. (5) as a learnable hyper-parameter instead of a tuneable constant**. Therefore, the tedious manual hyper-parameter tuning can be replaced with an efficient learning procedure. And, *LightNAS* automatically learns the optimal hyper-parameter configuration for λ during the search process, which maximizes the accuracy

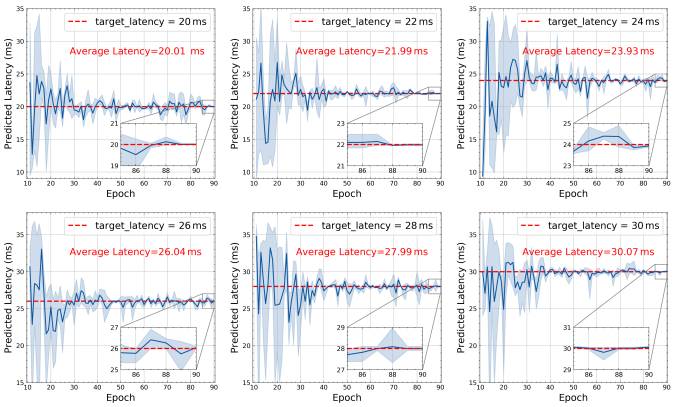

Fig. 11. Visualization of the search process under diverse latency constraints [8].

while satisfying the specified latency constraint $LAT(\alpha) = T$. For simplicity, we use $\mathcal{L}(w, \alpha, \lambda)$ to denote the objective defined in Eq. (5). Subsequently, w and α are updated with gradient descent, whereas λ is optimized using gradient ascent:

$$\begin{cases} w^* = w - \eta_w \cdot \frac{\partial \mathcal{L}(w,\alpha,\lambda)}{\partial w}, \ \alpha^* = \alpha - \eta_\alpha \cdot \frac{\partial \mathcal{L}(w,\alpha,\lambda)}{\partial \alpha} \\ \lambda^* = \lambda + \eta_\lambda \cdot \frac{\partial \mathcal{L}(w,\alpha,\lambda)}{\partial \lambda} = \lambda + \eta_\lambda \cdot \left(\frac{LAT(\alpha)}{T} - 1 \right) \end{cases} \tag{6}$$

where η_w, η_α, and η_λ are the learning rates of w, α, and λ, respectively. After demonstrating *what* the proposed method is, we then analyze *why LightNAS* searches for an architecture α with $LAT(\alpha) = T$. λ can adjust the complexity of the searched architecture, thereby affecting its latency. A larger λ derives the architecture with lower latency, whereas a smaller λ generates the architecture with higher latency as shown in Fig. 10. Then, during the search procedure, there are 2 possibilities.

- $LAT(\alpha) > T$: The architecture does not meet the latency requirement, so the gradient ascent scheme of λ increases λ to reinforce the latency regularization magnitude;
- $LAT(\alpha) < T$: The architecture meets the latency requirement. However, if the architecture's latency is smaller than the required latency, we may not get *the most complex model* which can maximize the attainable accuracy. Thus, the gradient ascent scheme then decreases λ to diminish the latency regularization magnitude.

In both cases, the search engine will strive to make latency $LAT(\alpha)$ towards latency constraint T, i.e., $LAT(\alpha) = T$. As a result, the search engine finally obtains an architecture α with $LAT(\alpha) = T$. Therefore, unlike previous hardware-aware NAS methods that require multiple trial-and-errors to find the desired architecture with latency T, *LightNAS* only needs to search once, greatly improving the search efficiency.

Results: To demonstrate the effectiveness of *LightNAS*, we visualize the search process under diverse latency constraints for a representative edge platform, Nvidia Jetson Xavier, in Fig. 11. The results clearly show that *LightNAS* is able to search for the required architecture that strictly satisfies the specified latency constraint in one single search, where the

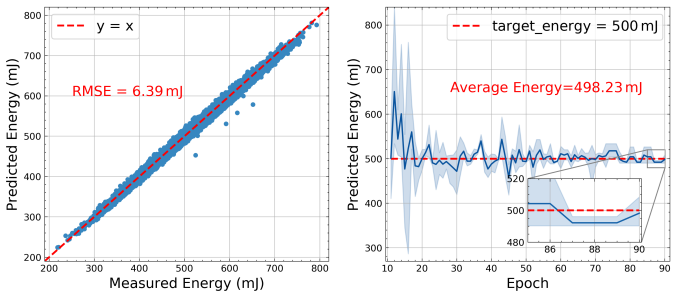

Fig. 12. Illustration of the generality of LightNAS to energy-critical search [8].

latency of the searched architecture gradually converges to the required one. In addition, we also find that *LightNAS* can be easily extended to deal with energy-critical tasks shown in Fig. 12, where we use the same method to model energy and replace the latency regularization with a new energy regularization. Experimental results clearly show the effectiveness of *LightNAS* over previous state-of-the-art NAS methods [8].

IV. Challenges

In this section, we discuss some possible and unaddressed challenges in the field of edge intelligence especially in terms of hardware-aware design.

A. Architecture-Aware Modelling

The hardware modelling techniques discussed in Section II and most literature [2] feature a '*black-box*' fashion, i.e., the modelling is unaware of what happens in the hardware and why different models or architectures perform so differently on the same hardware or on the same CNN model. Some work starts to look at how CNN computations are mapped into a underlying hardware and to analyze the performance of a model from hardware's perspective, like Timeloop [18]. Many factors can affect the performance from hardware's perspective:

- What are the key features of a target accelerator? Like, the number of processing units, the processing unit type, the underlying architecture, etc.
- What kind of underlying interconnect network does the accelerator deploy [4]?
- How are computations mapped to the targeting accelerators? In another word, what is the dataflow of the target accelerator?

Although black-box modelling techniques can achieve relatively good performance in terms of prediction accuracy, they require to collect a huge amount of data. Also, the black box methods may impede the portability of the built model from one hardware platform to another. The procedure of '*collecting-training-deployment*' has to be conducted for every new hardware. Moreover, if a new CNN architecture, new layers or new kernels are introduced, the existing prediction model may need to repeat the '*collecting-training-deployment*'. This is mainly due to the lack of understanding how the target hardware works with CNN models. If some analytical methods based on hardware analysis can be integrated, a more explainable model is possible. For example, an analytical model for the processing unit may explain the reason why a newly introduced operator performs in a certain way and a communication analysis method can explain the rationale of data movements between layers or processing units [4]. Such architecture-aware models should be more precise, have lower training overhead, and provide better portability.

B. Unified Integration

The methods discussed in Section III show different ways to optimize and design CNNs for the edge. We can either optimize an existing model or design a new model for a hardware. However, there is no evidence or conclusion which one is better or in which scenario a specific method should be deployed. This in turn leaves a new design decision for practitioners. If a task, a dataset, and an edge platform are given, a question may be immediately raised for engineers: which method should they use to implement the task on that edge platform? Besides performance and accuracy, many factors also have to be taken into account like training cost and portability. It thus would be desirable to have a methodological framework to integrate different design and optimization methods, then having a unified framework, in which different methods can be quantitatively compared in order to help practitioners to select the optimal method for their own tasks according to their requirements and consideration.

C. Beyond CNN

Most of works in edge intelligence focus on CNN-like models [2], because the majority of applications at the edge are vision-based. Prior to 2021, CNNs held a dominant position in the field. However, since the introduction of transformer in computer vision was first proposed in 2021, the landscape has undergone a significant shift. Moreover, generative AI models with strong capability are a new wave, like stable-diffusion models and ChatGPT models. These models are much more complex than CNN models, up to hundred billion parameters, and also collect more sensitive data from users, e.g., some personal information for generated images or dialogues, so they may be subject to more rigorous data protection. Thus, such models are expected to be increasingly implemented on edge systems and executed offline without the need to access remote servers. New models with architectures different from CNNs lead to new challenges for edge intelligence engineers. The concepts and insights we obtain for CNNs may still apply, but these methods need to undergo significant modifications or some new methods should be proposed for these emerging models.

V. Conclusions

It is envisioned that more diverse AI applications will emerge and be integrated into our daily life in the near

future. Edge as the new computing platform close to sensors and actuators has been gradually becoming one important hardware platform for AI applications. We briefly showcase three hardware-aware methods and conclude this paper with some existing challenges in edge intelligence research. We hope this paper can bring a new perspective to the community.

ACKNOWLEDGEMENT

This study is partially 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 (I1801E0028). This work is also partially supported by Nanyang Technological University, Singapore, under its NAP (M4082282/04INS000515C130).

REFERENCES

[1] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, "Edge intelligence: The confluence of edge computing and artificial intelligence," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7457–7469, 2020.

[2] D. Liu, H. Kong, X. Luo, W. Liu, and R. Subramaniam, "Bringing ai to edge: From deep learning's perspective," *Neurocomputing*, vol. 485, pp. 297–320, 2022.

[3] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[4] S. M. Nabavinejad, M. Baharloo, K.-C. Chen, M. Palesi, T. Kogel, and M. Ebrahimi, "An overview of efficient interconnection networks for deep neural network accelerators," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 10, no. 3, pp. 268–282, 2020.

[5] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.

[6] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *2019 IEEE international symposium on high performance computer architecture (HPCA)*, pp. 331–344, IEEE, 2019.

[7] C. Li, Z. Yu, Y. Fu, Y. Zhang, Y. Zhao, H. You, Q. Yu, Y. Wang, and Y. Lin, "Hw-nas-bench: Hardware-aware neural architecture search benchmark," in *The 9th International Conference on Learning Representations 2021 (ICLR 2021)*, 2021.

[8] X. Luo, D. Liu, H. Kong, S. Huai, H. Chen, and W. Liu, "Lightnas: On lightweight and scalable neural architecture search for embedded platforms," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[9] H. Kong, S. Huai, D. Liu, L. Zhang, H. Chen, S. Zhu, S. Li, W. Liu, M. Rastogi, R. Subramaniam, *et al.*, "Edlab: A benchmark for edge deep learning accelerators," *IEEE Design and Test*, 2021.

[10] I. Radosavovic, R. P. Kosaraju, R. Girshick, K. He, and P. Dollár, "Designing network design spaces," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 10428–10436, 2020.

[11] S. Huai, L. Zhang, D. Liu, W. Liu, and R. Subramaniam, "Zerobn: Learning compact neural networks for latency-critical edge systems," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pp. 151–156, IEEE, 2021.

[12] S. Huai, D. Liu, H. Kong, X. Luo, W. Liu, R. Subramaniam, C. Makaya, and Q. Lin, "Collate: Collaborative neural network learning for latency-critical edge systems," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pp. 627–634, IEEE, 2022.

[13] H. Kong, D. Liu, S. Huai, X. Luo, W. Liu, R. Subramaniam, C. Makaya, and Q. Lin, "Smart scissor: Coupling spatial redundancy reduction and cnn compression for embedded hardware," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2022.

[14] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International Conference on Machine Learning (ICML)*, pp. 6105–6114, 2019.

[15] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE/CVF International Conference on Computer Vision (CVPR)*, pp. 618–626, 2017.

[16] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[17] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10734–10742, 2019.

[18] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to dnn accelerator evaluation," in *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pp. 304–315, IEEE, 2019.