HSCoNAS: Hardware-Software Co-Design of Efficient DNNs via Neural Architecture Search

***Abstract*—In this paper, we present a novel multi-objective hardware-aware neural architecture search (NAS) framework, namely HSCoNAS, to automate the design of deep neural networks (DNNs) with high accuracy but low latency upon target hardware. To accomplish this goal, we first propose an effective hardware performance modeling method to approximate the runtime latency of DNNs on target hardware, which will be integrated into HSCoNAS to avoid the tedious on-device measurements. Besides, we propose two novel techniques, *i.e.*, dynamic channel scaling to maximize the accuracy under the specified latency and progressive space shrinking to refine the search space towards target hardware as well as alleviate the search overheads. These two techniques jointly work to allow HSCoNAS to perform fine-grained and efficient explorations. Finally, an evolutionary algorithm (EA) is incorporated to conduct the architecture search. Extensive experiments on ImageNet are conducted upon diverse target hardware, *i.e.*, GPU, CPU, and edge device. Remarkably, HSCoNet-Edge-A/B achieves a speedup of x1.8/x1.2 on the edge device while maintaining +2.3%/+1.0% higher accuracy compared with MobileNetV2/V3. Besides, HSCoNet-GPU-B achieves +1.3% higher accuracy than ProxylessNAS-GPU with comparable latency.**

I. Introduction

Deep neural networks (DNNs) have become the *de facto* engine of artificial intelligence (AI). Over the past few years, DNNs have achieved remarkable success in a wide range of real-world embedded applications, such as image classification [6], autonomous driving [4], intelligent IoT [6], *etc*. However, to pursue competitive accuracy, DNNs are evolving deeply with more layers as well as widely with more channels, thereby incurring the *computational gap* between complicated DNNs and resource-limited hardware like edge devices, which are deemed as the key computing platform for future AI [17]. Nonetheless, designing resource-efficient DNNs for less capable hardware still remains highly challenging since hardware-aware DNNs need to be small and fast, yet still accurate.

Significant efforts have been dedicated to bridging the above *computational gap*. Among them, model compression is regarded as a promising solution, where redundant weights or channels are pruned or quantized to reduce the complexity of DNNs [1], [3]. However, recent work [8] demonstrates training a compact DNN model from scratch can achieve comparable accuracy as the compressed one. Meanwhile, training from scratch can avoid the tedious tuning-retraining procedures. Besides, another alternative to tackle the *computational gap* is to directly design lightweight DNNs, *e.g.*, ShuffleNet series [9], MobileNet series [5], [13], *etc*. However, designing such state-of-the-art DNNs involves a huge amount of trial and error by experts, including fabricating the structure and fine-tuning the hyper-parameters. To alleviate this burden, the paradigm of designing DNNs has been shifted from *manual* to *automatic*, *i.e.*, neural architecture search (NAS) [7], [10], [12], [19].

Nonetheless, early NAS works [7], [10], [12], [19] suffer from several crucial drawbacks, hindering the broad application of these approaches for diverse hardware, especially for those with resource limitations. To begin with, prior NAS works are hardware-agnostic since they only aim to search for DNNs in terms of accuracy, regardless of other performance metrics like latency and power [6], [14], which are critical for edge AI applications [17]. Thus, they derive complicated DNNs with high accuracy, which, unfortunately, are inapplicable to resource-limited hardware. Besides, regarding designing DNNs, prior NAS methods apply the FLOPs/Params count to denote the complexity of DNNs. However, the FLOPs/Params count is indirect as well as hardware-agnostic, and thus cannot directly translate to the runtime performance upon target hardware, which is illustrated in Fig. 3.

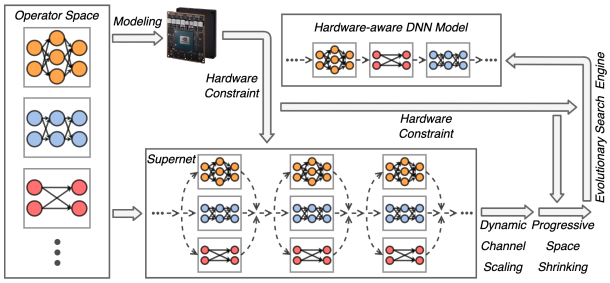

Fig. 1: Overview of the proposed HSCoNAS framework.

Several hardware-aware NAS works [2], [15], [16] are recently proposed to address the above issues. MnasNet [15] adopts the reinforcement learning (RL) scheme and formulates the runtime latency into the RL rewards to search for efficient DNNs, where the latency is directly measured on target hardware, which involves the tedious on-device measurements and incurs significant search overheads, approximately 40,000 GPU hours [2]. FBNet [16] and ProxylessNAS [2] both borrow the differentiable formula from DARTS [7] and incorporate the runtime latency into the training loss function, which serves as the latency regularizer, *i.e.*, they strive to optimize the latency, but unable to guarantee if the latency requirement is specified. However, in real-world scenarios, *just* meeting the specified latency would be better since it may derive more complicated DNNs which usually provide higher accuracy.

To tackle the aforementioned issues, unlike previous hardware-agnostic NAS methods, we propose an efficient and unified hardware-software co-design NAS framework, namely HSCoNAS, to automatically design efficient DNNs with high accuracy but low latency upon diverse target hardware, *i.e.*, GPU, CPU, and edge devices. The overview of HSCoNAS is illustrated in Fig. 1. Our novel contributions are as follows:

1) We propose a hardware performance modeling method to approximate the runtime latency of DNNs upon target hardware while introducing negligible overheads.
2) We formulate a multi-objective NAS approach, which strikes an effective trade-off between accuracy and la-

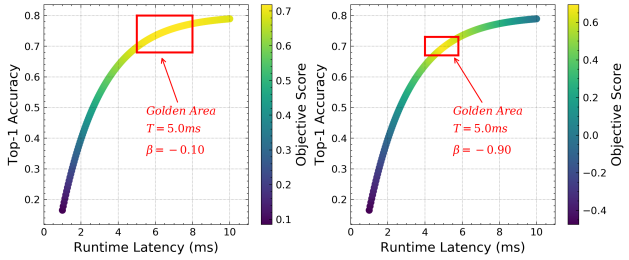

Fig. 2: Illustration of the objective defined in Eq (2) with different β values, assuming the latency constraint $T = 5.0$ms.

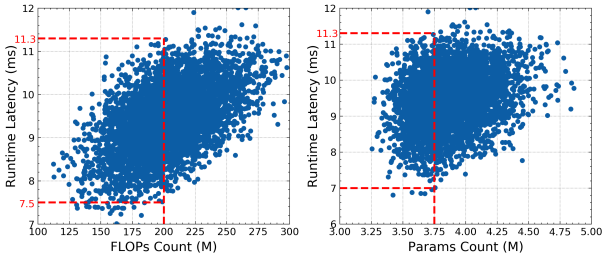

Fig. 3: Illustration of relationships between the runtime latency and FLOPs (*left*) / Params (*right*) count.

tency. Then, we propose a novel dynamic channel scaling scheme to enable the channel-level explorations (*i.e.*, fine-grained), thereby generating more superior DNNs. Besides, we present a progressive space shrinking to refine the search space towards target hardware as well as alleviate the search overheads, followed by an evolutionary algorithm to conduct efficient architecture search.

3) We perform extensive experiments on ImageNet with three hardware devices, *i.e.*, GPU, CPU, and edge device. Notably, HSCoNet-Edge-A/B obtains a speedup of x1.8/x1.2 on the edge device while with +2.3%/+1.0% higher accuracy compared with MobileNetV2/V3 [5], [13]. HSCoNet-GPU-B obtains +1.3% higher accuracy than ProxylessNAS-GPU [2] with comparable latency.

II. Preliminaries and Problem Formulation

A. Preliminaries

NAS has been regarded as a promising alternative to automate the design of competitive DNNs since manually-designed DNNs involve a huge amount of trial and error by experts. Prior NAS works [7], [10], [15] usually construct an over-parameterized network with L layers, namely supernet $\mathcal{N}$, to ease the search of optimal neural architectures. As illustrated in Fig. 1, the supernet can be formulated as a directed acyclic graph (DAG) based on a set of K operators, *i.e.*, $\mathcal{O} = \{op_i\}_{i=1}^{K}$, where each layer has K different operators. Note the operator can be the basic convolution, pooling, or building blocks from manually-designed DNNs like ShuffleNetV2 [9] and MobileNetV2 [13]. Finally, an architecture candidate $arch$ can be sampled from the supernet by selecting one operator for each layer, *i.e.*, $arch = \{op^l\}_{l=1}^{L} \in \mathcal{N} = \{\mathcal{O}^l\}_{l=1}^{L}$. Note we set $L = 20$ and $K = 5$ across this work.

Thus, once the supernet is well trained, we can evaluate different architecture candidates (*i.e.*, subgraphs) with inherited weights from the supernet by means of the weight-sharing technique [10], thereby avoiding the considerable overheads of training vast stand-alone DNNs. Following the weight-sharing paradigm [10], the objective of training the supernet is to minimize the expected loss over the search space $\mathcal{A}$ as follows:

$$\mathcal{W}^* = \arg\min_{arch \in \mathcal{A}} \mathbb{E}[\mathcal{L}(\mathcal{W}_{arch}, D_{train})] \text{ s.t., } arch \sim \mathcal{U}(\mathcal{A}) \quad (1)$$

where $arch = \{op^l\}_{l=1}^{L}$ denotes the architecture candidate with inherited weights $\mathcal{W}_{arch}$ from the supernet and D_{train} is the training dataset. $\mathcal{U}(\cdot)$ represents the uniform distribution. Such weight-sharing technique significantly reduces the search cost from thousands of GPU days [19] to several GPU days [2], [16] without loss of accuracy [10]. Therefore, in HSCoNAS, we also adopt the weight-sharing paradigm based on the supernet to efficiently search for hardware-aware DNNs.

B. Problem Formulation

Given a search space $\mathcal{A}$ and a target hardware platform, our objective is to search for $arch \in \mathcal{A}$, which can maximize the accuracy while guaranteeing the specified latency requirement. However, for some deep learning platforms, they may trade latency for higher accuracy, and vice versa. Thus, to accomplish flexibility, we present a multi-objective formula to achieve the trade-off between accuracy and latency as follows:

$$\underset{arch \in \mathcal{A}}{\text{maximize}} \; ACC(arch) + \beta \times \left| \frac{LAT(arch)}{T} - 1 \right| \quad (2)$$

where $ACC(\cdot)$ and $LAT(\cdot)$ denote accuracy on target task and runtime latency on target hardware, respectively. T is the specified latency constraint on target hardware. $\beta < 0$ is the trade-off coefficient. For simplicity, we denote the above objective as $\mathcal{F}(arch, T)$. Following the empirical rule in [15], [18], *i.e.*, larger DNNs usually lead to higher accuracy, we can either penalize the architecture with high latency or with low accuracy, thereby achieving an effective trade-off between accuracy and latency. An example to show the trade-off performance of Eq (2) is visualized in Fig. 2. In practice, HSCoNAS aims to discover hardware-aware DNNs lied in the *Golden Area*, approximately satisfying the specified latency constraint T as well as achieving competitive accuracy.

III. HSCoNAS Framework

In this paper, we propose an efficient and unified hardware-software co-design NAS framework, namely HSCoNAS, to automate the design of efficient DNNs with low latency upon target hardware while achieving competitive accuracy. The demonstrations of HSCoNAS are twofold. From *hardware's perspective*, we present an effective hardware performance modeling method to approximate the runtime latency of DNNs upon target hardware, avoiding the tedious on-device measurements. From *software's perspective*, we introduce a multi-objective evolutionary algorithm (EA) based NAS approach, where a novel dynamic channel scaling scheme is integrated to enable HSCoNAS to perform channel-level explorations since the number of channels has a nontrivial impact on both accuracy and latency [18]. Furthermore, we propose a novel progressive space shrinking method to improve the

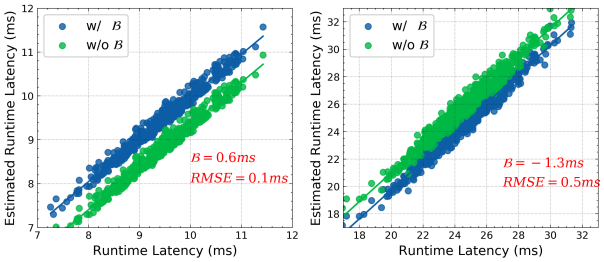

Fig. 4: Illustration of the effectiveness of proposed hardware performance modeling method on GPU (*left*) and CPU (*right*).

quality of the search space towards target hardware, thereby generating more superior hardware-aware neural architectures. We elaborate on HSCoNAS in the subsequent sections.

A. Hardware Performance Modeling

As illustrated in Fig. 3, we observe that neural architectures with the same FLOPs or Params count significantly differ regarding the runtime latency. Therefore, the FLOPs or Params count is a hardware-agnostic metric and is inadequate to reflect the runtime performance upon target hardware. However, directly measuring the runtime performance on target hardware for $arch \in \mathcal{A}$ is prohibitively expensive since the search space of NAS is immensely large, *e.g.*, $|\mathcal{A}| \approx 9.5 \times 10^{33}$ in HSCoNAS. To tackle this, we analytically model the runtime latency for $arch \in \mathcal{A}$ using the following formulation:

$$LAT(arch) = \sum_{l=1}^{L} op^l + \mathcal{B} \tag{3}$$

where op^l represents the operator of l-th layer in $arch$, *i.e.*, $arch = \{op^l\}_{l=1}^{L}$. $\mathcal{B}$ is empirically approximated as follows:

$$\mathcal{B} = \frac{1}{M}(\sum_{i=1}^{M} LAT^{+}(arch_i) - \sum_{i=1}^{M} LAT(arch_i)) \tag{4}$$

where $LAT^{+}(arch_i)$ denotes the on-device runtime latency of $arch_i$. M is the number of architectures sampled from $\mathcal{A}$.

The intuitions behind this performance model. Since one layer's output is the input of the next layer within DNNs, DNNs are executed layer by layer on most of hardware architectures. Thus, the total latency of a DNN model can be derived by summing up the latency across layers. However, the execution of DNNs involves a considerable amount of data movement, incurring high communication overheads between computing units and memory units as well as different computation units. Meanwhile, hardware architectures impose different advanced design techniques, *e.g.*, pipeline, weight buffer, cache, *etc*, to mitigate such communication overheads. Thus, to accurately capture the communication effects of diverse hardware, we incorporate a hardware-specific bias $\mathcal{B}$ to our hardware performance model as shown in Eq (3).

We perform extensive experiments on three hardware devices, *i.e.*, GPU (Nvidia Quadro GV100), CPU (Intel Xeon Gold 6136), and edge device (Nvidia Jetson Xavier). As illustrated in Fig. 4, we observe a strong correlation between the on-device and the estimated runtime latency after incorporating $\mathcal{B}$. Notably, the proposed method achieves an extremely low root-mean-squared-error (RMSE) of 0.1ms, 0.5ms, 1.7ms for CPU, GPU, edge device, respectively. However, empirically $\mathcal{B}$ should be positive since it serves to compensate for the latency bias incurred by the communication overheads. Nonetheless, this is only valid for GPU. Our conjecture is that it involves unique communication overheads when profiling different operators upon diverse hardware. After executing all layers, such communication overheads are mitigated or hidden.

TABLE I: Hardware characteristics under different modes.

	Power / mW				Latency / ms	$\mathcal{B}$ / ms
	CPU	DLA	SOC	Total		
Mode 3	941	5712	2902	13809	38.7	-3.1
Mode 4	1002	4592	2535	11863	35.5	-1.5
Mode 5	1491	7991	3649	18663	35.7	3.3
Mode 6	1802	8301	3526	18982	35.8	5.8

An interesting observation is the communication bias $\mathcal{B}$ is not a constant on heterogeneous platforms but depends on the hardware configurations. We conduct experiments on our edge device, *i.e.*, Xavier, by means of its configurable property. We switch Xavier to different power modes, *i.e.*, 3, 4, 5, 6, where only the number and frequency of CPU cores are different. The frequency of each CPU gradually increases from 1,200 MHz to 2,100 MHz from mode 3 to 6. Following the *Control Variable* rule, we set the number of CPU as 1 for each power mode throughout our experiments. As illustrated in TABLE I, we observe Xavier achieves the best speedup as well as lowest power consumption when $\mathcal{B}$ is around zero. Thus, for heterogeneous systems, regarding different configurations, we should specialize different DNNs to achieve the best runtime performance in terms of both latency and power consumption.

B. Dynamic Channel Scaling

In literature, the hardware-aware NAS works [2], [15], [16] merely search for the optimal configuration of the operator in each layer while keeping the number of channels in each operator fixed. However, as demonstrated in [13], [18], the number of channels has an essential impact on both accuracy and runtime efficiency upon target hardware. Nonetheless, the conventional channel scaling scheme [18] is performed after the neural architecture is determined, and a uniform scaling factor is imposed across layers as illustrated in Fig. 5 (*top*), thereby cannot achieve effective trade-offs between accuracy and efficiency. To alleviate these issues, we propose a dynamic channel scaling scheme as depicted in Fig. 5 (*bottom*), which is further integrated into HSCoNAS to enable the channel-level explorations, *i.e.*, determining the best channel configuration for each layer. To achieve this, we first define a list of n channel scaling factors in HSCoNAS as $C = \{c_1, c_2, ..., c_n\}$, *e.g.*, $\{0.1, 0.2, ..., 1.0\}$. Let S^l denote the maximum number of channels for the l-th layer. It is worth noting that the maximum channel number S^l can be varied according to the hardware characteristics since the parallelism capability among diverse hardware significantly differs as demonstrated in Section III-A.

Recall that NAS algorithms [2], [15] select one proper operator op^l from the operator set $\mathcal{O}$ for each layer l to

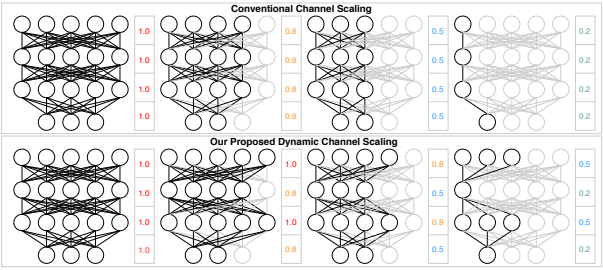

Fig. 5: Comparisons between the conventional (*top*) and the proposed dynamic channel scaling (*bottom*) scheme. Note the scaled number of channels is rounded (e.g., $5 \times 0.5 \approx 3$).

generate an architecture candidate. To begin with, we initialize the number of channels as S^l for layer l in the supernet, *i.e.*, initialized with the maximum number. In practice, the dynamic channel scaling is implemented via scaling down from the maximum number of channels. The reason behind this is the scaling down method can avoid collapses during training the supernet since we need to reconstruct the supernet topology and reload the inherited weights into memory once the scaled number of channels is larger than the initialized one. Throughout training the supernet, we leverage a masking mechanism with the vector $\mathbb{I}^l \in \{0,1\}^{S^l}$, where the scaling factor $c^l \in C$ is used to manipulate the number of channels for each operator within layer l, *i.e.*, assigning 1 to the selected channels and 0 to the masked ones. By means of the scaling down method, we can derive the output of op^l as $\mathbb{I}^l \times op^l(x)$, where x denotes the output of the previous layer. To incorporate the dynamic channel scaling into HSCoNAS, we adjust the supernet training accordingly, *i.e.*, we change $arch \in \mathcal{A}$ in Eq (1) from $\{op^l\}_{l=1}^L$ to $\{op^l, c^l\}_{l=1}^L$, where $c^l \in C$ is dynamically imposed across different layers. After the supernet is well trained, the proposed EA-based architecture search (refer to Section III-D) can automatically discover the optimal architecture candidate upon target hardware, *i.e.*, $arch^* = \{op^{l*}, c^{l*}\}_{i=1}^L$, using the weight-sharing technique.

There are twofold benefits by integrating the dynamic channel scaling scheme into HSCoNAS. First, by enabling the channel-level explorations, HSCoNAS can discover more superior and fine-grained architectures in terms of accuracy compared with [2], [16], which can be seen from extensive experimental results as illustrated in Section IV-B. Second, HSCoNAS can adaptively derive hardware-aware neural architectures upon diverse target hardware. Especially for resource-limited edge devices, HSCoNAS equipped with the hardware performance modeling can find the maximum number of channels for each layer under the specified latency since more channels usually imply higher accuracy [18].

C. Progressive Space Shrinking

Since the search space of NAS is combinatorially large, we propose an efficient space shrinking method to progressively prune and shrink the initial search space, which finally arrives at a well-designed subspace where it is much easier to find superior DNNs upon target hardware [11]. Thus, we only need to explore the well-designed subspace instead of the whole

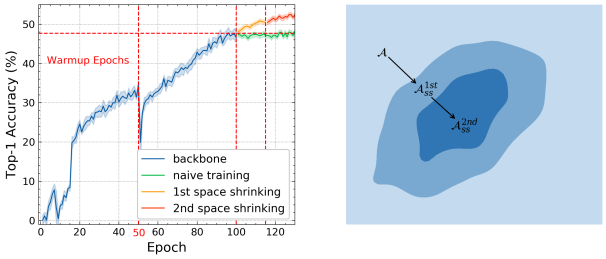

Fig. 6: Illustration of the progressive space shrinking scheme.

space, thereby significantly improving the search efficiency. In HSCoNAS, we characterize the quality of different subspaces upon target hardware with statistical distribution estimates.

Definition 1: Given a set of N architecture candidates uniformly sampled from a subspace $\mathcal{A}^{sub}$, the quality $Q(\mathcal{A}^{sub})$ of $\mathcal{A}^{sub}$ upon target hardware is defined as follows:

$$Q(\mathcal{A}^{sub}) = \frac{1}{N}\sum_{i=1}^{N} \mathcal{F}(arch_i, T), \text{ s.t. } arch_i \sim \mathcal{U}(\mathcal{A}^{sub}) \quad (5)$$

where $\mathcal{F}(\cdot)$ represents the objective defined in Eq (2). For instance, for two subspaces $\mathcal{A}^1$ and $\mathcal{A}^2$, if $Q(\mathcal{A}^1) > Q(\mathcal{A}^2)$, this indicates that we have a higher probability to find better DNNs in terms of the trade-off between latency and accuracy within subspace $\mathcal{A}^1$. We set N to 100 across our experiments, which is proven to be sufficient in [11].

The progressive space shrinking consists of three stages: the initial search space $\mathcal{A}$, the first space shrinking to $\mathcal{A}_{ss}^{1st}$, and the second space shrinking to $\mathcal{A}_{ss}^{2nd}$. These procedures are illustrated in Fig. 6 (*right*). To begin with, we train the supernet $\mathcal{N}$ for 100 epochs within the initial search space $\mathcal{A}$, which serves as the foundation for the subsequent space shrinking steps since we need to have architecture samples to approximate the quality of different subspaces. Then, we start the first space shrinking, where we sample a subspace for each operator within each layer and use Definition 1 to evaluate the quality of different subspaces. Finally, the operator with the highest quality is selected for that layer. For the first stage we apply this space shrinking to 20-th, 19-th, 18-th, 17-th, layer by layer. Note that when we evaluate the subspaces of the current layer, the operator of its subsequent layer should be fixed. For example, when evaluating the 19-th layer, we fix the operator of 20-th layer according to the subspace quality. After applying the space shrinking to the four layers, we complete the first stage space shrinking, reaching a smaller design space $\mathcal{A}_{ss}^{1st}$, which reduces the space size by three orders of magnitudes. Furthermore, we tune the supernet within subspace $\mathcal{A}_{ss}^{1st}$ for 15 epochs and then conduct the second space shrinking in order of 16-th, 15-th, 14-th, 13-th layer in the same way. At the end, we arrive at $\mathcal{A}_{ss}^{2nd}$, further reducing the space size by another three orders of magnitudes.

This shrinking policy is based on observations from our vast experiments on different combinations of space shrinking. In terms of complexity, if we evaluate the subspaces of four layers at the same time, it needs to evaluate 5^4 subspaces, whereas our method only needs to evaluate 5×4 subspaces. Moreover, as illustrated in Fig. 7 (*left*), we observe after each

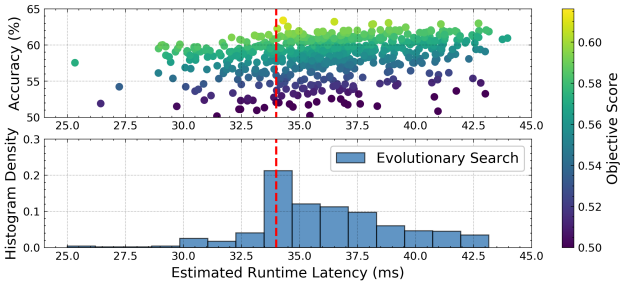

Fig. 7: Illustration of the evolutionary results. The red dashed line denotes the latency constraint for edge device, *i.e.*, 34ms.

space shrinking the supernet obtains higher accuracy[1], where *backbone* denotes training the supernet with the initial space $\mathcal{A}$ and *naive training* indicates to continue training the supernet within the initial space $\mathcal{A}$.

D. Evolutionary Architecture Search

With all the key components introduced before, this section presents our architecture search method. EA and RL are two widely used algorithms in literature. However, RL incurs a high search cost since it is hard to converge [19]. Thus, we adopt EA across this work, which is as effective as RL but with higher efficiency [12]. HSCoNAS aims to search for the architecture candidate with the highest objective score (refer to Eq (2)), which can be formulated as follows:

$$arch^* = \arg\max_{arch \in \mathcal{A}} \mathcal{F}(arch, T) \tag{6}$$

where $arch = \{op^l, c^l\}_{l=1}^{L}$ represents the architecture candidate sampled from the supernet with inherited weights. The pseudo-code of our EA algorithm is summarized in Algorithm 1. We set the number of generations G as 20, the size of population P as 50, the number of parents K as 20, respectively. During each evolution, crossover with a probability of $C_{prob} = 0.25$ and mutation with a probability of $M_{prob} = 0.25$ jointly work to yield efficient explorations not only on the operator level but also on the channel level.

We take the evolutionary results on the edge device as an example, which is illustrated in Fig. 7 (*top*), where a specified latency requirement of 34ms is given. Notably, HSCoNAS discovers an optimal neural architecture with the runtime latency of 34.3ms, which approximately meets the specified latency constraint. We visualize the results in a histogram as seen in Fig. 7 (*bottom*), where we find EA can find more architecture candidates which have the runtime latency closed to the specified latency constraint, *i.e.*, 34ms. This proves the effectiveness of our objective function in Eq (2).

IV. Experimental Settings and Results

In this section, we conduct extensive experiments to demonstrate the effectiveness of HSCoNAS on three hardware platforms, *i.e.*, GPU (Nvidia Quadro GV100), CPU (Intel Xeon Gold 6136), edge device (Nvidia Jetson Xavier), with the specified latency constraint of 9ms, 24ms, 34ms, respectively.

[1]The higher accuracy of the supernet indicates the architectures sampled from this search space are likely to obtain higher accuracy [11].

Algorithm 1 Evolutionary Architecture Search

Input: Supernet $\mathcal{N}$, number of generations/populations G/P, crossover/mutation probability C_{prob} / M_{prob}
Output: Architecture candidate with highest objective score

$POP_0 \leftarrow init\_population(P)$ ▷ initial population
for $i = 1$ **to** G **do**
  $LAT_{i-1} \leftarrow LAT(POP_{i-1})$ ▷ see Eq (3)
  $Score_{i-1} \leftarrow \mathcal{F}(POP_{i-1}, T)$ ▷ see Eq (2)
  Sort POP_{i-1} in terms of $Score_{i-1}$ ascendingly
  $POP_{i-1}^{parent} \leftarrow POP_{i-1}[P-K:]$
  $POP_{i-1}^{child} \leftarrow POP_{i-1}[:P-K]$
  for $j = 1$ **to** $P - K$ **do**
    In_a, In_b = $Random(POP_{i-1}^{parent})$ ▷ $In_a \neq In_b$
    $In_{new} \leftarrow Crossover(In_a, In_b, C_{prob}))$
    $POP_{i-1}^{child}[j] \leftarrow Mutation(In_{new}, M_{prob})$
  end for
  $POP_i \leftarrow POP_{i-1}^{child} \cup POP_{i-1}^{parent}$
end for
Derive $Score_G$ and sort POP_G ascendingly with $Score_G$
Return $POP_G[-1]$ ▷ with highest objective score

A. Experimental Settings

Dataset. We apply the widely used classification dataset, *i.e.*, ImageNet, across our experiments. Following [2], [16], we random sample 20,000 images from the training dataset as the validation dataset with 20 images from each class. Note the sampled 20,000 images are only used during the architecture search phase, whereas the original validation dataset consisting of 50,000 images is used to report the final accuracy as well as compared with other state-of-the-art works.

Training settings. For training the supernet, we adopt the SGD optimizer with a momentum of 0.9, a weight decay of 3×10^{-5}, a norm gradient clipping of 5, a batch size of 512, a learning rate of 0.5 annealed down to zero following the cosine schedule for 100 epochs, denoted as the supernet backbone. Besides, the standard data augmentations are applied throughout this work. Furthermore, after each stage of progressive space shrinking, we tune the supernet backbone within the shrunk search space for 15 epochs with an initial learning rate of 0.01 and 0.0035, respectively. Other training settings keep consistent with training the supernet backbone. The neural architecture discovered HSCoNAS for diverse target hardware is denoted as HSCoNet. To achieve fair comparisons with other works [2], [7], [15], [16], we train those hardware-aware HSCoNets from scratch, which are conducted on 8 $\times$ Tesla V100 GPUs. The training settings are the same as training the supernet backbone with two exceptions. The batch size is set to 1024 and the learning rate warm-up strategy is applied for the first five epochs.

B. Experimental Results

Similar to [5], [13], we apply two channels layouts, *i.e.*, $[48, 128, 256, 512]$ and $[68, 168, 336, 672]$, to generate two different sizes of HSCoNet denoted as HSCoNet-A and HSCoNet-B, respectively. The searched neural architectures for diverse target hardware are visualized in Fig. 8, *e.g.*, HSCoNet-Edge-A denotes the hardware-aware DNNs for the edge device, and vice versa. Note CB_3x3 and CBX_3x3

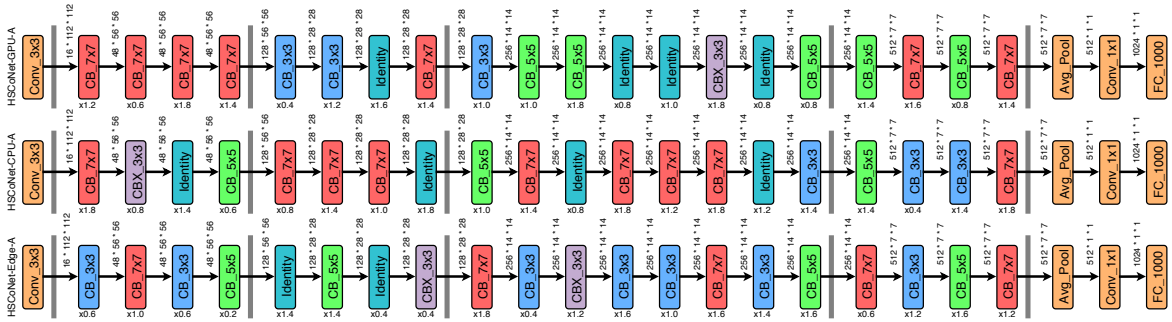

Fig. 8: Illustration of three neural architectures discovered by HSCoNAS on target hardware,*i.e.*, GPU, CPU, and edge device.

TABLE II: Comparisons with state-of-the-art approaches.

	Test Error (%)		Runtime Latency (ms)		
	Top-1	Top-5	GPU	CPU	Edge
Manually-Designed Models					
MobileNetV2 1.0× [13]	28.0	-	11.5	25.2	61.9
ShufflenetV2 1.5× [9]	27.4	-	10.5	34.3	65.9
MobileNetV3 (*large*) [5]	24.8	-	12.2	31.8	61.1
State-of-the-art NAS Models					
DARTS [7]	26.7	8.7	17.3	81.4	68.7
MnasNet-A1 [15]	24.8	7.5	10.9	26.4	51.8
FBNet-A [16]	27.0	9.1	10.5	21.6	48.6
FBNet-B [16]	25.9	8.2	13.6	25.5	57.1
FBNet-C [16]	25.1	7.7	15.5	28.7	66.4
ProxylessNAS-GPU [2]	24.9	7.5	12.0	24.5	57.4
ProxylessNAS-CPU [2]	24.7	-	16.1	29.6	70.1
ProxylessNAS-Mobile [2]	25.4	7.8	11.5	26.4	53.5
Hardware-Aware Models Discovered by HSCoNAS					
HSCoNet-GPU-A	**25.1**	7.7	**9.0**	26.5	43.4
HSCoNet-CPU-A	25.3	**7.6**	10.1	**22.8**	43.1
HSCoNet-Edge-A	25.7	8.1	9.9	25.8	**34.9**
HSCoNet-GPU-B	23.6	6.9	**12.0**	31.6	76.9
HSCoNet-CPU-B	**23.5**	**6.8**	13.4	**26.4**	69.1
HSCoNet-Edge-B	23.8	6.9	12.9	31.8	**52.7**

represent two types of building blocks in ShuffleNet [9] with a kernel size of 3×3, and vice verse. Besides, Identity is the skip-connection operation [2], [7]. In Fig. 8, the number under each operator, *e.g.*, $\times 1.4$, represents the channel scaling factor automatically searched by HSCoNAS for this operator, which is enabled by the proposed dynamic channel scaling.

Results and comparisons with recent state-of-the-art works are summarized in TABLE II. Notably, the HSCoNets outperform those manually-designed lightweight DNNs like ShuffleNetV2 [9] and MobileNetV2 [13] in terms of both accuracy and runtime latency on our three experimental hardware devices, respectively. HSCoNet-GPU-A obtains comparable accuracy as ProxylessNAS-GPU on ImageNet while being x1.3 faster on GPU. Besides, HSCoNet-GPU-B maintains similar runtime latency on GPU as ProxylessNAS-GPU but achieves +1.3% higher accuracy. Furthermore, HSCoNet-CPU-B yields the lowest top-1/5 error of 23.5%/6.8% among those state-of-the-art DNNs but with an inference speedup of x3.1 on CPU when compared with the hardware-agnostic NAS, *i.e.*, DARTS [7], which demonstrates the effectiveness of our hardware-software co-design paradigm. Moreover, HSCoNet-Edge-B reports a top-1/5 error of 23.8%/6.9% and the runtime latency of 52.7ms on the edge device, which is of +1.3%/+0.8% higher accuracy and x1.3 faster than FBNet-C [16].

Furthermore, we compare the runtime latency of searched HSCoNets on different hardware platforms, respectively. An interesting observation is that hardware-aware DNNs searched for the edge device do not necessarily run fast on GPU and CPU, and vice versa. This demonstrates that HSCoNets discovered by HSCoNAS are device-specific. Thus, it is essential to design specialized DNNs for different hardware architectures to achieve the best accuracy and runtime efficiency.

V. Conclusion

This paper presents HSCoNAS, a hardware-aware evolutionary algorithm (EA) based neural architecture search (NAS) framework. HSCoNAS is integrated with an effective hardware performance modeling method and two NAS improvements to automate the design of efficient deep neural networks (DNNs) upon target hardware. Extensive experimental results show HSCoNAS outperforms the state of the arts in terms of latency and accuracy. In future, we plan to extend HSCoNAS to different constraints like power consumption.

References

[1] B. Ahn and et al. Deeper weight pruning without accuracy loss in deep neural networks. In *DATE*, 2020.
[2] H. Cai and et al. Proxylessnas: Direct neural architecture search on target task and hardware. *ICLR*, 2019.
[3] R. Ding and et al. Flightnns: Lightweight quantized deep neural networks for fast and accurate inference. In *DAC*, 2019.
[4] C. Hao and et al. Nais: Neural architecture and implementation search and its applications in autonomous driving. *ICCAD*, 2019.
[5] A. Howard and et al. Searching for mobilenetv3. In *ICCV*, 2019.
[6] W. Jiang and et al. Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search. In *DAC*, 2019.
[7] H. Liu and et al. Darts: Differentiable architecture search. *ICLR*, 2019.
[8] Z. Liu and et al. Rethinking the value of network pruning. *ICLR*, 2019.
[9] N. Ma and et al. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *ECCV*, 2018.
[10] H. Pham and et al. Efficient neural architecture search via parameter sharing. *ICML*, 2018.
[11] I. Radosavovic and et al. On network design spaces for visual recognition. In *ICCV*, 2019.
[12] E. Real and et al. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.
[13] M. Sandler and et al. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, 2018.
[14] D. Stamoulis and et al. Hyperpower: Power-and memory-constrained hyper-parameter optimization for neural networks. In *DATE*, 2018.
[15] M. Tan and et al. Mnasnet: Platform-aware neural architecture search for mobile. In *CVPR*, 2019.
[16] B. Wu and et al. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *CVPR*, 2019.
[17] C. Wu and et al. Machine learning at facebook: Understanding inference at the edge. In *HPCA*, 2019.
[18] J. Yu and et al. Slimmable neural networks. *ICLR*, 2019.
[19] B. Zoph and et al. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.