

---

# COMBINING NEURAL ARCHITECTURE SEARCH AND AUTOMATIC CODE OPTIMIZATION: A SURVEY

---

**Inas Bachiri\***

Ecole nationale Supérieure d’Informatique  
Algiers, Algeria  
ji\_bachiri@esi.dz

**Hadjer Benmeziane**

IBM Research Europe  
Zurich, Switzerland  
Hadjer.Benmeziane1@ibm.com

**Riyadh Baghdadi**

New York University Abu Dhabi  
Abu Dhabi, UAE  
baghdadi@nyu.edu

**Smail Niar**

Université Polytechnique Hauts-de-France,  
LAMIH, INSA  
Valenciennes, France  
Smail.Niar@uphf.fr

**Hamza Ouarnoughi**

Université Polytechnique Hauts-de-France,  
LAMIH, INSA  
Valenciennes, France  
Hamza.Ouarnoughi@uphf.fr

**Abdelkrime Aries**

Ecole nationale Supérieure d’Informatique  
Algiers, Algeria  
ab\_aries@esi.dz

## ABSTRACT

Deep Learning (DL) models have experienced exponential growth in complexity and resource demands in recent years. Accelerating these models for efficient execution on resource-constrained devices has become more crucial than ever. Two notable techniques employed to achieve this goal are Hardware-aware Neural Architecture Search (HW-NAS) and Automatic Code Optimization (ACO). HW-NAS automatically designs accurate yet hardware-friendly Neural Networks (NNs), while ACO involves searching for the best compiler optimizations to apply on NNs for efficient mapping and inference on the target hardware. This survey explores recent works that combine these two techniques within a single framework. We present the fundamental principles of both domains and demonstrate their sub-optimality when performed independently. We then investigate their integration into a joint optimization process that we call **Hardware Aware-Neural Architecture and Compiler Optimizations co-Search (NACOS)**.

**Keywords** Efficient Neural Networks · Hardware-aware Neural Architecture Search · Automatic Code Optimization · Compiler Auto Scheduler

## 1 Introduction

In the quest for achieving state-of-the-art performance, DL researchers have been designing highly complex Neural Networks (NNs) that have revolutionized our lives. However, these progressive improvements come with significant increases in NNs’ size, complexity, energy consumption, and inference latency, making their deployment on resource-constrained devices challenging. Given the plethora of available hardware platforms, ranging from powerful clusters to tiny IoT devices, it is essential to design efficient and hardware-friendly NNs. This can be achieved by intervening at three levels: *model design, software stack, and hardware stack* [1]:

---

\*Contact author

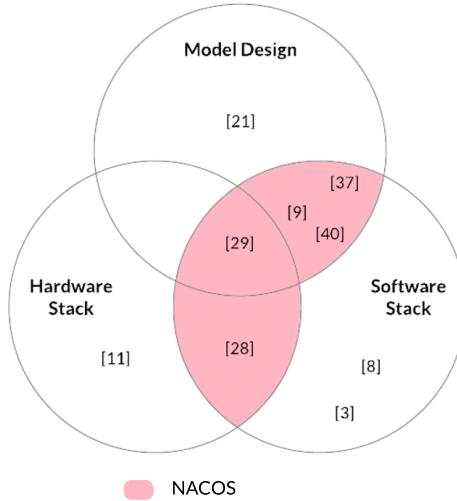


Figure 1: Cross-level joint deep learning optimization methods. In this paper, we explore works that combine HW-NAS and Automatic Code Optimization (NACOS)

1. Methods for improving **model design** include compression techniques like Quantization [2], learning techniques like Distillation [3] and optimization techniques, such as Hardware-aware Neural Architecture Search (HW-NAS). The latter is a multi-objective optimization process automating the design of NNs through maximization of accuracy and hardware efficiency metrics.
2. At the **software stack** level, we can either optimize the implementation provided by DL frameworks such as PyTorch [4], or the code optimization strategies employed by the compiler on the low-level code of NNs [5, 6].
3. The **hardware level** mainly includes designing specialized hardware and tuning accelerator configurations for training and running NN models inference [7].

Each of these techniques contributes in optimizing NNs. However, independent execution of these techniques leads to sub-optimal results. For example, designing a network using HW-NAS, and then applying quantization on top of it may lead to a drop in accuracy [2]. Therefore, combining techniques within and across the three design levels has been explored. For instance, at the model design level, NAS has been combined with Quantization [8]. Also, between the model design level and the hardware level, HW-NAS has been combined with the exploration of hardware accelerator designs on FPGA [9]. Finally, between the model design level and the software stack level, compression has been combined with compiler optimizations [10], illustrating the potential of synergies. Figure 1 summarizes some cross-level works that have been published.

A promising yet under-explored design methodology is to jointly explore HW-NAS and Automatic Code Optimization (ACO). For the sake of brevity, we give it the name of NACOS (Neural Architecture and Compiler Optimizations co-Search).

In their hardware efficiency evaluation, HW-NAS techniques consider the implementation provided by the DL framework as it is. This gives a rough approximation of the latency and energy consumption before applying code optimizations, resulting in the risk of overlooking optimal architectures during the search. On the other hand, ACO usually works on a given network with a fixed architecture, regardless of its performance and optimality. Combining HW-NAS and ACO would allow an accurate approximation of the hardware efficiency metrics in HW-NAS and an ACO that is adapted to the best network.

This survey aims to explore NACOS methods and provide insights into the state-of-the-art methodologies by highlighting the following attributes:

- We first cover the fundamental principles of Hardware-aware Neural Architecture Search (HW-NAS) and Automatic Code Optimization (ACO) in Section 2.
- In Section 3, we highlight the drawbacks of performing each technique independently and motivate their joint optimization.

- We propose a taxonomy to classify NACOS methods based on their key components in Section 4. In addition, we provide a detailed overview of existing works with a focus on their search space design.
- We then discuss the exploration algorithms and the candidate evaluation strategies employed in existing methods in Section 5.
- In the last section, we address the challenges and limitations of the existing works and propose some research directions to advance this area further.

## 2 Background

In this section, we provide a high-level overview of HW-NAS and ACO techniques.

### 2.1 Hardware-aware Neural Architecture Search

The end-to-end design process of NN architectures proves challenging when performed manually, as it requires time and expertise. This is why many researchers resort to Neural Architecture Search (NAS) [11], which is a set of techniques for automatically designing NN architectures. Formulated as an optimization problem, NAS searches for the best architecture within a set of NN candidates, referred to as the **search space**, using a **search algorithm** that optimizes one objective, such as accuracy, or multiple objectives, such as execution time, energy consumption, memory usage, etc. Due to the time-consuming training of every sampled architecture, an **evaluation strategy** is used to estimate the objective(s) with minimal or no training.

NAS has been successful in many tasks, including Image Classification [12], object detection [13], and keyword spotting [14]. However, following the DL trend, the found models require intensive compute power and memory resources due to their complex nature. This complexity also makes it hard to deploy them to hardware for efficient inference. This challenge has spurred the development of novel multi-objective NAS methods, known as Hardware-aware Neural Architecture Search (HW-NAS) [15]. HW-NAS is described with the same components as NAS. The search space can be tailored to the target hardware and task [16], and the search algorithm finds the Pareto front, which represents the best architectures in terms of trade-off between the considered objectives. These objectives include hardware metrics such as latency [17, 12], FLOPs [18, 19], energy consumption [20] and so on. The complexity of HW-NAS is hindered by the evaluation of these objectives, which requires estimation techniques such as using lookup tables [17], analytical approximations [21], and surrogate models [12]. For in-depth details about HW-NAS techniques, we refer readers to [22].

### 2.2 Automatic Code Optimization

Automatic code optimization (ACO), also known as auto-scheduling, refers to a set of techniques that are implemented at the compiler level with the aim of automating the optimization of high-level code. ACO is achieved by identifying the most effective schedule, which is essentially a sequence of selected code transformations such as parallelization, loop tiling, interchange, and vectorization, applied under a specific order with specific parameters. This schedule enhances the efficiency of the code’s implementation and hardware mapping without altering its semantics.

One example is the Halide Auto-scheduler [23], which automatically explores the best schedule for Halide programs. The search space includes various code transformations like parallelization, loop unrolling, interchange, fusion, and tiling. This exploration is performed using Beam Search, guided by a NN-based cost model predicting the speedup achievable with a particular schedule, given manually engineered features representing both the program and the schedule.

On the other hand, to automate finding the optimal sequence of transformations to apply on a Tiramisu [24] program, a DL-based auto-scheduler [5] has been implemented. The search process is conducted using both Beam Search and Monte Carlo Tree Search, leveraging a cost model as a speedup estimator to navigate the space. The cost model predicts the speedup that we’d get from applying a schedule on the program, given a set of automatically-extracted features.

Another aspect of auto-scheduling involves searching for the best parameter configuration of a given schedule. Code transformations typically involve one or more parameters that need tuning. Choosing appropriate parameters is crucial to effectively apply the transformation and achieve desired results. An example is AutoTVM [6]—a compiler infrastructure tool implemented for the TVM [25] compiler that uses simulated annealing to search the parameter space, and leverages a NN model to rank program versions with different transformation parameters based on high-level computation graph abstractions, which significantly optimizes tensor programs.

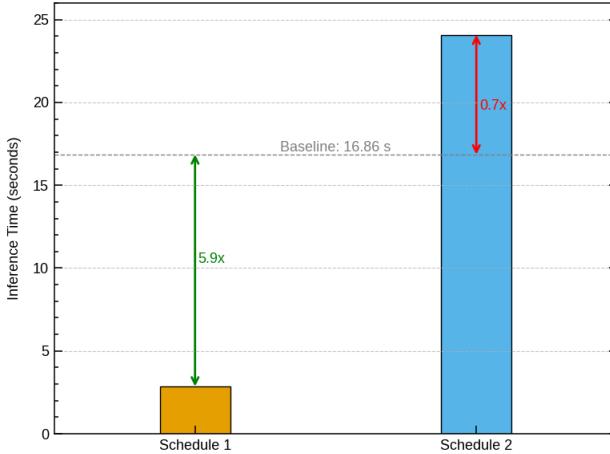


Figure 2: The inference latency of VGG16 using two different compiler schedules, on 100 image samples from ImageNet [27]. Schedule 1 and Schedule 2 are obtained by applying different sequences of parallelization, loop tiling, and fusion with various parameters using MLIR [28], and executing on an Intel Core i7 processor with 32 GB of RAM. The two schedules perform differently on the network; Schedule 1 outperforms Schedule 2 by making the inference on VGG16 faster. The values on the arrow lines represent the acceleration relative to the baseline time, with green indicating acceleration and red indicating deceleration. The baseline time represents the original inference time of VGG16 before applying the schedules.

### 3 Motivation

Given a hardware platform with specific efficiency constraints like latency or memory usage specifications, a classical HW-NAS framework will search for the best possible NN architecture that satisfies these constraints. This involves evaluating candidate NN architectures under a fixed scheduling strategy from a deep learning library (e.g., TensorFlow or PyTorch). However, the discovered architecture may not be optimal for the target hardware when executed with a certain scheduling strategy.

As illustrated in Figure 2, we observe variations in inference latency values for the same NN when different schedules are applied. For instance, applying two different schedules to VGG16 [26] results in largely distinct latency values. This highlights the significance of the schedule choice in latency estimation, and the risk for the HW-NAS process to overlook optimal architectures if it does not consider compiler optimizations. For example, if we were constrained by a maximum of 10s latency, and the fixed compiler schedule that we are using is schedule 2, the VGG16 architecture would be considered to not meet the constraint and will be overlooked, even though it could fit in the constraint really well if schedule 1 was used.

On the other hand, a scheduling strategy that optimizes performance for one NN may not yield similar benefits for others. In Figure 3, Schedule 1 significantly reduces the inference latency of VGG16 [26], achieving a speed-up of approximately 5.9 times compared to the original execution latency. However, this same schedule does not universally optimize the performance across all networks. For networks like ResNet18, ResNet34, MobileNet [29], and MNASNet [18], it results in inference latencies that are comparable to or worse than their original latencies. Similarly, Schedule 2 shows a noticeable speed-up for ResNet18 [30] and ResNet34. Yet, for other networks, it may increase the execution latency slightly. This shows that searching for a good schedule, whether manually or through ACO, can be sub-optimal if the NN design is not considered in the process.

Methods such as NACOS are thus primordial to explore both search spaces and come up with a design of NN architectures and their associated schedules for optimal performance and hardware efficiency on a given task.

### 4 Taxonomy

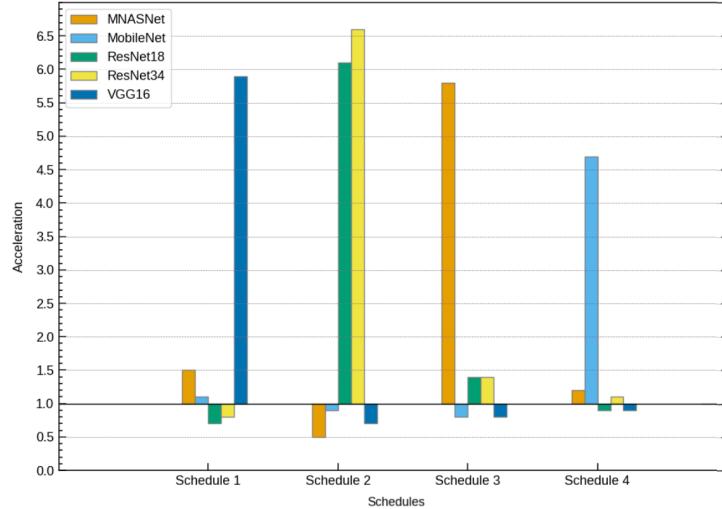


Figure 3: The accelerations of various neural networks under different scheduling strategies. Schedule 1 significantly speeds up VGG16, but does not improve and even worsens performance for other networks. Similarly, Schedule 2 optimizes ResNet18 and ResNet34 but slightly increases VGG16’s latency. This demonstrates that a well-optimized schedule for one neural network does not necessarily optimize other networks.

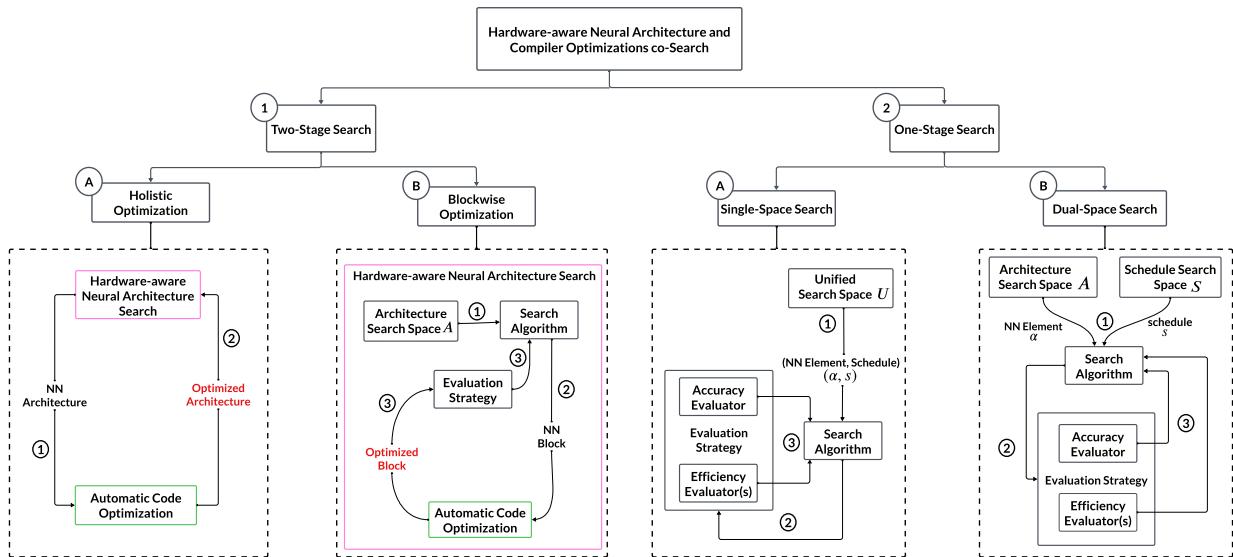


Figure 4: Hardware Aware Neural Architecture and Compiler Optimizations co-Search (NACOS) Taxonomy

In this section, we propose a taxonomy to categorize NACOS techniques based on their search flow, providing a structured foundation for future work. Our taxonomy is summarized in Figure 4. At a high level, we categorize NACOS methods into two classes: two-stage search methods and one-stage search methods. The first treats HW-NAS and ACO as separate search processes, while the second treats them as a single one. We describe these two classes of methods in the rest of this section. In addition, an overview of existing NACOS methods is presented in Table 1.

#### 4.1 Two-stage Search

This class treats HW-NAS and ACO as separate stages, engaging in two distinct search processes that interactively contribute to the final outcome. For each sampled NN architecture candidate in the HW-NAS stage, the ACO optimizes its code and provides the HW-NAS with precise efficiency metrics. Equation 1 shows the optimization problem solved

by the HW-NAS stage. In this equation,  $\alpha$  represents a candidate architecture from the architecture search space denoted as  $A$ . The search seeks to minimize the loss function of the neural network denoted as  $\mathcal{L}$  and the cost of executing the neural network on hardware denoted as  $\mathcal{C}$ . Several metrics, such as latency, energy consumption, or memory footprint, can be combined in the evaluation of  $\mathcal{C}$ .

$$\min_{\alpha \in A} \mathcal{L}(\alpha), \mathcal{C}(\alpha) \quad (1)$$

In the ACO stage, the search for the best transformation sequence applied to  $\alpha$  is formulated in Equation 2. Here,  $s$  represents a schedule (a sequence of code optimizations) in the schedules space, denoted as  $S$ . The problem is then cast as a constrained optimization one. We seek to minimize the cost of executing the candidate architecture  $\alpha$  on the hardware when  $s$  is applied,  $\mathcal{C}(s(\alpha))$ , under the constraint of keeping the same loss or reducing it.

$$\begin{aligned} & \min_{s \in S} \mathcal{C}(s(\alpha)) \\ & \text{subject to } \mathcal{L}(s(\alpha)) \leq \mathcal{L}(\alpha) \end{aligned} \quad (2)$$

Depending on the nature of the candidates given by the HW-NAS stage, i.e. whether they are blocks or entire architectures, techniques within this category can be classified into:

#### 4.1.1 Holistic Optimization

The NAS stage within this category first explores the associated architecture space and discovers an entire NN architecture  $\alpha$ . The resulting architecture  $\alpha$  is then handed over to the ACO stage, which optimizes its code, allowing the expansion of the search space to accommodate larger models with greater accuracy. Optimizing the NN and handing it back to HW-NAS allows it to find a more accurate model while still satisfying the efficiency constraints, making full use of the target hardware's resources. The process is illustrated in Figure 4(1.A).

In this category, we find MCUNet [31], a model-software co-design framework for microcontrollers (MCUs). This framework jointly designs an efficient NN and its corresponding inference schedule to fit the tight memory resources on MCUs. This is achieved through two interactive components : TinyNAS and TinyEngine. TinyNAS is a HW-NAS module that first prunes the architecture search space, then performs one-shot search (refer to Section 5) in the optimized space for the best architecture that satisfies memory and latency constraints using the evolutionary algorithm. The selected architecture by TinyNAS is handed to TinyEngine, a memory-efficient inference library that eliminates the unnecessary memory overhead and performs ACO. TinyEngine optimizes the whole architecture and adapts its schedule to make full use of the limited resources in the MCUs, which will give TinyNAS more space to search for a larger, better performing architecture.

#### 4.1.2 Blockwise Optimization

In this category of approaches, the ACO stage searches for the best compiler schedule for each building block of the NN architecture independently during the search process of NAS, instead of the entire architecture altogether. As shown in Figure 4 (1.B), at each iteration of the joint search process, the NN block proposed by HW-NAS is handed to the ACO module for optimization. The optimized neural network blocks are subsequently used by HW-NAS to construct the final architecture.

An example of this category of techniques is CHANAS [32], a framework that co-searches for an NN architecture and a corresponding compiler scheduling policy that maps the model onto the target hardware. To do this, it constructs a super-network [33] to represent the architecture search space by stacking medium sized neural blocks, constituted of elastic MBConv cells. Such cells consist of a  $1 \times 1$  convolution, a  $k \times k$  depthwise convolution, a Squeeze and Excitation [34] block and another  $1 \times 1$  convolution. All candidate NN architectures can be extracted from this super-network, allowing for quick accuracy evaluation through weight sharing [35] (refer to Section 5). The framework follows a hierarchical exploration process centered around neural blocks, involving two coordinated search phases. First, the compiler scheduler goes through the super-network, transforms each block into a computational sub-graph and optimizes it for the target hardware. The optimization is done at two levels: graph-level optimization, where the block is optimized using layer fusion and data layout transformations, and then using lower-level optimizations where the best schedule for each operator is independently searched using an evolutionary-like heuristic with a Gaussian Process-based cost model. Then, the HW-NAS component explores different sequences of the neural blocks (sub-networks of the super-network) that have already been optimized by ACO. For this, the search space is automatically pruned according to the performance constraints. After that, an evolutionary algorithm is used to search for the solution in the reduced space, guided by a NN accuracy predictor and a latency lookup table.

## 4.2 One-stage Search

In this category, neural architectures and compiler schedules are explored using one single search algorithm that simultaneously searches for both of them in each iteration. Equation 3 illustrates how the optimization process is cast. The exploration can take place in a unified search space, denoted as  $U$ , or in two distinct spaces  $A$  and  $S$

$$\min_{\substack{(\alpha, s) \in U \\ \text{or} \\ (\alpha, s) \in (A, S)}} \mathcal{L}(s(\alpha)), \mathcal{C}(s(\alpha)) \quad (3)$$

Based on that, we can further classify techniques within this category into:

### 4.2.1 Single-Space Search

In this sub-category, both NN candidates and compiler optimization possibilities share a unified search space, employing the same representation for both types of searched elements. In the unified search space, for each NN operator, a list of possible compiler transformations is associated. The transformations can be architecture-related and changes the hyper-parameters such as number of groups, dilation rate in a convolution 2D, or schedule-related such as loop tiling or unrolling. The search process, as indicated in Figure 4 (2.A), iteratively samples candidate couples from this unified search space and evaluates them for accuracy and efficiency.

For instance, in [36], NN operators are represented in the same way as code transformations, i.e. as nested loops. These transformations are represented in the polyhedral model [37] and implemented in the TVM compiler [25]. The method replaces the convolution operators by trying out different sequences of transformations present in the search space, leveraging their inherent nested loop nature. The goal is to maintain comparable accuracy while reducing latency. AutoTVM [6], the ACO module of TVM, is employed to perform auto-parameter-tuning on these transformations. To reduce the number of candidates, the search space is pruned by only keeping candidates that pass the Fisher Potential [38] test. Among the retained candidates, the best performing one is selected after the evaluation process, which is done by execution. Fisher Potential serves as a cost-effective zero-shot metric that rejects damaging network changes and transformations without training them. It is noteworthy, though, that despite its computational efficiency, this metric is proved to be sub-optimal [39].

### 4.2.2 Dual-Space Search

In this sub-category NN architectures and compiler schedule candidates, while part of the same search process, are defined in their distinct search spaces,  $A$  and  $S$  respectively. As illustrated in Figure 5 (2.B), a (architecture operator, schedule) tuple candidate is sampled for accuracy and efficiency evaluation at each iteration. The search process is hindered by the large combined search spaces.

A notable work within this sub-category is NAAS [40]. NAAS proposes an evolution-based framework that combines the architecture search process with compiler-level optimizations and hardware accelerator design. An accelerator is a hardware design specifically tailored to speed up certain computational tasks, such as convolution operations in NNs. The considered compiler transformations in this framework are Loop Ordering and Loop Tiling, which are used to optimize each convolution layer independently during the search. Based on these transformations, candidate optimizations proposed by the compiler are encoded as vectors assigning an order and a tile size to each convolution dimension (e.g., spatial width, kernel height, input channel). Accelerator design elements and compiler optimizations are integrated with the search space of the HW-NAS method called Once-For-All (OFA) [41], using the evolutionary algorithm.

First, a pool of accelerator candidates is generated, and for each accelerator candidate, a network candidate that satisfies the pre-defined accuracy requirement is sampled from the OFA space. Then, for each (architecture, accelerator) pair, the algorithm searches for the optimal corresponding compiler optimization strategy. In order to take both latency and energy into consideration during the search, the Energy-Delay Product (EDP) [42] is used as a minimization objective. The framework outputs a tuple of neural architecture, compiler schedule, and accelerator architecture. The accelerator architecture refers to the specific design and configuration of the hardware accelerator, which is optimized to maximize efficiency and performance.

## 5 Search Strategies and Evaluation

Table 1: Summary of existing NACOS methods and their characteristics

Method	Type	Target Hardware	Target NN	Objectives	Target Task	Open-Source	Compiler
[31]	1.A	MCU	CNN	Latency, memory & storage usage, accuracy	Image Classification, Speech Recognition	Yes	TinyEngine
[36]	2.A	mCPU, mGPU, CPU, GPU	CNN	latency	Image Classification	Yes	TVM
[43]	1.B	mGPU, mDSP	SRCCNN	Latency, PSNR	Super Resolution	Yes	XGen
[32]	1.B	Mobile, CPU, GPU	CNN	Accuracy, latency	Image Classification	No	TVM
[40]	2.B	NVDLA, Shidiannao, Eyeriss, Edge TPU	CNN	accuracy, EDP	Image Classification	No	MAESTRO

Given the large size of search spaces in NACOS, in terms of NN architectures and compiler schedules, it is important to use well-tailored exploration algorithms. Depending on the method, we can opt for a unified exploration algorithm to search for both NN design units (operators, blocks or architectures) and compiler schedules simultaneously, or use distinct algorithms for each.

The most used strategy in this context is the Evolutionary Algorithm [44]. It can be used for both One-stage and Two-Stage search techniques. In NAAS [40], it is used to sample (architecture, schedule) tuples for each explored hardware configuration. It can also be used for independently performing HW-NAS like in MCUNet [31], or for applying ACO like in CHANAS [32].

Continuous optimization through gradient descent has also been used in [43] by making the searched NN hyper-parameters, such as the depth of the network and the per-layer width, differentiable.

A notable observation in two-stage search methods is that, it is common for the search strategy in the ACO stage to be rule-based. For example, in [43], rules have been developed to perform different compiler optimizations, like fusing convolution operations with depth-to-space operations.

It is worth mentioning that some methods do not prioritize working on the search algorithm, but rather focus on designing and pruning the search space. That's the case for [36], where the search strategy is just enumerating random sequences of candidates from the search space and evaluating them.

Even with a well-crafted search space and an efficient exploration algorithm, it is unrealistic to measure the actual accuracy and hardware efficiency metrics during the search. Therefore, using performance and efficiency estimators becomes necessary to avoid training NN candidates and executing them with sampled schedules at each search iteration.

For accuracy, the dominating estimation strategy in NACOS methods [31, 32, 40] is One-shot evaluation, also referred to as weight sharing. It is often used when the architecture space  $A$  is a super-network, and all possible candidate architectures are sub-networks of that super-network. In this method, the super-network is trained, then the sampled architectures, i.e sub-networks, are evaluated by inheriting the trained super-network weights, without having to train them separately. It is not the only one, though. In CHANAS [32], a lookup table is used in the HW-NAS stage and a Gaussian process-based cost model is used in the ACO stage. In [43], a NN speed model is used. Meanwhile, latency is measured through execution in [31] and [36]. Memory footprint in MCUNet [31] is also measured through execution.

## 6 Challenges & Limitations

In this section, we lay down the challenges and limitations present in current NACOS methods. We believe that addressing these issues is necessary for the progression of this research area, paving the way to unlock its full potential in generating optimal, efficient and hardware-friendly NN architectures.

### 6.1 Hardware heterogeneity

As shown in Table 1, existing NACOS methods target a variety of hardware platforms, spanning CPUs, GPUs, DSPs, mobile phones, NN accelerators, and even MCUs. While this versatility allows for the generation of NN models that can run on a large set of hardware platforms, from the most powerful hardware to the smallest IoT devices, it also exhibits several challenges.

A primary concern is that these methods are limited by their use of compilers that are specific to the targeted hardware platforms. For example, in the case of MCUNet, TinyEngine is exclusively compatible with MCUs. This restricts the applicability of each method in terms of the used configuration specifications and hardware platforms.

With the evolving set of hardware and the growing need for neural network transferability across different devices, addressing this challenge of hardware heterogeneity becomes imperative for the advancement of this research area. One promising approach, is promoted by the development of the MLIR [28] infrastructure. Indeed, MLIR [28] strives to support any high-level Domain Specific Language or DL framework and compatible with different hardware devices. This will ensure that the co-searched neural network and compiler schedule pairs become usable and applicable on a wide range of off-the-shelf hardware.

### 6.2 Generalization

All of the existing NACOS methods focus on generating and optimizing Convolutional Neural Network (CNN) architectures. These methods address architectures from both groups of CNNs [22] : standard CNNs that use standard

convolutions, and extended CNNs that incorporate special convolution variants such as depthwise and grouped convolutions. Some methods even extend their scope to specific types of CNNs, such as Super Resolution (SR) CNNs, which are designed for super-resolution tasks and feature the same convolution blocks arranged in a different pyramidal structure [43].

The inclination towards targeting CNN architectures can be attributed to the loop-nest structure and affine nature of convolution operations [36], which makes searching for compiler schedules and applying code optimizations on them easier and more natural. This is mainly due to the availability of well-defined loop representations in compiler Intermediate Representations (IRs) and the plethora of modules that optimize loop nests for different compilers (e.g. [24]).

However, several NAS works have emerged to explore the search space for capsule networks [21], transformers [45], and GANs [46]. Despite these strides in generalization, none of the currently available NAS techniques designed for tasks other than Computer Vision incorporate compiler optimizations within their process. This motivates interesting future works in which compiler optimizations can improve hardware efficiency across various domains. This exploration could potentially enable specialized language models to run efficiently on hardware-constrained devices

### 6.3 Large search space exploration

One of the most significant challenges in NACOS methods is the exploration of large search spaces. With standard NAS strategies, the search process requires substantial computational resources. This limiting factor forces methodologies to stick to tiny machine learning tasks such as image classification (IC) with small datasets like CIFAR-10.

The scope of NACOS can significantly be increased with pre-search strategies to reduce the search space size. In MCUNet [31], authors employ a search-space pruning heuristic to reduce its size, based on the assumption that a search space accommodating higher FLOPs under the specified memory constraint can produce better models.

Besides, the unified search space requires careful definition. The application of different transformation and order on the operator needs heuristically based strategies. Over-aggressive loop unrolling or inappropriate tiling can lead to increased code size, cache misses, or even degraded performance. Strategies that balance these aspects use insights from ACO methods [6].

### 6.4 Lack of benchmarks

In the context of NAS, datasets containing architectures along with their corresponding accuracy and hardware metrics, are known as NAS Benchmarks [47, 48, 49]. They are used to compare the performance of various NAS algorithms while serving as search spaces.

A significant challenge in NACOS is the absence of such benchmarks to reproduce and compare existing methods. This is mainly due to their nature, as they focus more on designing the search space rather than working on the search strategy. There are multiple ways in which the joint space can be designed and explored by the search algorithm, forming the basis of the entire method. This diversity makes it difficult to standardize the search spaces. Additionally, as shown in Table 1, these techniques target different hardware platforms and employ various compilers to perform compiler scheduling. Creating a benchmark would thus require measuring the performance of models across numerous hardware platforms and using a variety of compilers. The dataset would also need to include (architecture, schedule) pairs, which further complicate this task because of the infinite possibilities of applicable compiler schedules.

## 7 Conclusion

This survey explores the combination of two efficient deep learning techniques: Hardware-Aware Neural Architecture Search (HW-NAS) and Automatic Code Optimization (ACO). The combination aims to alleviate the sub-optimality issues observed when these techniques are performed independently. We conduct a literature review and identify key challenges and limitations posed in existing works. Additionally, we propose potential solutions and improvements to further advance this research area. Unlocking the full potential of this area holds promise for enhancing not only neural network design but also the underlying infrastructure with minimal human intervention.

## References

- [1] Gaurav Menghani. Efficient deep learning: A survey on making deep learning models smaller, faster, and better. *ACM Comput. Surv.*, 55(12), mar 2023. ISSN 0360-0300. doi:10.1145/3578938. URL <https://doi.org/10.1145/3578938>.
- [2] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.
- [3] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.
- [4] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. URL <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [5] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. A deep learning based cost model for automatic code optimization. *Proceedings of Machine Learning and Systems*, 3:181–193, 2021.
- [6] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS’18, page 3393–3404, Red Hook, NY, USA, 2018. Curran Associates Inc.
- [7] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights. *Proceedings of the IEEE*, 109(10):1706–1752, 2021. doi:10.1109/JPROC.2021.3098483.
- [8] Yukang Chen, Gaofeng Meng, Qian Zhang, Xinbang Zhang, Liangchen Song, Shiming Xiang, and Chunhong Pan. Joint neural architecture search and quantization. *arXiv preprint arXiv:1811.09426*, 2018.
- [9] Weiwen Jiang, Lei Yang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Shouzhen Gu, Sakyasingha Dasgupta, Yiyu Shi, and Jingtong Hu. Hardware/software co-exploration of neural architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4805–4815, 2020. doi:10.1109/TCAD.2020.2986127.
- [10] Hui Guan, Shaoshan Liu, Xiaolong Ma, Wei Niu, Bin Ren, Xipeng Shen, Yanzhi Wang, and Pu Zhao. Cocopie: enabling real-time ai on off-the-shelf mobile devices via compression-compilation co-design. *Commun. ACM*, 64(6):62–68, may 2021. ISSN 0001-0782. doi:10.1145/3418297. URL <https://doi.org/10.1145/3418297>.
- [11] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: a survey. *J. Mach. Learn. Res.*, 20(1):1997–2017, jan 2019. ISSN 1532-4435.
- [12] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *International Conference on Learning Representations*, 2019. URL <https://arxiv.org/pdf/1812.00332.pdf>.
- [13] G. Ghiasi, T. Lin, and Q. V. Le. Nas-fpn: Learning scalable feature pyramid architecture for object detection. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7029–7038, Los Alamitos, CA, USA, jun 2019. IEEE Computer Society. doi:10.1109/CVPR.2019.00720. URL <https://doi.ieeecomputersociety.org/10.1109/CVPR.2019.00720>.
- [14] Bo Zhang, Wenfeng Li, Qingyuan Li, Weiji Zhuang, Xiangxiang Chu, and Yujun Wang. Autokws: Keyword spotting with differentiable architecture search. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2830–2834, 2021. doi:10.1109/ICASSP39728.2021.9414848.
- [15] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. A comprehensive survey on hardware-aware neural architecture search. 2021. URL <https://arxiv.org/abs/2101.09336>.
- [16] Li Lyina Zhang, Yuqing Yang, Yuhang Jiang, Wenwu Zhu, and Yunxin Liu. Fast hardware-aware neural architecture search. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 692–693, 2020.
- [17] B. Wu, K. Keutzer, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, and Y. Jia. Fbnets: Hardware-aware efficient convnet design via differentiable neural architecture search. In *2019 IEEE/CVF Conference on Computer*

- Vision and Pattern Recognition (CVPR)*, pages 10726–10734, Los Alamitos, CA, USA, jun 2019. IEEE Computer Society. doi:10.1109/CVPR.2019.01099. URL <https://doi.ieee.org/10.1109/CVPR.2019.01099>.
- [18] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2815–2823, 2019. doi:10.1109/CVPR.2019.00293.
  - [19] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *ArXiv*, abs/1905.11946, 2019. URL <https://api.semanticscholar.org/CorpusID:167217261>.
  - [20] Chengyue Gong, Zixuan Jiang, Dilin Wang, Yibo Lin, Qiang Liu, and David Z. Pan. Mixed precision neural architecture search for energy efficient deep learning. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–7, 2019. doi:10.1109/ICCAD45719.2019.8942147.
  - [21] Alberto Marchisio, Andrea Massa, Vojtech Mrazek, Beatrice Bussolino, Maurizio Martina, and Muhammad Shafique. Nascaps: a framework for neural architecture search to optimize the accuracy and hardware efficiency of convolutional capsule networks. In *Proceedings of the 39th International Conference on Computer-Aided Design, ICCAD ’20*, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380263. doi:10.1145/3400302.3415731. URL <https://doi.org/10.1145/3400302.3415731>.
  - [22] Hadjer Benmeziane, Kaoutar El Maghraoui, Hamza Ouarnoughi, Smail Niar, Martin Wistuba, and Naigang Wang. Hardware-aware neural architecture search: Survey and taxonomy. In *IJCAI*, pages 4322–4329, 2021.
  - [23] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédéric Durand, and Jonathan Ragan-Kelley. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4), jul 2019. ISSN 0730-0301. doi:10.1145/3306346.3322967. URL <https://doi.org/10.1145/3306346.3322967>.
  - [24] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019*, page 193–205. IEEE Press, 2019. ISBN 9781728114361.
  - [25] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI’18*, page 579–594, USA, 2018. USENIX Association. ISBN 9781931971478.
  - [26] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, Conference Track Proceedings*, 2015. URL <http://arxiv.org/abs/1409.1556>.
  - [27] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi:10.1109/CVPR.2009.5206848.
  - [28] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization, CGO ’21*, page 2–14. IEEE Press, 2021. ISBN 9781728186139. doi:10.1109/CGO51591.2021.9370308. URL <https://doi.org/10.1109/CGO51591.2021.9370308>.
  - [29] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
  - [30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi:10.1109/CVPR.2016.90.
  - [31] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. Mcunet: Tiny deep learning on iot devices. In *Proceedings of the 34th International Conference on Neural Information Processing Systems, NIPS’20*, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.
  - [32] Weiwei Chen, Ying Wang, Ying Xu, Chengsi Gao, Cheng Liu, and Lei Zhang. A framework for neural network architecture and compile co-optimization. *ACM Trans. Embed. Comput. Syst.*, 22(1), oct 2022. ISSN 1539-9087. doi:10.1145/3533251. URL <https://doi.org/10.1145/3533251>.

- [33] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. Efficient neural architecture search via parameters sharing. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4095–4104. PMLR, 7 2018. URL <https://proceedings.mlr.press/v80/pham18a.html>.
- [34] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018. doi:10.1109/CVPR.2018.00745.
- [35] Lingxi Xie, Xin Chen, Kaifeng Bi, Longhui Wei, Yuhui Xu, Zhengsu Chen, Lanfei Wang, An Xiao, Jianlong Chang, Xiaopeng Zhang, and Qi Tian. Weight-sharing neural architecture search: A battle to shrink the optimization gap, 2020.
- [36] Jack Turner, Elliot J. Crowley, and Michael F. P. O’Boyle. Neural architecture search as program transformation exploration. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’21, page 915–927, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi:10.1145/3445814.3446753. URL <https://doi.org/10.1145/3445814.3446753>.
- [37] Richard M Karp, Raymond E Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967. URL <http://dl.acm.org/citation.cfm?id=321418>.
- [38] Jack Turner, Elliot J. Crowley, Michael O’Boyle, Amos Storkey, and Gavin Gray. Blockswap: Fisher-guided block substitution for network compression on a budget. In *International Conference on Learning Representations*, 2020. URL <https://openreview.net/forum?id=Sk1kDkSFPB>.
- [39] Colin White, Arber Zela, Robin Ru, Yang Liu, and Frank Hutter. How powerful are performance predictors in neural architecture search? *Advances in Neural Information Processing Systems*, 34:28454–28469, 2021.
- [40] Yujun Lin, Mengtian Yang, and Song Han. Naas: Neural accelerator architecture search. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1051–1056. IEEE, 2021.
- [41] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once for all: Train one network and specialize it for efficient deployment. In *International Conference on Learning Representations*, 2020. URL <https://arxiv.org/pdf/1908.09791.pdf>.
- [42] Mark Horowitz, Thomas Indermaur, and Ricardo Gonzalez. Low-power digital design. In *Proceedings of 1994 IEEE symposium on low power electronics*, pages 8–11. IEEE, 1994.
- [43] Yushu Wu, Yifan Gong, Pu Zhao, Yanyu Li, Zheng Zhan, Wei Niu, Hao Tang, Minghai Qin, Bin Ren, and Yanzhi Wang. Compiler-aware neural architecture search for on-mobile real-time super-resolution. In *Computer Vision – ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XIX*, page 92–111, Berlin, Heidelberg, 2022. Springer-Verlag. ISBN 978-3-031-19799-4. doi:10.1007/978-3-031-19800-7\_6. URL [https://doi.org/10.1007/978-3-031-19800-7\\_6](https://doi.org/10.1007/978-3-031-19800-7_6).
- [44] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0262082136.
- [45] Nikita Klyuchnikov, Ilya Trofimov, Ekaterina Artemova, Mikhail Salnikov, Maxim Fedorov, Alexander Filippov, and Evgeny Burnaev. Nas-bench-nlp: neural architecture search benchmark for natural language processing. *IEEE Access*, 10:45736–45747, 2022.
- [46] Chen Gao, Yunpeng Chen, Si Liu, Zhenxiong Tan, and Shuicheng Yan. Adversarialnas: Adversarial neural architecture search for gans. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.
- [47] Chris Ying, Aaron Klein, Eric Christiansen, Esteban Real, Kevin Murphy, and Frank Hutter. NAS-bench-101: Towards reproducible neural architecture search. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 7105–7114. PMLR, 09–15 Jun 2019. URL <https://proceedings.mlr.press/v97/ying19a.html>.
- [48] Xuanyi Dong and Yi Yang. Nas-bench-201: Extending the scope of reproducible neural architecture search. *arXiv preprint arXiv:2001.00326*, 2020.
- [49] Chaojian Li, Zhongzhi Yu, Yonggan Fu, Yongan Zhang, Yang Zhao, Haoran You, Qixuan Yu, Yue Wang, and Yingyan (Celine) Lin. {HW}-{nas}-bench: Hardware-aware neural architecture search benchmark. In *International Conference on Learning Representations*, 2021. URL [https://openreview.net/forum?id=\\_0kaDkv3dVf](https://openreview.net/forum?id=_0kaDkv3dVf).