NN-Stretch: Automatic Neural Network Branching for Parallel Inference on Heterogeneous Multi-Processors

Jianyu Wei*
noob@mail.ustc.edu.cn
University of Science and Technology of China
Microsoft Research

Ting Cao†
ting.cao@microsoft.com
Microsoft Research

Shijie Cao
shijiecao@microsoft.com
Microsoft Research

Shiqi Jiang
shijiang@microsoft.com
Microsoft Research

Shaowei Fu
fushw@mail.ustc.edu.cn
University of Science and Technology of China

Mao Yang
maoyang@microsoft.com
Microsoft Research

Yanyong Zhang
yanyongz@ustc.edu.cn
University of Science and Technology of China

Yunxin Liu†
liuyunxin@air.tsinghua.edu.cn
Institute for AI Industry Research (AIR), Tsinghua University
Shanghai Artificial Intelligence Laboratory

ABSTRACT

Mobile devices are increasingly equipped with heterogeneous multi-processors, e.g., CPU + GPU + DSP. Yet existing Neural Network (NN) inference fails to fully utilize the computing power of the heterogeneous multi-processors due to the sequential structures of NN models. Towards this end, this paper proposes *NN-Stretch*, a new model adaption strategy, as well as the supporting system. It automatically branches a given model according to the processor architecture characteristics. Compared to other popular model adaption techniques such as model pruning that often sacrifices accuracy, NN-Stretch accelerates inference while preserving accuracy.

The key idea of NN-Stretch is to *horizontally stretch* a model structure, from a long and narrow model to a short and wide one with multiple branches. We formulate the model branching into an optimization problem. NN-Stretch attempts to narrow down the design space by taking into account the hard latency constraints through varying where the branches converge and how each branch is scaled to fit heterogeneous processors, as well as the soft accuracy constraints through maintaining the model skeleton and expressiveness of each branch. According to the constraints, NN-Stretch can efficiently generate accurate and efficient multi-branch models. To facilitate easy deployment, this paper also devises a *subgraph-based spatial scheduler* for existing inference frameworks to parallelly execute the multi-branch models. Our experimental results are very promising, with up to 3.85× speedup compared to single CPU/GPU/DSP execution and up to 0.8% accuracy improvement.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing**; • **Computing methodologies → Concurrent algorithms**.

KEYWORDS

Neural networks, Heterogeneous processors, Mobile devices, Multiple branch, Model parallelism

ACM Reference Format:
Jianyu Wei, Ting Cao, Shijie Cao, Shiqi Jiang, Shaowei Fu, Mao Yang, Yanyong Zhang, and Yunxin Liu. 2023. *NN-Stretch*: Automatic Neural Network Branching for Parallel Inference on Heterogeneous Multi-Processors. In *ACM International Conference on Mobile Systems, Applications, and Services (MobiSys '23), June 18–22, 2023, Helsinki, Finland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3581791.3596870

1 INTRODUCTION

Mobile SoCs (System on a Chip) are becoming increasingly heterogeneous to pursue higher performance within power constraints. The heterogeneous processors on a SoC, including the CPU, GPU and DSP, have a unified memory and deliver comparable performance [20]. Therefore, there is the opportunity for concurrent/-parallel heterogeneous computing to improve the NN (Neural Network) inference quality on mobile devices, in terms of both accuracy and efficiency.

*Work is done during internship at Microsoft Research.
†Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MobiSys '23, June 18–22, 2023, Helsinki, Finland
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0110-8/23/06…$15.00
https://doi.org/10.1145/3581791.3596870

Fundamentally speaking, parallel NN model execution relies on the parallelism of the model structure. For example, existing works [21, 42] often leverage the intra-operator parallelism of a model by partitioning an operator and running these partitions in parallel on different processors. Such a method is more suitable for heavy-computation operators; otherwise the communication cost between the processors (including synchronization, data mapping, data transformation, etc.) may easily overshadow the gain from parallel execution. However, heavy-computation operators are rare to find in mobile-side NN models. For example, our measurements show that the communication cost between the CPU and GPU is around 1 ms, but the latency of each MobileNetV1 [17] operator is normally <1 ms.

Inter-operator parallelism, on the other hand, could work for all operators. It concurrently executes operators that do not have data dependency on different processors, rather than partitioning a single operator. However, the challenge here is that the widely-used NN models such as ResNet [12] and EfficientNet [32] are all composed of sequential operators, i.e., a single branch, and thus cannot apply inter-operator parallelism. Additionally, even the multi-branch models can hardly benefit from concurrent execution on heterogeneous processors, as the computation loads on different branches are not balanced, such as the Inception series [34].

Meanwhile, many model adaption techniques [8, 14, 22, 47], such as model pruning, have been proposed to optimize the model structure to facilitate mobile-side deployment, but they often focus on inference acceleration on a single processor. None of these techniques has considered adapting models for heterogeneous computing so far. Besides, these methods often trade-off model accuracy for latency.

We thus raise the important research question: *can we automatically transform a given single-branch model to a balanced multi-branch structure* for efficient parallel execution on heterogeneous processors? Compared to pruning or distillation which sacrifices model capacity for latency reduction, such a model branching operation can accelerate inference with no loss on model capacity (i.e., accuracy).

To answer the above question, this paper proposes *NN-Stretch*, a novel model adaption strategy and the supporting system for model deployment on mobile/edge devices. To the best of our knowledge, this is the first model adaptation technique realizing automatic model branching for parallel execution on heterogeneous multiprocessors, i.e., *branch parallelism*, as shown in Fig. 1. In order to facilitate better model deployment, we argue that the proposed model adaption technique should have the following properties: (1) achieving latency reduction with no accuracy loss; (2) not requiring any additional efforts from model designers; and (3) incurring no more overhead than other model adaption techniques, such as pruning.

The key idea of NN-Stretch is to transform a given single-branch model that is usually long and narrow to a short and wide one with multiple branches through the horizontal stretch as shown in Fig. 1, with each branch extracting different groups of features. The effectiveness of this transformation is supported by the finding that scaling a model in depth or width within a certain degree can successfully preserve its accuracy as discussed in previous research [1, 6, 10, 25, 29].

The problems are then to determine (1) where to converge the branches, named as *meeting point*, to merge features extracted by each branch, and (2) how to scale each branch. We formulate this problem into an optimization problem. Similar as other model design space, the number of design options is too large to search efficiently. To decrease the number of design options, we engage the "hard" latency constraints (that are directly correlated with the above two factors), as well as the "soft" accuracy constraints (whose correlations with these two factors are more indirect and unpredictable). Only those design options that satisfy the constraints need to be considered.

More specifically, we adopt the following constraints. For latency, we have (1) cost-amortized meeting point identification, to amortize the processor communication cost with branch depth, and (2) heterogeneity-aware depth-width scaling, to scale down the depth (number of layers) or width (number of filters) on each branch to reduce the total computation cost and improve the utilization of heterogeneous processors. For accuracy, we have (1) structure-preserved meeting point identification, to keep the model skeleton by taking the model stages (i.e., the update of feature-map size) as meeting points, and (2) capacity-guaranteed depth-width scaling, to keep the expressiveness of each branch by setting the lower bounds for depth and width. The combination of these constraints rapidly shrinks the design options so that the multi-branch model can be quickly generated.

The main challenge in executing such multi-branch models stems from the fact that current inference frameworks sequentially run a model on one processor at a time. To address this challenge, we devise a *sub-graph-based spatial scheduler* to complement the available inference frameworks. It uses the concept of processor-level sub-graph (i.e., a sequence of operators, with only the first one having a dependency on other processors) as the scheduling unit to perform processor assignment. Each model branch is thus a sub-graph scheduled to run on a specific processor. The scheduler design also addresses the different processor communication mechanisms for correctness and efficiency, such as synchronous- or asynchronous-run of different processors to the host CPU.

In this work, NN-Stretch is evaluated for popular CNN models, including EfficientNet [32], RegNet [31], and ResNet [12], using ImageNet [9] datasets. The comparison baselines are TFLite [24] inference framework, and CoDL [19], the state-of-the-art parallel inference system on the CPU+ GPU processors. Our on-device evaluation runs on three different mobile phones with three different processors, including CPU, GPU and DSP. Results show that NN-Stretch can achieve up to 2.3×, 3.85× and 2× speedup on CPU+GPU+DSP compared to CPU-only, GPU-only and DSP-only, respectively. The model accuracy is preserved.

In fact, NN-Stretch is the first to enable flexible combination of available processors for inference, such as GPU+DSP and CPU+GPU +DSP. It gives more scheduling options based on the real device usage. NN-Stretch is orthogonal to other model deployment optimizations, such as quantization, model pruning, and operator kernel optimization. NN-Stretch is open-sourced[1].

The main contributions of this paper are as follows:

[1] https://github.com/caoting-dotcom/multiBranchModel

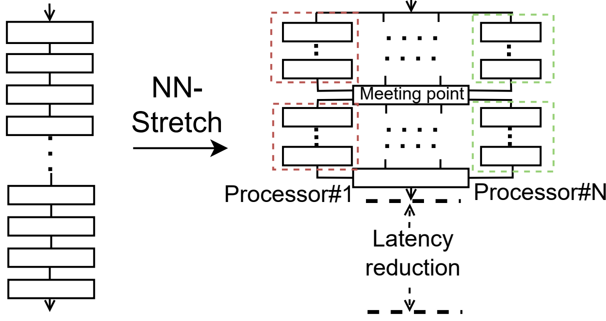


Figure 1: NN-Stretch transforms a long and lean model (usually with a single branch) into a short and wide one with multiple branches through horizontal stretching.

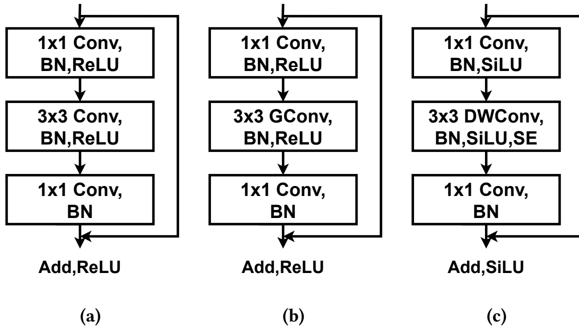


Figure 2: Representative building blocks from state-of-the-art NN models (a) ResNetV1, (b) RegNetX, and (c) EfficientNet, are sequentially connected.

- We propose a new model adaption strategy that can transform a single-branch model into a multi-branch one to enjoy the benefit of branch parallelism;
- We devise the meeting-point identification and depth-width scaling techniques to achieve effective model branching while considering both model and hardware characteristics;
- We design a sub-graph-based scheduler in inference frameworks to support the parallel inference on heterogeneous processors.
- We implement NN-Stretch system that demonstrates both latency reduction and accuracy gain.

2 BACKGROUND AND RELATED WORK

In this section, we describe the characteristics of mobile SoCs and typical NN model architectures, as well as the mismatch between the two. Namely, mobile SoCs are unique in that their CPU cores and GPUs have comparable performance and can execute different branches of a model concurrently. However, popular NN models are sequentially connected and cannot take advantage of this feature directly (Sec. 2.1). Most of the earlier model adaption efforts have assumed execution on a single processor, not on heterogeneous processors (Sec. 2.2). In addition, we also explain that existing multi-branch NN design cannot be used for general automatic model branching that suits our need (Sec. 2.3).

Table 1: Stages of RegNetX-4GF model.

Stage	Size of output feature map	Stride	# of repeated blocks
Stage 1	56×56	2	1
	56×56	1	1
Stage 2	28×28	2	1
	28×28	1	4
Stage 3	14×14	2	1
	14×14	1	13
Stage 4	7×7	2	1
	7×7	1	1

2.1 Mismatch between Heterogeneous Architecture and Model Structure

Concurrent heterogeneous computing architecture. Different from servers, mobile devices employ a unique heterogeneous computing architecture. Take the widely-used mobile CPU and GPU as an example. (1) Comparable performance. Different from server GPUs which run orders-of-magnitude faster than the CPUs, because of the chip and power limitation, mobile CPUs and GPUs have similar performance especially for deep learning model inference, as shown in Fig. 3. (2) A unified memory. Different from server machines which usually have separate memory units for the CPU and GPU, the mobile CPU and GPU share a unified memory [20]. Thus, expensive data copying costs can be avoided. Because of these two properties, there are opportunities to adopt heterogeneous concurrent computing to speed up the model inference.

Sequential model structure. Current widely-used NNs are composed of operators sequentially dependent on each other. The general practice to design a NN is to design a layer first, which can be a single operator or a building block, and then stack the layer repeatedly in each *stage*. A *stage* is a sequence of layers with the same size of a feature map. Fig. 2 shows the building blocks of representative NN models. Table 1 lists the stages of RegNetX-4GF as an example, which is mainly composed of four stages with the number of blocks as 2, 5, 14 and 2, respectively. Within a stage, the first building block is applied with convolution (short as Conv) stride=2, while the following blocks are stride=1. Other hyperparameters within a stage stay the same, such as the number of filters.

Communication-intensive intra-operator parallelism. To concurrently execute a sequential NN, three research papers μlayer [21], Optic [42] and CoDL [19] explore to utilize the intra-operator parallelism of a NN. They partition each operator (along the output channel or height/width dimensions) to run on different processors, as illustrated in Fig. 3a. Upon the execution of an operator, the processors communicate to share the output between each other for the next operator. Though the CPU/GPU/DSP processors share a unified memory on mobile SoC, the communication normally introduces the following overheads: (1) data transformation, to transform the data to the type used by each processor; (2) data mapping, to map the data to each processor's address space; and (3) processor synchronization, to inform other processors the completion of an action.

The processor communication overhead is significant, particularly for the light-weight mobile-side NNs. Fig. 3 compares the communication latency and the single-processor execution time

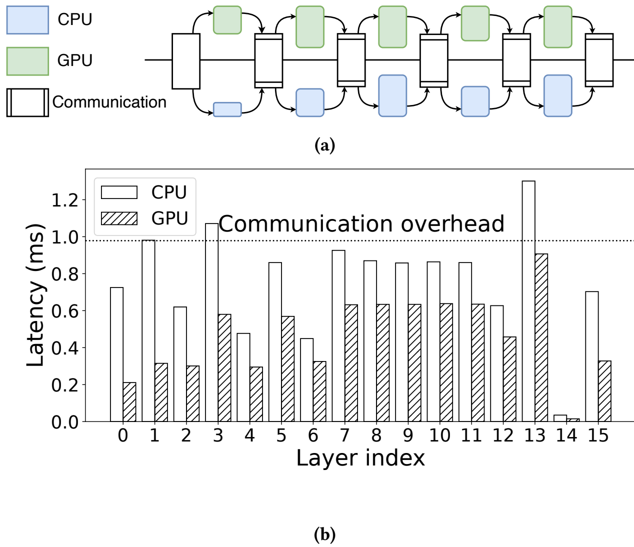


Figure 3: (a) Intra-operator parallelism. Communication is needed for each operator. (b) Execution latency when executed on a single CPU or GPU vs. CPU-GPU communication latency when executed on both processors in parallel, for each layer of MobileNetV1 running on Snapdragon 855. Communication latency is longer than most layers' execution times.

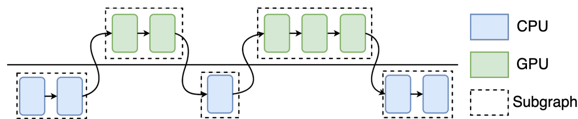


Figure 4: The sequential operator execution across processors in a typical today's industrial on-device inference framework.

for each operator of MobileNetV1. The communication latency is relatively stable around 1 ms, more than the inference latency of most operators on CPU or GPU only.

To address this challenge, this paper explores branch parallelism to avoid frequent processor-to-processor communications.

Unsupported processor concurrency. Though the research work μlayer and Optic have proposed concurrent processor execution, in real production scenarios, current industrial NN inference frameworks do not support multi-processor concurrent execution due to the high communication cost, either the mobile-side ones such as TFLite [24], Mace [27], and ncnn [38] or the server-side ones such as ONNX runtime [28] and TensorFlow [11]. Instead, they sequentially run operators on each processor as shown in Fig. 4.

This paper proposes a sub-graph-based modification for available mobile-side inference frameworks to enable multi-processor concurrent execution.

2.2 Lack of Model Adaption Suitable for Multi-Processor Execution

A NN model designed by data scientists normally has a gap for real-world usage, and requires model adaption for deployment on different devices. The gap can be: (1) unsatisfactory running time, (2) too-large model size and memory footprint, and (3) unsuitable operator types for hardware.

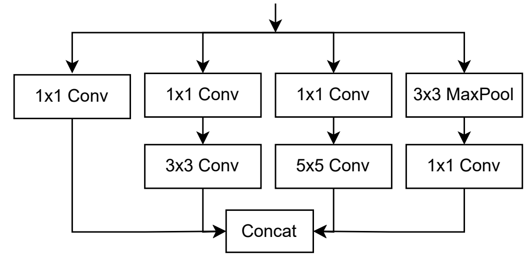


Figure 5: The handmade multi-branch block of InceptionV1.

Therefore, different model adaption techniques are proposed, which can be categorized into three major directions. (1) Model scaling [12, 17, 37]. It scales the depth (the number of layers), width (the number of filters i.e., output channels) or resolution of a given model to generate models of different sizes and FLOPs (the number of floating point operations) for diverse devices. For example, EfficientNets uniformly scales all dimensions, with the model size ranging from 7.8 MB to 66 MB, and accuracy 79% to 84%. (2) Model compression. Different from model scaling that scales model depth or width up/down for different devices and tasks, model compression mostly focus on removing the weight redundancy to compress a model into a smaller size. It includes a range of techniques, such as model pruning [13, 14, 22, 46] and quantization [4, 43, 45]. (3) Processor-tailored NN design [8, 36, 47, 50]. It designs NNs using hardware-efficient operators or hyperparameters. For example, EfficientNetEdgeTPU [32] and MobileNetEdgeTPU [2] aim to design efficient models for Edge TPU by augmenting the design space with expert-selected building blocks that run efficiently on Edge TPU.

These techniques above adapt models to either reduce the memory and computation usage, or better utilize the characteristics of a single processor. They greatly facilitate the deployment of NN models to mobile devices. However, none of them has considered model adaption for the unique feature of mobile devices, i.e., CPU-GPU heterogeneous computing.

There are also techniques that leverage the existing independent subgraphs. DUET [51] targets a subgraph-partition-compiling -scheduling problem. DUET depends on the existing parallel branches in a model, and then compiles the kernels for multiprocessors. DUET cannot generate branches for a model. BlastNet [23] depends on the multi-DNN concurrent execution. BlastNet partitions each model into blocks and schedules the blocks across different processors. Both techniques heavily rely on existing independent subgraphs for concurrent execution and are not suitable for sequential DNN models.

To address this challenge, this paper proposes a new model adaption strategy, which branches a model to utilize the heterogeneous processors.

2.3 Call for General and Automatic Model Branching

There are a few handmade multi-branch NNs, such as the Inception series [18, 33–35], and the NAS (Neural Architecture Search)-searched modes such as NASNet [3] and Cai et al. [52]. Inception, shown in Fig. 5, uses kernels of different sizes in different

branches to extract global and local information. Achieving compelling accuracy with low latency, the hand-crafted multi-branch NN is carefully designed and the blocks are customized stage-by-stage, though it is unclear how to adapt the Inception architectures to new datasets/tasks [44]. Therefore, the technique cannot be extended for general automatic model branching. The NAS-based methods search optimal NNs from a large space composed of basic multi-branch blocks. Though it can save manual labor, it is too costly in terms of computation, e.g., requiring 2000 GPU hours for NASNet [3] to find a proper multi-branch NN, rendering it impractical for deployment.

To address this challenge, our paper proposes a general, practical, and automatic branching technique.

3 NN-STRETCH SYSTEM OVERVIEW

Fig. 6 shows the overview of NN-Stretch system. It covers both model adaption/design and model inference. During the design phase, NN-Stretch takes as input a sequential model, its training function with dataset, and the target deployment platforms (both the device types and inference frameworks). NN-Stretch outputs the branched model. The model format is augmented with two new attributes for each operator to facilitate parallel inference: (1) a meeting point or not, and (2) the recommended running processor.

For inference, we design the sub-graph-based spatial scheduler to support parallel inference on multiple processors. It partitions each branch into a sub-graph, as the basic scheduling unit to run on a processor. The processor is picked according to the model design recommendation, processor availability, and the latency or energy priorities.

The first step of model design is to identify the meeting points, considering both the communication cost for latency and the model structure preservation for accuracy. The model is segmented by these meeting points. A *duplicate-and-scale-down* process is applied to these segments for branching, considering both processor heterogeneity for latency and capacity guarantee for accuracy. The process duplicates a segment into multiple branches first leading to an inflated model size, and then scales down the width i.e., the number of filters, or depth i.e., the number of layers, of each branch to shrink the model size back to the original. After the multi-branch model is generated, like other model adaption techniques, the new model is trained by the given training function and dataset.

The meeting point and scaling ratio selection is guided by latency reduction. The latency can be measured on devices or can also be predicted. We extend the state-of-the-art nn-Meter [49] latency predictor to predict latency on the target device.

4 MODEL BRANCHING

In order to transform a deep and narrow network to a short and wide one with multiple branches, the questions are where to branch the model, and what each branch looks like. We at first explored different NN design methods, such as the processor-tailored ones [8, 36, 47, 50], by which we tried to replace some layers by searched processor-friendly layers in different branches. However, we find that these methods can easily be against NN-Stretch's principles, leading to accuracy losses, heavy model adaption overheads, or

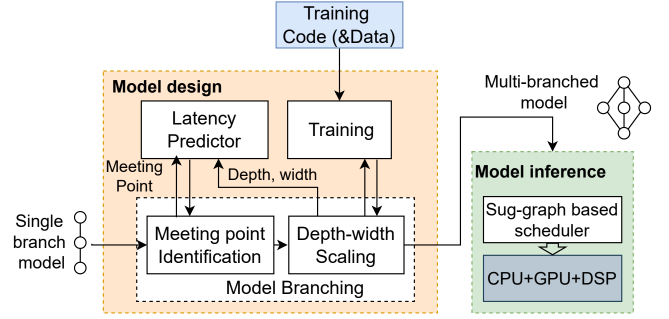


Figure 6: NN-Stretch system overview.

model designer involvements. They are thus not practical as a general and automatic deployment strategy.

Consequently, we conclude the important guideline for NN-Stretch's model branching: *to mostly maintain the given model's structure and capacity* (the size of weights and the total amount of computations, i.e., FLOPs). In this end, the model branching of NN-Stretch is devised as a duplicate-and-scale process. More specifically, NN-Stretch employs the following 3 steps: (1) identifying the meeting points to partition the sequential (deep and narrow) network to multiple segments, (2) duplicating each segment to form a multi-branch network structure, and (3) scaling down the depth and width of each branch to reduce the inference cost. This section will introduce how we formulate and solve the model branching issue.

4.1 Large Design Space for Model Branching

Problem formulation. NN-Stretch introduces a series of hyper-parameters needed to be specified for model branching. Specifically, Table 2 lists all hyper-parameters, as well as their types, shapes and potential values: the number of meeting-points (N_{meet}), the number of branches (N_{branch}), the locations of meeting points (L_{meet}), the depth-scaling ratios (R_{depth}) and width-scaling ratios (R_{width}) of all branches.

With all these hyper-parameters specified, NN-Stretch transforms a given model (represented as a graph, G_{seq}) to a multi-branch one (G_{branch}), represented as:

$$G_{branch} = F(G_{seq}, N_{meet}, N_{branch}, L_{meet}, R_{depth}, R_{width}) \quad (1)$$

where F denotes the graph transformation. Notably, different width-scaling factors could lead to different output channel numbers across branches. To tackle the divergence problem and restore the channel number, the graph transformation F inserts a 1×1 Conv operator after each branch, and concatenates all branches (an example Fig. 8 shown at the end of this section).

In NN-Stretch, we target to find a proper set of hyper-parameters ($N_{meet}, N_{branch}, L_{meet}, R_{depth}, R_{width}$) for a given model ($G_{seq}$), generating a multi-branch model (G_{branch}) with minimum latency, subject to the accuracy of the multi-branch model is larger than or equal to a pre-defined target accuracy (e.g., the G_{seq} accuracy, or with an acceptable relaxation). As discussed in [1, 6, 10, 25, 29], scaling a model in depth and width has similar expressive power and can successfully preserve its accuracy within a certain range. In general, model branching can be formulated as the following optimization problem:

Table 2: Estimation of the Hyper-parameter design space of meeting-point identification and depth-width scaling for model branching. The total design space for popular CNN models easily exceeds 10^{50}.

Hyper-Parameter	Description	Type	Potential Values
N_{meet}	number of meeting points	int	1 to N_{layer}
N_{branch}	number of branches	int	1 to $N_{processor}$
L_{meet}	meeting-point locations (indices of layer_id)	int 1-d array (shape = $[N_{meet}]$)	each location: 1 to N_{layer}
R_{depth}	depth-scaling ratios	float 2-d vector (shape = $[N_{meet}, N_{branch}]$)	each factor: 0.1 to 1.0
R_{width}	width-scaling ratios	float 2-d vector (shape = $[N_{meet}, N_{branch}]$)	each factor: 0.1 to 1.0

Table 3: Both accuracy and latency improved with the meeting point number N_{meet} (evaluated on ResNet56 with dataset CIFAR-100 and input size NHWC=<1, 96, 96, 3>).

N_{meet}	Top-1 Acc.	Top-5 Acc.	Latency (ms)
1	70.32	90.93	**13.2**
2	71.03	91.13	13.9
3	71.95	91.52	14.5
4	72.24	91.67	15.1
5	72.20	91.69	15.6
6	72.31	91.74	16.1
9	**72.46**	**91.85**	17.9

$$\min_{N_{meet}, N_{branch}, L_{meet}, R_{depth}, R_{width}} Latency(G_{branch}) \quad (2)$$

$$s.t. \quad Accuracy(G_{branch}) \geq target_accuracy \quad (3)$$

Challenges. However, it is extremely challenging to quickly find a set of hyper-parameters for multi-branch transformation that satisfies Equations 2 and 3, due to the following two reasons. First, the design space is huge. For a popular CNN model with 30-50 layers, the hyper-parameter design choices for model branching easily exceed 10^{50}. Second, even though the inference latency of a transformed multi-branch model can be quickly and accurately predicted or tested on actual devices, its accuracy can only be verified after the training phase which requires a large amount of GPU resources and training time, not to mention the $\sim 10^{50}$ design choices.

Directly searching all combinations of potential hyper-parameter choices is simply infeasible. In NN-Stretch, we propose a systematic way to drastically narrow down the search space according to: (1) the analysis of the influence of hyper-parameters on inference efficiency, (2) the observations of the influences of hyper-parameters on model accuracy.

4.2 Narrowing Down Design Space from Inference Latency Perspectives

Typically, the hyper-parameters of meeting points and scaling ratios have a direct and statistical correlation to the inference latency of the transformed multi-branch model, which also can be accurately and quickly predicted or tested on real devices. Therefore, we can explicitly formulate the relationship between latency and hyper-parameters, pruning away solutions with unsatisfactory latency.

Cost-amortized meeting point identification. When running multi-branch models on mobile SoCs, each meeting point naturally becomes a communication point across processors. A larger number of meeting-points results in higher communication costs, which may even offset the latency reduction gained from branch parallelism. As demonstrated in Table 3, the increase of meeting points results in a significant increase in overall latency.

In order to avoid exploring long latency solutions caused by high communication costs, we add constraints on the number of and locations of meeting points, as shown in Eq. 4. The rationale is to amortize the communication cost by the depth of a segment.

$$\begin{aligned} &Latency(communication) <= \\ &\alpha \cdot \sum_{j=L[i]}^{L[i+1]-1} Latency(layer_j), \forall i \in [0, N_{meet} - 1) \end{aligned} \quad (4)$$

where $Latency(communication)$ is the communication latency across processors; $Latency(layer_j)$ is the latency of a specific layer; α is a user-defined amortization ratio (e.g., 0.1, meaning that the communication cost is less than 10% of the latency of each segment).

Heterogeneity-aware depth-width scaling. In NN-Stretch, multiple branches are running on heterogeneous processors. Hence, the number of branches (N_{branch}) is determined by the number of processors ($N_{processor}$) on the mobile SoC.

Similar to cost-amortized meeting point identification, we set lower bounds on the depth/width-scaling ratios to avoid straggler branches that can increase the overall latency.

$$\begin{aligned} &Latency_{proc_i}(branch_i) <= \\ &\beta \cdot Latency_{proc_j}(original_segment), \forall i \in [1, N_{branch}] \end{aligned} \quad (5)$$

where β denotes an adjustment coefficient (normally as 1.1 or 1.2) to restrict the FLOPs of each branch and $proc_j$ is the fastest processor to execute the original segment.

As mobile SoCs are inherently heterogeneous, diverse processors (e.g., CPU, GPU, DSP) have different architectures and prefer branches with different scaling options to fully match their architectural characteristics. For example, the depth-scaling leads to a relatively wider branch shape, which is suitable for highly-paralleled GPU and DSP. As Fig. 7 shows, the achieved performance (GFLOPs/second) on the GPU increases greatly as the operator becomes wider. On the other hand, the width-scaling leads to a relatively deeper branch, which is more suitable for CPU as it is

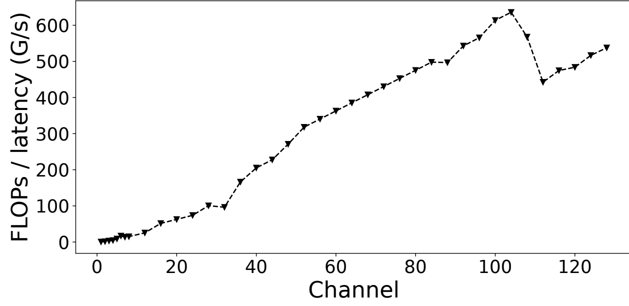


Figure 7: The performance (GFLOPs/second) increases greatly as the number of channels increases on the GPU. Operator setting: 3×3 Conv, the same number of input and output channels, height=width=28.

good at sequential small operators. Therefore, according to different hardware characteristics, corresponding constraints on scaling strategies can be applied to further narrow down the search space.

Last but not least, as processors in a mobile SoC have similar but slightly different performance power, the best way to fully saturate all processors is to align the FLOPs of each branch to the performance of the processor that executes this branch, as formulated in Eq. 6. (Note again that FLOPs means the number of floating point operations, and FLOPS is the throughput, i.e., FLOPs Per Second).

$$\begin{aligned} &FLOPs(branch_1) : FLOPs(branch_2) : \cdots : FLOPs(branch_n) \\ &\approx FLOPS(proc_1) : FLOPS(proc_2) : \cdots : FLOPS(proc_n) \end{aligned} \tag{6}$$

4.3 Narrowing Down Design Space from Model Accuracy Perspectives

Unlike latency-related constraints listed in Section 4.2 that are well formulated and analyzed, the accuracy-related constraints are 'soft' constraints due to the uncertainty and unpredictability of model inference accuracy. According to the characteristics of the sequential model itself and observations on preliminary experiments, model-specific soft constraints can be applied to further narrow down the search space.

Structure-preserved meeting point identification. In popular CNN models (e.g., ResNet, MobileNet, ShuffleNet), the models are often composed of multiple stages, with all the building blocks in each stage sharing the same architecture. In order to preserve the expert-designed model structure as much as possible, we first take the connection points between stages as meeting points, rendering each stage a segment to be branched.

Apart from inference efficiency, the number of meeting points also influences model accuracy. Intuitively, having more meeting points leads to more frequent feature exchanges, which in turn likely leads to better accuracy. While having fewer meeting points tends to hurt the model accuracy due to insufficient information exchange. The experimental result on ResNet56 with the N_{meet} from 1 to 9 matches our intuition, as shown in Table 3. Therefore, for a specific model, we set a lower bound on the number of meeting points, a number below which would result in violation of the accuracy requirement in Equation 3.

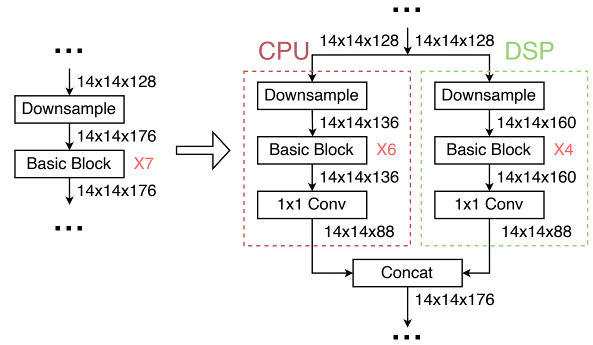


Figure 8: A generated multi-branch stage for EfficientNet.

Capacity-guaranteed depth-width scaling. Previous works [7, 26] point out that extremely shallow or narrow networks have limited expressive power and undermine the model accuracy. Inspired by this finding, we can also introduce lower bounds on depth-scaling and width-scaling ratios to avoid searching extremely shallow or narrow branches that cannot maintain the original accuracy. Our experiment on ResNet also demonstrates that branched models with extremely low scaling ratios (0.3 in Table 4) perform worse than those scaling ratios that result in a similar model size under a similar number of FLOPs. In general, lower bounds on scaling ratios can be obtained through preliminary experiments for different models.

Table 4: The low scaling ratio harms accuracy even when the total model size is similar (evaluated on ShuffleNetV2 with dataset CIFAR-100). (n,m) is (R_{depth}, R_{width}).

$Branch_1$	$Branch_2$	Top-1 Acc.	Top-5 Acc.	GFLOPs	#Params (M)
(0.5,1)	**(0.75,0.4)**	**72.81**	**92.91**	0.83	1.38
(0.5,1)	(0.3,1)	71.52	92.21	0.90	1.50
(1,0.5)	(1,0.3)	70.39	91.80	0.83	1.37

Notably, the constraints are not limited to Eq. 4, 5, and 6 introduced in Section 4.2 and those soft constraints introduced in Section 4.3. Further constraints on narrowing down the search space can be proposed and explored for various NN models. In NN-Stretch, with these constraints, we can effectively narrow down the search space from 10^{50} to less than 10 or 20 for popular CNN models. Therefore, we are able to efficiently find appropriate branching hyper-parameters using grid search.

For example, Figure 8 shows a generated multi-branch stage in NN-Stretch for EffcientNet. EfficientNet has 7 stages, and we use the 5th one for illustration. This stage has 7 blocks and 176 channels, i.e., (depth, width) as (7, 176). Here, we set α as 0.1, β as 1.0, and the lower bound of scaling ratios as 0.5. Among all the candidates, a CPU branch with (depth, width) as (6, 136) and a DSP branch (depth, width) as (4, 160) achieve the best latency reduction.

5 SUB-GRAPH-BASED SPATIAL SCHEDULER

For inference systems, a model is a graph containing a set of operators, which is the scheduling unit within a processor. To efficiently

support the parallel execution of multiple processors and reduce communication costs, NN-Stretch leverages a coarser granularity, i.e., *sub-graph*, as the scheduling unit among processors. The definition of a sub-graph is a sequence of operators run on a processor with only the first operator depending on the output from other processors. Each model branch is thus a sub-graph, and scheduled to run on a processor (Fig. 8 as an example).

Challenge of scheduler design. The challenge stems from the *different communication mechanisms among processors*, which require a careful design for both correctness and efficiency.

Take the GPU as an example. Since it provides low-level programming APIs, such as OpenCL, it can be programmed to run synchronously or asynchronously to the host CPU. However, the DSP library provides only the operator-level and graph-level APIs in a synchronous manner, which blocks the CPU thread until it finishes.

Similarly, considering data sharing among processors, processors on mobile SoCs have one unified memory, and there is no need for memory copy. However, there is no hardware-supported cache coherence. The coherence has to be guaranteed in the software [5]. The GPU has an explicit API, i.e., memory mapping, to make sure all the changed data in the cache is written back to memory, and ready to be read by other processors. On the other hand, DSP does not provide this explicitly, buried in the high-level APIs.

Threadpool for processor communication. The thread management is therefore specifically designed for different communication mechanisms. To address the synchronous-only manner on the DSP, we implement a threadpool, and assign a separate CPU thread to communicate with the DSP. The threadpool is created only once during initialization, to avoid repeated thread creation costs for each sub-graph.

Unexpectedly, we observe this communication thread greatly competes for the big cores. For example, the inference becomes very unstable and can be 2× slower after adding a thread for DSP communication. Thus, we bind the communication thread to the little cores and leave the big ones for operator-execution threads.

For processors that can run asynchronously, such as the GPU, our scheduler does not use a separate communication thread to avoid competition.

To address different data-sharing mechanisms, we explicitly call the memory mapping API for the GPU for cache coherence. For the DSP, we use the C++ *future* synchronization method to wait for the DSP communication thread to finish and then access the data for coherence.

Scheduling process. Algorithm 1 shows the scheduling process. During the model parsing phase of inference initialization, NN-Stretch partitions the graph into topology-ordered sub-graphs. Based on the design recommendation, processor availability, and latency/energy priority, it can decide the execution processor for each sub-graph.

During inference, if a sub-graph is a meeting point (Line 3 to 6), it starts the CPU sub-graph execution, waits for all the processors to finish, and then concatenates all the results. If the sub-graph targets the GPU, the scheduler does not start a separate thread. It enqueues the necessary commands (Line 15 to 19), including memory mapping, data transformation, and operator kernels, to the GPU queue and returns. If the sub-graph targets the DSP, it is pushed to a separate thread in the threadpool. The *future* object will be set after this thread is done, and the shared data with the DSP can be read by the CPU (Line 21).

Algorithm 1: Sub-graph-based spatial scheduler

```
1  for subgraph ∈ TopoSorted(Subgraphs) do
2      if subgraph.isMeetingPoint then
3          ExecuteOnCpu(CpuSubgraph);
4          OpenCLQueue.Finish();
5          DspFuture.Wait();
6          Execute(subgraph);
7      else
8          ParallelExecute(subgraph);
9      end
10 end
11 Function ParallelExecute(subgraph):
12     if subgraph.isCpuSubgraph then
13         CpuSubgraph = subgraph;
14     else if subgraph.isGpuSubgraph then
15         EnqueueInputMapBuffer();
16         EnqueueInputTransformKernel();
17         EnqueueGpuKernels(subgraph);
18         EnqueueOutputTransformKernel();
19         EnqueueOutputMapBuffer();
20     else if subgraph.isDspSubgraph then
21         DspFuture = ThreadPool.push(subgraph);
```

6 DISCUSSION

Model training cost reduction. The heterogeneity-aware model design of NN-Stretch performs the best when customized to the target processor. It may introduce re-design and re-train cost if the model needs to be deployed on diverse mobile devices. The supernet training method [48] from neural architecture search field can be leveraged to solve the issue. The supernet contains a range of subnets with different model configurations. By weight sharing technique, all the subnets in the supernet can be trained together only once. We use this method in Sec. 7.4 Ablation Study, since we need to compare many different models designed by different methods. All the model variants are trained together once. The supernet training costs 3× more GPU hours compared to training one model from scratch.

Also, we observe that for the same SoC series, such as Qualcomm Snapdragon 855 and 888, the processor design is relatively stable, but just scales up the processor performance. We thus avoid model redesign if the model behaves as expected on a new device.

More scheduling opportunities. NN-Stretch enables the flexible combination of different processors for inference, based on the real usage of processors by other applications on the devices. For small models that already achieve real-time on one-single processor, NN-Stretch also gives the space to increase the model size and accuracy with the same latency cost. We will also show the accuracy and latency tradeoff results of NN-Stretch in the ablation study.

Adaption to dynamic workloads. A key design goal of NN-Stretch is to adapt to dynamic workload, by generating parallel model structure for combinations of available processors. For example, if GPU is busy rendering, we can use CPU+DSP for inference. Similarly, if CPU is busy, we can use GPU+DSP. By comparison, many DNN models use sequential structures, and can only run on one processor. This bottlenecks the flexibility and speedup upper bound according to Amdahl's law. Our model structure does have processor affinity for better performance. But this does not prohibit it from running on other processors. We will show the results of NN-Stretch adaption to dynamic workloads in Sec. 7.4 ablation study.

Integration with the current mobile software stack. Our model design only updates the model structure, decoupled with the inference systems. After the updated model file is input into the inference system, any model optimizations or compressions can be applied. Our scheduler is integrated into the inference system scheduler, to enqueue the operators to the available processors for parallel execution. The inference systems we used are TFLite from Google, and Mace from Xiaomi, which are all widely used mobile inference systems.

Application to other tasks and models. One design principle of NN-Stretch is to maintain the model accuracy, by keeping the total number of FLOPs, the basic model structure, and the capacity-guaranteed branch scaling. We evaluate NN-Stretch on the complex ImageNet dataset as a showcase. There are certainly numerous tasks in the real world. We expect NN-Stretch could maintain accuracy. This paper targets the widely used convolution models for mobile devices. For Transformer models, NN-Stretch could also be leveraged, e.g., to parallelize multiple attention heads, and FFN layers on the CPU, GPU, and DSP. The exploration of more tasks, datasets, and transformer models can be promising future works.

7 EVALUATION

7.1 NN-Stretch Implementation

The model design is implemented based on Facebook's *pycls* [31], an image classification codebase written in PyTorch. It was originally developed for the RegNet models. pycls uses basic training settings without any training or testing enhancements and thus provides simple, strong and reproducible baselines. We integrate a multi-branch model builder, TFLite model generator, latency predictor and supernet trainer into pycls.

The parallel inference is implemented in TFLite 2.8.0 [24]. The major updates in TFLite is as following. The augmented operator parsing is in `GetOpsToReplace`; sub-graph partitioning in `PartitionGraphIntoIndependentNodeSubsets`; and sub-graph scheduling to different processors in `Invoke`. The sub-graph is wrapped by `HexagonDelegateKernel` object for DSP and `TfLiteGpuDelegateV2` for GPU.

To measure the communication latency of the CPU and GPU, we leverage the OpenCL Event object [30], which gives the timing of a GPU command execution, i.e., queued (command is enqueued by the host), submitted (command is submitted by the host to the device), running (command starts execution on the device), and complete (command finishes execution on the device). We time between the queued `clEnqueueMapBuffer` command and the start of the first GPU computing kernel, as well as the completion of the

Table 5: Test platforms in the evaluation.

	Xiaomi 9	Pixel 6	Xiaomi 11
SoC	Snapdragon 855	Google Tensor	Snapdragon 888
CPU	Kyro 485	Cortex-X1/A76/A55	Kyro 680
GPU	Adreno 640	Mali-G78	Adreno 660
DSP	Hexagon 690	-	Hexagon 780

Table 6: FLOPs and model size of our test models.

Model	FLOPs (G)	Params (M)
ResNet34	3.7	21.8
ResNet50	4.1	25.6
RegNetX-1.6GF	1.6	11.2
RegNetX-4GF	4.0	20.6
EfficientNet-Lite4	2.6	13.0
EfficientNet-B5	10.3	30.4

last GPU computing kernel and `clEnqueueUnmapObject`, as the communication latency. In this way, the communication latency includes kernel enqueue overhead, data transformation, and memory map/unmap. For CPU and DSP, as they share the same physical address space, there is no memory map/unmap cost. The hexagon DSP provides no APIs for fine-grained profiling. We get the communication latency by subtracting the elapsed time of `hexagon_nn_execute_new` in a normal DSP-only execution from the total latency of the DSP communication thread in the threadpool.

The total lines of code (LOC) include 2860 lines of modifications to pycls and Tensorflow, and 1058 of testing tools.

7.2 Experiment Setup

Test models. The NNs selected for evaluation are ResNet, RegNet and EfficientNet. They are widely-used models and also their model size is suitable for mobile deployment. They are composed of different backbone blocks such as residual bottleneck and MBConv, and different backbone operators such as vanilla convolution, grouped convolution and depth-wise convolution. We also choose two variations of each to show NN-Stretch's effectiveness for various model lengths and widths, i.e., ResNet-34/50, RegNetX-1.6GF/4GF and EfficientNet-Lite4/B5. The model FLOPs and sizes are shown in Table 6.

To deploy RegNet onto the mobile GPU, we follow the common practice to replace the grouped convolution with a split and several small convolutions, each of which is corresponding to one group of the grouped convolution [39]. To deploy EfficientNet-B5, we remove the SE operators and replace swish operators on the CPU branch with ReLU6 [40].

Since the DSP only supports INT8 computation, all the models are INT8-quantized. The CPU also has hardware-supported INT8 instructions. Mobile GPU does not support INT8 execution, and the models are executed in FP16.

ImageNet classification task. The application selected is the popular image classification for the large-scale ImageNet dataset. ImageNet consists of $1.28M$ for training and $50K$ images for validation. The input image size for inference is NHWC = <1, 224, 224, 3>. Model training code and configurations for every model are from pycls. The training configurations are as follows. All the models use SGD with a momentum of 0.9, a half-period cosine schedule,

Table 7: Comparison of Top-1 accuracy of the multi-branch models from NN-Stretch on all three devices, with the base sequential models. NN-Stretch can maintain the accuracy of the models. R, RX, EN stand for ResNet, RegNetX, EfficientNet and C, G, D stand for CPU, GPU, DSP, respectively.

	R-34	R-50	RX-1.6	RX-4	EN-L4	EN-B5
Original	73.3	76.7	77.0	78.6	77.8	78.0
Mi9/C+G	72.8	76.2	76.9	78.4	77.9	77.8
Mi9/C+D	73.2	76.2	77.5	78.9	77.6	77.9
Mi9/G+D	73.3	76.6	76.9	78.4	77.5	77.8
Mi9/C+G+D	72.9	76.1	76.6	78.3	-	-
Pixel6/C+G	73.6	76.5	-	-	77.6	77.9
Mi11/C+G	72.8	76.2	76.9	78.4	77.9	77.8

and 100 training epochs. ResNet and RegNet use a learning rate of 0.2, a batch size of 256, and a weight decay of 5e-5. EfficientNet uses a reference learning rate of 0.4, a batch size of 256, and a weight decay of 1e-5. Training takes the same amount of time as the baseline models.

Hardware devices. We evaluate NN-Stretch on three different mobile SoCs, integrated with different CPUs, GPUs and DSPs. Table 5 lists the detailed specification. Pixel 6 does not have DSP equipped.

We try the best to run different processor combinations for each device. However, there are some settings not supported by the inference system. The DSP on Xiaomi 11 cannot be evaluated, as Snapdragon 888 is currently unsupported for TFLite Hexagon DSP delegate. Additionally, split operators with more than four output tensors are not supported by the TFLite OpenCL backend for Mali-GPUs. That means RegNetX-1.6GF/4GF cannot be evaluated on the Pixel 6.

Energy measurement. To measure the energy consumption, we monitor the real-time power of the phone during inference by reading /sys/class/power_supply. We sample the power with an interval of 0.1ms, and then integrate the power with respect to time to get energy.

Comparison baselines. We have two baselines. The first is the single processor latency using default TFLite. The second is the CoDL system, i.e., the state-of-the-art parallel inference system on the mobile CPU and GPU, leveraging intra-operator parallelism. For a fair comparison, we update the CoDL code directly to support our multi-branch parallel inference. The baseline CoDL results use the settings as image-based GPU memory type, heuristic chain-search policy, and output channel as partition dimension.

7.3 Overall Results

Accuracy. Table 7 compares the accuracy of the multi-branch models generated by NN-Stretch with the given baseline models. We design different models for different devices by adjusting the depth/width scaling ratios. As there is similar CPU vs GPU computing performance between Mi11 and Mi9, they share the multi-branch models. We can see that NN-Stretch can preserve the accuracy of the models.

It's possible to reuse the weights of the given model by applying knowledge distillation (KD). Furthermore, as our stretched model has a similar structure to the original model, KD can perform even better in such a situation. But in our evaluation section, to prove that the stretched models have the same or even better expressive power, we train them from scratch with the same training configurations as the original models. By applying KD to ResNet-34 (Mi9/C+G) and ResNet-50 (Mi9/C+G), we can achieve 73.7% and 76.9% respectively using the weighted soft label distillation [15].

Speedup to TFLite. Fig. 9 shows the inference latency comparison between the single-processor (CPU/GPU/DSP) baseline on default TFLite, and NN-Stretch. By the multi-branched model design and sub-graph scheduling, NN-Stretch enables the flexible processor combinations based on availability, i.e., CPU+GPU, CPU+DSP, GPU+DSP, and CPU+GPU+DSP, for parallel inference. We can achieve up to 2.2×, 1.4×, and 1.8× speedup compared to the fastest single processor, on the three platforms respectively. The fastest inference is achieved obviously when all three processors are used. The following is CPU+DSP, and then CPU+GPU. Note that the GPU runs in FP16, and thus its performance can be worse than the INT8 CPU and DSP for some models.

Considering just the speedup of branches, we can achieve near-ideal speedup by adding more processors. However, for this end-2-end speedup comparison, there is surely a speedup difference from ideal. For example, ResNet-50 on Mi9/C+D achieves a 1.5× speedup. Based on the latency of the single CPU and DSP, a 1.9× speedup is expected. The major causes are as follows. (1) The stem and head blocks of the model are not parallelized. The percentage of these blocks depends on the model structure. In ResNet-50, this accounts for 11% of the total latency. For EfficientNet, this percentage is 15%. Therefore, EfficientNet is not worthy for three-processor parallelism. (2) Another reason is the communication overhead among processors. In this ResNet-50 on Mi9/C+D example, this overhead contributes to 17.9% of the total latency.

The missing bars on Pixel 6 and Mi11 are not supported by the platforms due to the reasons explained in Sec. 7.2.

Speedup to CoDL. Fig. 11 shows the latency comparison with the CoDL baseline. Since it is only for CPU+GPU parallel run, we also run NN-Stretch on the CPU+GPU. NN-Stretch achieves a speedup of up-to 1.9× compared to GPU and 1.6× compared to CoDL. RegNetX and EfficientNet are not evaluated, as grouped convolution and INT8 depth-wise convolution are currently not included in CoDL. Pixel6 is not evaluated because OpenCL 3.0 is unsupported by Mace framework utilized by CoDL.

For these relatively small models, CoDL fails to achieve latency reduction due to the high communication cost brought by intra-op parallelism. However, as NN-Stretch only requires synchronization at meeting points, this overhead is dramatically reduced. For instance, when running ResNet-50 on Mi11, CoDL takes 12.2ms for communication, accounting for 35.3% of the total latency. In contrast, NN-Stretch reduces the communication overhead to 3.0ms.

Power and energy. We also evaluate the power and energy consumption of NN-Stretch. As shown in Fig. 10, for the power of a single processor, running the model on the CPU consumes the highest power, ranging from 5.1 to 6.5w. The DSP costs the lowest, ranging from 1.2 to 2.0 W. This is due to the simpler architecture of the DSP. Interestingly, compared with the single CPU, the use of multi-processors by NN-Stretch does not increase power consumption greatly. The most significant increase is seen in EfficientNet-Lite4, which increases by 24%. This phenomenon occurs because

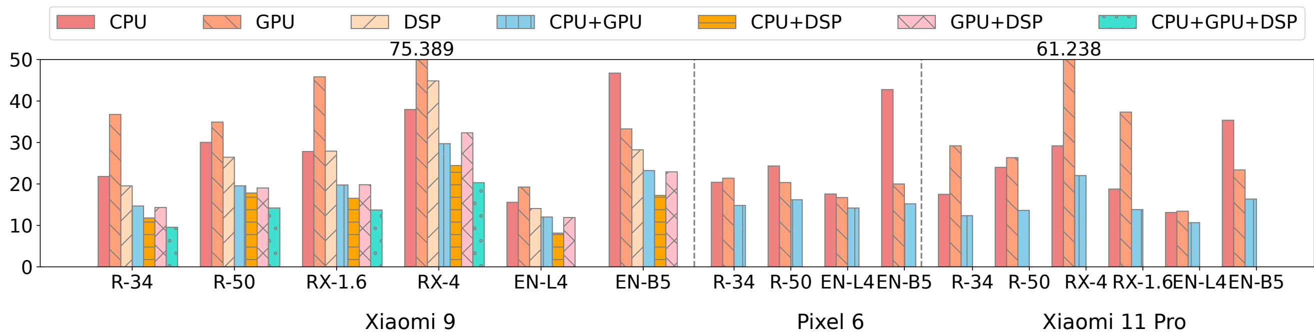
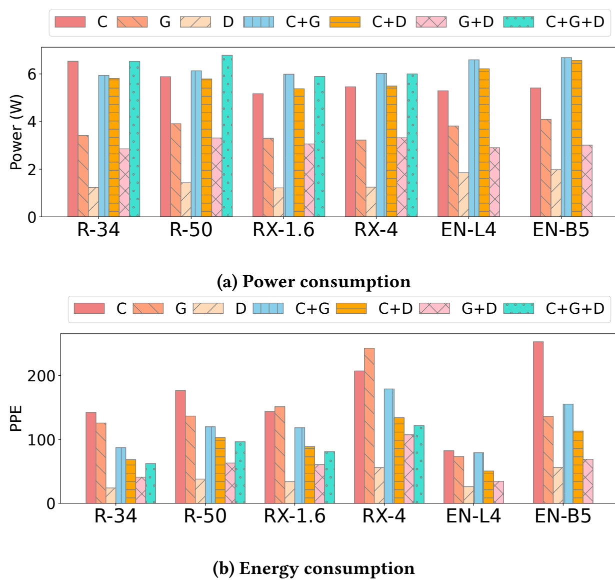


Figure 9: Latency comparison of the TFLite single-processor with NN-Stretch multi-processors.



(a) Power consumption

(b) Energy consumption

Figure 10: Power and energy comparison of the single processor and NN-Stretch multi-processors on Xiaomi 9.

the inference computation is distributed across multiple processors, and the CPU power and utilization are reduced. For instance, when designing ResNet-34 for CPU+DSP, the widths of the convs in the CPU branch are scaled down to 50%, the smallest scaling ratio among all the models, resulting in less power consumption than the CPU.

Similarly, for energy, the lowest cost is running the model on the DSP, followed by the DSP with other processors, i.e., GPU+DSP and CPU+GPU+DSP. Compared with the single CPU, CPU+GPU+DSP can reduce the energy cost by 56.3%.

Discussion. Though the DSP consumes less power and energy, its applicability is quite limited. It only supports common operators in INT8 precision, long-vector computation, and no low-level programmable APIs [41]. The CPU and GPU are still the most widely used processor for DNN inference on mobile devices. All in all, NN-Stretch enables the capability of DNN inference to flexibly select the proper processors to run, based on the model and hardware characteristics, and the latency and energy priority.

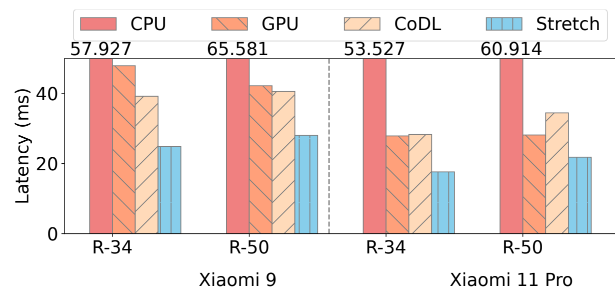


Figure 11: Latency comparison of the single processor, CoDL and NN-Stretch on CPU+GPU on Mi9 and Mi11.

Table 8: The accuracy of just branching without depth/length scaling on ResNet-50.

Network	Top-1 Acc.
Original	76.7
Two Branch	78.2

7.4 Ablation Study

Since this section evaluates many different designed models, we use the supernet training method to train all the model variations together. As you will see, the supernet training can achieve competitive accuracy and latency results compared to training each model from scratch.

Effectiveness of segment duplication. To evaluate whether each branch can effectively extract the features after meeting point identification and segment duplication, we generate a model before scaling down to train and evaluate the accuracy as shown in Table 8. Results show that by only duplicating the segment, the inflated model can achieve 1.5% higher accuracy, which leaves space for further scaling down for latency reduction.

Different scaling strategies. Sec. 4.3 suggests scaling on different dimensions for each branch. This diversifies the branches potentially for better accuracy, as well as better utilization of heterogeneous hardware features. This section also evaluates the accuracy and latency for other possibilities, i.e., only depth scaling for both branches, and only width scaling for both branches, to compare with the length and width scaling for each, as shown in

Table 9: Accuracy and speedup comparison of different scaling strategies: depth-only, width-only, and depth+width for ResNet-50.

Strategy	Top-1 Acc.	Speedup
Original	76.7	1
Depth-only	75.7	1.3
Width-only	76.2	1.2
Depth + Width	**76.2**	1.5

Table 9. The results show that NN-Stretch can achieve higher performance and better accuracy compared to the other two.

Accuracy improvement. Besides latency reduction and accuracy preservation, NN-Stretch also provides the opportunity for accuracy improvement by setting different latency constraints and different branch scaling ratios. Fig. 12 shows the latency and accuracy tradeoff line from NN-Stretch for ResNet-50 on Mi9/C+D. It is above the original model, which means at similar latency, NN-Stretch could achieve higher model accuracy, and thus gives shows more options for the users based on the latency and accuracy budget.

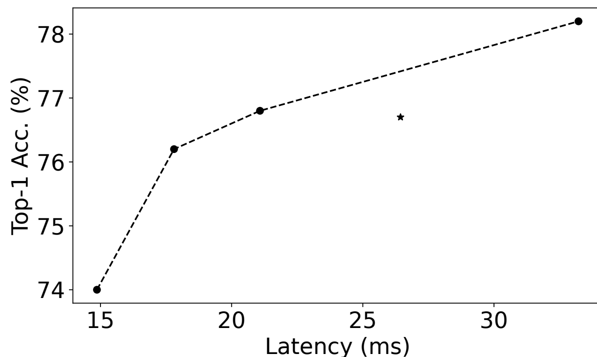


Figure 12: NN-Stretch improves the accuracy-latency tradeoff (marked as dots) compared to the original ResNet-50 on Mi9/C+D (marked as the star).

Adaption to dynamic workloads. NN-Stretch can adapt to dynamic workloads by employing different availale processors for execution. To validate this, we introduce interference workload, and compare the latency response of NN-Stretch with the single-processor baseline, shown in Fig. 13. The interference workload used is Gables [16], which executes roofline model benchmarks on either CPU or GPU, including both memory- and computation-intensive operations. NN-Stretch monitors processor's utilization after each model inference. If NN-Stretch detects resource contention with other processes on a processor, it will move the model to run on other processors.

The figure shows three phases, marked by the two dotted lines. The interference workload runs on the GPU during [0, 50) model inference. The baseline inference runs on the CPU, while NN-Stretch runs on GPU+DSP. During [50, 155), the interference runs on the CPU, as well as the baseline. The baseline latency becomes flunctuated because of the interference. The worst latency can reach 222 ms. That is possibly when Gables is running computation intensive operations. On the other hand, NN-Stretch moves the model to the GPU+DSP when contention is detected after the 50_{th} inference. Though the latency cost of running ResNet-34 on GPU+DSP is higher than CPU+DSP by NN-Stretch, it is still much faster than the baseline. During [155, 200), the intereference is moved back to run on the GPU. The latency response of the baseline and NN-Stretch becomes similar as phase one, i.e., [0, 50).

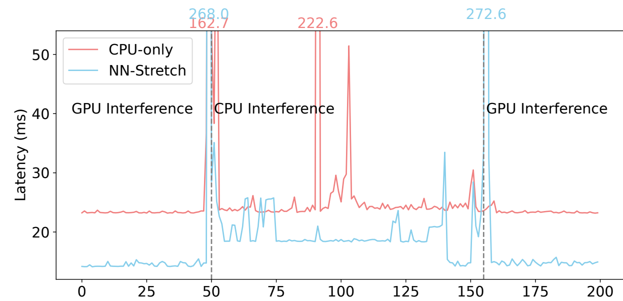


Figure 13: The latency response of NN-Stretch and baseline with interference workload for ResNet-34 on Mi9. NN-Stretch can adapt to dynamic interference. The baseline always runs on the CPU. For [0, 50) and [155, 200) inferences, interference workload runs on the GPU, while NN-Stretch runs on the CPU+DSP. During [50, 155), the interference runs on the CPU, and NN-Stretch moves the model to run on the GPU+DSP.

8 CONCLUSION

This paper carefully studies how to accelerate DNN model inference on mobile devices by levering the heterogeneous processors (i.e., CPU, GPU, DSP) equipped on these devices. In particular, we propose to restructure the models that are often sequentially structured to multi-branch structures and assign each branch to a processor for concurrent execution.

To the best of our knowledge, NN-Stretch is the first work that exploits the branch parallelism for a model to speed up its inference. Moving forward, we will continue our exploration in this direction, by investigating how to adopt our method on different hardware platforms such as robots or autonomous cars, as well as different deep learning models such as natural language processing models. We would like to see the branch parallelism evolves to be a general type of parallelism that benefits many different deep learning model inference scenarios.

9 ACKNOWLEDGEMENT

The authors thank the anonymous shepherd and reviewers for their valuable comments. The work of Yunxin Liu was supported by the National Key R&D Program of China (No. 2022YFF0604501) and Xiaomi Foundation.

A APPENDIX

The research artifact accompanying this paper is available via https://doi.org/10.5281/zenodo.7907009.

REFERENCES

[1] Md. Zahangir Alom, Theodore Josue, Md Nayim Rahman, Will Mitchell, Chris Yakopcic, and Tarek M. Taha. 2018. Deep Versus Wide Convolutional Neural

Networks for Object Recognition on Neuromorphic System. In *2018 International Joint Conference on Neural Networks, IJCNN 2018, Rio de Janeiro, Brazil, July 8-13, 2018*. IEEE, 1–8. https://doi.org/10.1109/IJCNN.2018.8489635

[2] Suyog Gupta Andrew Howard. 2019. *Introducing the Next Generation of On-Device Vision Models: MobileNetV3 and MobileNetEdgeTPU*. https://ai.googleblog.com/2019/11/introducing-next-generation-on-device.html

[3] Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. 2018. Path-Level Network Transformation for Efficient Architecture Search. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer G. Dy and Andreas Krause (Eds.). PMLR, 677–686. http://proceedings.mlr.press/v80/cai18a.html

[4] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep Learning with Low Precision by Half-Wave Gaussian Quantization. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 5406–5414. https://doi.org/10.1109/CVPR.2017.574

[5] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. 2014. Hexagon DSP: An Architecture Optimized for Mobile Multimedia and Communications. *IEEE Micro* 34, 2 (2014), 34–43. https://doi.org/10.1109/MM.2014.12

[6] Nadav Cohen, Or Sharir, and Amnon Shashua. 2015. On the Expressive Power of Deep Learning: A Tensor Analysis. *CoRR* abs/1509.05009 (2015). arXiv:1509.05009 http://arxiv.org/abs/1509.05009

[7] Nadav Cohen, Or Sharir, and Amnon Shashua. 2016. On the Expressive Power of Deep Learning: A Tensor Analysis. In *29th Annual Conference on Learning Theory (Proceedings of Machine Learning Research, Vol. 49)*, Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir (Eds.). PMLR, Columbia University, New York, New York, USA, 698–728. https://proceedings.mlr.press/v49/cohen16.html

[8] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, et al. 2019. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[10] Fenglei Fan, Rongjie Lai, and Ge Wang. 2020. Quasi-Equivalence of Width and Depth of Neural Networks. *arXiv: Learning* (2020). https://doi.org/arXiv:2002.02515

[11] Google. 2019. *TensorFlow: An end-to-end open source machine learning platform*. https://www.tensorflow.org/

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[13] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. 2019. Filter Pruning via Geometric Median for Deep Convolutional Neural Networks Acceleration. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*.

[14] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel Pruning for Accelerating Very Deep Neural Networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

[15] Zhou Helong, Song Liangchen, Chen Jiajie, Zhou Ye, Wang Guoli, Yuan Junsong, and Qian Zhang. 2021. Rethinking soft labels for knowledge distillation: a bias-variance tradeoff perspective. In *International Conference on Learning Representations (ICLR)*.

[16] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A Roofline Model for Mobile SoCs. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 317–330. https://doi.org/10.1109/HPCA.2019.00047

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

[18] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML (JMLR Workshop and Conference Proceedings, Vol. 37)*, Francis R. Bach and David M. Blei (Eds.). JMLR.org, 448–456. http://dblp.uni-trier.de/db/conf/icml/icml2015.html#IoffeS15

[19] Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. CoDL: Efficient CPU-GPU Co-execution for Deep Learning Inference on Mobile Devices. In *The 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22)*. ACM. https://doi.org/10.1145/3498361.3538932

[20] Shiqi Jiang, Lihao Ran, Ting Cao, Yusen Xu, and Yunxin Liu. 2020. Profiling and Optimizing Deep Learning Inference on Mobile GPUs. In *APSys '20*. Association for Computing Machinery, New York, NY, USA, 75–81.

[21] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *EuroSys '19*. Association for Computing Machinery, New York, NY, USA, Article 45, 15 pages.

[22] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning Filters for Efficient ConvNets. The International Conference on Learning Representations (ICLR).

[23] Neiwen Ling, Xuan Huang, Zhihe Zhao, Nan Guan, Zhenyu Yan, and Guoliang Xing. 2023. BlastNet: Exploiting Duo-Blocks for Cross-Processor Real-Time DNN Inference. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems* (Boston, Massachusetts) *(SenSys '22)*. Association for Computing Machinery, New York, NY, USA, 91–105. https://doi.org/10.1145/3560905.3568520

[24] Tensorflow Lite. 2020. https://www.tensorflow.org/lite/

[25] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. 2017. The Expressive Power of Neural Networks: A View from the Width. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 6231–6239. https://proceedings.neurips.cc/paper/2017/hash/32cbf687880eb1674a07bf717761dd3a-Abstract.html

[26] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. 2017. The Expressive Power of Neural Networks: A View from the Width. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6232–6240.

[27] MACE. 2020. https://github.com/XiaoMi/mace

[28] Microsoft. 2019. *ONNX Runtime*. https://github.com/microsoft/onnxruntime

[29] Thao Nguyen, Maithra Raghu, and Simon Kornblith. 2021. Do Wide and Deep Networks Learn the Same Things? Uncovering How Neural Network Representations Vary with Width and Depth. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net. https://openreview.net/forum?id=KJNcAkY8tY4

[30] OpenCL. 2021. https://www.khronos.org/opencl/

[31] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. 2020. Designing Network Design Spaces. In *CVPR*.

[32] Mingxing Tan Suyog Gupta. 2019. *EfficientNet-EdgeTPU: Creating Accelerator-Optimized Neural Networks with AutoML*. https://ai.googleblog.com/2019/08/efficientnet-edgetpu-creating.html

[33] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, Satinder P. Singh and Shaul Markovitch (Eds.). AAAI Press, 4278–4284. http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14806

[34] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[35] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. 2015. Rethinking the Inception Architecture for Computer Vision. *CoRR* abs/1512.00567 (2015). arXiv:1512.00567 http://arxiv.org/abs/1512.00567

[36] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[37] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6105–6114. http://proceedings.mlr.press/v97/tan19a.html

[38] Tencent. 2018. *Tencent ncnn deep learning framework*. https://github.com/Tencent/ncnn

[39] Tensorflow. 2020. https://github.com/titu1994/Keras-ResNeXt

[40] Tensorflow. 2022. https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet/lite

[41] TensorFlow. 2022. Hexagon Delegate. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/delegates/hexagon/README.md

[42] S. Wang, G. Ananthanarayanan, and T. Mitra. 2019. OPTiC: Optimizing Collaborative CPU–GPU Computing on Mobile Devices With Thermal Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 3 (2019), 393–406.

[43] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. 2015. Quantized Convolutional Neural Networks for Mobile Devices. *CoRR* abs/1512.06473 (2015). arXiv:1512.06473 http://arxiv.org/abs/1512.06473

[44] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 5987–5995. https://doi.org/10.1109/CVPR.2017.634

[45] Yuhui Xu, Yongzhuang Wang, Aojun Zhou, Weiyao Lin, and Hongkai Xiong. 2018. Deep Neural Network Compression With Single and Multiple Level Quantization. *Proceedings of the AAAI Conference on Artificial Intelligence* 32, 1 (Apr. 2018). https://ojs.aaai.org/index.php/AAAI/article/view/11663

[46] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. 2017. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[47] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. 2018. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[48] Jiahui Yu, Pengchong Jin, Hanxiao Liu, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Thomas Huang, Xiaodan Song, and Quoc V. Le. 2020. BigNAS: Scaling Up Neural Architecture Search with Big Single-Stage Models. In *European Conference on Computer Vision*.

[49] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices. In *MobiSys 2021*. https://www.microsoft.com/en-us/research/publication/nn-meter-towards-accurate-latency-prediction-of-deep-learning-model-inference-on-diverse-edge-devices/

[50] Li Lyna Zhang, Yuqing Yang, Yuhang Jiang, Wenwu Zhu, and Yunxin Liu. 2020. Fast Hardware-Aware Neural Architecture Search. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2020, Seattle, WA, USA, June 14-19, 2020*. Computer Vision Foundation / IEEE, 2959–2967. https://doi.org/10.1109/CVPRW50498.2020.00354

[51] Minjia Zhang, Zehua Hu, and Mingqin Li. 2021. DUET: A Compiler-Runtime Subgraph Scheduling Approach for Tensor Programs on a Coupled CPU-GPU Architecture. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 151–161. https://doi.org/10.1109/IPDPS49936.2021.00024

[52] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. In *CVPR*. 8697–8710. http://openaccess.thecvf.com/content_cvpr_2018/html/Zoph_Learning_Transferable_Architectures_CVPR_2018_paper.html