AdaptQNet: Optimizing Quantized DNN on Microcontrollers via Adaptive Heterogeneous Processing Unit Utilization

Yansong Sun, Jialuo He, Dirk Kutscher, Huangxun Chen
Hong Kong University of Science and Technology (Guangzhou)
{ysun012,jhe212}@connect.hkust-gz.edu.cn,{dku,huangxunchen}@hkust-gz.edu.cn

Abstract

There is a growing trend in deploying DNNs on tiny microcontroller (MCUs) to provide inference capabilities in the IoT. While prior research has explored many lightweight techniques to compress DNN models, achieving overall efficiency in model inference requires not only model optimization but also careful system resource utilization for execution. Existing studies primarily leverage arithmetic logic units (ALUs) for integer-only computations on a single CPU core. Floating-point units (FPU) and multi-core capabilities available in many existing MCUs remain underutilized. To fill this gap, we propose AdaptQNet, a novel MCU neural network system that can determine the optimal precision assignment for different layers of a DNN model. AdaptQNet models the latency of various operators in DNN models across different precisions on heterogeneous processing units. This facilitates the discovery of models that utilize FPU and multi-core capabilities to enhance capacity while adhering to stringent memory constraints. Our implementation and experiments demonstrate that AdaptQNet enables the deployment of models with better accuracy-efficiency trade-off on MCUs.

CCS Concepts

• **Computer systems organization → Embedded software**.

Corresponding Author: Huangxun Chen.

ACM MOBICOM '25, Hong Kong, China

ACM ISBN 979-8-4007-1129-9/2025/11
https://doi.org/10.1145/3680207.3765247

Keywords

On-device AI, Neural Network Architecture Search

ACM Reference Format:
Yansong Sun, Jialuo He, Dirk Kutscher, Huangxun Chen. 2025. AdaptQNet: Optimizing Quantized DNN on Microcontrollers via Adaptive Heterogeneous Processing Unit Utilization. In *The 31st Annual International Conference on Mobile Computing and Networking (ACM MOBICOM '25), November 4–8, 2025, Hong Kong, China.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3680207.3765247

1 Introduction

With advancements in deep learning, improving DNN inference efficiency has become an important research topic. Driven by the increasing demand for real-time services and emphasis on data privacy, the focus has expanded from merely enhancing DNN inference efficiency on GPU servers to prioritizing efficiency on edge devices, such as mobile and IoT devices [13, 14, 17–21, 23, 27, 38, 51]. Fundamentally, DNN inference efficiency is determined by two key aspects:
• **DNN's inherent properties.** Larger models (i.e., more parameters) are generally less efficient. Therefore, many efforts have been made to develop efficient DNNs. These include manually designed DNNs such as ResNet [9] and MobileNet [12, 37]; model compression techniques such as pruning [10, 52] and quantization [6, 25, 33] to reduce the model size; and neural architecture search (NAS) [5, 34] to automatically identify efficient DNNs within a search space.
• **Target platform's system support.** Systems that better utilize hardware resources can execute a model more efficiently. For example, on mobile devices, to complement efficient DNN design, prior works [17, 18, 23] design advanced compilation and runtime methods to coordinate heterogeneous resources (CPU/GPU/NPU/DSP), greatly boosting DNN inference performance. NN-Stretch [48] further adapts DNNs to align with the hardware architecture.

Compared with the progress in mobile devices, **efficient DNN inference for MCUs is relatively less mature**. As shown in Table 1, prior studies [1, 2, 7, 22, 27] targeting MCU platforms primarily adopted quantization techniques

Table 1: State-of-the-art works for efficient DNN inference on edge devices, including mobile and MCU platforms. (in the last column, H-agnostic indicates hardware-agnostic and H-aware indicates hardware-aware.)

	NN Model		Deployed System		Improvement		
	Architecture	Precision	Hardware Platform (H)	Processors	H-agnostic Model	H-aware Model	System Optimize
CoDL[18]	RetinaFace/YOLOv2/ VGG16/PostNet/FST	FP32	Mobile Phone (Xiaomi/Redmi/Honor)	CPU/GPU	-	-	✓
Band[17]	RetinaFace/MobileNet/ ResNet50/FSRNet	FP32	Mobile Phone (Google Pixel/Redmi/Samsung Galaxy)	CPU/GPU/ NPU/DSP	-	-	✓
μlayer[23]	GoogLeNet/SqueezeNetV1/ VGG16/AlexNet/MobileNetV1	INT8/FP16	Mobile Phone (Samsung Exynos 7420/7880)	CPU/GPU	-	-	✓
NN-stretch[48]	ResNet-34/50,RegNetX-1.6GF/4GF, EfficientNet-Lite4/B5	INT8/FP16	Mobile Phone (Xiaomi/Pixel)	CPU/GPU/DSP	-	✓	✓
Rusci et al. [36]	MobilenetV1	INT-2/4/8	MCU (NUCLEO-F767ZI@ARM-M7)	CPU	✓	-	-
TinyEngine[27]	MobileNetV2/ProxylessNAS/ MnasNet	INT8	MCU (STM32F412@M4, STM32F746/65, STM32H743@M7)	CPU	-	-	✓
RT-MDM[22]	self-trained CNN	INT8	MCU (ArduinoNano@M4)	CPU	-	-	✓
MoteNN[2]	NASNet/DARTS/MobileNetV2/ MCUNetV2/BERT-Tiny	INT8	Intel(R) Xeon(R) Silver 4210R	CPU	-	-	✓
Liberis et al.[26]	SwiftNetCell/MobileNetV1	INT8	MCU (NUCLEO-F767ZI@M7)	CPU	-	-	✓
QuantMCU[45]	self-trained CNN	INT2-INT8	MCU(ArduinoNano@M4, STM32H743@M7)	CPU	✓	-	-
CMix-NN[1]	MobilenetV1	INT-2/4/8	MCU (STM32H743@M7)	CPU	-	-	✓
MCU-MixQ[7]	VGG-Tiny/MobileNet-Tiny	INT2-INT8	MCU (STM32F746@M7)	CPU	-	✓	✓
AdaptQNet (Ours)	MobileNetV2/V3, EfficientNet	**FP32/16 & INT-2/4/8**	MCU (STM32H747@M7+M4)	**CPU w/ FPU & dual-core**	-	✓	✓

Table 2: Specifications of MCUs from the top five manufacturers [39]. (✓* indicates the presence of a single-precision FPU, and "-" denotes models that lack internal Flash storage and rely on external Flash.)

Vendor	Chipset Model	Processor	FPU?	Release	Clock	Flash	RAM
STM	STM32H7(45BI)	dual Cortex-M7+M4	✓	2019.6	M7-480MHz, M4-240MHz	2048kB	1024kB
Infineon	XMC7100(D-F100K2112AA)	dual Cortex-M7	✓	2022.11	dual M7-250MHz	2112 kB	384kB
NXP	i.MX-RT(1170)	dual Cortex-M7+M4	✓	2021.1	M7-1GHz, M4-400MHz	-	2048kB
Microchip	SAM4Cx	dual Cortex-M4	✓	2022.6	dual M4-120MHz	2048kB	304kB
Espressif	ESP32-P4(NRW16)	dual RISC-V	✓*	2023.1	dual RISC-V-400MHz	-	768kB

to produce integer-only models. They primarily use an arithmetic logic unit (ALU) within a single CPU core to execute DNNs. Another challenge for MCUs is their stringent resource constraints, typically limited to hundreds or, at most, thousands of kilobytes. Thus, even when mixed-precision models are considered [1, 7, 36, 45], precision options are usually confined to INT2-8.

However, we observed that FPUs are widely available in modern MCU SoCs (Table 2). With hardware support, the latency of FP operations can be comparable to that of INT operations (Table 3). Our study further confirms that enabling some critical operations to run with FP precision allows others to be even lower bit widths. This mixed FP-INT model can be even smaller than INT8 model with higher accuracy (Figure 2). Besides underutilized FPUs, many modern MCUs also feature multiple cores. For instance, STM32H747 has ARM Cortex M4+M7 cores, each equipped with its own ALU and FPU. The extra core can perform additional computations to boost DNN capacity, and can be executed in parallel to minimize the extra latency cost (Figure 3).

Thus, this work proposes AdaptQNet to automatically identify efficient DNNs that effectively leverage modern MCUs' FPU and multi-core capabilities, which have rarely been explored in prior studies, to advance efficient DNN inference on MCUs. Basically, AdaptQNet preserves the original DNN meta-structure, which has been proven to be effective in full-precision settings, and adjusts the precision configuration across layers to identify an optimal mixed-precision model, potentially including both FP and INT operations that can be executed by ALU/FPU on available cores (Figure 5).

Despite the adjustable components being limited to precision without altering operator dimensions, key challenges must be addressed owing to the vast search space, which arises from the compound effect of multiple precision options and potential multi-core utilization, as detailed in §2.3. This necessitates a carefully designed search strategy. AdaptQNet primarily features a two-stage neural architecture search (NAS) framework to address the challenge. The first stage focuses on single-core cases, determining the precision configuration for various layers in the DNN. Based on the results,

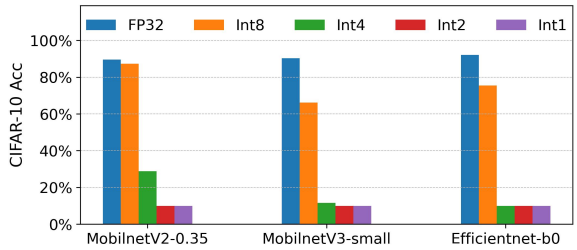


Figure 1: Accuracy of DNN models from QAT.

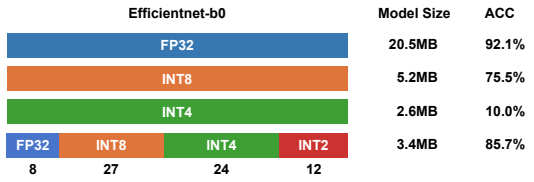


Figure 2: Mixed FP-INT EfficientNet from AdaptQNet.

the layers that are likely to benefit from utilizing additional core computations are identified, and a dedicated second-stage search is conducted for these layers. By decoupling the search into two stages, the vast original search space is effectively reduced. Our main contributions are as follows.

- To our best knowledge, we are the first to identify the underutilization of FPU and multi-core capabilities that have long existed on MCUs.
- We propose AdaptQNet, a two-stage differentiable NAS-based approach combined with rigorous latency modeling for various DNN operators at various precision on single-/multi-cores. AdaptQNet identifies accurate and efficient DNNs optimized for MCUs with FPUs and multi-cores.
- Our evaluations demonstrate that AdaptQNet achieves significant improvements across different DNNs and benchmarks. Compared to INT8 QAT models, AdaptQNet achieves an average accuracy improvement of 7.6% across all cases. For MobileNetV3-SMALL on CIFAR-10, the accuracy improves from 62.7% to 75.9%. In comparison to mixed-INT models, AdaptQNet also shows significant improvements. For MobileNetV2 on CIFAR-10, AdaptQNet achieves an accuracy of 83.8%, surpassing the mixed-INT model's 80.4%. Similarly, for MobileNetV3-SMALL on CIFAR-10, AdaptQNet reaches 75.9% accuracy compared to the mixed-INT model's 63.1%. These improvements are achieved while maintaining comparable or better compression ratios and latency reductions. For EfficientNet-B0 on Mini-ImageNet, AdaptQNet achieves a compression ratio of 0.189x with 78.5% accuracy, outperforming the mixed-INT model's 74.4% accuracy at 0.223x compression ratio.

2 Motivation and Challenges

In this section, we illustrate the motivation and challenges involved in developing accurate and efficient mixed FP-INT DNNs for modern MCU platforms.

2.1 Limitation of Unified Quantization

Unlike servers and mobile devices, MCUs face stringent memory constraints, as listed in Table 2. Thus, there is a high demand for aggressive model compression. However, most DNNs struggle to maintain their original accuracy when they are significantly compressed. Figure 1 shows our experiments applying quantization-aware training (QAT) [15] to produce quantized MobileNetV2/V3 [11, 37] and EfficientNet [44]. While INT models achieve lower memory footprints than FP32 models, this reduction often comes with accuracy degradation to varying degrees. Nevertheless, we found that elevating some critical layers to FP precision helps preserve model accuracy, whereas less important weights can be further compressed into lower-bits, thereby reducing the overall memory footprint. Figure 2 shows a mixed FP-INT model found by our method, achieving better accuracy-efficiency tradeoff superior to that of INT8 model.

Table 3: Operator Latency (ms) on STM32H747@M7.

Operator	FP32 w/o FPU	FP32 w/ FPU	INT8	INT4	INT2
Conv	196.97	19.70	4.47	3.32	2.15
DW-Conv	128.59	12.85	3.27	2.23	1.43
MaxPool	1.45	0.14	1.14	N/A	N/A
FC	0.52	0.05	0.03	N/A	N/A

2.2 Underutilized Processing Units

Mixed-precision is not a novel concept, but recent studies [1, 7, 36, 45, 45] on MCUs are often restricted to mixed INT precisions (INT2-INT8), primarily relying on ALU for computation. As shown in Table 2, modern MCUs widely feature FPUs; however, these remain underutilized in prior research. We empirically measured the execution latency of typical DNN operators on MCUs, and presented the results in Table 3[1]. With FPU hardware support, FP operations can execute comparable to their integer counterparts. It is noted that Conv/FC operators are dense-arithmetic operations, highly sensitive to precision and FPU support. This opens up the opportunity to mixed FP-INT models on MCUs, with better accuracy-efficiency balance.

Unlike GPUs and CPUs on mobile devices, ALU and FPU on MCUs are integrated differently. Figure 3(a) illustrates the typical architecture of two-core MCU, where each core is equipped with an ALU and FPU. Each core has local memory

[1]INT4/2 cases were measured using CMixNN [1], which supports Conv operators but not max-pooling or fully connected (FC) operators.

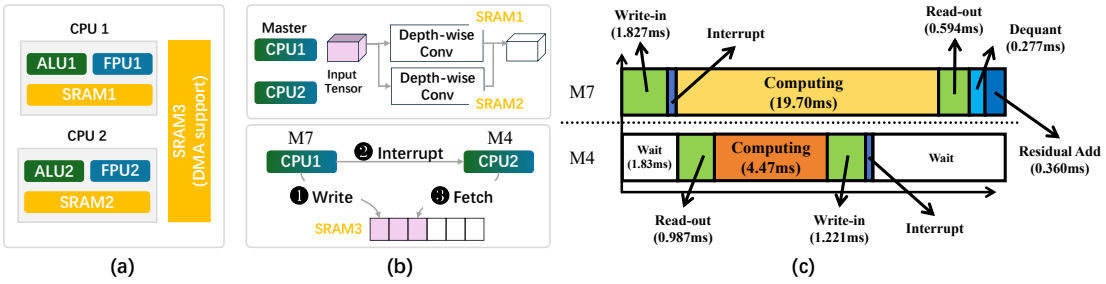


Figure 3: Dual-core MCU modeling: (a) Typical architecture for a dual-core MCU; (b) Interrupt-based coordinated execution model; (c) Latency modeling empirically evaluated for a 3x3 convolution operation using CubeAI, where a Cortex M7 core executes in FP32 precision, and a M4 core executes in INT8 precision.

for model execution. GPUs and CPUs on mobile devices are independent processing units with distinct memory spaces, resulting in non-negligible synchronization overhead, as highlighted by prior works [17, 18]. However, ALU and FPU on a single-core MCU, e.g., ALU1 and FPU1 in Figure 3(a), are closely coupled, with minimal synchronization overhead, but they cannot execute simultaneously.

For dual-core MCUs, however, ALU and FPU from different cores, e.g., ALU1 and FPU2 in Figure 3(a), can execute in parallel to accelerate DNN inference. To coordinate execution, we can leverage a dedicated memory space that supports 2-core access to facilitate inter-core communication. As shown in Figure 3(b), core 1 ❶ writes the input tensor to the shared memory; ❷ sends an interrupt signal to core 2; then core 2 ❸ fetches the tensor from shared memory and performs its computation. After sending the interrupt, core 1 executes its own computation. When core 2 completes its computation, it uses a similar interrupt-based scheme to send the results back to core 1. Figure 3(c) presents the empirical results of dual-core execution, where M7 core serves as core 1 performing an FP32 3x3 Conv, while M4 core acts as core 2 executing an INT8 3x3 Conv, with 32x32x3 input tensor. Homogeneous dual-core MCU, e.g., two M7 cores executing an FP32 Conv and an INT8 Conv, exhibit a similar pattern as in Figure 3(c). Multi-core execution can enhance model capacity without incurring substantial overhead, if parallel execution is effectively implemented.

2.3 Design Challenge

The composite complexity introduced by FPU and multi-core execution creates unique problem space. FPU/ALU's functioning on multi-core MCUs differs significantly from GPU/TPU/FPGA architectures [4] and is not addressed in prior works[1, 7, 36, 45, 50] that only utilize ALUs on single-core MCUs. Besides, the search space is vast, and we find that optimal models cannot be effectively discovered using a straightforward model search strategy.

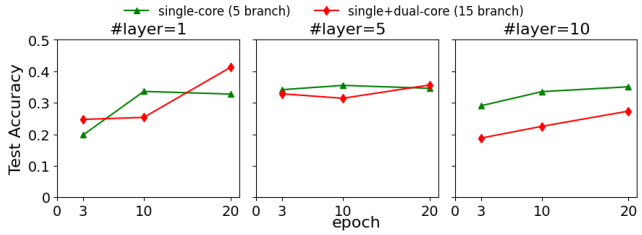


Figure 4: Pilot Study: Optimization Efficiency of SuperNet with Various #Precision Branches.

For a DNN model $\{W_1, W_2, ..., W_{N_L}\}$ with N_L layers, each layer has precision candidates S_p, resulting in $\|S_p\|^{N_L}$ precision configurations. Furthermore, if each layer leverages multi-core capacity, assuming N_c cores, we can execute up to N_c precision branches in parallel. Then, we have additional configurations for each layer as $\sum_{i=2}^{N_c} C(\|S_p\|, i)$, where $C(\cdot,\cdot)$ represents the combinatorial function. Thus, the total number of candidate precision in the search space $\mathcal{S}$ is:

$$\|\mathcal{S}\| = (\|S_p\| + \sum_{i=2}^{N_c} C(\|S_p\|, i))^{N_L} \quad (1)$$

If S_p = {FP32, FP16, INT8, INT4, INT2}, $N_c = 2$, and $N_L = 52$ (MobileNetV2), then $\|\mathcal{S}\| = (6+15)^{52} \approx 5.69 \times 10^{68}$.

We conduct a pilot study to empirically demonstrate the challenge of model search within such a large space. We construct DNN models with varying depths (1/5/10), where each layer consists of a Conv (kernel size 3×3) followed by ReLU activation. Each layer is then expanded to multiple precision branches with two cases: (a) single-core case (supernet@5): we construct a super-network with 5 precision options: FP32, FP16, INT8, INT4, INT2 (see Figure 6) (b) single+dual-core case (supernet@15): The super-network is expanded to include pairwise combinations, such as FP32+INT8, FP16+INT4 (see Figure 7 (b)), yielding 15 options in total (5 single-precision + 10 combination-precision). We apply NAS to search for an

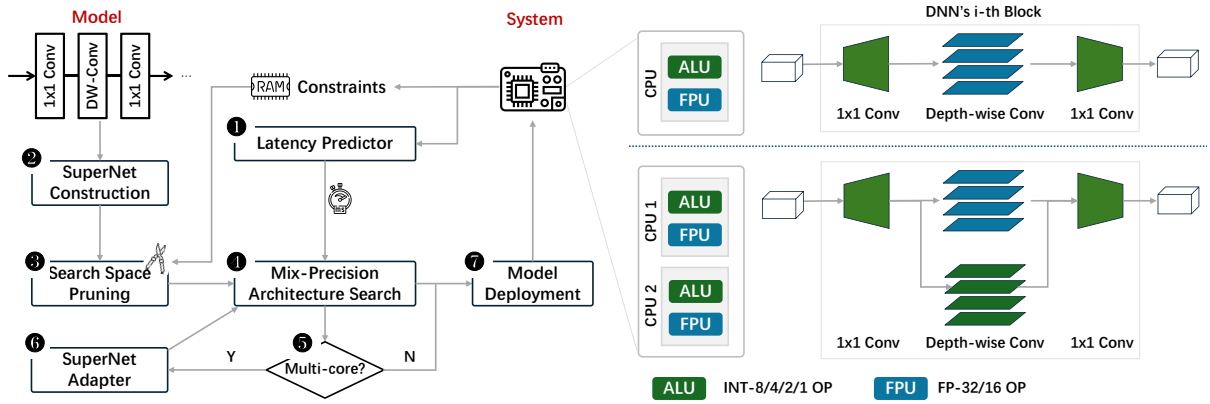


Figure 5: AdaptQNet Overview.

optimal model, training the constructed supernet on CIFAR-10 for multiple epochs (3/10/20) with Adam optimizer. The search space of Supernet@15 is a superset of that of Supernet@5, however, directly searching within Supernet@15 not only requires more GPU memory and computation time, but also suffers from lower optimization efficiency, often resulting in a sub-optimal sub-network. As shown in Figure 4, as network depth increases to 5 layers, supernet@15 shows slower optimization progress, requiring 20 epochs to reach 0.3562, whereas supernet@5 achieves similar performance (0.3551) in just 10 epochs. This trend becomes more pronounced at depth 10, where supernet@15 struggles to maintain performance (dropping to 0.2734) even with 20 epochs, while supernet@5 remains stable at 0.3506. In practical DNNs with dozens of layers, e.g., MobileNetV2 with 50+ layers, search within a supernet with all precision options is neither efficient nor achieving an optimal final model.

Existing methods cannot readily address this problem. Tao et al.[45] employ mixed precision to address a specific issue, differing from our focus: they identify less sensitive patches and enforce lower precision (e.g., INT4) to reduce patch-induced latency. Rusci et al.[36] adopt an iterative strategy to reduce bitwidth layer by layer; however, it is difficult to generalize for handling combination-precision in dual-core execution in our case. Pham et al.[31] propose weight sharing among all child models to improve NAS efficiency; however, in our case, the child models correspond to the same operator instantiated with different precisions (or their combinations), as illustrated in Figure 7. Xu et al. [50] optimize operator placement across multiple cores without altering the model itself, making their approach complementary to ours.

Our Idea. Inspired by above pilot study, we propose a two-phase NAS method (Figure 5). In first phase, we restrict the search to selecting single precision branch per layer (Figure 6). This eliminates the need to consider multi-branch cases at this stage, significantly reducing initial search space, and making it easier to converge to an optimal model. In second phase, we build upon the previous results. If a specific precision branch demonstrates an overwhelming advantage over others for a given layer, further exploration of multi-branch cases for that layer is deemed unnecessary. In such case, multi-core execution would introduce unnecessary synchronization/memory overhead without much gain. Conversely, if multiple precision branches show comparable gains, we explore two or more branch cases to identify the better configuration (Figure 7). For both phases, we perform search space pruning by applying hard RAM constraints to eliminate infeasible branches or combinations, thereby reducing the overall search time. By disentangling the search space into two phases, we not only enhance search efficiency but also improve search effectiveness in identifying a mixed-precision DNN that fully exploits FPU and multi-cores.

3 AdaptQNet System Design

Figure 5 illustrates AdaptQNet's complete workflow that involves two main roles: the model to be optimized and the target system where the model will be deployed. Before model optimization, we (1) identify all the constraints imposed by the system hardware and conduct hardware profiling to construct a latency predictor for various primitive operators executed on the processing units of the target system. Based on the meta-structure of the model, we (2) construct a supernet that represents the entire search space with candidate precision configurations. We (3) apply hardware RAM constraints to prune the search space, and then (4) conduct a phase-one mix-precision architecture search. Following this, we will (5) examine the phase-one results to determine whether the model could potentially benefit from multi-core enhancements. If so, we will (6) adapt the original supernet to incorporate precision combinations for multi-core cases on selected layers and (4) execute a phase-two mixed-precision architecture search. Finally, we (7) sample

the optimal model from the supernet and dispatch the operations to heterogeneous processing units for inference.

3.1 System Constraints

The major constraint posed by MCUs is memory, specifically the flash size for storing model parameters and the RAM size for holding dynamic activations.

Flash Constraints. In MCU-based TinyML systems, flash memory stores both model parameters, including weights and biases, as well as auxiliary components such as operator libraries and code, which can be categorized as follows:

$$M_{\text{flash}} = M_{\text{para}} + M_{\text{aux}} \tag{2}$$

where M_{para} and M_{aux} represent the memory footprints of the model parameters and the auxiliary components, respectively. M_{para} is determined by the number of parameters and bits used to represent each parameter, that is, precision, and can be expressed as $M_{\text{para}} = N_{\text{p}} \times \text{bw}_{\text{p}}$, where N_{p} and bw_{p} denote the number of parameters and bit width, respectively. For example, if the model uses FP32, $\text{bw}_{\text{p}} = 32$ bits $= 4$ bytes, whereas for INT8, $\text{bw}_{\text{p}} = 8$ bits $= 1$ byte. Thus, quantization [6, 25, 29, 35] is widely used for compressing DNNs to reduce flash usage.

Although unified quantization has become the de facto standard for model compression, recent research has shown that different DNN layers exhibit varying quantization sensitivities. Thus, mixed-precision solutions [32, 33] offer potential advantages. In this case, for a DNN with L layers containing learnable parameters $\{W_1, W_2, ..., W_L\}$, the flash usage can be characterized as follows:

$$M_{\text{flash}} = \sum_{i=1}^{L} \left(N_{\text{p}}^{i} \times \text{bw}_{\text{p}}^{i} \right) + M_{\text{aux}} \tag{3}$$

where N_{p}^{i} and bw_{p}^{i} represent the number of parameters and bit width (precision) for the i-th layer, respectively.

RAM Constraints. In addition to static model parameters, RAM is required to store intermediate output tensors (activations) during the model inference. For sequential DNNs, inference libraries, e.g., X-Cube-AI [40] allocate buffers based on the maximum activation tensor size across all operators. Specifically, we have:

$$\begin{aligned} M_{\text{RAM}} &= M_{\text{buff}} + M_{\text{I/O}} + M_{\text{lib}} \\ &= \text{Max}(M_{\text{act}}^{1}, M_{\text{act}}^{2}, \ldots, M_{\text{act}}^{L}) + M_{\text{I/O}} + M_{\text{lib}} \end{aligned} \tag{4}$$

where M_{act}^{i} denotes RAM for i-th operator/layer, $M_{\text{I/O}}$ denotes RAM for Input/Output and M_{lib} denotes RAM for inference library. During inference, the system repurposes the reserved input/output RAM to store intermediate tensors, thereby optimizing RAM usage. This strategy is taken into account when determining the search space.

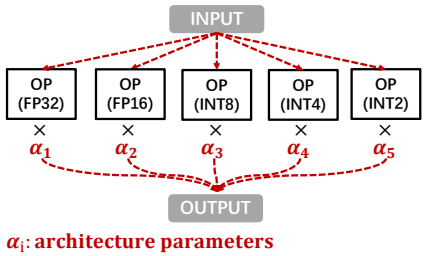


Figure 6: Primitive to construct supernet for mix-precision architecture search.

To deploy a DNN model on an MCU, we should satisfy:

$$M_{\text{flash}} \leq \text{FLASH\_MAX}; M_{\text{RAM}} \leq \text{RAM\_MAX} \tag{5}$$

where FLASH_MAX and RAM_MAX denote the maximum available flash and RAM, respectively. The RAM usage modeling for multi-core cases is elaborated in §3.3.

3.2 Mixed-Precision Architecture Search

Search Space (SuperNet Construction). AdaptQNet provides a mixed-precision NAS approach called *PrecisionNAS* that automatically searches for the best-precision configuration across layers. We focus on CNN-based backbones, such as MobileNets [11, 37] and EfficientNet [44]. Inspired by prior work [49], we design the search space using the primitives illustrated in Figure 6. Specifically, for each selected layer in a full-precision trained model, we first perform post-training quantization (PTQ) to generate its different precision branches, and then combine all branches into a unified architecture. Each branch is associated with an architecture parameter α_i. By integrating all the expanded layers, the resultant model, referred to as the supernet NN_{sup}, spans from the first layer's input to the final layer's prediction output. In addition, we retain the quantization and dequantization operations within each precision branch, while the residual branch maintains identical precision between input and output without requiring any additional conversion. During deployment, the dequantization and merge operations are executed by a more capable core, as in Figure 3(c). We enforce unified bit-width for weights/activations within each layer/operator, as prior works [55] suggest that maintaining consistency is beneficial. Many viable subnets exist. Generally, higher-precision branches improve model capacity, and with FPU support, the latency gap can be reduced, but still have a higher memory footprint. Our system aims to balance these trade-offs and automatically search for the optimal precision configuration, ensuring that the resultant subnet achieves high capacity while effectively utilizing the underlying hardware to enable efficient inference.

Algorithm 1 Search Space Pruning

Input: Supernet NN_{sup}, RAM_MAX
Output: Simplified Supernet NN'_{sup}
for each precision branch $b_p \in \text{NN}_{\text{sup}}$ **do**
  if $M^p_{\text{RAM}} >$ RAM_MAX **then**
    Prune precision branch b_p from NN_{sup}
  end if
end for

Algorithm 2 Searching Algorithm

Input: Pruned supernet NN_{sup} with model parameters w and architecture parameters α , cost function/coefficient $C(\cdot)$ and λ, training/validation dataset $\mathcal{X}_{\text{train}}$ and $\mathcal{X}_{\text{val}}$
Output: Optimized supernet NN'_{sup}.
for epoch $e = 0, \ldots, N$ **do**
  // Update w on $\mathcal{X}_{\text{train}}$
  pred = $\text{NN}_{\text{sup}}(w, \alpha, \mathcal{X}^e_{\text{train}})$
  loss = get_loss(pred, $C(\cdot)$, λ)
  Perform backpropagation to update w (not update α)
  // Update α on $\mathcal{X}_{\text{val}}$
  pred = $\text{NN}_{\text{sup}}(w, \alpha, \mathcal{X}_{\text{val}})$
  loss = get_loss(pred, $C(\cdot)$, λ)
  Perform backpropagation to update α (not update w)
end for
Return: Optimized supernet NN'_{sup}.

Search Space Pruning. Unlike previous hardware-agnostic approaches [3, 49], our work focuses on MCUs with stringent memory constraints. Thus, we prioritize memory restrictions upfront to eliminate infeasible branches in the supernet, thereby avoiding unnecessary computations during the search process. As described in Equation 4, RAM usage is dominated by the layer with the maximal RAM requirements. Thus, we can use the layer-wise maximum RAM usage as the pruning criterion: if a single layer of a candidate incurs RAM usage exceeding the total RAM limits, it can be safely excluded from the search space, thereby reducing the subsequent search time. A detailed procedure is presented in Algorithm 1. The primary reason for using RAM as the pruning criterion, rather than flash, is its layer-wise maximum usage indicated in Equation 4. Flash memory, which accumulates across multiple layers (Equation 3), is more challenging to use as a criterion for pruning. We ensure that no potentially optimal subsets within the memory constraints are wrongly excluded. Due to space limitation, please refer to Figure 11 in Appendix to see the remaining precision branches in the supernet across various precisions after pruning using Algorithm 1.

Search Algorithm. Given the pruned supernet, our goal is to search for an optimal configuration of the architectural parameters α in Figure 6 across different layers. The overall search procedure is presented in Algorithm 2.

The architecture parameter α serves as a branch mask. Formally, for a pair of input activation v_i and output activation v_j in Figure 6, we define:

$$v_j = \sum_{i,k} \alpha^{ij}_k \cdot v_i \cdot w^{ij}_k \tag{6}$$

where $\alpha^{ij}_k \in \{0, 1\}$ represents the mask for k-th branch between between v_i and v_j, and $\sum_k \alpha^{ij}_k = 1$. We then adopt a differentiable NAS [49] approach to convert the supernet into a stochastic version in which branches are executed stochastically. For each branch, execution occurs when α^{ij}_k is sampled to be 1. To enhance training stability, instead of hard sampling, where only one branch is executed, we apply the Gumbel-Softmax trick [16, 28] to enable soft sampling. Each branch is assigned a parameter θ^{ij}_k, and we have:

$$\alpha^{ij}_k = \text{GumbelSoftM}(\theta^{ij}_k | \theta^{ij}) = \frac{\exp((\theta^{ij}_k + g^{ij}_k)/\tau)}{\sum_k \exp((\theta^{ij}_k + g^{ij}_k)/\tau)} \tag{7}$$

where g^{ij}_k is sampled from Gumbel(0,1), and a temperature coefficient τ is used to control the behavior of the Gumbel-Softmax distribution. When $\tau \to \infty$, α^{ij}_k becomes a continuous random variable that follows a uniform distribution. As described in Equation 6, all branches are executed, and their outputs are averaged. Conversely, as $\tau \to 0$, α^{ij}_k approaches a discrete random variable following a categorical distribution, and only one branch in Equation 6 is sampled and executed. This also makes the supernet and its loss function directly differentiable with respect to θ^{ij}_k, thereby enabling efficient optimization during training.

To search for an accurate and efficient model, we define the loss function for the search process as follows:

$$\text{Loss} = \mathcal{L}_{task} + \lambda \cdot C_{latency} \tag{8}$$

where $\mathcal{L}_{task}$ represents task-specific loss, such as cross-entropy for classification tasks, and $C_{latency}$ measures the latency cost of the DNN, and λ is a balancing hyperparameter that adjusts the trade-off between accuracy and efficiency.

For a DNN with L layers, since MCU systems lack complex operation fusion as seen in mobile systems [56], the overall latency is calculated as an accumulation across layers $C_{latency} = \sum_{i=1}^{L} C^i_{latency}$, where $C^i_{latency}$ represents the latency of the i-th layer.

In a supernet with multiple precision branches, the latency for each layer is calculated as the weighted average of each precision option, which can be expressed as:

$$C^i_{latency} = \sum_k \alpha^{ij}_k \cdot C^i_{latency}(k) \tag{9}$$

where α_k^{ij} represents the weighting factor for k-th branch in i-th layer and $C^i_{latency}(k)$ is the latency of k-th branch.

Hardware Profile and Latency Predictor. $C^i_{latency}(k)$ depends on the precision, operator type, and hardware configuration (e.g., CPU and FPU). To integrate precise latency values into the loss function for training optimization, we must accurately and efficiently measure $C^i_{latency}(k)$. Formally:

$$C^i_{latency}(k) = C(\#\text{OP}, \text{bit-width}) = w_i^k \times \#\text{OP} \quad (10)$$

where $\mathrm{C}(\cdot)$ models the inference time as directly proportional to the number of primitive operations (e.g., multiply-accumulate operations) and precision (bit-width), and w_i^k denotes the ratio between the actual latency and the number of operations (OP) for k-th branch in i-th layer, that can serve as a normalized metric to compare the efficiency across different precision branches.

Within a layer, different branches differ only in precision while sharing the same input/output dimensions and operator types. Therefore, to efficiently estimate w_i^k for different branches, we can measure the latency of a reference branch executed by the target hardware and interpolate proportionally for other precisions using the same hardware unit. Specifically, we can measure the latency of operators with FP32, INT8, and INT2 using the X-CUBE-AI [40] and CMix-NN [1] frameworks on the target hardware. For FP16 operators, we obtain the latency by taking half of the FP32 counterpart. To incorporate more precision options, such as INT6 or INT4, we can apply linear interpolation based on the measured latencies of the existing precision levels.

3.3 Incorporating Multi-Core Capability

After the phase-one search, we obtain a supernet with optimized architecture parameters, as shown in Figure 7(a). For a target system with a single core, the best-precision branch with the highest associated weight from each layer can be sampled to form the subnet for deployment. However, if the hardware features two cores, we can leverage duplicated operators with different precisions to compensate for the accuracy loss introduced during quantization. It is worth noting that natively selecting the best two branches for dual-core execution is incorrect, as the latency and RAM usage modeling for dual-core scenarios differ significantly from the single-core case. Therefore, a dedicated round of NAS is required for proper optimization.

New Search Space (SuperNet Adapter). As shown in Figure 7(b), we adapt the previous supernet to construct a new search space aimed at finding efficient DNNs that better utilize multi-core capacities. We use the trained maximum α of each layer to determine whether to further explore multi-core utilization for this layer. If one layer has a small $max(\alpha)$ compared with the other layers, it indicates that the layer has more exploration space. However, we find that the α distribution varies and is highly sensitive across models. Thus, simply applying a threshold on α for layer selection is unsatisfactory, as it either retains too many layers (low search efficiency) or too few (missing better candidates). Thus, we adopt a top-k strategy: all layers are sorted by their $-\max(\alpha)$ in descending order, and a cut-off ratio k is applied. This is agnostic to the specific α values while providing predictable control over the size of the two-core search space. For selected layers, we further sample all combinations of branches based on the number of available cores (e.g., two in this case). For others, we retained only the best branch with $max(\alpha)$.

It is important to note that although dual-core hardware provides the potential to execute a two-branch DNN, it does not guarantee that two-branch configurations are always better than single-branch configurations. For instance, two-branch models may exceed memory constraints, rendering them infeasible, or may incur higher latency with minimal accuracy gains. Therefore, we still include the single branch in the search space to ensure a comprehensive optimization. Similar to PrecisionNAS described in §3.2, each single/dual-precision configuration (branch) is associated with an architectural parameter α_i for optimization.

Multi-core RAM Usage Modeling. Before applying PrecisionNAS to the new supernet, we also perform search space pruning as outlined in Algorithm 1 using RAM usage as the pruning criterion. However, for multi-branch DNNs, the input tensors associated with a branching point can be shared by multiple operators. Therefore, unlike in sequential DNNs, input tensors cannot be evicted immediately from RAM until all the associated branches complete their computations.

$$M_{\text{RAM}} = M_{\text{buff}} + M_{\text{branch}} + M_{\text{I/O}} + M_{\text{lib}} \quad (11)$$

where M_{branch} denotes the extra memory footprint of the tensors associated with branching, differing from Equation 4.

Multi-core Latency Modeling. To guide the search process over the new supernet shown in Figure 7, accurate latency measurements for two-branch cases should be incorporated into the loss function (Equation 8). As illustrated in Figure 3, for two-core coordinated execution, we calculate the latency as the total span of computations, considering the maximal possible overlap between their execution timelines. For ALU/FPU assignment across cores, in homogeneous systems (e.g., M7+M7), configurations such as ALU1+FPU2 and FPU1+ALU2 are equivalent; in heterogeneous ones (e.g., M7+M4), heavier operators (e.g., FP) are assigned to the higher-capacity core (M7), guided by our latency profiling.

4 Evaluation Methodology

Hardware Prototype. We use the STM32H747 dual-core MCU as our evaluation platform, as shown in Figure 8(a). It

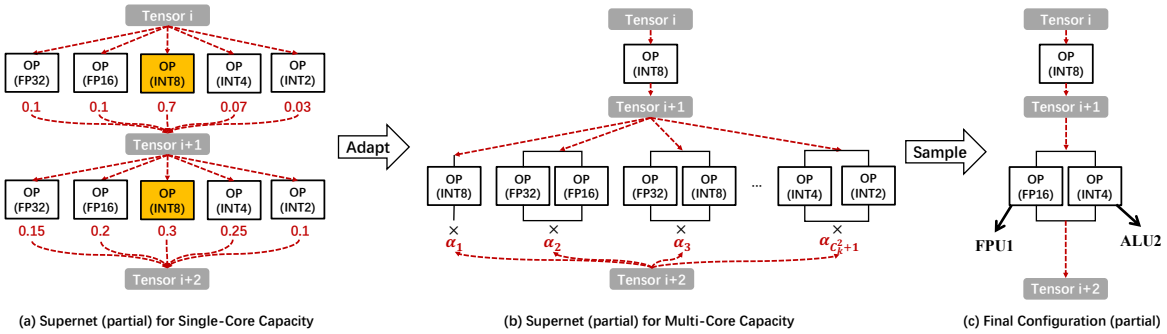


Figure 7: SuperNet Adapter: For layers with a dominating branch, sample only the best one. Otherwise, sample both the best and combinations #cores (e.g., 2), then conduct PrecisionNAS on the new supernet.

features an ARM Cortex-M7 core and a Cortex-M4 core, with 2048KB Flash and 1024KB RAM. We use CubeMX [41] for hardware configuration. To deploy mixed FP-INT models, we enable the FPU on both cores. For M7 core, we set bits 20 to 23 of the Coprocessor Access Control Register (CPACR) to 1 using code snippet `SCB->CPACR |= ((3UL«20)|(3UL«22))`. Similarly, for M4 core, we configure the relevant CPACR bits to enable its FPU. For low-bitwidth, Cortex-M4/M7 are both SIMD-enabled units that can pack INT4/2 within INT8 registers and perform vectorized operations for efficiency. M7 core operates at 400MHz, while M4 core runs at 240MHz. To facilitate dual-core co-execution, we designate SRAM4 (64KB) in power domain D3 [42] as the tensor exchange region, as it is accessible by both cores. We primarily use X-CUBE-AI [40] for DNN model deployment and leverage OpenAMP [30] to implement multi-core coordination. We run DNNs on a bare-metal MCU without any OS. Hardware modeling of RAM and latency required by AdaptQNet is universally applicable across MCUs, not limited to our evaluation board. Moreover, our framework is inherently generalizable and can be readily extended to other MCUs through appropriate profiling.

Evaluation Model&Benchmarks. Since CNNs are predominantly used in MCU applications, we evaluate three well-known lightweight CNN models: EfficientNet [44], MobileNetV2 [37], and MobileNetV3 [11]. We will consider others DNNs in future. For benchmarking, we adopt two representative dataset, CIFAR-10 [24] and MINI-IMAGENET [46]. The maximum available RAM for AI inference on our hardware STM32H747 is 784 KB. Accordingly, we set the RAM constraint to 512 KB for CIFAR-10 and 784 KB for Mini-ImageNet, considering their differing input resolutions (32×32 and 84×84). Each dataset is divided into training, validation, and test sets. Training set is used to learn model weights with a 256 batch size, validation set is used to evaluate training objectives and search for architecture parameters with a 200 batch size, and test set is used for final evaluation. We reserve 4,000 samples from validation set as test set for CIFAR-10, and use an overall split ratio of 7:2:1 for Mini-ImageNet.

Baselines and Evaluation Metrics. We compare our proposed method with the following baselines:

- Original Model (FP32): This is the full-precision model, which generally achieves the highest accuracy but is less computationally feasible and efficient on MCUs.
- QAT Model (INT8): This is a fully INT8 model obtained by uniformly quantizing FP32 models using QAT [29] technique.
- Mixed-INT Model (INT8/4/2): As prior works [7, 36, 53] are not open-sourced, we implement Mixed-INT models following their descriptions, with the precision space restricted to integer only and the number of candidates is constrained to powers of two to facilitate computation.

We ran all baselines and our method under the same hardware constraints and benchmark data. Upon model convergence, we selected models that achieved the optimal accuracy and efficiency while meeting memory constraints for a fair comparison. Specifically, we use test set accuracy, compression ratio (CR) relative to full-precision models, average inference latency, and effective bitwidth (Eff-BW = $BW_{ref} \times CR$) as evaluation metrics, where BW_{ref} denotes the bitwidth of the reference model, which is 32 in our evaluation.

Model Search Settings. We implement model training and architecture search algorithm using PyTorch and run them on a server equipped with NVIDIA A6000 GPU with 48GB memory. We use separate Adam optimizers with cosine learning rate (lr) scheduling for updating model weights (w) and architecture parameters (α), with hyperparameters setting as shown in Figure 9(a). Additionally, to stabilize the training process, we conduct a warm-up phase for model weight training, 5 epochs for CIFAR-10 and 10 epochs for Mini-ImageNet, before proceeding with architecture parameter searching. For regularization term λ in Equation 8, we set it relatively small so that the initial regularization term $\lambda \cdot \mathcal{C}_{latency}$ is about 20% of the task loss $\mathcal{L}_{task}$, achieving a final model that balances both efficiency and accuracy. BatchNorm/ReLU is

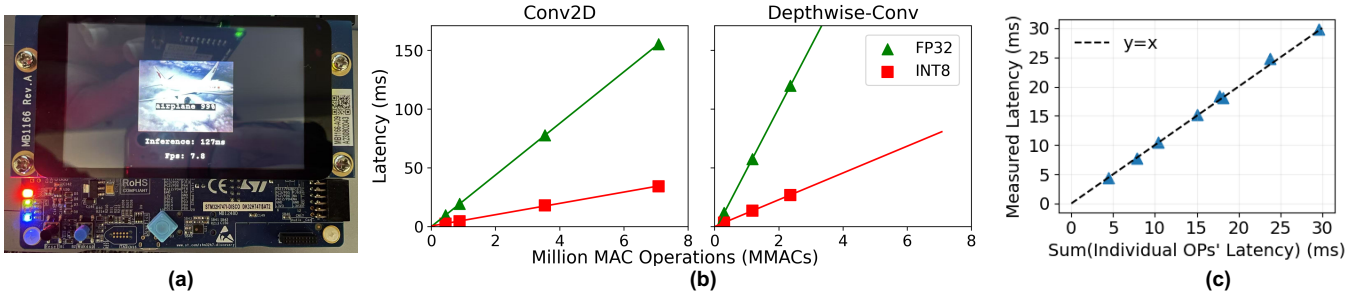
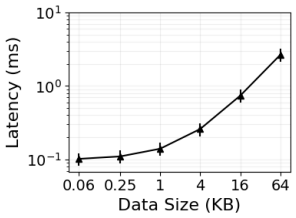


Figure 8: (a) Hardware Platform: STM32H747 Dual-Core MCU with enabled FPUs; (b) Computation Latency Profiling (Per-Operator): Conv2D/DW-Conv under FP32/INT8 precisions; (c) DNN Computation Latency Modeling: measured latency v.s. sum of individual operators' predicted latency.

Setting \ Dataset	CIFAR-10		Mini-ImageNet	
	w	α	w	α
Optimizer	Adam w/ cosine lr scheduling			
lr (initial)	1e-3	1e-4	5e-4	5e-5
lr (min)	1e-6	1e-5	5e-7	5e-6
decay	1e-4	1e-4	5e-4	1e-4
gradient clip	1.0	0.1	0.5	0.05

(a)



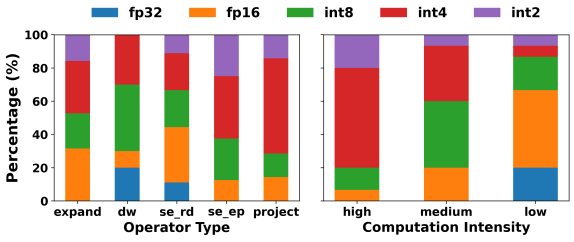


Figure 9: (a) Hyper-parameters for Model Search; (b) Inter-core Communication Latency Profiling: data transfer overhead in multi-core execution; (c) Precision Distribution@OperatorType (MobileNetV3-Small, CIFAR-10); (d) Precision Distribution@Computation Intensity (MobileNetV3-Small, CIFAR-10).

folded into Conv and frozen after the first epoch to mitigate quantization errors and stabilize supernet training.

5 Evaluation Result

5.1 Hardware Profiling Micro-Benchmark

Per-Operator Computation Latency Profiling. To measure per-operator computation latency, we construct DNNs with a single targeted operator, e.g., Conv2D or DW-Conv. We enumerate feasible combinations of input sizes (32×32 to 128×128) and channel numbers (3 to 64), recording their #MAC and latency. We measure the computation latency separately for M7 and M4 cores. The results for M7 core are shown in Figure 8(b). We observe a linear correlation between MMACs and latency for each operator. Since we run DNNs on a bare-metal MCU without any OS, computation latency is primarily determined by the computational complexity of operators. Figure 8(b) also shows that INT8 operators reduce execution latency by 3–4 times compared to their FP32 counterparts. Given the same MMACs, DW-Conv incurs slightly higher latency than conventional Conv2D, as X-CUBE-AI framework internally treats DW-Conv as two separate Conv, introducing additional memory access overhead. In a nutshell, for each operator at each precision, we can derive a linear function to characterize the relationship between MMACs and latency.

DNN Computation Latency Modeling. To model a DNN's overall latency, we randomly select the number of layers and also randomly select operators from candidate pools to compose DNNs, then measure their overall latency and compare against the sum of individual operators' predicted latencies based on previously derived linear functions. The results, shown in Figure 8(c), validate that a DNN's overall latency can be accurately approximated by the sum of its individual operators' latencies. In this way, we can estimate the latency overhead of a DNN, enabling more efficient model architecture search.

Inter-core Communication Latency Modeling. To leverage dual-core capacity, we must also model the communication latency induced by data transfer between two cores. We enumerate tensors of different sizes, ranging from 0.06 to 64 KB (the maximum capacity of shared RAM), and measure the latency from the sender's write operation to shared RAM until the receiver fully loads the data into its own RAM. The

Table 4: Comparison of our method against baselines across different DNN models and benchmarks.

DNN Model	Benchmark	Method	Accuracy	Model Size(MB)/CR	Latency(ms)	Eff-BW
MobileNetV2 Width/1.0	CIFAR-10	Original Model (FP32)	84.3%	8.53/1x	504.42/1x	32
		QAT Model (INT8)	83.2%	2.17/0.25x	114.03/0.226x	8
		Mixed-INT Model (INT8/4/2)	80.4%	**1.09/0.128x**	**81.74/0.162x**	4.08
		Ours (FP32/16+INT8/4/2)	**83.8%**	1.50/0.175x	102.39/0.203x	5.58
	Mini-ImageNet	Original Model (FP32)	78.3%	8.61/1x	1010.07/1x	32
		QAT Model (INT8)	76.0%	2.18/0.25x	230.81/0.228x	8
		Mixed-INT Model (INT8/4/2)	72.5%	**1.22/0.14x**	**161.90/0.160x**	4.47
		Ours (FP32/16+INT8/4/2)	**76.9%**	1.64/0.19x	219.18/0.217x	6.11
MobileNetV3 -SMALL	CIFAR-10	Original Model (FP32)	86.2%	5.81/1x	235.71/1x	32
		QAT Model (INT8)	62.7%	1.47/0.25x	51.15/0.217x	8
		Mixed-INT Model (INT8/4/2)	63.1%	1.13/0.194x	**42.67/0.181x**	6.21
		Ours (FP32/16+INT8/4/2)	**76.2%**	**1.16/0.200x**	53.44/0.227x	6.42
	Mini-ImageNet	Original Model (FP32)	79.8%	5.90/1x	467.85(1x)	32
		QAT Model (INT8)	58.5%	1.48/0.25x	102.4/0.219x	8
		Mixed-INT Model (INT8/4/2)	60.1%	1.12/0.187x	100.44/0.217x	5.97
		Ours (FP32/16+INT8/4/2)	**72.9%**	**1.06/0.178x**	**95.91/0.205x**	5.69
EfficientNet -B0	CIFAR-10	Original Model (FP32)	92.1%	20.49/1x	649.59/1x	32
		QAT Model (INT8)	75.5%	5.20/0.25x	143.56/0.221x	8
		Mixed-INT Model (INT8/4/2)	73.8%	4.43/0.216x	**129.26/0.199x**	6.92
		Ours (FP32/16+INT8/4/2)	**85.7%**	**3.41/0.167x**	134.47/0.207x	5.35
	Mini-ImageNet	Original Model (FP32)	88.1%	20.59/1x	1293.65/1x	32
		QAT Model (INT8)	72.4%	5.22/0.25x	293.66/0.227x	8
		Mixed-INT Model (INT8/4/2)	74.4%	4.59/0.223x	283.32/0.219x	7.15
		Ours (FP32/16+INT8/4/2)	**78.5%**	**3.91/0.189x**	**269.08/0.208x**	6.04

Table 5: Comparison of Phase 1 (single-core) and Phase 2 (dual-core) Models across MobileNetV2-1.0, MobileNetV3-Small and EfficientNet-B0. #2-Core denotes #layers utilizing dual-core capacity. k is threshold for supernet adaption.

DNN Model	Phase	Accuracy	Model Size(MB)/CR	Latency(ms)	Eff-BW	#2-Core	k
MobileNetV2-1.0	Phase-1 (single-core)	83.8	1.50 / 1×	102.39 / 1×	5.58	-	-
	Phase-2 (dual-core)	**84.2**	1.61 / 1.07×	109.11 / 1.06×	5.97	2	6
MobileNetV3-Small	Phase-1 (single-core)	76.2	1.16 / 1×	53.44 / 1×	6.42	-	-
	Phase-2 (dual-core)	74.6	**1.12 / 0.97×**	**52.19 / 0.98×**	**6.29**	1	5
EfficientNet-B0	Phase-1 (single-core)	85.7	3.41 / 1×	134.47 / 1×	5.35	-	-
	Phase-2 (dual-core)	82.8	**3.19 / 0.94×**	**128.75 / 0.96×**	**5.02**	1	7

results are shown in Figure 9(b). Compared to the computation latency, inter-core communication latency is minimal, <3ms. We can interpolate the measured curve to estimate it and follow the execution workflow shown in Figure 3(c) to determine the execution latency for dual-core cases.

5.2 Discovered Architectures

Overall Performance. The overall results are presented in Table 4. In terms of accuracy and efficiency trade-off, AdaptQNet demonstrates consistent advantages across various DNNs and benchmarks. On CIFAR-10, compared to INT8 and mixed-INT models, our method achieves higher accuracy while maintaining a similar latency. For Mini-ImageNet, we observe that the accuracy degradation from FP32 to INT8 is more pronounced for MobileNetV3-Small. However, our method significantly improves accuracy (from 58.5% to 72.9%) while achieving even better efficiency in model size and latency. This highlights that AdaptQNet optimizes precision at each layer to achieve better accuracy-efficiency trade-off.

Precision Distribution@ComputationIntensity. We sort all layers/operators by the number of operations in descending order and categorize them into three equally sized groups: high, medium, and low computational intensity. The precision distribution for MobileNetV3-Small is shown in Figure 9(d). Specifically, in the high-complexity group, INT4 precision dominates at 60%, followed by INT2 at 20%, showing that high-complexity layers favor medium-to-low precision configurations. This suggests that these layers are optimized

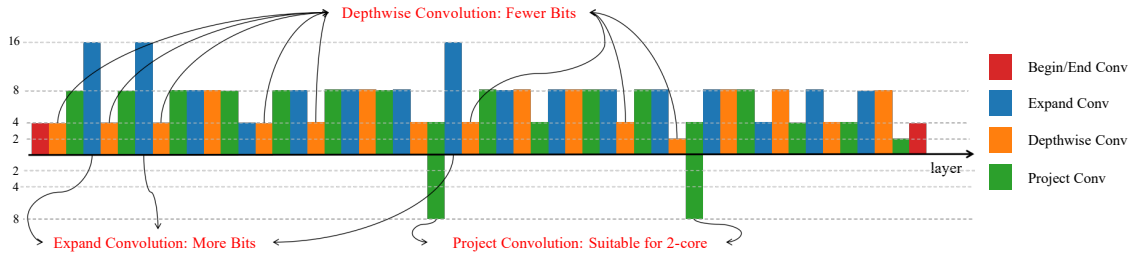


Figure 10: The architecture of a MobileNetV2 variant identified by AdaptQNet with 17 Inverted Residual Blocks (IRBs). Each IRB comprises three sub-parts: Expand Conv, Depthwise Conv, and Project Conv.

to maintain accuracy even with reduced precision through our model search process. Similarly, medium-complexity layers prefer medium-range precision, with INT8 and INT4 together accounting for 73.3%. Low-complexity layers, on the other hand, prefer higher precision, with FP16 having the highest proportion (46.7%) and FP32 accounting for 20%. This suggests that maintaining higher precision in these layers is crucial for overall model accuracy, despite their lower computational demands.

Precision Distribution@OperatorType. We analyzed the precision distribution across different convolution types, with the results shown in Figure 9(c). Expanded Conv exhibits a balanced distribution between high and medium precision, with FP16 and INT4 each accounting for 31.6% of cases, while INT8 and INT2 represent 21.1% and 15.8%, respectively. Depthwise Conv shows a preference for medium precision, with INT8 dominating at 40% and INT4 at 30%. Squeeze-and-excitation (SE) operators exhibit distinct precision preferences. SE-reduce favors higher precision, with FP32 and FP16 together accounting for 44.4%, while maintaining a balanced distribution across other precision levels. In contrast, SE-expand tends toward lower precision, with INT4 being the most common at 37.5%, followed by equal proportions (25%) of INT8 and INT2. Project Conv prefer INT4 precision (57.1%), with others (FP16, INT8, and INT2) each accounting for 14.3%. These patterns indicate that precision requirements are closely tied to operator roles/types. While expand Conv benefit from a balanced precision distribution, project Conv can effectively operate at lower precision. SE modules illustrate how precision requirements can vary even within closely related operations, likely due to their differing roles in feature refinement.

Generalization@Hardware Setting. We evaluate the generalization of AdaptQNet across different hardware settings using MobileNetV2/1.0 as a case. Specifically, we apply AdaptQNet under three configurations: (1) 384 KB RAM, (2) 512 KB RAM (the same as in Table 4), (2) 1024 KB RAM, and (3) 512 KB RAM with half CPU frequency, i.e., higher latency. The results are summarized in Table 6). With stricter memory constraints or higher latency penalties, our system is driven to search for a more compact model, with reduced accuracy.

Table 6: Evaluation on Different Hardware Settings.

Setting	Accuracy (%)	Model Size	Latency	Eff-BW
(384KB, f_{cpu})	76.4	1.41	92.33	5.43
(512KB, f_{cpu})	83.8	1.50	102.39	5.58
(1024KB, f_{cpu})	84.7	1.89	156.54	6.14
(512KB, $\frac{1}{2}f_{\text{cpu}}$)	81.9	1.46	97.49	5.47

5.3 Evaluation for Design Effectiveness

Effectiveness of Search Space Pruning. For MobileNetV2 model with an input size of 84×84×3 under a 384 KB RAM constraint, we remove search space pruning (see Figure 5) to examine its benefits. Without the hard memory constraint enforced by pruning, the resulting model consumes 662 KB of peak memory, far exceeding the 384 KB limit. In contrast, with search space pruning, the final model achieves a peak memory usage of 341 KB. This validates the contribution of our design. Search space pruning not only reduces search cost but also complements the search algorithm by ensuring feasible solutions, as the algorithm applies only a soft penalty to hardware constraints and cannot fully guarantee the feasibility of final model.

Effectiveness of Dual-core SuperNet Search. We apply our design based on results of all three models in Table 4 to demonstrate the benefits of our specialized search for dual-core utilization. The results are shown in Table 5. For $k = 6$(i.e., 6 layers with the smallest $max(\alpha)$), the phase-two variant outperforms the phase-one model in accuracy while also remain efficiency in model size and latency. As shown in Figure 10, two project Conv layers in MobileNetV2/1.0 were initially assigned INT4 precision in phase one. In phase-one search, INT8 and INT4 represent similar trade-offs with different emphases: INT8 offers higher precision at the cost of increased latency, whereas INT4 provides faster computation with slightly reduced accuracy. During phase-two search,

the architecture parameters are reset, effectively creating a new optimization space. Combined with the incorporation of dual-core features, it led to the adoption of previously suboptimal precision (INT8) as part of the optimal solution. INT4+INT8 combination ultimately outperforms other precision pairs. For MobileNetV3-Small and EfficientNet-B0, our design yields more efficient models while maintaining their accuracy. These results confirm the benefits of our design.

5.4 System Overhead Measurement

NAS Time Consumption. We monitor Algorithm 2 and observe that it typically requires about 70 epochs (2.5 hours) to converge on precision settings, and around 100 epochs (3.6 hours) to fully converge on model weights.

Energy Consumption. We empirically measure the energy consumption of MobileNetV2 at various precision. Specifically, we connect a 0.22Ω resistor with the external power supply circuit and use an oscilloscope to measure the current. The power is equal to the current × MCU's supply voltage, and energy consumption is determined by multiplying the root mean square power with the execution time. FP32 model has the highest energy consumption (1250mJ) due to heavy FP operations. INT8 model consumes the least energy (300mJ), as quantized operations significantly reduce computational overhead. The mixed FP32-INT8 model consumes 490mJ, closer to the INT8 model than the FP32 model. While energy efficiency is not our primary focus, introducing a few FP operations in critical layers only lead to certain increase in energy consumption, while offering notable gains in accuracy. Moreover, FPU and multi-core features are often found in high-end MCUs with greater power capacity, and we will explicitly incorporate operator-level energy profiling for model search in future work.

Precision Conversion Overhead. Precision conversion operations introduce some overhead. However, our empirical measurements show that such conversions take minimal time, ranging from 0.003 to 0.011 ms (0.02–0.07% of total inference latency). Thus, despite their frequent occurrence, conversion operations do not impact the overall performance.

6 Related Work

DNN quantization and mix-precision quantization. Quantization [6, 25, 29, 35] is widely recognized as an effective method for DNN model compression. By reducing the bit width (precision) of the model parameters, quantization not only decreases the model size but also typically improves inference efficiency. Initial research focused on unified quantization [8, 15], e.g., from FP32 to INT8/4. However, different layers in DNNs exhibit varying sensitivities to quantization, which is advantageous for exploring mixed-precision quantization [32, 33, 47, 49, 54]. Mixed-precision DNNs have also been explored for MCUs [1, 7, 36, 45]. However, as summarized in Table 1, they are limited to integer-only models. AdaptQNet identifies an opportunity to utilize FPU to explore mixed FP-INT models on MCUs, achieving a better trade-off between accuracy and efficiency.

Multiple xPU utilization for DNN inference. When the system is equipped with multiple processing units, it is natural to optimally utilize them for computational acceleration. Significant efforts have been made to leverage heterogeneous processor cores on mobile devices to co-execute DNN models. CoDL[18] and μLayer[23] focused on efficiently utilizing CPU/GPU on mobile devices to accelerate DNN inference. Band [17], NN-stretch [48] and FYESR [13] go further by incorporating DSP and NPU available on the latest devices. However, little attention has been given to the heterogeneous processing units widely available on MCU SoCs, and there is a lack of designs to effectively utilize them. AdaptQNet aims to fill this gap by modeling the capabilities of multi-core MCU systems with ALU and FPU, and incorporating them into the neural network architecture search process.

7 Limitation and Future Work

Mix-precision Granularity. In our work, the granularity for mixed-precision configuration is set at the operator level, primarily due to constraints posed by existing inference framework [40]. However, our hardware features dual-issue pipeline [43] to potentially support instruction-level granularity. However, it would significantly enlarge the search space and make the search algorithm more challenging. Additional efforts are required to balance these trade-offs.

More Model Supports. To extend our method for efficiently supporting more complex model, e.g., RNNs or Transformers, the number of precision options per layer should be reduced; otherwise, the supernet size grows significantly. A potential solution is to perform per-layer sensitivity analysis to identify partial bit-widths or combinations for search.

Further System Optimization. AdaptQNet currently models hardware capacity and arrange operators across cores accordingly to maximize parallelization. We plan to explore fine-grained operator scheduling to minimize idle time.

8 Conclusion

AdaptQNet is a novel MCU DNN system that determines optimal precision assignments per layer to effectively utilize FPUs and multi-core capabilities. Our approach automatically searches for the best precision configuration by modeling operator latency on heterogeneous processing units, providing a more efficient solution than baselines. The resultant mixed FP-INT models can achieve significant accuracy gains while maintaining efficiency on MCUs.

Acknowledgement

We sincerely thank the anonymous shepherd and reviewers for their comments and suggestions. This work is supported by Guangzhou-HKUST(GZ) Joint Funding Program (No. SL2024A03J01192), Guangzhou Municipal Science and Technology Project (Young Scholar Set-Sail Research, No. SL2024A04J01392), Guangdong General Higher Education Institution (Innovation Team, No. 2024KCXTD008), and Guangdong Provincial Key Lab of Integrated Communication, Sensing and Computation for Ubiquitous Internet of Things (No. 2023B1212010007).

References

[1] Alessandro Capotondi, Manuele Rusci, Marco Fariselli, and Luca Benini. 2020. CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices. *IEEE Transactions on Circuits and Systems II: Express Briefs* 67, 5 (2020), 871–875.

[2] Renze Chen, Zijian Ding, Size Zheng, Meng Li, and Yun Liang. 2024. MoteNN: Memory Optimization via Fine-grained Scheduling for Deep Neural Networks on Tiny Devices. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[3] Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. 2020. Hawq-v2: Hessian aware trace-weighted quantization of neural networks. *Advances in neural information processing systems* 33 (2020), 18518–18529.

[4] Jordan Dotzel, Gang Wu, Andrew Li, Muhammad Umar, Yun Ni, Mohamed S Abdelfattah, Zhiru Zhang, Liqun Cheng, Martin G Dixon, Norman P Jouppi, et al. 2024. FLIQS: One-Shot Mixed-Precision Floating-Point and Integer Quantization Search. In *International Conference on Automated Machine Learning*. PMLR, 6–1.

[5] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural architecture search: A survey. *Journal of Machine Learning Research* 20, 55 (2019), 1–21.

[6] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. 2022. A survey of quantization methods for efficient neural network inference. In *Low-Power Computer Vision*. Chapman and Hall/CRC, 291–326.

[7] Junfeng Gong, Cheng Liu, Long Cheng, Huawei Li, and Xiaowei Li. 2024. MCU-MixQ: A HW/SW Co-optimized Mixed-precision Neural Network Design Framework for MCUs. *arXiv preprint arXiv:2407.18267* (2024).

[8] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[10] Yang He and Lingao Xiao. 2023. Structured pruning for deep convolutional neural networks: A survey. *IEEE transactions on pattern analysis and machine intelligence* (2023).

[11] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. 2019. Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*. 1314–1324.

[12] Andrew G Howard. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[13] Kai Huang, Xiangyu Yin, Tao Gu, and Wei Gao. 2024. Perceptual-Centric Image Super-Resolution using Heterogeneous Processors on Mobile Devices. In *Proceedings of the 30th Annual International Conference on Mobile Computing And Networking*.

[14] Yushan Huang, Taesik Gong, SiYoung Jang, Fahim Kawsar, and Chulhong Min. 2024. Energy Characterization of Tiny AI Accelerator-Equipped Microcontrollers. In *Proceedings of the 2nd International Workshop on Human-Centered Sensing, Networking, and Multi-Device Systems*. 1–6.

[15] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2704–2713.

[16] Eric Jang, Shixiang Gu, and Ben Poole. 2017. Categorical Reparametrization with Gumble-Softmax. In *International Conference on Learning Representations (ICLR 2017)*. OpenReview. net.

[17] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun. 2022. Band: coordinated multi-dnn inference on heterogeneous mobile processors. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 235–247.

[18] 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 *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 209–221.

[19] Hong Jia, Young D Kwon, Dong Mat, Nhat Pham, Lorena Qendro, Tam Vu, and Cecilia Mascolo. 2024. UR2M: Uncertainty and resource-aware event detection on microcontrollers. In *2024 IEEE International Conference on Pervasive Computing and Communications (PerCom)*. IEEE, 1–10.

[20] Hong Jia, Young D Kwon, Alessio Orsino, Ting Dang, Domenico Talia, and Cecilia Mascolo. 2024. TinyTTA: Efficient Test-time Adaptation via Early-exit Ensembles on Edge Devices. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.

[21] Liuyi Jin, Tian Liu, Amran Haroon, Radu Stoleru, Michael Middleton, Ziwei Zhu, and Theodora Chaspari. 2023. Emsassist: An end-to-end mobile voice assistant at the edge for emergency medical services. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*. 275–288.

[22] Sukmin Kang, Seongtae Lee, Hyunwoo Koo, Hoon Sung Chwa, and Jinkyu Lee. 2024. RT-MDM: Real-Time Scheduling Framework for Multi-DNN on MCU Using External Memory. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[23] 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 *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–15.

[24] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. https://api.semanticscholar.org/CorpusID:18268744

[25] Min Li, Zihao Huang, Lin Chen, Junxing Ren, Miao Jiang, Fengfa Li, Jitao Fu, and Chenghua Gao. 2024. Contemporary advances in neural network quantization: A survey. In *2024 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–10.

[26] Edgar Liberis and Nicholas D Lane. 2019. Neural networks on microcontrollers: saving memory at inference via operator reordering. *arXiv preprint arXiv:1910.05110* (2019).

[27] Ji Lin, Wei-Ming Chen, Yujun Lin, Chuang Gan, and Song Han. 2020. Mcunet: Tiny deep learning on iot devices. *Advances in Neural Information Processing Systems* 33 (2020).

[28] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. 2022. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. In *International Conference on Learning Representations*.

[29] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart Van Baalen, and Tijmen Blankevoort. 2021. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295* (2021).

[30] OpenAMP Project. 2023. OpenAMP Library - Open Asymmetric Multi Processing Framework. Online Documentation. https://openamp.readthedocs.io/en/latest/doxygen/openamp/index.html Accessed: 2024-01-10.

[31] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient neural architecture search via parameters sharing. In *International conference on machine learning*. PMLR, 4095–4104.

[32] Mariam Rakka, Mohammed E Fouda, Pramod Khargonekar, and Fadi Kurdahi. 2022. Mixed-precision neural networks: A survey. *arXiv preprint arXiv:2208.06064* (2022).

[33] Mariam Rakka, Mohammed E Fouda, Pramod Khargonekar, and Fadi Kurdahi. 2024. A Review of State-of-the-Art Mixed-Precision Neural Network Frameworks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2024).

[34] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. 2021. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)* 54, 4 (2021), 1–34.

[35] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. 2023. A comprehensive survey on model quantization for deep neural networks in image classification. *ACM Transactions on Intelligent Systems and Technology* 14, 6 (2023), 1–50.

[36] Manuele Rusci, Alessandro Capotondi, and Luca Benini. 2020. Memory-driven mixed low precision quantization for enabling deep network inference on microcontrollers. *Proceedings of Machine Learning and Systems* 2 (2020), 326–335.

[37] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[38] Shakhrul Iman Siam, Hyunho Ahn, Li Liu, Samiul Alam, Hui Shen, Zhichao Cao, Ness Shroff, Bhaskar Krishnamachari, Mani Srivastava, and Mi Zhang. 2024. Artificial Intelligence of Things: A Survey. *ACM Transactions on Sensor Networks* (2024).

[39] Statista. [n. d.]. Leading microcontroller unit (MCU) manufacturers worldwide. https://www.statista.com/statistics/1327509/top-mcu-suppliers-worldwide/. https://www.statista.com/statistics/1327509/top-mcu-suppliers-worldwide/

[40] STMicroelectronics. [n. d.]. X-CUBE-AI: AI expansion pack for STM32CubeMX. https://www.st.com/en/embedded-software/x-cube-ai.html. https://www.st.com/en/embedded-software/x-cube-ai.html

[41] STMicroelectronics. 2023. STM32CubeMX - STM32Cube initialization code generator. Software Tool. https://www.st.com/en/development-tools/stm32cubemx.html Accessed: 2024-01-10.

[42] STMicroelectronics. 2023. STM32H7 Introduction: Dual-Core Architecture. Application Note DM00733995. https://www.st.com/resource/en/application_note/dm00733995.pdf Accessed: 2024-01-10.

[43] STMicroelectronics. 2024. ARM Cortex-M7 Microcontroller. https://www.st.com/content/st_com/en/arm-32-bit-microcontrollers/arm-cortex-m7.html. Accessed: 2024-12-10.

[44] Mingxing Tan and Quoc Le. 2019. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*. PMLR, 6105–6114.

[45] Wei Tao, Shenglin He, Kai Lu, Xiaoyang Qu, Guokuan Li, Jiguang Wan, Jianzong Wang, and Jing Xiao. 2024. Value-Driven Mixed-Precision Quantization for Patch-Based Inference on Microcontrollers. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[46] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. 2016. Matching networks for one shot learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 3637–3645.

[47] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. Haq: Hardware-aware automated quantization with mixed precision. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 8612–8620.

[48] 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 multiprocessors. In *Proceedings of the 21st Annual International Conference on Mobile Systems, Applications and Services*. 70–83.

[49] Bichen Wu, Yanghan Wang, Peizhao Zhang, Yuandong Tian, Peter Vajda, and Kurt Keutzer. 2018. Mixed precision quantization of convnets via differentiable neural architecture search. *arXiv preprint arXiv:1812.00090* (2018).

[50] Chaonong Xu, Min Liu, Chao Li, and Weiming Kong. 2024. Low-Latency Deep Learning Inference Schedule on Multi-Core MCU. In *2024 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[51] Daliang Xu, Mengwei Xu, Qipeng Wang, Shangguang Wang, Yun Ma, Kang Huang, Gang Huang, Xin Jin, and Xuanzhe Liu. 2022. Mandheling: Mixed-precision on-device dnn training with dsp offloading. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*. 214–227.

[52] Sheng Xu, Anran Huang, Lei Chen, and Baochang Zhang. 2020. Convolutional neural network pruning: A survey. In *2020 39th Chinese Control Conference (CCC)*. IEEE, 7458–7463.

[53] Hongyi Yao, Pu Li, Jian Cao, Xiangcheng Liu, Chenying Xie, and Bingzhang Wang. 2022. Rapq: Rescuing accuracy for power-of-two low-bit post-training quantization. *arXiv preprint arXiv:2204.12322* (2022).

[54] Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael Mahoney, et al. 2021. Hawq-v3: Dyadic neural network quantization. In *International Conference on Machine Learning*. PMLR, 11875–11886.

[55] Haibao Yu, Qi Han, Jianbo Li, Jianping Shi, Guangliang Cheng, and Bin Fan. 2020. Search what you want: Barrier panelty nas for mixed precision quantization. In *European Conference on Computer Vision*. Springer, 1–16.

[56] 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 *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 81–93.

A APPENDIX

A.1 Pruning Efficiency

Here, b0(49) denotes the variant b0 of EfficientNet with 49 layers/operators, each with multiple precision branches. For certain MobileNet variants, most FP32 branches were deemed infeasible and were pre-excluded to save time in subsequent searches. This demonstrates the effectiveness of the pre-search pruning.

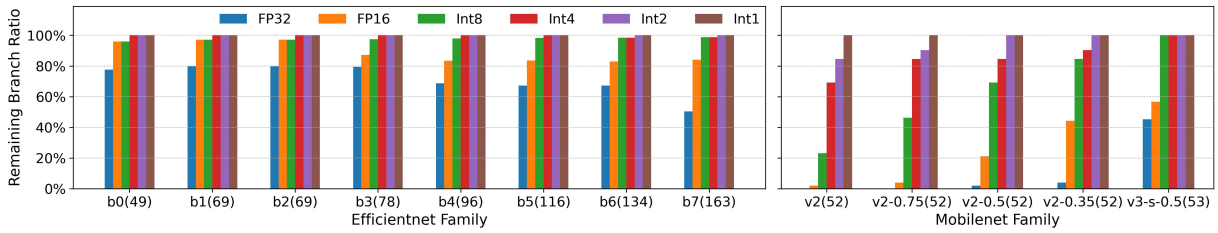


Figure 11: Search space pruning results for EfficientNet and MobileNet families.

B Artifact Appendix

B.1 Abstract

AdaptQNet is a comprehensive hardware-aware neural network optimization system that implements advanced mixed-precision neural architecture search for efficient deployment on resource-constrained microcontrollers with FPU and dual-core features. The artifact provides a complete implementation featuring: (1) Multi-stage DNAS (Differentiable Neural Architecture Search) with first-stage mixed-precision search (FP32/FP16/INT8/INT4/INT2) and second-stage dual-core optimization; (2) Support for MobileNetV2 architecture with configurable precision options. The system achieves 78-85% accuracy on CIFAR-10 with 2-8x memory reduction and 2-16x model size compression compared to FP32 baseline. The artifact enables reproduction of key results including: two-stage DNAS convergence (50-100 epochs), dual-core precision distribution optimization, and hardware-constrained performance analysis.

B.2 Artifact check-list (meta-information)

- **Run-time environment:** Linux/Ubuntu 18.04+, CUDA 11.0+
- **Execution:** Sequential training and search, 2-8 hours per experiment
- **Metrics:** Top-1 accuracy, memory usage, latency, model size, quantization efficiency
- **Output:** Trained models, search results, accuracy curves, memory/latency
- **Experiments:** Automated scripts with configurable parameters
- **How much disk space required (approximately)?:** 10 GB (including datasets)
- **How much time is needed to complete experiments (approximately)?:** 2-8 hours per model
- **Publicly available?:** Yes

B.3 Description

B.3.1 How to access.

- The artifact is publicly available on GitHub: https://github.com/jacksun12/aq-artifact

The repository contains all source code, configuration files, and documentation. Dataset will be automatically downloaded during the first run. The total disk space required after unpacking is approximately 5-10 GB, including the datasets and trained models.

B.3.2 Hardware dependencies. The artifact can run on standard computing hardware. To match configurations in this paper:

- CPU: Intel(R) Xeon(R) Platinum 8369B CPU @ 2.90GHz
- GPU: NVIDIA A6000 48GB x2
- Memory: 8GB RAM minimum, 16GB recommended
- Storage: 10GB free space
- OS: Linux/Ubuntu 18.04+

B.3.3 Software dependencies.

- All dependencies are specified in `requirements.txt` and can be installed via pip.

B.4 Installation

Follow these steps to set up the environment:

(1) Clone the repository:

```
git clone https://github.com/jacksun12/aq-artifact.git
cd adaptqnet-artifact
```

(2) Create a conda environment (recommended):

```
conda create -n adaptqnet_env python=3.9
conda activate adaptqnet_env
```

(3) Install dependencies:

```
pip install -r requirements.txt
```

The installation process takes approximately 10 minutes, including dependency downloads.

B.5 Experiment workflow

The experimental workflow consists of two main stages. We provide two execution methods:

B.5.1 Method 1: Bash Script (Recommended for One-time Execution).

For automated fast execution of Stage 1 and Stage 2, use the provided bash script:

```
# Navigate to the DNAS search directory
cd Search_CIFAR10/dnas

# Run the automated bash script for all DNAS searches
bash run_all_dnas.sh

#Wait until the first bash finished, then navigate to the Dual-Core DNAS search directory
cd Search_CIFAR10/dnas_multicore

# Run the automated bash script for all DNAS searches
bash run_all_dnas.sh
```

The bash script automatically starts multiple DNAS search processes:

- MobileNetV2 mixed-precision search
- MobileNetV2 integer-only search
- MobilenetV2 multi-core search

B.5.2 Method 2: Python Script (Manual Control).

Phase 1: First Stage DNAS Search (2-6 hours per model)

```
cd Search_32x32_10_CIFAR10/dnas

PYTHONPATH=/root python dnas_search_cifar10_mbv2.py \
    --tensor_analysis_json ../tensor_analysis_result/mbv2_cifar10_tensor_analysis_results.json \
    --input_size 32

PYTHONPATH=/root python dnas_search_cifar10_mbv2.py \
    --tensor_analysis_json ../tensor_analysis_result/mbv2_cifar10_tensor_analysis_results.json \
    --input_size 32 \
```

```
    --int_only
```

Phase 2: Second Stage Dual-Core DNAS Search (1 hour per model)

```
cd Search_32x32_10_CIFAR10/dnas_multicore

python dnas_cifar10_search_multicore_mbv2.py     \
--supernet_path ../dnas/mbv2_supernet_models/final_supernet_mbv2_mixed.pth     \
--topk 2     --epochs 10     --lr 0.01     --temperature 0.5    \
--hardware_weight 0.0001     --batch_size 284
```

Phase 3: Evaluation and Visualization (5 minutes)

```
cd tensor_analysis
python tensor_latency_predictor.py
python quick_compression_calc.py
```

All experiments are automated through shell scripts and Python modules. The workflow produces trained models, search results, accuracy curves, and performance analysis plots.

B.6 Evaluation and expected results

B.6.1 Key Results to Reproduce.

1. Mixed-Precision Quantization Performance:

- CIFAR-10 accuracy: 78-85% (depending on model and precision)
- Model size reduction: 2-16x depending on quantization level

2. Hardware-Aware Search Results:

- DNAS convergence: 50-100 epochs
- Precision distribution: INT4/INT2 for less critical layers
- Search time: 2-6 hours per model on single GPU

3. Expected Output Files:

- `best_dnas_cifar10_*.pth`: Best searched models
- `training_curves_*.png`: Training progress plots
- `dnas_results_*/`: Complete search results

B.6.2 Maximum Allowable Variation. Due to the stochastic nature of neural network training and search, some small differences are expected.

- Accuracy variation: ±4%
- Training time variation: ±20% depending on hardware
- Memory usage variation: ±10% depending on certain hyperparameters, such as `–input_size`.

B.7 Experiment customization

The above commands can be customized in many ways:

B.7.1 Search Methods.

- `–input_size` can be replaces with different input resolutions to explore the balance between RAM usage, latency and quantization effect
- `–int_only` can be removed for mixed-precision search (FP32/FP16/INT8/INT4/INT2)
- `–tensor_analysis_json` can be replaced with different analysis result files for various hardware constraints
- `–epochs` can be modified from 20-200 for different convergence requirements
- `–lr` can be replaced with values from 0.0001-0.01 for different learning rates
- `–temperature` can be adjusted from 0.0001-1.0 for different softmax sharpness
- `–hardware_weight` can be modified from 0.0001-0.01 for different hardware penalty emphasis
- `–batch_size` can be adjusted from 64-512 based on available GPU memory

B.7.2 Multi-Core Search Parameters.

- `-supernet_path` can be replaced with different first-stage supernet models
- `-topk` can be adjusted from 1-10 to control the number of layers for dual-core optimization
- `-epochs` can be modified from 20-200 for different convergence requirements
- `-lr` can be replaced with values from 0.0001-0.01 for different learning rates
- `-temperature` can be adjusted from 0.0001-1.0 for different softmax sharpness
- `-hardware_weight` can be modified from 0.0001-0.01 for different hardware penalty emphasis
- `-batch_size` can be adjusted from 64-512 based on available GPU memory