TinyFormer: Efficient Sparse Transformer Design and Deployment on Tiny Devices

Jianlei Yang, *Senior Member, IEEE,* Jiacheng Liao, Fanding Lei, Meichen Liu, Lingkun Long, Junyi Chen, Han Wan, Bei Yu, *Senior Member, IEEE* and Weisheng Zhao, *Fellow, IEEE*

***Abstract*—Developing deep learning models on tiny devices (e.g. Microcontroller units, MCUs) has attracted much attention in various embedded IoT applications. However, it is challenging to efficiently design and deploy recent advanced models (e.g. transformers) on tiny devices due to their severe hardware resource constraints. In this work, we propose *TinyFormer*, a framework specifically designed to develop and deploy resource-efficient transformer models on MCUs. TinyFormer consists of *SuperNAS*, *SparseNAS*, and *SparseEngine*. Separately, SuperNAS aims to search for an appropriate supernet from a vast search space. SparseNAS evaluates the best sparse single-path transformer model from the identified supernet. Finally, SparseEngine efficiently deploys the searched sparse models onto MCUs. To the best of our knowledge, SparseEngine is the first deployment framework capable of performing inference of sparse transformer models on MCUs. Evaluation results on the CIFAR-10 dataset demonstrate that TinyFormer can design efficient transformers with an accuracy of 96.1% while adhering to hardware constraints of 1MB storage and 320KB memory. Additionally, TinyFormer achieves significant speedups in sparse inference, up to $12.2\times$ comparing to the CMSIS-NN library. TinyFormer is believed to bring powerful transformers into TinyML scenarios and to greatly expand the scope of deep learning applications.**

***Index Terms*—TinyFormer, TinyML, Transformer, NAS, Deployment, Sparsity.**

I. Introduction

As IoT applications are becoming increasingly popular recently, microcontroller units (MCUs) have received extensive attention among various kinds of application scenarios. These low-cost, low-power tiny devices are wildly used in plug-and-play scenarios with extreme resource constraints. The devices are usually deployed near the sensor end, gathering the freshest data once produced. Accordingly, Tiny Machine Learning (TinyML) is a growing field in computer science, aiming to apply machine learning technology on MCUs, thereby enabling various applications [1]. Several well-established TinyML applications, such as *Keyword Spotting* [2], *Anomaly Detection* and *Raise to Wake*, only involve simple machine learning algorithms. Some higher-end applications, such as *Wildlife Detection* and *Food Edibility Detection*, usually require powerful deep learning models [3]. However, most of these scenarios only have MCUs available to be exploited, which poses new challenges to TinyML.

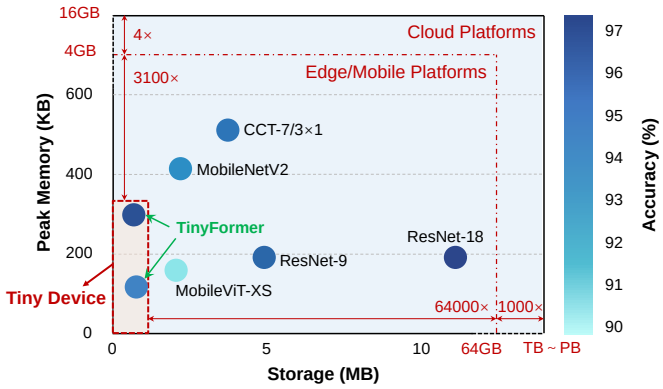


Fig. 1: Accuracy and resource usage comparison among different compact models when evaluated on CIFAR-10. Tiny devices only have MB-level storage and KB-level memory, which is a huge difference between the resources available on edge or cloud platforms.

Bringing powerful deep neural networks to MCUs can greatly expand the scope of deep learning applications [4, 5]. However, the available resources of MCUs are strictly limited. For example, ARM Cortex-M7 has only 1MB storage (Flash) and 320KB memory (SRAM) The resources of ARM Cortex-M7 are even less than that of mobile devices (such as mobile phones, Raspberry Pi) which are up to GB-level. As shown in Fig. 1, there is a large gap between the required resources of deep learning models and the available hardware capacities of tiny devices. To deploy ResNet-18 [6] (with 11M parameters) on MCUs, at least 90% of weights have to be shrunk, which leads to significant accuracy degradation. Moreover, weight pruning does not reduce the peak memory of deep learning models. It is necessary to redesign the neural network to reduce the peak memory. Therefore, it is difficult to deploy powerful models on such resource-constrained devices.

Recently, transformers have achieved great performance in various fields, including computer vision, speech recognition, and natural language processing [7–9]. Deploying these powerful transformer models on MCUs can be exciting for satisfying the requirement of high-demanding scenarios in TinyML.

Manuscript received on August, 2024, revised on March 2025, accepted on November 2025. This work is supported in part by the Beijing Natural Science Foundation (Grant No. L243031), the National Key R&D Program of China (Grant No. 2023YFB4503704 and 2024YFB4505601), and the National Natural Science Foundation of China (Grant No. 62572036). *Corresponding authors are Jianlei Yang and Han Wan.*

J. Yang, J. Liao, F. Lei, M. Liu, L. Long, J. Chen and H. Wan are with School of Computer Science and Engineering, Beihang University, Beijing 100191, China, and Qingdao Research Institute, Beihang University, Qingdao 266104, China. Email: jianlei@buaa.edu.cn, wanhan@buaa.edu.cn

B. Yu is with Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong SAR, China.

W. Zhao is with Fert Beijing Research Institute, School of Integrated Circuit Science and Engineering, Beihang University, Beijing, 100191, China.

However, transformers contain a large number of parameters. Even the lightweight transformer models [10, 11] can not satisfy the strict resource constraints. Deploying powerful transformer models on edge devices or even MCUs remains difficult.

Aiming to bring powerful transformers to MCUs, TinyFormer is proposed as an efficient framework to design and deploy sparse transformers on resource-constrained devices. To be noticed, a sparse transformer in this paper refers to a hybrid model that consists of transformer encoders and convolution layers, with a weight pruning strategy. TinyFormer consists of SuperNAS, SparseNAS, and SparseEngine. SuperNAS aims to produce an appropriate supernet from a large search space for further searching. SparseNAS performs single-path model searching in the supernet and model compressing for evaluating hardware constraints and accuracy. SparseEngine automatically deploys and optimizes the compressed models with the highest accuracy on targeted MCUs. The main contributions of this paper are as follows:

- TinyFormer is proposed as an efficient framework to develop transformers on resource-constrained devices. TinyFormer brings powerful transformers into TinyML scenarios by making it extremely small and efficient on MCUs.
- With the joint search and optimization of sparsity configuration and model architecture, TinyFormer produces a sparse transformer with the best accuracy while satisfying hardware constraints.
- We propose SparseEngine, an automated deployment tool for sparse transformers-based hybrid models. To the best of our knowledge, it is the first deployment tool capable of performing sparse inference for transformers, with a guaranteed latency on targeted MCUs.

With the cooperation of SuperNAS, SparseNAS, and SparseEngine, TinyFormer brings powerful transformers into resource-constrained devices, and enables a faster sparse inference process compared with existing inference engines. Experimental results on CIFAR-10 show that TinyFormer could achieve an accuracy of 96.1%, with an inference latency of 3.9 seconds running on STM32F746. Compared to the light-weight transformer CCT-7/3$\times$1 [12], TinyFormer achieves higher accuracy improvement while saving 74% storage. Benefiting from the automated SparseEngine, TinyFormer could obtain up to $12.2\times$ speedup in sparse model inference compared to CMSIS-NN [13].

The rest of the paper is organized as follows. Section II reviews the related background and provides our motivations. Section III demonstrates the details of the TinyFormer framework. Experimental results are presented in Section IV and conclusions are given in Section V.

II. Related Works and Motivations

Aiming to bring transformers to TinyML scenarios, the following issues have to be comprehensively considered. Firstly, the transformer architecture should be light-weighted to satisfy the extremely demanding resource constraints. Moreover, the sparsity configuration and architecture of the model have coupled effects on accuracy. Thus, we are supposed to consider these coupled effects in the model architecture exploration and compression stage. Finally, it is essential to provide specialized deployment support for targeting to MCUs. These representative investigations motivate us to enable powerful deep-learning models on MCUs through various technological paradigms. Our three primary motivations are listed below:

Motivation ❶: Design light-weight transformer for MCU.

Existing deep learning models on MCU for computer vision tasks are mostly CNNs. Aiming to meet the resource constraints, researchers use mixed precision quantization to deploy CNN models on MCUs [14, 15]. MCUNet [16, 17] is a system-algorithm co-design framework to search and deploy extremely tiny models on the MCU platform. MCUNet aims to find the models with low resources required, and their architectures are derived from the basic structures of CNNs. Some researchers even attempt to train deep learning models on edge devices [18].

Besides CNNs, Transformers have demonstrated success in a large amount of downstream tasks [7–9] including computer vision ones. However, there are fewer works to deploy Transformer on MCUs due to the large amount of parameters and high peak memory during inference. Even the light-weight transformers require resources far beyond the upper limit of MCU constraints [10–12]. Pulp-transformer [19] introduces a library of attention kernels to accelerate transformers' inference. MCUFormer [20] brings transformer on MCU and achieves high accuracy in image classification on ImageNet-1k. However, MCUFormer is mainly focused on optimizing the computational graph to accelerate the inference process with dense models. Applying pruning methods and exploiting the sparsity of the model can further improve the performance of transformers on MCU.

It is still challenging to deploy very small transformers on MCUs. From our perspectives, these efforts will bring transformers into TinyML scenarios, enhancing the applicability of numerous tiny devices.

Motivation ❷: Joint search with model architecture and sparsity configuration.

Neural Architecture Search (NAS) is an advanced approach in automatically model architecture design [21] and achieves remarkable results in various fields. Since model deployment is bounded by hardware constraints (including storage, memory and inference latency), Hardware-Aware Neural Architecture Search (HW-NAS) algorithm is proposed to adopt NAS algorithms for target hardware devices. Most HW-NAS approaches are targeting GPU [22], mobile devices [23, 24] or custom hardware [25].

Most NAS approaches aim to search for a dense model for better performance. However, the lack of considering model sparsity in NAS limits the potential benefits of efficient TinyML exploration. SpArSe proposed a sparse architecture search algorithm for resource-constrained MCUs [26]. However, SpArSe optimizes the network's morphology and performs searching and pruning on respective stages. Hence, it ignores the coupled influence of pruning parameters and network architecture on model accuracy. Aiming to fully

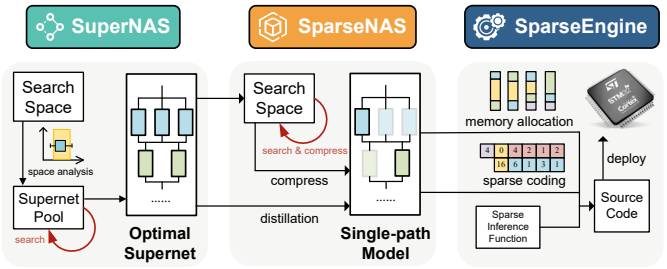


Fig. 2: TinyFormer is a hardware-aware framework. SuperNAS is co-designed with SparseNAS to produce sparse models with transformers under resource limits. SparseEngine enables sparse inference on MCUs.

consider the interaction between sparsity configuration and dense single-path model, we add pruning steps in two-stage NAS for validation.

Motivation ❸: Deploy efficient sparse models on MCU.

Model compression can significantly reduce the model size while maintaining the accuracy. Accordingly, deploying compressed models on MCUs requires particular support from deployment tools. For example, deployment tools should support sparse model storing and execution to accommodate their extremely severe resource constraints. Existing deployment tools and libraries for MCUs include TensorFlow Lite Micro [27], CMSIS-NN [13], MicroTVM [28], CMix-NN [29]. TinyEngine [16] is a code generator-based library to save the resource consumed by the interpreter during inference.

However, these frameworks are mainly targeted at small-scale computing by utilizing the locality in dense form for acceleration. Support of sparse coding and inference enables deployment tools to accommodate larger deep-learning models. To the best of our knowledge, there are few deployment tools or libraries that support sparse model inference, limiting the scale and performance of models. To further exploit the advantage of sparsity, a specialized deployment tool should be developed for sparse model inference on MCUs.

III. TinyFormer: A Framework of Resource-Efficient Model Searching and Deployment

Based on these motivations, we propose TinyFormer, a resource-efficient framework to design and deploy sparse transformer-based hybrid models on resource-constrained devices. In Sec. III-A, we provide a macro view of the TinyFormer's architecture. In the rest of the subsection, we discuss about the key components of TinyFormer: SuperNAS, SparseNAS, and SparseEngine. We present essential algorithms in SuperNAS and SparseNAS and illustrate how they corporate to search the best sparse model in Sec. III-B and Sec. III-C. The implementation method of SparseEngine and the procedure of sparse inference is provided in Sec. III-D.

A. Overview

As shown in Fig. 2, TinyFormer consists of three parts: SuperNAS, SparseNAS, and SparseEngine. SuperNAS aims

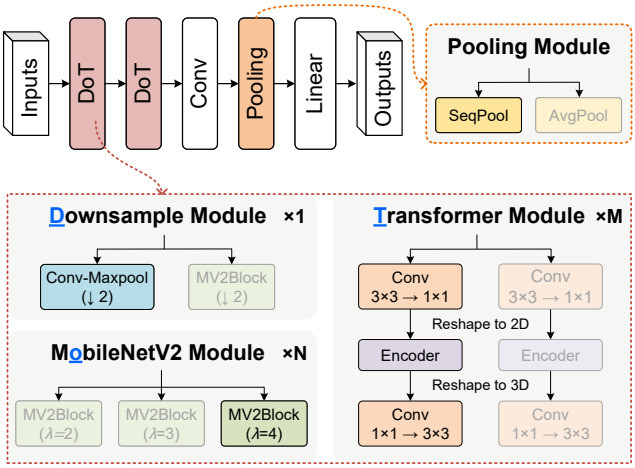


Fig. 3: Our supernet architecture design. We design four types of choice modules: *Downsample Module*, *MobileNetV2 Module*, *Transformer Module* and *Pooling Module*. Each choice module contains 2 or 3 architecture candidates inside. The single-path model is sampled from the supernet with only one architecture candidate invoked per choice module.

to automatically find an appropriate supernet in a large search space $\mathbb{S}_{sup(er)}$. In this work, a supernet is built as a pre-trained over-parameterized model, where the following single-path models are sampled from the supernet. SparseNAS is adopted to find a sparse transformer from the supernet. SparseNAS searches for a single-path model in the supernet with a set of sparse configurations in search space $\mathbb{S}_{spa(rse)}$. Weight pruning is performed in `Conv2d` (convolution in 2d) and `Linear` operators, and full-integer quantization with `INT8` format is applied on all layers. Finally, SparseEngine automatically generates binary code on STM32 MCUs with several functional implementations. SparseEngine deploys the obtained sparse model to MCUs, enabling sparse inference to save hardware resources.

B. SuperNAS: Supernet Architecture Search

When designing a supernet structure to search models and deploy on MCUs, we are facing a trade-off between model sparsity and capacity. The accuracy of the smaller dense models is limited by capacity, while pruning with higher sparsity on the larger models could lead to a drastic drop in accuracy. The balance between model sparsity and capacity should shift according to hardware resource constraints, challenging the design of the supernet.

Therefore, we propose SuperNAS to automatically design the supernet. SuperNAS searches appropriate supernet in search space $\mathbb{S}_{sup}$. Appropriate supernet should satisfy two following conditions. Firstly, most of the sparse models obtained from supernet are supposed to satisfy the resource constraints strictly. Secondly, the average validation accuracy of these models should be achieved as high as possible.

With the design of the search space for supernet ($\mathbb{S}_{sup}$), SuperNAS first analyses the probability of accepting the designed

TABLE I: Search space design in Sec. III-B and III-C. Configurations of supernet are sampled in $\mathbb{S}_{sup(er)}$, and configurations of the single-path model is sampled in $\mathbb{S}_{spa(rse)}$ with a given supernet. MHSA is the abbreviation of Multi-head Self Attention.

Symbol	Invovled Configurations	Choices
$\mathbb{S}_{sup(er)}$	Number of heads in MHSA	[1, 2, 4]
	Number of MobileNetV2s in `DoT`	[1, 2]
	Number of Transformers in `DoT`	[1, 2]
	Embedding dimension size in `DoT1`	[32, 44]
	Embedding dimension size in `DoT2`	[96, 128, 256]
	Dimension of the Last Linear	[512, 768, 1024]
$\mathbb{S}_{spa(rse)}$	Single-path choice of module	Shown in Fig. 3
	Pruning block size	[2, 4]
	Pruning sparsity	[0, 0.4∼0.8:0.1]

search space. Then SuperNAS randomly samples supernet configurations from search space and evaluates the average accuracy of the sparse single-path models in the supernet. The supernet with the best accuracy will be sent to SparseNAS for further search.

1) Supernet Architecture design: The search space of SuperNAS $\mathbb{S}_{sup}$ contains hyper-parameters that are related to the supernet architecture design, shown in Tab. I. The supernet architecture design is shown in Fig. 3, using the same approaches as the single-path-one-shot (SPOS) [30]. Supernet contains choice modules in different branches, and the single-path model is sampled by selecting a choice module at each branch. We build supernet architecture based on four types of choice modules: *Downsample module*, *MobileNetV2 module*, *Transformer module* and *Pooling module*. The four types of modules contain multiple different architecture candidates respectively. The condidate parameter set in $\mathbb{S}_{sup}$ is determined based on emperical enginerring practices, and takes multiple aspects into account, including memory usage, storage usage, hardware efficiency, and the trade-off between searched results and the search time in total.

In Fig. 3, `Conv` represents a standard 3×3 convolution with 1×1 padding unless stated otherwise. In *Downsample module*, `Conv-Maxpool` refers to a standard convolution following a 2×2 max pooling. Architecture candidates that perform downsampling are tagged with $\downarrow 2$. *MobileNetV2 module* refers to inverted residual block in [31] with expansion factor λ. In *Transformer module*, `Conv` 3×3→1×1 is a standard 3×3 convolution following a 1×1 convolution with no padding. `Conv` 1×1→3×3 is similar to the expression above. Encoder is a standard transformer encoder similar to [9], with ReLU [32] as the activation operation for efficient calculation on MCUs. `SeqPool` and `AvgPool` in *Pooling module* indicate sequence pooling in [12] and 2×2 average pooling, respectively.

Unlike CNNs, transformers lack spatial inductive biases and rely heavily on massive datasets for large-scale training. Therefore, we insert *MobileNetV2 module* before *transformer module* to address this issue. Referring to MobileViT [10], we insert `Conv` 3×3→1×1 and `Conv` 1×1→3×3 before- and after- the encoder instead of positional encoding.

Algorithm 1: Search Space Analysis in SuperNAS

Input: Search space $\mathbb{S}_{sup}$,$\mathbb{S}_{spa}$, memory limit $\mathcal{L}_m$ and storage limit $\mathcal{L}_s$, lower bound ratio λ_{lo} and upper bound ratio λ_{up}, iteration count $\mathcal{T}_{sup}$.
Output: Probability to accept $\mathbb{S}_{sup}$ for search space.
$[n_{accpt}, n_{eval}] \leftarrow [0, 0]$
for $i = 1$ **to** $\mathcal{T}_{sup}$ **do**
  $c_{sup}, c_{spa} \leftarrow RandomSample(\mathbb{S}_{sup}, \mathbb{S}_{spa})$
  $\mathcal{M}^i \leftarrow CreateModel(c_{sup}, c_{spa})$
  $[l_m^i, l_s^i] \leftarrow ResourceEval(\mathcal{M}^i)$
  if $l_m^i \leq \mathcal{L}_m$ **and** $l_s^i \leq \mathcal{L}_s$ **then**
    $n_{eval} \leftarrow n_{eval} + 1$
    $N_{param}^i \leftarrow ParamsEval(\mathcal{M}^i)$
    if $\lambda_{lo}\mathcal{L}_m \leq N_{param}^i \leq \lambda_{up}\mathcal{L}_m$ **then**
      $n_{accpt} \leftarrow n_{accpt} + 1$
    end
  end
end
return n_{accpt}/n_{eval}

We take `DoT`, an architecture stacked by *Downsample module*, *MobileNetV2 module* and *Transformer module*, as the basic layer of the supernet. *MobileNetV2 module* and *Transformer module* in `DoT` structure are repeatable, which makes `DoT` a more flexible feature extraction architecture. The supernet architecture adopts two `DoT`s as the backbone, followed by some post-processing operators.

2) Search Space Analysis: As mentioned in III-B, the balance between model sparsity and capacity is essential in search space design, and the configurations of search space determine the supernet and single-path model architecture. Therefore, we need to evaluate search space before sampling supernet. Alg. 1 shows how we analyze the search space. Before the analysis, we set the hyper-parameter λ_{lo} and λ_{up} as the lower and upper bound of the single-path model's capacity. We randomly sample configurations c_{sp}, including sparsity and block-pruning configuration in each layer, from the search space and build sparse single-path models. If the sparse single-path model meets the hardware constraint, we evaluate the count of parameters in the model and accept the model if the count is between the given boundary. Finally, we calculate the statistical probability to represent how many sampled single-path models are acceptable in the search space. If the statistic probability (n_{aceept}/n_{eval}) is higher than 90%, we accept the search space for further deployment. Otherwise, we adjust the search space in advance to avoid unnecessary searches. The adjustment of search space is performed by:

$$y = Round(k(x - \overline{x}) + b), \tag{1}$$

where x and y are dimensional settings before- and after-adjustment. Symbols with overline refer to the average values. $Round$ functions projects floats to their nearest integers. The

Algorithm 2: Supernet Architecture Seach

Input: Search space $\mathbb{S}_{sup}$, memory limit $\mathcal{L}_m$ and storage limit $\mathcal{L}_s$, iteration count $\mathcal{T}_i$, $\mathcal{T}_j$.
Output: An optimal supernet $\mathcal{A}_{sup}$.
for $i = 1$ **to** $\mathcal{T}_i$ **do**
  $c^i_{sup} \leftarrow EvolutionarySample(\mathbb{S}_{sup})$
  $\mathcal{A}^i \leftarrow CreateSupernet(c^i_{sup})$
  if $TestSupernet(\mathcal{A}^i) \neq True$ **then**
    continue
  end
  Train $\mathcal{A}^i$ for 10 epochs
  for $j = 1$ **to** $\mathcal{T}_j$ **do**
    $c_{spa} \leftarrow RandomSample(\mathbb{S}_{spa})$
    $M^{i,j} \leftarrow CreateModel(c_{spa}, \mathcal{A}^i)$
    $[l_m, l_s] \leftarrow ResourceEval(\mathcal{M}^{i,j}, c^k_{sp})$
    if $l_m > \mathcal{L}_m$ **or** $l_s > \mathcal{L}_s$ **then**
      continue
    end
    $\mathcal{M}^{i,j,k}_{prun} \leftarrow OneShotPrune(\mathcal{M}^{i,j}, c^k_{sp})$
    $Acc \leftarrow AccuracyEval(\mathcal{M}^{i,j,k}_{prun})$
    $AvgAcc^{i,j} \leftarrow UpdateAvgAcc(Acc)$
    $AvgAcc^{i} \leftarrow UpdateAvgAcc(AvgAcc^{i,j})$
  end
  $\mathcal{A}_{sup} \leftarrow UpdateBest(\mathcal{A}^i, AvgAcc^i)$
end
return $\mathcal{A}_{sup}$

slope k in equation 1 is given by:

$$k = \frac{n_{accept}}{n_{eval}} + 10\%, \tag{2}$$

where n_{accept} and n_{eval} is counted in Alg. 1. The intercept b in equation 1 is given by:

$$b = x_{min} + \frac{\overline{N_{params}} - N_{min}}{N_{max} - N_{min}}(x_{max} - x_{min}), \tag{3}$$

where N_{min} and N_{max} are the minimum and maximum value in Parameters Evaluations (N_{params}). x_{min} and x_{max} are named in the same way.

To be specific, only the dimensional setting will be adjusted, while the number of heads in MHSA and number of modules in DoT are fixed. The search space can be also tuned manually to adapt the implementation of calculation (e.g set dimension to a multiple of 4).

In detailed implementations, we choose $\lambda_{lo} = 0.8$ and $\lambda_{up} = 2.8$ based on our preliminary experience. We conducted experiments in three types of search space: *Small*, *Normal*, and *Large*, which denote the model size sampled from them. The experiments results prove that the search space analysis allows SuperNAS to optimally balance sparsity and accuracy trade-offs. Detailed experiments of search space analysis can be found in Sec. IV-B.

3) Supernet Architecture Search: The search process of supernet is shown in Alg. 2. For each sampled supernet, we perform a simple test on it before the actual search (expressed as *TestSupernet* in Alg. 2). Specifically, we randomly sample 100 single-path models from the supernet and evaluate the memory usage of each model. If half of the models exceed the memory limit of hardware constraint, we skip the search procedure for this supernet.

Algorithm 3: Single-path Model Search in SparseNAS

Input: Search space $\mathbb{S}_{spa}$, supernet architecture $\mathcal{A}$, memory limit $\mathcal{L}_m$ and storage limit $\mathcal{L}_s$, iteration count $\mathcal{T}_{spa}$.
Output: An optimal pruned single-path model $\mathcal{M}_{prun}$.
for $i = 1$ **to** $\mathcal{T}_{spa}$ **do**
  $c_{spa} \leftarrow EvolutionarySample(\mathbb{S}_{spa})$
  $M^i \leftarrow CreateModel(c_{spa}, \mathcal{A})$
  $[l_m, l_s] \leftarrow ResourceEval(\mathcal{M}^i)$
  if $l_m > \mathcal{L}_m$ **or** $l_s > \mathcal{L}_s$ **then**
    continue
  end
  Train $\mathcal{M}^i$ for 10 epochs
  $\mathcal{M}^i_{prun} \leftarrow IterativePrune(\mathcal{M}^i)$
  $Acc^i \leftarrow AccuracyEval(\mathcal{M}^i_{prun})$
  $\mathcal{M}_{prun}, c_{sp} \leftarrow UpdateBest(\mathcal{M}^i_{prun}, Acc^i)$
end
Return $\mathcal{M}_{prun}$

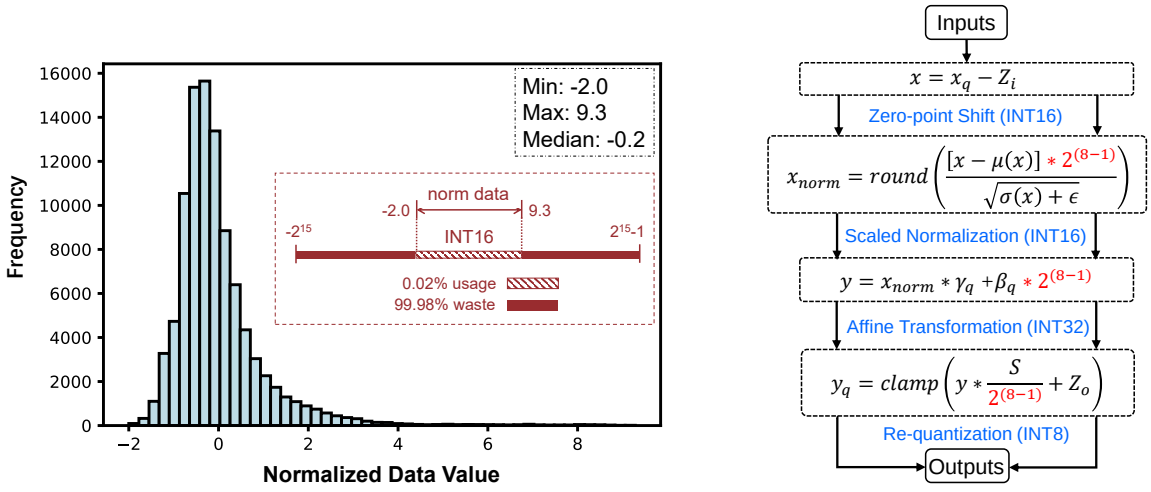
After the simple test of supernet, we take two steps to evaluate its sensitivity to sparsity. Firstly, we randomly select single-path models from the supernet. For each single-path model, compression is conducted by generated sparse configuration to check whether the model occupies more resources than the practical scenario. Then we take the average accuracy as the performance metric of the supernet. The configuration sampled in $\mathbb{S}_{sup}$ is updated with evolutionary algorithm (expressed as *EvolutionarySample* in Alg. 2). The supernet with the highest performance metric will be adopted for the next search stage.

To reduce the search cost, we evaluate the storage and peak memory usage of the sparse models for skipping the ones that do not satisfy the resource requirements. In addition, the one-shot weight-magnitude pruning algorithm without tuning is adopted to reduce the runtime cost (presented as *OneShotPrune* in Alg. 2).

C. SparseNAS: Hardware-Aware Sparse Model Search

SparseNAS aims to obtain the best sparse single-path model from the supernet. Differently from the original SPOS approaches from [30], the compression procedure is performed, including pruning and quantization, among the searching steps. Moreover, in order to reduce the cost of training and compression procedures, SparseNAS is divided into two stages. In first stage (selection stage), SparseNAS aims to find the best sparse single-path model with *pruning* and *quantization* procedures. For the second stage (compression stage), SparseNAS only performs pruning and fine-tuning operations to recover the model's accuracy.

1) Two-stage process: SparseNAS consists of the selection stage and the compression stage. The selection stage of Sparse-



(a) (b)

Fig. 4: (a) Distribution of normalized activation in the first LayerNorm layer. (b) *Scaled-LayerNorm* inference process. S and Z are the scaled and zero-point parameter of quantization respectively. $clamp$ function clamps the result to range of signed `INT8`.

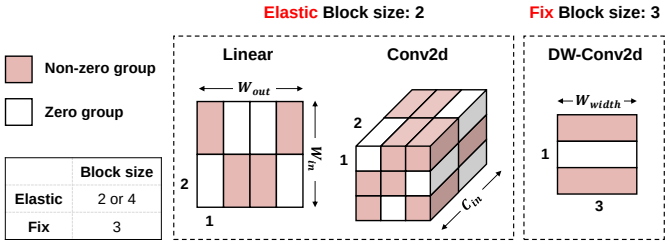


Fig. 5: Blockwise pruning method. We abbreviate depthwise convolution in 2d as `DW-Conv2d`. W_{in} and W_{out} denote the dimensions of linear layer, C_{in} means the number of input channels and W_{width} is the kernel width of depthwise convolution.

NAS is presented in Alg. 3 for single-path model selection. In the selection stage, SparseNAS trains randomly-sampled sparse single-path models for a few epoch, then performing iterative pruning and accuracy evaluation. The model with highest accuracy is sent to the compression stage for further training. We skip the model candidates that do not satisfy hardware constraints by evaluating the storage and memory usage.

For the compression stage, SparseNAS performs iterative pruning again and fine-tuning on the model for accuracy improvement. After the compression stage, the model will be sent to SparseEngine for efficient deployment. Two-stage procedures reduce the training and compression cost while maintaining the obtained single-path model's accuracy.

2) Pruning Method: Different from the one-shot pruning method in SuperNAS, AGP iterative pruning method [33] is utilized in two-stage NAS. AGP method can avoid significant accuracy degradation caused by pruning. We only perform weight pruning in `Conv2d` and `Linear` operators.

SparseNAS utilizes a blockwise pruning method, grouping multiple continuous weights as a block to prune. Pruning configuration (sparsity and block size) affects both accuracy and hardware resource usage in deployment, and the effects vary from different layers. Therefore, we adopt a mixed-blockwise pruning strategy in `Conv2d` and `Linear` layers, as shown in Fig. 5. In mixed-blockwise pruning procedure, SparseNAS selects elastic block size (2 or 4) for each layer to prune. When sampling the single-path model, each choice module is set to a random configuration of sparsity and block size.

In SparseEngine, we applied the blockwise convolution in width direction to exploit spatial locality in computation. Therefore, as an exception, the block size of blockwise convolution is set to 3, for the kernel size is fixed to 3×3.

3) Quantization Method: Floating-point calculations on MCUs are inferior in latency and power consumption compared to integer calculations. Therefore, we quantize the model weights and activations to `INT8` by Post-Training Quantization (PTQ) algorithm [34]. However, LayerNorm calculations in transformer is not suitable for directly quantized. Performing linear quantization on LayerNorm will cause a significant accuracy drop. The original LayerNorm is defined as:

$$y = \frac{x - \mu(x)}{\sqrt{\sigma(x) + \epsilon}} * \gamma + \beta, \tag{4}$$

where $\mu(x) = \frac{1}{c}\Sigma_{i=1}^{c} x_i$ and $\sigma(x) = \sqrt{\frac{1}{c}\Sigma_{i=1}^{c}(x_i - \mu)^2}$ are the mean and variance values of input x in channel-wise direction (c is the number of channels). $x_{norm} = \frac{x-\mu(x)}{\sqrt{\sigma(x)+\epsilon}}$ is the normalization result before affine transformation. γ and β are the learnable parameters in affine transformation. ϵ is a significantly small value to prevent the denominator to be zero. In linear quantization, the normalization result x_{norm} is supposed to be rounded to integer. However, directly rounding x_{norm} to `INT16` incurs a significant loss of precision.

To verify the conjecture, we count the distribution of x_{norm} of LayerNorm in first `DoT` Architecture. As shown in Fig. 4(a),

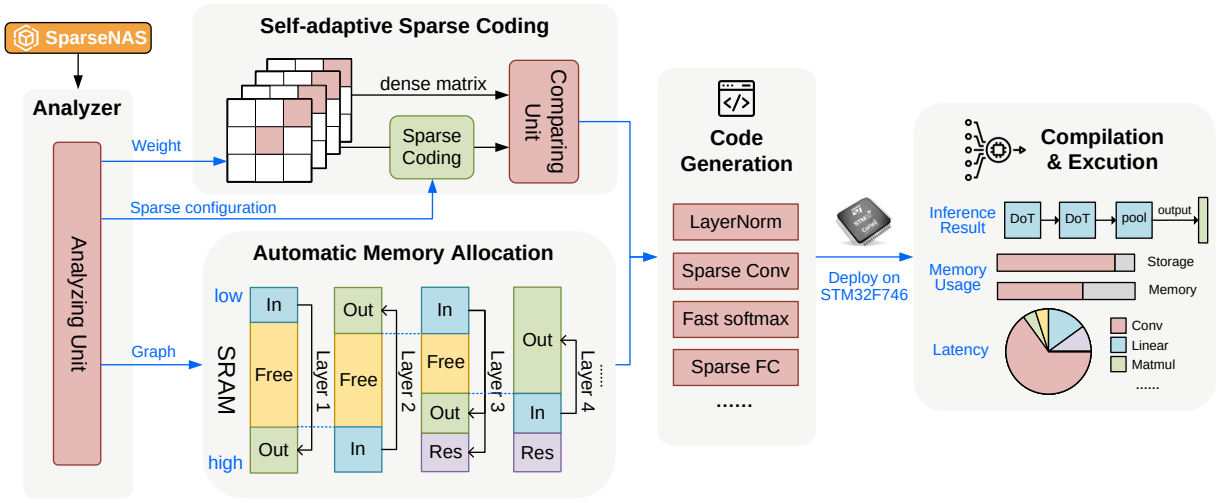


Fig. 6: The workflow overview of SparseEngine. Sparse models obtained from SparseNAS are first analyzed for sparse coding and memory allocation. Then, different kinds of layers' computations are transformed via automatic code generation for the following compilation and execution on targeted MCUs.

the normalized data is mapped to the range of -2.0 to 9.3. Rounding the small range of normalization results to `INT16` causes a large loss of precision. On the other hand, the range of `INT16` is not fully utilized by x_{norm}. Adopting `INT16` to store the normalized data will waste 99.98% of integer values, presented in Fig. 4(a).

To tackle this problem, *Scaled-LayerNorm* is proposed to perform integer-only inference instead of naive quantized LayerNorm. As shown in Fig. 4(b), we enlarge the normalized results by $2^{(8-1)}$ to reduce the numeric precision loss in quantization. After the linear transformation, the results are stored as `INT16` format, while it also includes the $2^{(8-1)}$ factor. In re-quantization step, we fold $\frac{1}{2^{(8-1)}}$ into scaling factor S to ensure mathematical equivalence. Expanding the numerical range of x_{norm} prevents significant accuracy drop caused by large precision loss. With the approaches above, we reach a significant speed-up in LayerNorm inference, with only slight accuracy drop. Detailed experimental results of Scaled-LayerNorm is presented in Sec. IV-C.

D. SparseEngine: Efficient Deployment Library of Sparse Transformers

SparseEngine is a deployment tool for sparse transformers on MCUs. It consists of a deployment library based on C++ language, and a code generator. SparseEngine can automatically allocate memory to each layer and generate codes that can directly deploy on STM32 MCUs. Compared with CMSIS-NN and TinyEngine, SparseEngine supports extra functions, including dynamic sparse calculate of `Conv2d` and `Linear` operations, to efficiently inference a sparse transformer. Moreover, SparseEngine optimizes *softmax* operation, which is more frequently used in transformer inference. Comprehensive details of SparseEngine are illustrated in Fig. 6.

Firstly, the models obtained from SparseNAS are analyzed to extract the required information, including sparse configuration, model architecture and memory usage, etc. To efficiently utilize the available memory on MCUs, a head-tail-alternation allocation strategy is adopted to automatically allocate memory for models inference, presented in Fig. 6. Meanwhile, inspired by the run-length coding algorithm [35], we perform the self-adaptive sparse strategy with blockwise run-length coding in `UINT8`. With self-adaptive strategy, weights are stored as sparse format only if sparse coding could reduce the storage occupation. Finally, targeted codes could be generated for deployment on MCUs. The code generator in SparseEngine converts ONNX model to C-like code executable on on MCU. The adoption of ONNX as an intermediate representation enables TinyFormer to maintain interoperability across multiple machine learning pipelines.

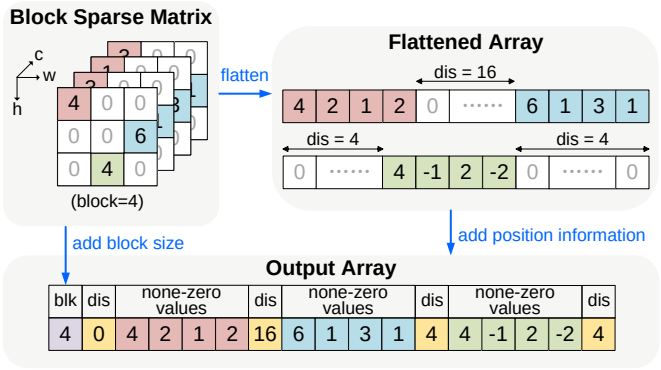


Fig. 7: Blockwise run-length coding for 3D matrix.

Compared with other deployment approaches on MCUs, SparseEngine aims to further exploit the sparsity on TinyML deployment and model inference. The sparsity exploitation includes sparse encoding/decoding, and sparse calculations (both `Conv2d` and `Linear` layers). Moreover, targeting on model inference with transformer modules, `Softmax` operator is also optimized to accelerate the inference on MCUs.

1) Sparse Coding: In sparse coding, based on run-length coding algorithm [35], we perform blockwise run-length coding in 8-bit to adopt block pruning. At first. the original 3D matrix (tensor) is flattened into array format. The spare weights are stored in $(dis, val_1, val_2..., val_b)$ format, as presented in Fig. 7, where dis represents the distance between non-zero blocks as the position information, val_i indicates the i-th weight in the non-zero block whose length is b. We insert zero elements if dis is beyond the maximum value of `INT8`.

Compared with Coordinate (COO) and Compressed Sparse Row (CSR) formats, blockwise run-length coding has a higher compression ratio. It only requires one element to represent the position of adjacent non-zero weights. The compression ratio of this encoding format could be obtained by

$$\eta = \frac{1}{(1-\rho) \times (1+\frac{1}{b})}, \tag{5}$$

where η indicates the compression ratio, ρ refers to the sparsity and b is the block size of pruning. Consequently, the compression ratio of blockwise run-length encoding could be larger along with the increasing of sparsity and block size. Cooperating with mixed-blockwise pruning in SparseNAS, blockwise run-length encoding significantly reduce the required size of sparse coding.

2) Sparse Convolution and Linear: Decoding sparse weights and then computing convolution in dense format occupies a large amount of memory footprint [36, 37]. To avoid unnecessary memory usage, we perform sparse convolution calculation directly in sparse format. Fig. 8 presents the details of the sparse `Conv` calculation. Sparse weights are decoded to obtain the coordinates and values. Each weight value corresponds to a sub-matrix of the input matrix. After extracting the sub-matrix, we execute element-matrix multiplication and accumulate the results as the output. Specifically, we decode two weight values simultaneously and extract two corresponding elements from the sub-matrix. Then, two `INT8` values are sign-extended to `INT16` and concatenated as `INT32` format. Thus, the above multiply-accumulate calculations can be performed by SIMD instructions. Meanwhile, the sparse linear layers are implemented by the same manner, despite the difference in dimensions.

3) Softmax Optimization: According to our profiling, the `Softmax` layer is very time-consuming due to the required exponential operations. The computation of `Softmax` can be described as

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} = \frac{e^{x_i - x_{max}}}{\sum_{j=1}^{n} e^{x_j - x_{max}}}, \tag{6}$$

where x_i indicates the i-th value and x_{max} is the maximum value in x (n is the number of elements in one channel). Since the inputs of `Softmax` are in `INT8` format, the exponents range from -256 to 0. Consequently, redundant calculations will be increased according to the input size of `Softmax`. Therefore, we optimize the `Softmax` operator with a lookup table to reduce redundant computations.

Targeting at calculating the negative exponential functions, SparseEngine adopts the absolute value of the exponential factor as an index to query the bitmap. If the corresponding bit exists in the bitmap, SparseEngine obtains the result from the lookup table and reuses it in `Softmax` computations. Otherwise, it calculates the exponential function and stores the result in the table, updating the corresponding bit in the bitmap. According to the evaluation of SparseEngine, its memory usage on bitmap and lookup table is less than 1.2KB. Since SparseEngine reuses the buffer memory of convolution when performing `Softmax` optimization, there is no extra memory cost in calculations.

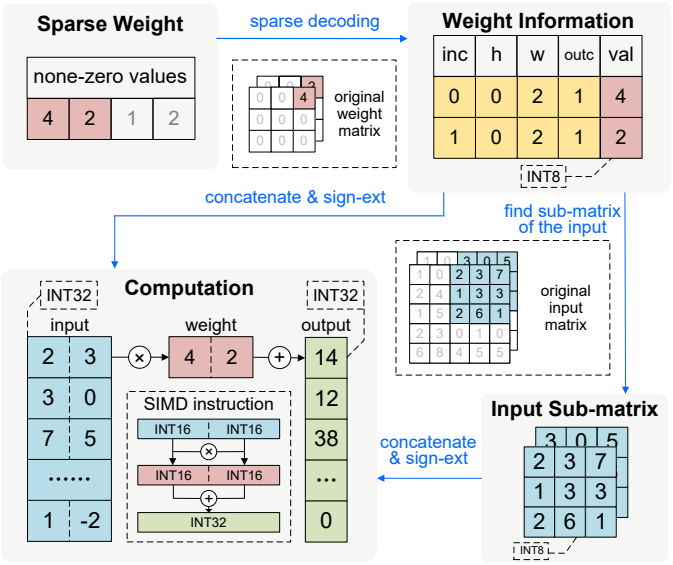


Fig. 8: Sparse Convolution with SIMD instruction. Each block's weight information are extracted by sparse decoding procedure. Among them, the location information are adopted to find the sub-matrix in original input matrix. After concatenate & sign extension 8(sign-ext) operations, sub-matrix and weight values are sent for computations with SIMD instructions acceleration.

IV. Experimental Results

Here we present experimental results to demonstrate the effectiveness of our TinyFormer framework. The results mainly consist of offline evaluation and runtime validation. In Sec. IV-B, we provide offline evaluation results with algorithmic performance to demonstrate the effectiveness of SuperNAS and SparseNAS. In Sec. IV-C, we are focused on the runtime efficiency on MCU platform, and present the runtime validation of SparseEngine.

A. Evaluation Setup

1) Dataset: According to requirements in TinyML scenario, our TinyFormer is mainly evaluated on CIFAR-10 [39] and ImageNet-32 [40] dataset for image classification. CIFAR-10 contains 50 thousand and 10 thousand images for training and validation respectively. ImageNet-32 contains 1.28 million and 50 thousand images for training and validation respectively. Images in both datasets are set at 32×32 resolution in RGB format.

TABLE II: Top-1 accuracy comparison and hardware resources evaluation. Different models are evaluated under different hardware constraints. TinyFormer-120K and TinyFormer-300K indicate that the peak memory limit is set as 120KB and 300KB, respectively. Texts that end with $*$ indicate that the peak memory or storage usage is beyond the limitations of STM32F746 MCU.

Model	Peak Mem.	Storage	CIFAR-10	ImageNet-32
MobileNetV2 [31]	416KB$*$	2.13MB$*$	94.61%	38.22%
CCT-7/3×1 [12]	512KB$*$	3.52MB$*$	95.72%	39.04%
MobileViT-XS [10]	160KB	1.91MB$*$	90.11%	34.16%
MobileViTV2-0.75 [38]	172KB	3.48MB$*$	91.30%	34.99%
MCUNet-in3 [16, 17]	22KB	0.89MB	84.26%	26.60%
TinyFormer-120K (ours)	120KB	0.94MB	94.52%	38.53%
TinyFormer-300K (ours)	300KB	0.91MB	**96.10%**	**39.37%**

2) Training Settings: In the training process, we use AdamW [41] as the optimizer. The learning rate starts with a warm-up phase, increasing from 1×10^{-6} to 5.5×10^{-4} for the first 10 epochs, and then follows a cosine annealing schedule, gradually decreasing to 1×10^{-5}. During the model's training, label smoothing with a probability of 0.1 is employed, as suggested by Szegedy et al. [42]. Unless otherwise specified, all involved models are trained from scratch for 300 epochs using a batch size of 128.

The weights of models are initialized using the method from [43]. All models are performed `INT8` quantization. The reported top-1 accuracy of image classification task is evaluated on full-integer models if not specifically stated.

Some baseline models are designed towards the ImageNet dataset with higher resolution. To ensure a fair comparison, we maintain the original architecture of baseline models, except for the number of classes in classification.

3) Platforms: The offline experiments in Sec. IV-B are conducted on 8 NVIDIA V100 GPU. The runtime experiments in Sec. IV-C are conducted on with SparseEngine for accuracy and efficiency evaluation. We select STM32F746 MCU platform (Cortex-M7 core @ 216MHz, with 320KB RAM and 1MB Flash) and STM32H743 MCU platform (Cortex-M7 core @ 480MHz, with 512KB RAM and 2MB Flash) to conduct the runtime experiments. All program is burned by Keil5 MDK supporting ARM-based microcontrollers.

B. Offline Evaluation

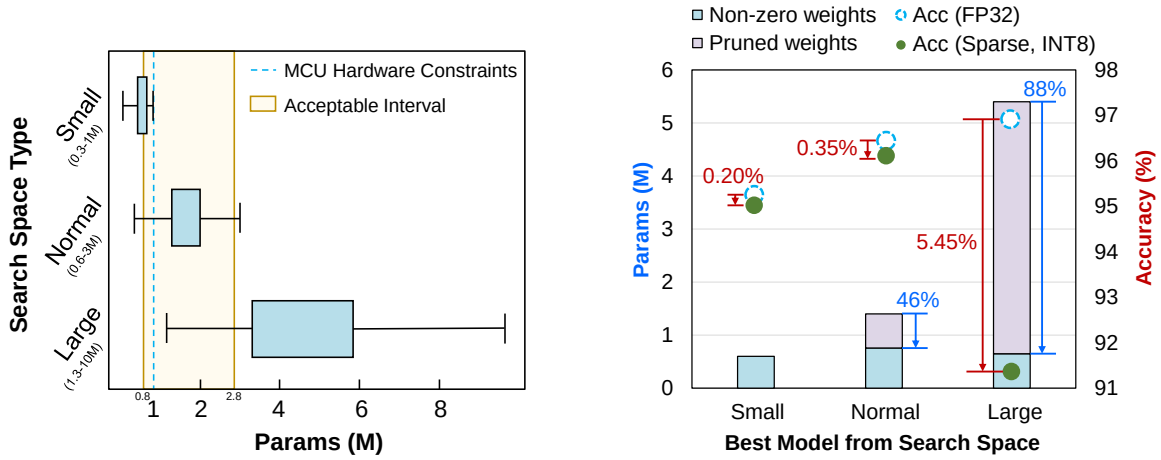
In offline evaluation, we have mainly trained two model with different hardware constraints (TinyFormer-300K and TinyFormer-120K). In the NAS process, SuperNAS takes 15.2 GPU hours per sampling, and SparseNAS takes 1.5 GPU hours per sampling.

Tab. II shows the comparison results of our TinyFormer and other state-of-the-art lightweight models. With the co-optimization of SuperNAS and SparseNAS, TinyFormer-300K could satisfy the hardware constraints of STM32F746 MCU platform and achieve a record accuracy with CIFAR-10 and ImageNet-32 on MCUs. As illustrated in Fig. 3, TinyFormer is designed as a hybrid model that contains both convolution and transformer encoder layers. Compared to other lightweight hybrid models, such as MobileViT-XS, TinyFormer better combines the advantages of CNN and transformer to achieve higher accuracy with limited resources. Compared with CCT-7/3×1 designed for CIFAR-10, TinyFormer-300K achieves higher accuracy while reducing peak memory and storage by 41% and 74%, respectively. Meanwhile, TinyFormer-120K is searched for stricter peak memory constraints. Compared with MobileViT-XS, TinyFormer-120K improves the accuracy by 4.4% with less peak memory and storage.

TABLE III: Ablation of TinyFormer in offline evaluation. Basic TinyFormer structure has two `DoTs` and mixed block size. All the models are searched with hardware constraints of 300KB Memory and 980KB Storage in CIFAR-10. #Params indicate the number of effective weights.

Model	Accuracy	#Params	Storage
TinyFormer	**96.10%**	731K	942KB
TinyFormer (w/o Tr.)	94.62%	756K	963KB
TinyFormer (Single `DoT`)	93.92%	572K	655KB
TinyFormer (Block Size = 2)	95.88%	730K	950KB
TinyFormer (Block Size = 4)	95.52%	807K	962KB

1) Search Space: The search space design affects the sampled model size. An inappropriate search space have a negative impact on the experimental results. As shown in Fig. 9(a), we conduct experiments on three different sizes of search spaces based on the parameters counts in sampled supernet, including *small* (0.3∼1M), *normal* (0.6∼3M), and *large* (1.3∼10M). With hyper-parameter $\lambda_{lo} = 0.8$ and $\lambda_{hi} = 2.8$, only *normal* space has acceptance probability higher than 90%. The *small* and *large* space has an acceptance probability less than 30%. The *small* search space mostly contains dense models with low capacity, while models in *large* search space have both higher parameter counts and higher sparsity. Figure 9(b) illustrates trade-off between sparsity and accuracy. We set three search space: *Small*, *Normal*, and *Large*. The models in *Small* search space are with lower scale and lower sparsity, The *Large* search space demonstrates the inverse relationship.



(a) (b)

Fig. 9: (a) The obtained model size from three types of search space. The model sampled from a small search space satisfies the hardware constraints without pruning. The models sampled from large search spaces require compression to meet the constraints. (b) The best-performing model in each search space before/after the quantization and pruning process.

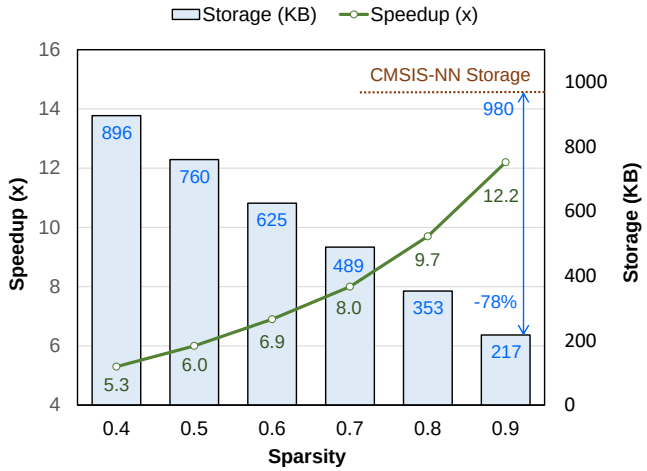


Fig. 10: Performance comparison of SparseEngine and CMSIS-NN on CIFAR-10. SparseEngine is $5.3\times\sim12.2\times$ faster than CMSIS-NN under different sparsity, while saving $9\%\sim78\%$ storage.

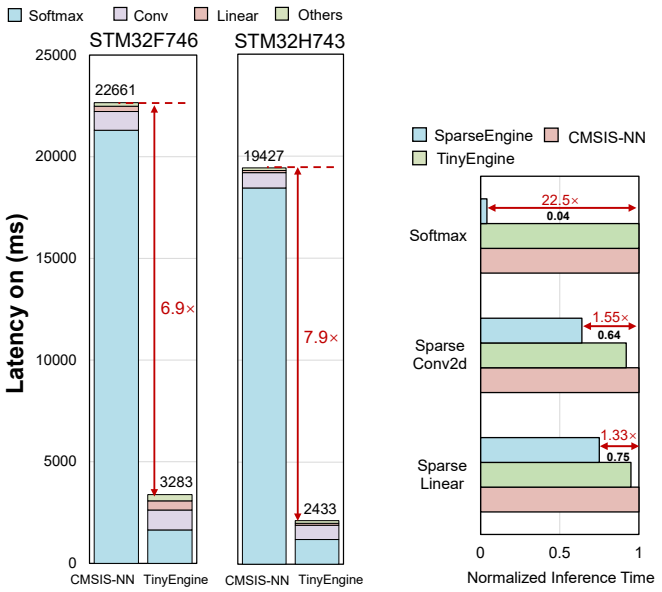
The *Normal* search space represents an intermediate configuration. Within the *Small* search space, lower sparsity results in lower accuracy degradation (0.20%). However, the accuracy of model in *Small* search space is substantially constrained by the hardware constraints. By contrast, the *Large* search space searches dense models with highest accuracy, but experiences more significant accuracy degradation (5.45%) during pruning for MCU deployment. The *Normal* search space optimally balances sparsity and accuracy trade-offs, with higher accuracy of dense model compared with small search space, and less accuracy degradation during pruning compared with large search space.

2) Ablation Study: In this part, we show the ablation study on TinyFormer from Tab. III, which consists of three perspectives: whether to use *transformer block*, number of DoTs, and whether to adopt mixed block size pruning in search stage. In Tab. III, TinyFormer refers to the TinyFormer-300K model in Tab. II, with *transformer block*, two DoTs, and mixed block size pruning.

To discover the impact of *transformer block* in TinyFormer, a CNN model without *transformer block* is searched and denoted as TinyFormer (w/o Tr.). TinyFormer (w/o Tr.) deletes the *transformer block* in DoT architecture, and only remains the *downsample block* and *MobileNetV2 block*. Compared to TinyFormer (w/o Tr.), TinyFormer obtains better accuracy under almost the same resource usage. These results suggest that incorporating transformer-related blocks or layers can offer advantages in achieving improved performance for TinyML.

Additionally, we conduct experiments to evaluate the impact of the number of DoT layers on the model's accuracy. TinyFormer (Single DoT) is derived from the supernet that exclusively consists of a single DoT architecture, while adhering to the same hardware constraints. With the same search space, TinyFormer (Single DoT) utilizes only 66.8% of the storage limitation, resulting in a decrease in accuracy by 1.48%. When employing a single DoT, the main bottleneck for TinyFormer (Single DoT) arises from memory constraints. By contrast, the model with two DoTs almost approaches the limits of both storage and memory, effectively maximizing resource utilization. Therefore, two DoTs are utilized in our basic experiments.

Finally, we evaluate the effectiveness of the mixed block size strategy in two stages of SparseNAS. In particular, TinyFormer (Block Size $=2$) and TinyFormer (Block Size $=4$) denote models with a fixed block size of 2 and 4 in block pruning respectively. Tab. III demonstrates that larger block sizes allow for more efficient storage of weights within the given limitations. However, setting all block sizes to 4 adversely affects the model's accuracy. Based on these results, the pruning method employing a mixed block size scheme strikes the best balance between block size and the number of effective weights, thereby yielding the most suitable model



(a) (b)

Fig. 11: (a) In CMSIS-NN, `Softmax` operator occupies 94% of the latency. SparseEngine optimizes the procedure of texttttSoftmax to achieve 6.9× acceleration on STM32F746 and 7.9× acceleration on STM32H43, with the average of 60% sparsity. (b) Normalized inference time of SparseEngine, TinyEngine and CMSIS-NN on difference operators.

with optimal performance.

C. Runtime Evaluation

At the runtime level, we have developed SparseEngine for performing sparse inference on MCUs. The implementation of SparseEngine has successfully reduced the inference time to 3.8 seconds on our highest-accuracy searched model. To evaluate the performance of SparseEngine, we deploy the same sparse model in both CMSIS-NN and SparseEngine. As illustrated in Fig. 10, SparseEngine outperforms CMSIS-NN both in terms of inference latency and storage occupation. By leveraging sparse inference support, SparseEngine achieves a significant reduction in storage requirements on MCUs, ranging from 9% to 78%. Through the utilization of specialized optimizations, SparseEngine achieves an impressive acceleration of inference, ranging from 5.3× to 12.2×. Specifically, in Fig. 10, we mainly focus on how sparse configurations affect the inference speed and storage usage, rather than the accuracy. For the simplicity of presentation, we have not shown the accuracy of models in the figure.

To identify the bottleneck in the inference process, we evaluate the runtime breakdown for different layers. As depicted in Fig. 11(a), our findings indicate that the `Softmax` operator is responsible for the majority of the inference time. Within the `Softmax` operator, the most time-consuming step is the negative exponential calculation. We discovered that as the input size increases, the results of the negative exponential can be reused. Leveraging this observation, SparseEngine employs a bitmap lookup table to reduce redundant calculations and

TABLE IV: Comparasions of LayerNorm (LN) inference. Experimental configurations are similar to TinyFormer-300K in Tab. II except for the LayerNorm implementations. We make an ablation study on different methods of LN: inference with `FP32` format (`FP32`-LN), inference with naive quantization to `INT8` (`INT8`-LN (naive)), and our proposed method in Sec. III-C3 (`INT8`-LN (scaled))

Method of LN	Accuracy	Shape	Latency
`FP32`-LN	96.11%	[256 × 44]	194ms
		[64 × 192]	208ms
`INT8`-LN (naive)	95.92%	[256 × 44]	5ms
		[64 × 192]	4ms
`INT8`-LN (scaled)	96.10%	[256 × 44]	5ms
		[64 × 192]	4ms

optimize the `Softmax` operator. Fig. 11(b) provides a comparison of the inference time for the main operators between CMSIS-NN and SparseEngine. Since the dense model can not be deployed on MCU using CMSIS-NN, to make a fair comparison, we use CMSIS-NN to perform sparse inference with the same algorithm as SparseEngine in Fig. 8, but without the SIMD acceleration. The setting allows all the engines has the same memory usage when calculating the same operators. The algorithm first decodes the sparse weight to get the weight information. After decoding, the algorithm extracts the corresponding sub-matrix from the activation map, transforming it to column and perform the multiple-add calculation. Compared with CMSIS-NN, SparseEngine achieves 1.33× to 1.55× acceleration rate in `Conv2d` and `Linear` operator, and 22.5× acceleration in `Softmax` operator.

Moreover, we conducted a comparative analysis of power consumption between CMSIS-NN and SparseEngine on the STM32F746. Under the same voltage, the average power consumption of TinyFormer was measured at 473 mW, with a 3% reduction compared to the average power consumption of CMSIS-NN. Adoption of a lookup table for softmax cut down the amount of calculation, and results in a slight reduction on power consumption. Although the power reduction effect of SparseEngine is not particularly significant, SparseEngine achieve much less inference time compared to CMSIS-NN. The total power consumption of SparseEngine for image recognition tasks was only 14% of that of CMSIS-NN. The analyzing results suggest that SparseEngine holds a strong advantage in the TinyML applications that use transformers and require low-energy consumption.

Additionally, we conduct an experiment involving Scaled-LayerNorm in TinyFormer. Scaled-LayerNorm executes integer-arithmetic calculations, making it more suitable for model inference on MCUs. Scaled-LayerNorm addresses the issue of precision loss in quantization by expanding the range of normalization results during the rounding operation. Tab. IV illustrates the impact of Scaled-LayerNorm on acceleration. Remarkably, the accuracy of TinyFormer using Scaled-LayerNorm is nearly equivalent to that of the normal

LayerNorm computed in `FP32` format, while the inference procedure is accelerated by a factor of $38.8\times$ to $52.0\times$. These experimental results align with our expectations, as Scaled-LayerNorm significantly enhances the efficiency of LayerNorm inference without any noticeable loss in accuracy. In summary, the experimental results for both `Softmax` and Scaled-LayerNorm validate our observations and highlight the substantial acceleration effects achieved by SparseEngine.

V. Conclusions

In this work, TinyFormer is proposed as an innovative framework for developing efficient transformers on MCUs by integrating SuperNAS, SparseNAS, and SparseEngine. One notable feature of TinyFormer is its ability to produce sparse models with high accuracy while adhering to hardware constraints. By integrating model sparsity and neural architecture search, TinyFormer achieves a delicate balance between efficiency and performance. Along with the automated deployment approaches, TinyFormer can further accomplish efficient sparse inference with a guaranteed latency on targeted MCUs. Experimental results demonstrate that TinyFormer could achieve 96.1% accuracy on CIFAR-10 under the limitations of 1MB storage and 320KB memory. Compared with CMSIS-NN, TinyFormer achieves a remarkable speedup of up to $12.2\times$ and reduces storage requirements by up to 78%. These achievements not only bring powerful transformers into TinyML scenarios but also greatly expand the scope of deep learning applications.

References

[1] M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, "TinyML: current progress, research challenges, and future roadmap," in Proceedings of 58th ACM/IEEE Design Automation Conference (DAC), 2021, pp. 1303–1306.

[2] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello Edge: Keyword spotting on microcontrollers," arXiv preprint arXiv:1711.07128, 2017.

[3] R. Kallimani, K. Pai, P. Raghuwanshi, S. Iyer, and O. L. López, "TinyML: Tools, Applications, Challenges, and Future Research Directions," arXiv preprint arXiv:2303.13569, 2023.

[4] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov et al., "Benchmarking TinyML systems: Challenges and direction," arXiv preprint arXiv:2003.04821, 2020.

[5] C. Banbury, C. Zhou, I. Fedorov, R. Matas, U. Thakker, D. Gope, V. Janapa Reddi, M. Mattina, and P. Whatmough, "MicroNets: Neural network architectures for deploying tinyml applications on commodity microcontrollers," in Proceedings of Machine Learning and Systems (MLSys), 2021, pp. 517–532.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in Proceedings of Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All You Need," in Proceedings of Advances in Neural Information Processing Systems (NeurIPS), 2017, pp. 5998–6008.

[8] A. Gulati, J. Qin, C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu, and R. Pang, "Conformer: Convolution-augmented Transformer for Speech Recognition," in Proceedings of 21st Annual Conference of the International Speech Communication Association (INTERSPEECH), 2020, pp. 5036–5040.

[9] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, U. Thomas, M. Dehghani, M. Minderer, H. Georg, S. Gelly, U. Jakob, and H. Neil, "An Image is Worth 16×16 Words: Transformers for Image Recognition at Scale," in Proceedings of 9th International Conference on Learning Representations (ICLR), 2021.

[10] S. Mehta and M. Rastegari, "MobileViT: Light-weight, General-purpose, and Mobile-friendly Vision Transformer," in Proceedings of 10th International Conference on Learning Representations (ICLR), 2022.

[11] H. Zhang, W. Hu, and X. Wang, "EdgeFormer: Improving Light-weight ConvNets by Learning from Vision Transformers," arXiv preprint arXiv:2203.03952, 2022.

[12] A. Hassani, S. Walton, N. Shah, A. Abuduweili, J. Li, and H. Shi, "Escaping the Big Data Paradigm with Compact Transformers," arXiv preprint arXiv:2104.05704, 2021.

[13] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient Neural Network Kernels for Arm Cortex-M CPUs," arXiv preprint arXiv:1801.06601, 2018.

[14] M. Rusci, A. Capotondi, and L. Benini, "Memory-Driven Mixed Low Precision Quantization For Enabling Deep Network Inference On Microcontrollers," in Proceedings of Machine Learning and Systems (MLSys), 2020.

[15] M. Rusci, M. Fariselli, A. Capotondi, and L. Benini, "Leveraging automated mixed-low-precision quantization for tiny edge microcontrollers," IoT Streams for Data-Driven Predictive Maintenance and IoT, Edge, and Mobile for Embedded Machine Learning, vol. 1325, pp. 296–308, 2020.

[16] J. Lin, W.-M. Chen, Y. Lin, C. John, C. Gan, and S. Han, "MCUNet: tiny deep learning on IoT devices," in Proceedings of Advances in Neural Information Processing Systems (NeurIPS), 2020, pp. 11 711–11 722.

[17] J. Lin, W. Chen, H. Cai, C. Gan, and S. Han, "Mcunetv2: Memory-efficient patch-based inference for tiny deep learning," CoRR, vol. abs/2110.15352, 2021.

[18] J. Lin, L. Zhu, W. Chen, W. Wang, C. Gan, and S. Han, "On-device training under 256kb memory," in Proceedings of Advances in Neural Information Processing Systems (NeurIPS), 2022.

[19] A. Burrello, M. Scherer, M. Zanghieri, F. Conti, and L. Benini, "A microcontroller is all you need: Enabling transformer execution on low-power iot endnodes," in Proceedings of IEEE International Conference on Omni-Layer Intelligent Systems (COINS), 2021, pp. 1–6.

[20] L. Yinan, W. Ziwei, X. Xiuwei, T. Yansong, Z. Jie, and L. Jiwen, "Mcuformer: Deploying vision tranformers on microcontrollers with limited memory," in Proceedings of Advances in Neural Information Processing Systems (NeurIPS), 2023.

[21] B. Zoph and Q. V. Le, "Neural Architecture Search with Reinforcement Learning," in Proceedings of International Conference on Learning Representations (ICLR), 2017.

[22] A. Zhou, J. Yang, Y. Qi, Y. Shi, T. Qiao, W. Zhao, and C. Hu, "Hardware-Aware Graph Neural Network Automated Design for Edge Computing Platforms," in Proceedings of 60th ACM/IEEE Design Automation Conference (DAC), 2023.

[23] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search," in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 10 734–10 742.

[24] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct neural architecture search on target task and hardware," in Proceedings of 7th International Conference on Learning Representations (ICLR), 2019.

[25] W. Jiang, L. Yang, E. H. Sha, Q. Zhuge, S. Gu, S. Dasgupta, Y. Shi, and J. Hu, "Hardware/software co-exploration of neural architectures," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 39, no. 12,

pp. 4805–4815, 2020.
[26] I. Fedorov, R. P. Adams, M. Mattina, and P. Whatmough, "SpArSe: Sparse architecture search for CNNs on resource-constrained microcontrollers," in Proceedings of Advances in Neural Information Processing Systems (NeurIPS), 2019, pp. 4978–4990.
[27] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlu, J. Levenberg, R. Monga, S. Moore, D. G.Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zhang, "TensorFlow: A system for large-scale machine learning," in Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016, pp. 265–283.
[28] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in Proceedings of 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2018, pp. 578–594.
[29] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices," IEEE Transactions on Circuits and Systems II (TCAS-II), pp. 871–875, 2020.
[30] Z. Guo, X. Zhang, H. Mu, W. Heng, Z. Liu, Y. Wei, and J. Sun, "Single path one-shot neural architecture search with uniform sampling," in Proceedings of 16th European Conference on Computer Vision (ECCV), vol. 12361, 2020, pp. 544–560.
[31] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 4510–4520.
[32] A. F. Agarap, "Deep learning using rectified linear units (ReLU)," arXiv preprint arXiv:1803.08375, 2018.
[33] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," in Proceedings of 6th International Conference on Learning Representations (ICLR), 2018.
[34] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," arXiv preprint arXiv:1806.08342, 2018.
[35] A. H. Robinson and C. Cherry, "Results of a prototype television bandwidth compression scheme," in Proceedings of the IEEE, 1967, pp. 356–364.
[36] J. Yang, W. Fu, X. Cheng, X. Ye, P. Dai, and W. Zhao, "S2Engine: A novel systolic architecture for sparse convolutional neural networks," IEEE Transactions on Computers (TC), vol. 71, no. 6, pp. 1440–1452, 2021.
[37] P. Dai, J. Yang, X. Ye, X. Cheng, J. Luo, L. Song, Y. Chen, and W. Zhao, "SparseTrain: Exploiting dataflow sparsity for efficient convolutional neural networks training," in Proceedings of 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1–6.
[38] S. Mehta and M. Rastegari, "Separable self-attention for mobile vision transformers," Trans. Mach. Learn. Res., vol. 2023, 2023.
[39] A. Krizhevsky, G. Hinton et al., "Learning Multiple Layers of Features from Tiny Images," 2009.
[40] P. Chrabaszcz, I. Loshchilov, and F. Hutter, "A downsampled variant of imagenet as an alternative to the CIFAR datasets," 2017.
[41] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," in Proceedings of 7th International Conference on Learning Representations (ICLR), 2019.
[42] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 2818–2826.
[43] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in International Conference on Computer Vision, (ICCV), 2015, pp. 1026–1034.

Jianlei Yang (S'11-M'14-SM'20) received the B.S. degree in microelectronics from Xidian University, Xi'an, China, in 2009, and the Ph.D. degree in computer science and technology from Tsinghua University, Beijing, China, in 2014.

He is currently a Professor in Beihang University, Beijing, China, with the School of Computer Science and Engineering. From 2014 to 2016, he was a post-doctoral researcher with the Department of ECE, University of Pittsburgh, Pennsylvania, USA. His current research interests include emerging computer architectures, hardware-software co-design and machine learning systems.

Dr. Yang was the recipient of the First/Second place on ACM TAU Power Grid Simulation Contest in 2011 and 2012. He was a recipient of IEEE ICCD Best Paper Award in 2013, ACM GLSVLSI Best Paper Nomination in 2015, IEEE ICESS Best Paper Award in 2017, ACM SIGKDD Best Student Paper Award in 2020.

Jiacheng Liao received the B.S. degree in Computer Science and Technology from Beihang University, Beijing, China, in 2022, and M.S. degree in Computer Science and Technology in Beihang University, Beijing, China, in 2025. His current research interests include TinyML and ML systems.

Fanding Lei received the B.S. degree in Information and Electronics from Beijing Institute of Technology, Beijing, China, in 2020, and M.S. degree in Computer Science and Technology in Beihang University, Beijing, China, in 2023. His current research interests include TinyML and ML systems.

Meichen Liu received the B.S. degree in Computer Science and Technology from Beihang University, Beijing, China, in 2021, and M.S. degree in Computer Science and Technology in Beihang University, Beijing, China, in 2024. Her current research interests include TinyML and ML systems.

Lingkun Long received the B.S. degree in Computer Science and Technology from Beihang University, Beijing, China, in 2024. He is currently working toward the M.S. degree in Computer Science and Technology in Beihang University, Beijing, China. His current research interests include LLM inference and ML systems.

Junyi Chen received the B.S. degree in Computer Science and Technology from Beihang University, Beijing, China, in 2024. He is currently working toward the M.S. degree in Computer Science and Technology in Shanghai Jiao Tong University, Shanghai, China. His current research interests include ML systems.

Han Wan received the B.S. degree and Ph.D. degree in Computer Science and Technology from Beihang University, Beijing, China, in 2003 and 2011, respectively. She is currently an Associate Professor with the School of Computer Science and Engineering at Beihang University. From 2015 to 2016, she was a visiting scholar with the Education Research Group, Massachusetts Institute of Technology (MIT).

Her research interests include computer architectures and systems, educational data mining.

Bei Yu (M'15-SM'22) received the Ph.D. degree from The University of Texas at Austin in 2014. He is currently a Professor in the Department of Computer Science and Engineering, The Chinese University of Hong Kong.

He has served as TPC Chair of ACM/IEEE Workshop on Machine Learning for CAD, and in many journal editorial boards and conference committees. He is Editor of IEEE TCCPS Newsletter. He received nine Best Paper Awards from DATE 2022, ICCAD 2021 & 2013, ASPDAC 2021 & 2012, ICTAI 2019, Integration, the VLSI Journal in 2018, ISPD 2017, SPIE Advanced Lithography Conference 2016, and six ICCAD/ISPD contest awards.

Weisheng Zhao (Fellow, IEEE) received the Ph.D. degree in physics from the University of Paris Sud, Paris, France, in 2007.

He is currently a Professor with the School of Integrated Circuit Science and Engineering, Beihang University, Beijing, China. In 2009, he joined the French National Research Center, Paris, as a Tenured Research Scientist. Since 2014, he has been a Distinguished Professor with Beihang University. He has published more than 200 scientific articles in leading journals and conferences, such as *Nature Electronics*, *Nature Communications*, *Advanced Materials*, IEEE Transactions, ISCA, and DAC. His current research interests include the hybrid integration of nanodevices with CMOS circuit and new nonvolatile memory (40-nm technology node and below) like MRAM circuit and architecture design.

Prof. Zhao was the Editor-in-Chief for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEM I: REGULAR PAPER from 2020 to 2023.