Deep Learning on MCUs: Comparative Analysis of Compile and Interpreter based Execution Methods

Gunju Park* , Seungtae Hong*† , Jeong-Si Kim*
*Artificial Intelligence Computing Research Lab.
Electronics and Telecommunications Research Institute, Deajeon, Republic of Korea
Emails : parkgj@etri.re.kr , sthong@etri.re.kr , sikim00@etri.re.kr
†University of Science and Technology, Daejeon, Korea
Emails : sthong@etri.re.kr

***Abstract*—With the rapid advancements in Edge Deep Learning research, the focus has shifted from optimizing on-device deep learning inference on smartphones and mobile boards, like Nvidia's Jetson, to executing deep learning models on highly constrained computational resources of a Micro Control Unit (MCU). However, the limited operational resources of these MCU devices pose significant challenges. The Flash ROM memory, which contains the weights of deep learning models, is usually around 1-2MB, while the SRAM memory, used for managing runtime tensors, ranges approximately from 300kB to 1MB. These constraints make conventional on-device inference techniques challenging. This paper offers a comprehensive guide for performing AI inference within the restricted computational confines of an MCU and juxtaposes the efficiency of two runtime methods for implementing deep learning models within the MCU: the Compile-based method and the Interpreter-based method.**

***Index Terms*—Edge AI, MCU, Lightweight CNN, TinyML**

I. Introduction

The need for deep learning inference in various edge industries such as autonomous driving, smart home, healthcare, augmented reality (AR), and robotics has been growing recently [1]. The traditional Cloud-Edge method [2], which requires network communication, has several limitations. It cannot operate in environments without network availability, and issues regarding privacy and the energy consumption used for network communication in battery-based independent power systems are significant.

To address these issues, many researches are being conducted on on-device edge inference, which optimizes complex deep learning model inference by utilizing only the computational resources within the device to save energy consumption.

However, Micro Control Units (MCUs) are used in modules responsible for sensor processing in autonomous vehicles, intelligent robots, and smart IoT modules in smart homes. Compared to the Edge ML level equipped with traditional embedded GPUs or NPUs, they possess significantly limited computational resources. Thus, executing deep learning models on MCUs presents considerable challenges. The methodology to solve these problems is referred to as TinyML and is a subject of active research [3].

This paper introduces the overall process for executing deep learning models in the extremely constrained computational environment of MCUs, and analyzes and compares intrepreter-based and compile-based runtime inference approaches.

II. Methods and Analysis

A. MCU-Based AI Model Inference: An Overview

In order to perform deep learning model computations such as Convolution Neural Networks (CNNs), including MobileNet, on a Micro Control Unit (MCU), the model must first be trained and optimized at the host level before being converted into a TFLite model format.

Several model optimization methods are available. First, there is the Quantization technique [4]. The extremely limited memory space of MCU's SRAM (300KB - 1MB) and Flash ROM (1-2MB) necessitates this step as using the standard 32-bit floating point tensors of conventional AI models would exceed memory capacity. By applying Full Integer Quantization to both the model weight tensors and the input/output (activation) tensors, the required memory can be compressed to a quarter of its original size. If the normalization of input image data distribution is considered during Quantization, preprocessing can be integrated into the scale factor of the 8-bit model, eliminating the need for a separate preprocessing phase in the application layer.

Secondly, Pruning [5] sets less influential values in the Conv Filter's weights to zero or eliminates less significant channels within the filter, achieving up to 80% sparsity. Techniques such as Graph Fusion consolidate operations like Convolution, Batch-normalization, and ReLU into a single operation, thus reducing the number of necessary layers. Neural Architecture Search (NAS) [6] can also be used to balance the memory constraints of the MCU and high accuracy.

Post these optimizations, the model is converted to a TFLite format, which requires a Runtime Backend for model inference on the MCU. There are two primary types of runtime systems: Interpreter-based, exemplified by Google's open-source Tensorflow Lite Micro (TFLM) [7], and Compile-based, which includes the open-source like TinyEngine [6] from MIT's SongHanLab, micro-TVM from Apache. Once the runtime setting has been established as either type 2-a or 2-b as per Fig.1, An application is then deployed as firmware onto the MCU.

After initializing the MCU device, the input data, which can either be read from a JPEG compressed image file on an SD card or obtained as raw image data from a camera module, is

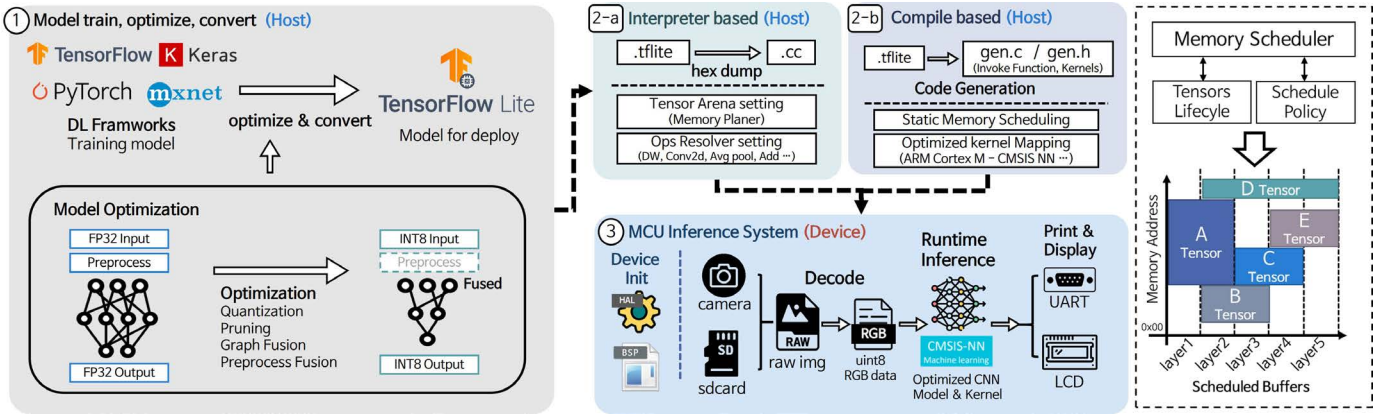

Fig. 1: MCU Inference system overview

decoded using a lightweight libJPEG and stored in unsigned char array buffers. This array is then used as the model input. When this array is used as the model input and the Runtime functions for model execution are run, the system internally performs computations using optimized low-level kernels for each layer operation.

Finally, memory planning is utilized as illustrated in Fig. 1's right-hand side. This involves mapping each tensor, used for model inference, to the address of the SRAM buffer memory, considering each tensor's lifecycle or the points required for inference operation. Through memory planning, the Tensor memory space used for overall inference can be reused, reducing the peak memory value.

B. Comparative Analysis : Interpreter-Based and Compile-Based Methods

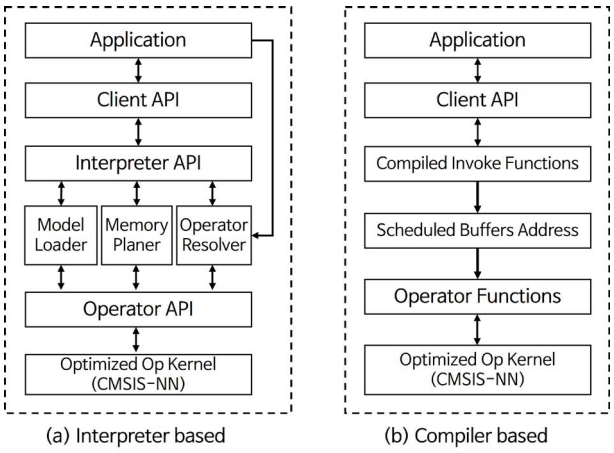

Fig. 2: Runtime execution flow diagrams

We compare the operational mechanisms of both runtimes at the Host level and the Device level respectively. The tasks performed at the Host level are those indicated as 2-a and 2-b in Fig. 1.

1) Host - Interpreter method: Interpreter-based methods, such as TFLM, perform simple tasks at the Host level, such as basic parameter setting or dumping the model and data in Hex format. They offer an API to set parameters like calculating the appropriate Tensor Arena size based on the given model. Since including all operations not contained in the model would increase firmware size, parameters are set in the Ops Resolver to include only necessary operations.

2) Host - Compile method: Compile-based methods like TinyEngine perform a relatively large number of tasks at the Host level. They write code for buffer (tensor) variables in advance for Read-only data such as weights, biases, and scale values based on parsed TFLite model data. At runtime, each tensor's address and size are determined in advance through Memory Scheduling based on Activation tensor information. They also select the optimal Kernel for each layer operation, map it, and generate the corresponding operation function code. Eventually, they generate the Invoke function that runs the entire model. Typically, codes are generated in C language.

Next, we analyze the tasks performed at the Device level (Fig.1's No. 3 and Fig.2's (a), (b)).

3) Device - Interpreter method: Interpreter methods carry out most tasks at the Device level, from mapping the model to usable data structure and building the interpreter to memory planning within the given Tensor Arena size, and Operator-Kernel mapping. These tasks are executed only once as an initialization stage for AI inference, but a slight overhead occurs at the Device. The actual model operation works through the interaction of the built Interpreter and the Operator API mapped to the Optimized Kernel, which perform the entire model's Invoke.

4) Device - Compile method: On the other hand, as most of the tasks are completed at the Host level in the Compile method, the Device firmware code simply calls the Invoke function of the generated code. Therefore, AI inference can be performed without any additional overhead at runtime.

III. Evaluation

We set up the experimental environment on the STM32F746G-Disco MCU board and performed experiments based on the CNN models (Table. 1 - Imagenet, VWW) from the MCUNet Model-zoo for performance measurement.

net id	MACs	Params	Top-1(int8)
mcunet-imagenet1	12.8M	0.64M	49.9%
mcunet-vww1	11.6M	0.43M	88.9%

TABLE I: MCUNet Networks Info

We conducted experiments using TinyEngine for the compile-based approach and Tensorflow Lite Micro for the interpreter-based approach.

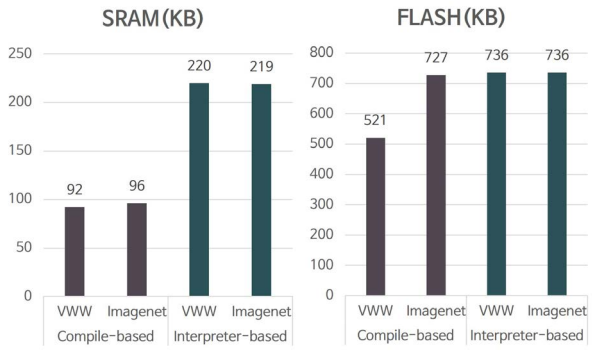

Fig. 3: Memory usage on MCU

First, we explored the difference in memory allocation for the tensors stored in the MCU device's FLASH and SRAM memory, based on the model-specific runtime methods. The size of model parameters in the FLASH memory showed little variation. However, the required SRAM memory space could be reduced by more than half depending on the efficiency of memory planning in each runtime algorithm.

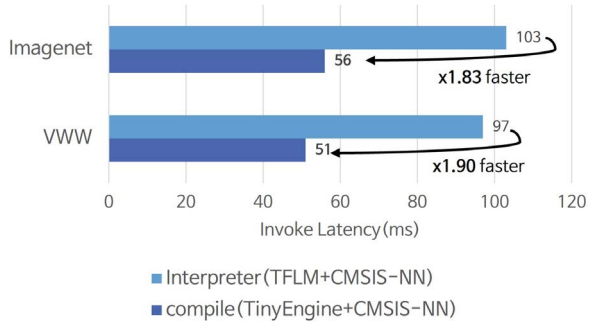

Fig. 4: Model invoke latency comparison

Next, we examined model execution time. We found that the latency of executing the Invoke operation for the entire model was about twice as fast in both models, likely due to additional optimizations by TinyEngine and potential suboptimal kernel parameter settings during each operation in the interpreter-based inference.

Lastly, we considered the latency during the setup stage in the interpreter-based runtime. Despite the setup process only occurring once, it may contribute a slight overhead when AI inference is triggered on MCUs predominantly deployed on sensors, affecting immediate inference performance. This

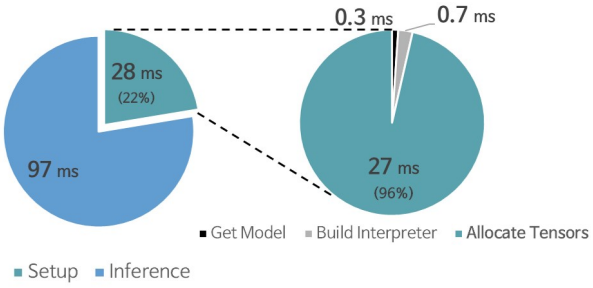

Fig. 5: Interpreter setup overhead on VWW model

overhead accounts for about 22% or 28ms of the VWW inference execution time, with the planning and allocation process being the most significant cause.

IV. Conclusion

In designing a model, the interpreter-based approach provides a suitable platform for efficient implementation and model verification. However, considering the challenges associated with optimizing and maintaining a new model in a compile-based environment, it is highly advantageous to meticulously design the model via TFLM and subsequently deploy using the compile-based method. Ultimately, a strategic integration of the strengths inherent to both the interpreter-based and compile-based methodologies can foster the development of efficient, optimized, and easily maintainable models. This approach has the potential to significantly accelerate advancements in machine learning applications

Acknowledgment

This research was supported by the Challengeable Future Defense Technology Research and Development Program through the Agency For Defense Development(ADD) funded by the Defense Acquisition Program Administration(DAPA) in 2022(No.915062201)

References

[1] C. Lee, S. Hong, S. Hong, and T. Kim, "Performance analysis of local exit for distributed deep neural networks over cloud and edge computing," *ETRI Journal*, vol. 42, pp. 658–668, 10 2020.

[2] S. Kim, T. Choi, S. Song, E. Strinati, and J.-M. Chung, "Special issue on 5g b5g enabling edge computing, big data and deep learning technologies," *ETRI Journal*, vol. 42, pp. 639–642, 10 2020.

[3] A. N. Mazumder, J. Meng, H.-A. Rashid, U. Kallakuri, X. Zhang, J.-S. Seo, and T. Mohsenin, "A survey on the optimization of neural network accelerators for micro-ai on-device inference," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 11, no. 4, pp. 532–547, 2021.

[4] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "A survey of quantization methods for efficient neural network inference," *arXiv preprint arXiv:2103.13630*, 2021.

[5] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag, "What is the state of neural network pruning?" *Proceedings of machine learning and systems*, vol. 2, pp. 129–146, 2020.

[6] J. Lin, W.-M. Chen, H. Cai, C. Gan, and S. Han, "Mcunetv2: Memory-efficient patch-based inference for tiny deep learning," in *Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[7] R. David, J. Duke, A. Jain, V. Janapa Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, T. Wang *et al.*, "Tensorflow lite micro: Embedded machine learning for tinyml systems," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 800–811, 2021.