Power-Performance Characterization of TinyML Systems

Yujie Zhang, Dhananjaya Wijerathne, Zhaoying Li, Tulika Mitra
School of Computing, National University of Singapore
{zyujie, dmd, zhaoying, tulika}@comp.nus.edu.sg

***Abstract*—TinyML systems are enabling machine learning (ML) inference at the edge. However, there exists little quantitative analysis of such systems. This paper presents a systematic performance and power characterization of diverse TinyML applications on micro-controllers (MCUs), spanning neural network models, software libraries, operating systems, and hardware architectures. We focus on the impact of the multiple layers of abstractions that provide higher programmability at the expense of performance and energy efficiency. We propose a model to estimate the costs of different abstraction layers and make recommendations for minimizing those costs. Our findings can help designers with Neural Architecture Search (NAS) and CNN inference optimization on edge devices.**

***Index Terms*—TinyML, MCU, general matrix multiplication**

I. Introduction

Machine learning (ML) inference in the cloud is the most prevalent approach today. However, inference on edge devices closer to the data source is becoming increasingly important due to security and privacy concerns. Moreover, processing the data at the edge saves network bandwidth and energy. Finally, some applications require a real-time response that cannot be guaranteed with unpredictable network delay and availability. TinyML [1] is a special class of inference on ultra resource-constrained edge devices that can achieve data collection and processing in real-time with very little power [2].

There exists a large body of work on power and performance analysis of ML training systems [3], [4] to provide insights into future system design. There is, however, very little quantitative analysis of the capabilities of existing TinyML systems. Even though [5]–[7] analyze deep neural network (DNN) implementation on MCU or Edge TPU, they do not adopt a full-stack view to study the limitation of DNN implementation on edge devices. A systematic and fine-grained power-performance characterization can help us identify the bottleneck of the current TinyML system and improve the performance of the edge device.

More importantly, various TinyML optimizations complicate the performance analysis. Severely resource-constrained devices require significant neural network models, software, and hardware-level optimizations to support complex ML applications. For instance, model-level optimizations, such as weight pruning and quantization, reduce memory footprint and inference computation complexity with negligible loss of accuracy. Architecture-specific libraries and compilers like TensorFlow Lite Micro (TFLM) [8] framework for micro-controllers and CMSIS-NN library [9] for ARM Cortex-M processors influence inference latency. Moreover, micro-controller architecture and platform details also substantially impact the power-performance behavior of ML inference.

While each level (model, framework, compiler/library, architecture) can significantly affect the performance of TinyML applications, their subtle interactions also have considerable importance due to the ultra resource-constrained nature of these devices. Thus the final inference performance is simultaneously determined by the whole stack, from model structure, compilation and software optimization, and architecture. For instance, matrix multiplication implementation of DNN layers using single instruction multiple data (SIMD) instructions can significantly accelerate model inference and improve energy efficiency. In this case, latency/energy benefit is obtained from hardware-specific model design or optimization.

We provide a systematic power and performance analysis of TinyML systems, considering deep neural networks, the software library, the operating system, and hardware architectures. This analysis identifies the bottleneck of the existing systems and indicates how to optimize the system further.

We evaluate five representative TinyML applications [10], [11] on different MCU platforms regarding inference latency, power efficiency, and energy consumption. We explore the impact of abstraction layers (operating system and ML framework) on model inference. Next, we investigate the general matrix multiplication (GEMM) implementation in software libraries and suggest strategies to optimize the corner cases. We study how hardware characteristics affect inference performance and provide insights on supporting larger NN models in ML frameworks. Finally, we propose a model to predict the ML inference performance on MCUs considering the impact of the full stack. Our study provides insights into future system optimizations and can be easily integrated into the platform-specific neural architecture search (NAS) process, for example.

II. Related Work

Table I shows a comparison of related works in system benchmarking for ML applications with our study.

There is much systematical analysis of DNN training platforms to identify the weaknesses and strengths of various targeted hardware architectures under different model structures. In terms of ML inference on edge devices, there is very little quantitative analysis, which to some extent impedes the TinyML system development. Banbury et al. [2] point out that the main difficulty is the lack of a universally acknowledged

benchmark, and they present four TinyMLPerf benchmarks. They provide a way to benchmark TinyML systems but do not analyze the system behaviors.

Based on the observation on MCU platforms that model latency is proportional to model operation count, Banbury et al. [5] propose MicroNet models and achieve state-of-the-art performance for three TinyMLPerf benchmark tasks. Heim et al. [6] show that perceptible performance metrics, like inference latency and energy consumption, should be a viable proxy for model design and present a toolchain for TensorFlow Lite (TFLite) [12] model implementation on MCU platforms. Yazdanbakhsh et al. [7] evaluate three Edge TPUs and present a learned ML model to predict inference latency for convolutional kernels. While these works only focus on MCUs or Edge TPU, Baller et al. [13] provide an inclusive analysis of deep learning inference performance on various edge devices, like GPU, NPU, and MCU. Their analysis mainly focuses on model latency/energy with or without utilizing AI units. Wang et al. [14] provide quantitative power-performance analysis of different components on mobile SoCs for deep learning inference and propose synergistic engagement of all the components concurrently to improve throughput [15]. Compared with these works, we systematically analyze the overhead and limitation of model inference on MCUs from a full-stack perspective, discuss how to improve the performance and provide a prediction model.

TABLE I
SUMMARY OF STATE-OF-THE-ART

	Platforms	Highlight
[5]	MCU	State-of-the-art MicroNets based on differentiable NAS for three TinyML tasks.
[6]	MCU	A toolchain of TFLite model implementation on MCUs using Mbed OS for NAS.
[7]	Edge TPU	1. Microarchitectural insights of Edge TPUs. 2. Prediction model for faster evaluation of accelerators.
[13]	Four SoC, one MCU	Performance comparison of four SoCs and one MCU for different models and frameworks.
Our work	MCU	Recommendation to overcome the performance costs from a full-stack view.

III. Experimental Setup

To evaluate the model inference performance on edge devices, we select five representative TinyML applications and evaluate them on multiple MCU platforms.

MCU Platforms. Considering the compute capability and memory availability, we select four MCU platforms in Table II for the evaluation. The mature toolchains supporting model inference on these MCUs also facilitate our evaluation. We adopt Arduino Nano 33 BLE Sense [16], SparkFun Edge Apollo3 Blue [17], Nucleo-144 STM32F746ZG [18], and Nucleo-64 STM32L476RG [19] (Fig. 1). Their 32-bit ARM processor cores, Cortex-M4F or Cortex-M7F, are designed for cost and energy-efficient micro-controllers. Cortex-M7F can dual issue specific pairs of ALU instructions to achieve better performance. The devices vary in clock speeds, SRAM sizes, etc., which contributes to a compelling study of how hardware architectures affect inference performance. We will use their acronyms in Table II (MCU-S, MCU-M, MCU-B, MCU-D) to represent them for simplicity in the rest of the paper.

TABLE II
MCU ARCHITECTURE INFORMATION

	Arduino Nano 33 BLE Sense	Nucleo-64 STM32L476RG	SparkFun Edge	Nucleo-144 STM32F746ZG
Acronym	**MCU-S**	**MCU-M**	**MCU-B**	**MCU-D**
Processor	Cortex-M4F	Cortex-M4F	Cortex-M4F	Cortex-M7F
Frequency	64 MHz	80 MHz	96 MHz	216 MHz
Bandwidth	67.3 MB/s	86.9 MB/s	88.6 MB/s	251.6 MB/s
SRAM	256 KB	128 KB	384 KB	320 KB
Flash	1 MB	1 MB	1 MB	1 MB

TinyML Benchmarks. We use five benchmarks [10], [11] presented in Table III. These applications vary in input types, model structures, and computation intensity. Evaluating them on edge devices facilitates understanding the fine-grained impact of model structures on inference characteristics.

Model Implementation. We use TFLite to obtain the quantized int8 version. Next, TensorFlow Lite Micro (TFLM) [8] and CMSIS-NN library are adopted to run these applications on MCUs. TFLM does not need operating system support to run complex TinyML applications on resource-constrained MCUs, reducing the memory requirement. CMSIS-NN kernels leverage SIMD instructions to implement matrix multiplication, improving the inference latency and energy.

Experimental Measurement. We use Mbed Timer API and clock registers to measure inference latency on Mbed OS and Bare Metal, respectively. We use SmartPower2 [23] to record the average power during model execution and then multiply it with inference time to obtain the energy. We use J-Trace Pro Debug Probe in ARM Keil MDK Professional Edition to trace MCU processor registers (e.g., CYCCNT register) and count the instructions executed during model evaluation.

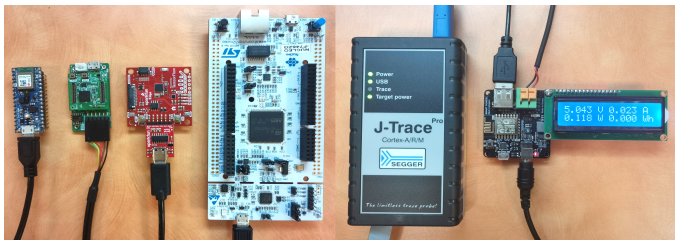

Fig. 1. Picture of Arduino Nano 33 BLE Sense, Nucleo-64 STM32L476RG, SparkFun Edge Apollo3 Blue, Nucleo-144 STM32F746ZG, J-Trace Pro Debug Probe, and SmartPower2 (from left to right).

IV. Impact of Abstraction Layers

We first explore the impact of multiple abstraction layers of the operating system and DNN framework on model inference performance on MCUs. The real-time operating system (RTOS) on embedded devices provides an interface between hardware peripherals and software programs. With multiple abstractions and implementation of Application Programming Interfaces (APIs), it greatly facilitates neural network inference programming. However, redundant support of APIs, like driver APIs and platform APIs, unavoidably causes application performance and energy efficiency degradation. Similarly, the

TABLE III
MODEL DESCRIPTION OF MAIN BENCHMARKING APPLICATIONS

	Micro Speech	Person Detection	Keyword Spotting	Image Classification	Anomaly Detection
Description	Recognizing two wake words	Recognizing Person	10 keyword spotting —— "No", "Up",...	Small image classification	Detecting anomalies in machine operating sounds
Model	——	MobileNet [20]	DS-CNN [21]	ResNet [22]	Deep AutoEncoder
Layers	Conv+FC	Conv+13 DSC+FC	Conv+4 DSC+FC	9 Conv+FC	10 FC
Dataset	Speech Commands	COCO	Speech Commands	Cifar10	ToyADMOS
#MACC	336k	7158k	2657k	12502k	264k
OPs	676k	15029k	5541k	25271k	538k
TFLite Model	18.7 KB	300.6 KB	59.6 KB	98.5 KB	277.0 KB

deep learning framework provides a unified way for neural network definition, interpretation, and execution, to improve program portability and generalization on different devices. However, programmability and generalization lead to higher inference latency and energy inefficiency.

We use *Micro Speech* as an example to show the impact of the above abstractions. *Micro Speech* consists of typical convolution and fully connected layers. Table IV summarizes the characteristics of different implementation choices on MCU-M: (1) naive model implementation in C without OS, DNN optimization libraries, and framework support, (2) C code with CMSIS-NN optimization, (3) C code with CMSIS-NN optimization on Mbed OS, (4) C code with CMSIS-NN optimization on simplified Mbed OS, and (5) TFLM framework with CMSIS-NN optimization on Mbed OS.

For implementation choice (1), we implement the model inference in pure C code and execute it bare-metal, i.e., we do not use any deep learning library, framework, or rely on an operating system. For implementation choice (2), we utilize CMSIS-NN kernels, which provide optimized implementation of convolution and fully connected operations. The inclusion of optimized kernels leads to around 10x improvement in runtime and energy efficiency. The reason is that CMSIS-NN utilizes SIMD instructions and matrix multiplication to reduce memory access and enable data reuse, significantly improving the model inference performance.

TABLE IV
OVERHEAD ANALYSIS OF ABSTRACTION LAYERS BASED ON MICROSPEECH ON MCU-M

	Latency (ms)	SRAM (KB)	Binary (KB)	Energy (mJ)
1) Unoptimized code+Bare-Metal	488.3	6.6	37	54.2
2) CMSIS-NN+Bare-Metal	46.2	4.1	28	5.1
3) CMSIS-NN+Mbed OS	47.5	8.1	83	8.4
4) CMSIS-NN+Simpl. Mbed OS	47.5	7.7	53	5.6
5) TFLM+CMSIS-NN+Mbed OS	48.9	11.8	99	9.5

Next, we explore the impact of the operating system in implementation choice (3) where we run the application with CMSIS-NN optimization on Mbed OS. Although Mbed OS support does not affect the latency much, adding OS support increases SRAM and Flash memory requirements by approximately 98% and 196%, respectively. Besides, as bare metal implementation does not need extra peripheral settings of Mbed OS, the latter also requires 65% additional energy. Thus, without extra peripherals and abstractions layer support, bare metal implementation allows small MCU platforms to support large models with less energy. To quantize the overhead of unnecessary peripheral support, we remove all unnecessary peripheral APIs support of Mbed OS to obtain a simplified operating system in implementation choice (4). Compared with that of full-fledged Mbed OS, model inference on simplified Mbed OS achieves considerable improvement in memory and energy and offers a good compromise between programmability and efficiency.

Finally, to evaluate the overhead of deep learning framework on model inference, we use TFLM for model representation and interpretation instead of direct implementation in C for implementation choice (5). TFLM on Mbed OS causes around 3% increase in inference time, energy, and memory requirements. Clearly, the advantages of TFLM abstraction far outweigh the overhead on performance efficiency.

Conclusion and Recommendation. Our study on the impact of operating system and ML framework on inference latency/energy and memory requirements quantifies the trade-off between programmability and efficiency. Specifically, the OS causes considerable overhead in terms of memory and energy. However, it can be mitigated to a large extent by carefully removing the support for redundant peripherals from the OS even if it requires some engineering effort. On the other hand, the deep learning framework itself adds little overhead and substantially reduces the designer's effort.

V. OPTIMIZATION LIBRARY LIMITATION

We now study a widely used and representative deep learning library CMSIS-NN on MCUs to analyze its performance limitations due to the interaction between the optimizations and the underlying architecture. Based on the study, we provide suggestions for NAS to avoid resource under-utilization due to inappropriate model parameter settings and further optimizations to utilize the limited resource.

A. SIMD Optimization Usage

Matrix multiplication is the most important and computationally intensive kernel for neural network inference [24]. Currently, many industry-standard backend NN libraries, such as CMSIS-NN, CMix-NN [25], X-CUBE-AI [26], and PULP-NN [27], utilize matrix multiplication to enable ML on resource-limited edge devices. In CMSIS-NN implementation, matrix multiplication kernels are used for most neural network layers (convolution, depthwise separable convolution,

and fully connected layer). However, this conversion is not always suitable.

Fig. 2 shows how 2D convolution operation with 2x2 filter weights can be lowered to matrix-vector multiplication. Input activations are replicated to create a matrix where each row corresponds to the elements in one convolution window (2x2 window). Therefore, one output activation can be generated by performing the dot product of the flattened filter weights with every row of the matrix. Similarly, 3D convolution and depthwise separable convolution (DSC) can be converted to general matrix multiplication (GEMM). Input activation replication is also known as input expanding, and this transformation is performed using the image-to-column (IM2COL) process.

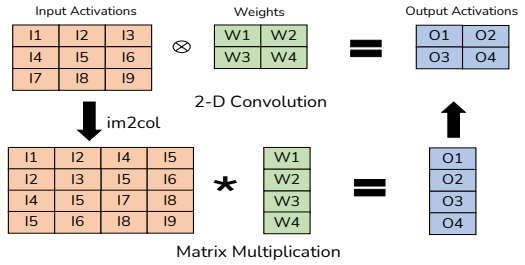

Fig. 2. An example of 2-D convolution implementation.

The actual implementation in CMSIS-NN is more complicated. For example, in the implementation of convolution operator, CMSIS-NN uses partial IM2COL, in which only a limited number of input columns are expanded to avoid high memory overhead. The matrix-multiplication kernel used by convolutions is implemented with a 2x2 kernel, which is carefully designed to fit the limited architectural registers of Cortex-M processors. Its computation is based on the dedicated Multiply-and-Accumulate (MAC) instruction *__SMLAD*. Computations that cannot utilize matrix-multiplication kernels due to tricky data alignment from inappropriate parameter settings are implemented through regular ALU instructions *MLA* and *ADD*. Compared with SIMD instructions, regular ALU instructions need more time for MAC computation. More extensive usage of SIMD instructions in convolution corresponds to improved runtime.

The following example layers of *Person Detection* application show this limitation of CMSIS-NN GEMM-based optimization. We list #MACC, #MEM, and latency of four Conv1x1 layers (A-D) in Table V. #MACC is the number of MACC operations while #MEM denotes the number of memory access operations. Both numbers are calculated based on the model description. As the executable memory needed by this application exceeds the SRAM size of MCU-M, we only show experimental results on other three MCUs.

As shown in Table V, layer B needs to execute nearly twice memory access and MACC operations compared to layer A. However, B's latency is only 0.69x times that of A's on MCU-B. On the other two platforms, this ratio is even lower. The small number of input channels of layer A fails to take advantage of GEMM conversion. Most of its computation has to be handled by normal ALU instructions without any SIMD optimization. In this case, layer A needs more inference time than some layers like layer B with large input channels, even though they have higher #MACC and #MEM. A similar observation applies to layer C and layer D.

TABLE V
CONV1X1 LAYER LATENCY ON MCU PLATFORMS

	#MACC	#MEM	MCU-S	MCU-B	MCU-D
Layer A	295k	627k	66 ms	31 ms	11 ms
Layer B	590k	1198k	40 ms	21 ms	6 ms
Ratio(B/A)	2x	1.91x	**0.61x**	**0.69x**	**0.53x**
Layer C	295k	608k	28 ms	15 ms	5 ms
Layer D	590k	1184k	31 ms	16 ms	4 ms
Ratio(D/C)	2x	1.95x	**1.09x**	**1.04x**	**0.92x**

Conclusion and Recommendation. Due to constant matrix multiplication kernel size, some computation of layers with inappropriate shapes has to be completed using normal instructions. However, naive computation without matrix multiplication kernels support increases inference latency due to slower MAC calculation and memory access. Careful parameter setting to enable converting all model computation to matrix multiplication can greatly facilitate inference performance.

B. Kernel Implementation Overheads

In this section, we investigate the overheads of CMSIS-NN kernel implementations for convolution, fully connected, and depthwise separable convolution layers.

Fig. 3 shows the inference latency of layers from five TinyML applications in Table III on MCU-S. The X-axis represents the number of MACC operations in each layer. Convolution layers typically require fewer parameters (weights and biases) than fully connected layers. Therefore operational intensity [28] (number of operations per memory access) of a fully connected layer is smaller than that of a convolutional layer. This indicates that if both the fully connected and convolution layers have the same number of MACC, the fully connected layer will have more memory accesses and then take more inference time. However, as shown in Fig. 3, the latency of the convolutional layer becomes higher than that of the fully connected layer with the same number of MACC operations. This inconsistency is also caused by the GEMM conversion in the CMSIS-NN kernel implementations.

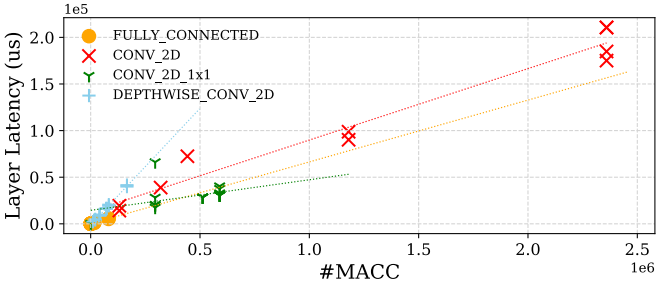

Fig. 3. Layer Latency on MCU-S.

As mentioned earlier, most NN libraries including CMSIS-NN, lower all three layers (convolution, fully connected, and depthwise separable convolution) to matrix multiplication

kernels. Convolution and DSC layers require the IM2COL process for input expansion. However, as fully connected layer implementation is inherently a matrix multiplication, it can be converted to matrix multiplication without input expansion. GEMM conversion in CMSIS-NN kernels generates overhead for convolution layer inference latency and memory footprint. The operational intensity of convolution decreases due to input expansion, which makes it easily limited by memory bandwidth. Moreover, reordering and expanding partial inputs also incur more time and memory costs than direct convolution. Even with partial IM2COL, extra memory cost is unavoidable.

With the same number of MACC operations, depthwise convolutions cause more overhead in the IM2COL process than convolutions. Although GEMM conversion decreases the operational intensity of the depthwise convolutions and the convolutions to a different extent, depthwise convolutions still exhibit higher latency per MACC than convolutions.

Next, we calculate the operational intensity of five applications based on model description and draw the roofline model of MCU-S in Fig. 4. The Y-axis on the left represents application performance (million operations per second, MOPS) and the right one denotes MOPS utilization, the ratio of application MOPS to peak MOPS. We utilize application's MOPS utilization to measure the computation capacity utilization efficiency on the platform. Based on the roofline model, these applications are all computation-bound. However, due to IM2COL overhead and decreased operational intensity from GEMM conversion, most applications, especially *Person Detection*, cannot fully utilize platform compute capability. For instance, the MOPS utilization of *Person Detection* and *Image Classification* are only 69.8% and 84.2%, respectively. Besides, *Anomaly Detection* consisting of fully connected layers achieves only 72.1% MOPS utilization, because of the incomplete utilization of SIMD optimization as mentioned in Section V-A. The layer inference computation without matrix-multiplication kernels requires more load instructions with one-byte inputs or weights access, significantly wasting memory bandwidth.

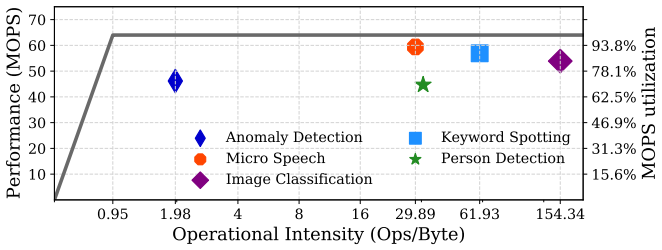

Fig. 4. Roofline model of MCU-S.

In order to investigate how GEMM conversion affects energy efficiency, we compare latency/energy for five applications on MCU-S, as shown in Fig. 5. The applications with more latency consume more energy due to the near-constant power during inference. Therefore, the overhead caused by GEMM conversion at runtime impedes energy efficiency.

Recommendation. In view of the cost of GEMM conversion in NN library kernels, direct convolution implementation with SIMD optimization should be explored as a promising alternative. For example, Zhang et al. [29] demonstrate that an implementation of direct convolution achieves better performance than existing state-of-the-art CPU-based convolution implementations. Gural et al. [30] propose memory-optimal direct convolutions by performing computations in place on resource-constrained devices.

VI. Architectural Impact

We now characterize the influence of hardware characteristics on power and performance in ML inference on MCUs.

A. Energy Efficiency

Fig. 5 shows the latency/energy comparison for applications on different MCUs. We cannot measure the power of MCU-B due to SmartPower2's restrictions, i.e., higher output voltage than what MCU-B can accept. So, we only report energy numbers of other three MCUs. MCU-D achieves the least model latency but the highest energy. Overall, the energy is proportional to the inference latency for all the MCUs. *Anomaly detection* has the lowest latency/energy and *Image classification* has the highest latency/energy. This is due to the near-constant power during inference. For instance, in Fig. 6, MCU-S power changes very little for different applications due to its relatively simple architecture.

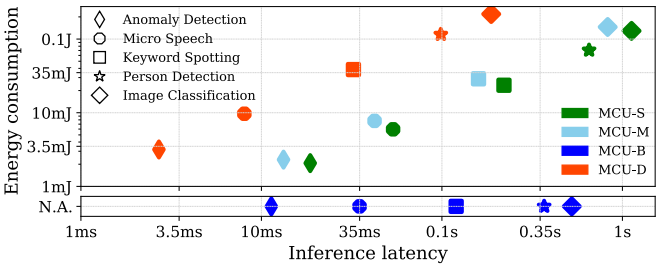

Fig. 5. Application latency/energy numbers on MCUs.

Fig. 6 shows the performance per watt for three MCUs, where the data for MCU-B is not shown as we cannot measure its power. The performance per watt of MCUs always follows the order: MCU-S>MCU-M>MCU-D. The high performance per watt of MCU-S is due to its low frequency and small memory. Although the SRAM size of MCU-M is only half of that of MCU-S, its higher frequency causes lower performance per watt. Moreover, as Cortex-M4F is designed for high energy efficiency, its performance per watt is higher than that of Cortex-M7F (MCU-D).

Recommendation. Executing the same model on an MCU with low computation/memory capability can achieve higher performance per watt.

B. Frequency and memory bandwidth

Next, we investigate the influence of hardware characteristics on inference performance. Table VI shows application

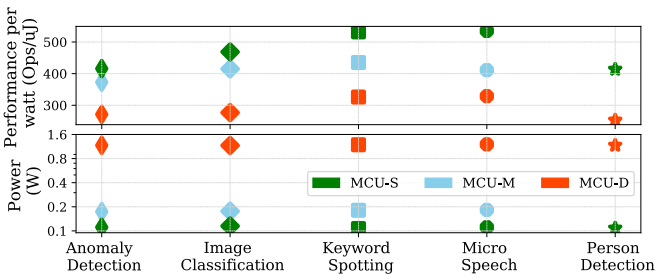

Fig. 6. Power and Performance per watt of MCUs.

throughput and hardware characteristics on MCUs, with MCU-S as the baseline. We observe that throughput is proportional to the bandwidth and frequency of MCU platforms. Note that MCU-D can dual-issue certain pairs of ALU instructions; so the throughput increase on MCU-D is twice its frequency increment compared to MCU-S. Also, as fully connected layers have more memory access per operation, higher memory bandwidth of MCU-D makes it achieve much lower latency. In contrast, convolution operations do not benefit much from high bandwidth. Therefore, among all the applications, *Image Classification*, which mainly consists of convolution layers, attains the best relative throughput on MCU-B and the worst relative throughput on MCU-D. Since the performance of *Image Classification* is more limited by compute capability instead of memory bandwidth, the impact of MCU memory bandwidth on application performance is weakened.

TABLE VI
APPLICATION THROUGHPUT AND HARDWARE CHARACTERISTICS COMPARISON ON MCUS (TAKING NUMBERS ON MCU-S AS BASELINE)

	MCU-M	MCU-B	MCU-D
Anomaly Detection	1.43	1.64	6.89
Micro Speech	1.27	1.53	6.67
Keyword Spotting	1.38	1.84	6.82
Person Detection	✗	1.77	6.63
Image Classification	1.36	**2.14**	**5.99**
Average Throughput	1.36	1.78	6.60
Processor Frequency	1.25	1.50	3.38
Memory Bandwidth	1.30	1.32	3.75

Recommendation. Neural networks with more fully connected or DSC layers are better suited for MCUs with large memory bandwidth. Models with more convolutions are better suited for MCU platforms with high frequency.

C. Memory Limitation

The ability of an MCU platform to support a specific ML application is dependent on memory requirements. Specifically, the application's working memory and executable binary have to reside in the platform's SRAM and Flash memory, respectively. We investigate these limitations with TFLM framework, the representative runtime for DNN on MCU.

The TFLM model's working memory mainly comprises intermediate tensors and persistent buffers, while generated binary includes model weights, graph definition, and operator implementation code [5]. To further explore factors affecting model working memory and binary size, we select three models, MicroNet-KWS-L, MicroNet-KWS-M, and MicroNet-KWS-S for *Keyword Spotting* [5]. Their specifications, including layer composition, TFLite model size, and #MACC, are listed in Table VII. We choose similar model structures and operators to eliminate the influence of software libraries on memory requirements for different operation implementations. We report the inference performance and memory requirements on MCU-S and MCU-D, considering memory availability and processor diversity as shown in Table VIII.

TABLE VII
THREE MICRONET MODELS FOR KEYWORD SPOTTING

	MicroNet-KWS-L	MicroNet-KWS-M	MicroNet-KWS-S
Model	659 KB	182 KB	115 KB
Arena	205 KB	105 KB	70 KB
Layers	Conv+7 DSC+FC	Conv+5 DSC+FC	Conv+5 DSC+FC
#MACC	65714k	15556k	8327k

TABLE VIII
THREE MICRONET MODEL INFERENCE ON TWO MCUS

Applications		MCU-S	MCU-D
MicroNet-KWS-L	Working Memory	254 KB	215 KB
	Binary	793 KB	738 KB
	Latency	3790 ms	566 ms
MicroNet-KWS-M	Working Memory	152 KB	112 KB
	Binary	316 KB	261 KB
	Latency	1118 ms	172 ms
MicroNet-KWS-S	Working Memory	116 KB	76 KB
	Binary	248 KB	193 KB
	Latency	646 ms	99 ms

Working Memory. TFLM defines a contiguous memory array called "arena" for intermediate model results and persistent variables. During initialization, TFLM sets up the memory by overlapping memory allocations that are not simultaneously needed during the same operator evaluation. Therefore, the arena size is the maximal working memory needed by intermediate tensors during operator execution. It is determined by the model width, i.e., the layer with most inputs, weights, and activations. Hence, if the SRAM size is greater than the tensor arena size, the application performance cannot be improved further. Table VII lists the arena size of three MicroNets models, which are close to the SRAM size needed on MCU-D in Table VIII. A balanced model structure design in NAS may overcome the SRAM size limitation while maintaining sufficient accuracy. Some recent works apply operator reordering [31] or memory swapping [32] to reduce working memory or virtually expand working memory, respectively.

Executable Binary. To further minimize the binary size to support model inference on MCUs with constrained flash memory, we investigate the composition of the executable binary and its relationship with the model structure. The generated binary is composed of the TFLite model and operator implementation code. Model structures like model depth and weight count are relative to the TFLite model size. The operator implementation code size is determined by model operator type and count. As shown in Table VIII, after excluding the TFLite model size, the remaining binary size is about 79 KB

for all three models on MCU-D. This is the implementation code size for convolutions, depthwise separable convolutions, and fully connected operations. Compared with MCU-D, there is an extra 40 KB and 55 KB overhead in working memory and executable binary, respectively, on MCU-S. This is due to Arduino Nano 33 BLE Sense platform-specific overhead.

Inference Latency. As the TFLite model is a composition of model graph definition and model weights, its size can, to some extent, reflect the inference latency. For the three models based on the same structure, model inference latency is proportional to the TFLite model size. For instance, the TFLite model size of MicroNet-KWS-L is 5.75 times that of MicroNet-KWS-S. Accordingly, the inference runtime of the large model on MCU-S is 5.86 times that of the small model, while on MCU-D, this number is 5.71. Compared with #MACC, TFLite model size is a better indicator of inference latency. However, for five applications with different structures in Table III, we do not observe an obvious relationship between latency and TFLite model size.

Recommendation. Balanced neural network model structure choice during NAS such that the largest layer can fit into on-chip memory is important for memory-constrained devices.

VII. Performance Prediction

We now present an analytical approach for MCUs to predict the latency of each layer by taking into account the impact of the model, software, and hardware. Prior works such as [5] assume that the model operation count can be regarded as an indicator for model performance. However, our evaluations found that a full-stack analysis is necessary for accurate prediction by capturing the impact of model parameters, software optimizations, and hardware characteristics.

A. Analytical Approach

Our analytical approach estimates inference time by estimating different types of assembly instructions executed during model inference. As shown in Fig. 7, we take in *Model Specification*, *Operator Implementation*, and *Architecture Specification* as inputs. We first generate the Low-Level Virtual Machine (LLVM) bitcode of operator implementation and obtain the corresponding intermediate representation (IR). Then based on IR, our LLVM pass counts assembly instructions for *Data Movement* and *Data Computation*, and loop iterations. Next, based on these data and architectural specifications, we estimate the inference time for *Data Movement*, *Data Computation*, and *Loop Execution* during layer inference. This estimated time constitutes the main latency of layer inference.

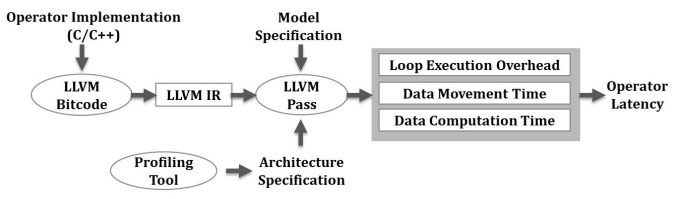

Fig. 7. Analytical approach for performance prediction.

Inputs. (1) *Model Specification.* Layer parameters such as filter height and width describe the layer structures, which offer us a high-level understanding of the compute and memory needs of the layers without any software optimization. (2) *Operator Implementation.* CMSIS-NN kernels provide different optimization implementations for different operators. As mentioned in section V-B, convolutions need IM2COL process to align data for matrix multiplication while fully connected layers do not need this extra input expansion. (3) *Architecture Specification.* With pipeline processing, Cortex-M4 CPUs can execute one ALU instruction or normal SIMD instruction each cycle. The integer-only model implementation based on CMSIS-NN kernels may need memory read or write instructions with different memory access sizes, like *LDR*, *STR*, *LDRSB*, *STRSB*, etc. Moreover, based on different destinations, memory read/write instructions require different execution cycles. For instance, on MCU-S, reading four bytes from flash memory needs five cycles, while writing one byte to SRAM needs one cycle. Therefore, both memory access size and data location of the memory access instructions should be considered. Profiling tools, like J-Trace Pro Debug Probe, can determine the number of execution cycles of these assembly instructions on a given MCU device.

Operator Latency. We divide operator latency into the following three parts. (1) *Loop Execution Overhead.* The number of control instructions required to manage the entire workflow correlates with the layer parameters. The layer with more computation requiring more GEMM conversion will cause more overhead from loop execution. We assume five cycles of overhead per iteration based on experimental evaluation, including the overhead of necessary jump instructions and potential instruction reloading under failed branch speculation. (2) *Computation Time.* There is a high correlation between IR and actual assembly instructions for computation operations. We directly use the IR instructions count for computation time. (3) *Data Movement Time.* We consider two data movement operations, loading or storing model weights, activations and loading or storing other variables defined in the program. The memory read/write instructions to load or store model weights, activations are easily recognized through data size and location, which are clearly shown in IR. In contrast, for the load/store of remaining variables, the LLVM IR cannot represent the process of loading these variables from SRAM to registers; so, we count the IR instruction *PHI* to estimate this overhead of data transmission. One *PHI* instruction corresponds to one variable load instruction from SRAM.

B. Performance Prediction Results

Fig. 8 shows the predicted and actual latency for FC, Conv, Conv1x1, and DWC layers on MCU-S. Clearly, our analytical approach can accurately predict layer inference latency. The absolute difference between predicted and actual latency is insignificant as shown in the Figure; the relative difference is 12.24% across all layers, and the mean absolute error (MAE) is 4.59%. We do not present results for *softmax*, *add*, and

reshape operators as they occupy only a tiny fraction of model inference latency.

Besides, the predicted latency shows a similar trend to actual latency for all layers. Moreover, the predicted error decreases for large FC, Conv, and DWC layers with more computation. For instance, for the FC layer with a predicted error of 5.31%, its actual latency is 63 μs while the predicted one is 66 μs. This slight difference in absolute latency causes a high relative prediction error. For larger layers, the relative prediction error is very low. All these observations reveal the effectiveness and robustness of our proposed performance prediction model.

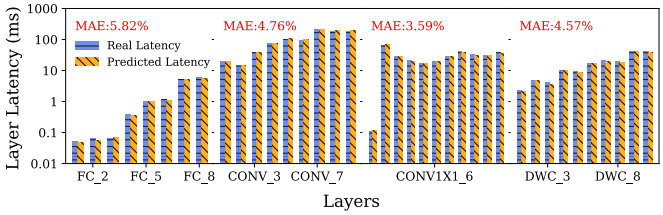

Fig. 8. Predicted and real latency of layers on MCU-S.

Based on predicted layer latency, we calculate model latency for five applications, with prediction error shown in Fig. 9. Among five applications, the maximal difference between predicted and actual latency is 5.12%, and MAE is 3.57%. Fig. 9 also shows prediction error percentage based on model #MACC or operation count (OPs). We obtain this number through linear regression analysis between model latency and #MACC or OPs. It is observed that our prediction considering influence from full stack provides far more accurate results.

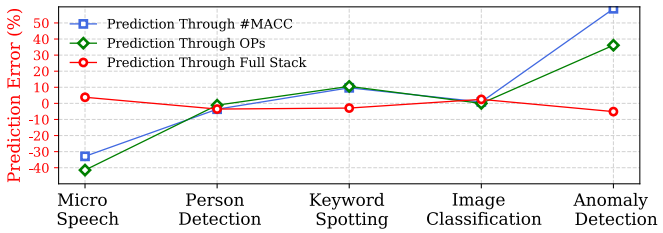

Fig. 9. Model prediction comparison on MCU-S.

VIII. Conclusion

In this paper, we present a systematic full-stack power-performance analysis of TinyML applications on multiple MCU platforms. Our analysis is performed from three different perspectives: the abstraction layers, the optimization library, and the hardware architectural impact. These observations provide insights and suggestions for NAS and system optimizations, as shown in each section. Finally, based on the analysis, we design a full-stack model to quickly estimate the latency of ML models accurately.

IX. Acknowledgment

This work was partially supported by the National Research Foundation, Singapore under its Competitive Research Programme Award NRF-CRP23-2019-0003 and Singapore Ministry of Education Academic Research Fund T1 251RES1905.

References

[1] TinyML. [Online]. Available: https://www.tinyml.org/
[2] C. R. Banbury *et al.*, "Benchmarking tinyml systems: Challenges and direction," *arXiv*, 2020.
[3] Y. E. Wang *et al.*, "Benchmarking TPU, GPU, and CPU platforms for deep learning," *arXiv*, 2019.
[4] Y. Wang *et al.*, "Benchmarking the performance and energy efficiency of AI accelerators for AI training," in *Proc. CCGRID*. IEEE, 2020, pp. 744–751.
[5] C. Banbury *et al.*, "Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers," in *Proc. MLSys*, vol. 3, pp. 517–532, 2021.
[6] L. Heim *et al.*, "Measuring what really matters: Optimizing neural networks for tinyml," *arXiv*, 2021.
[7] A. Yazdanbakhsh *et al.*, "An evaluation of edge TPU accelerators for convolutional neural networks," *arXiv*, 2021.
[8] R. David *et al.*, "Tensorflow Lite Micro: Embedded machine learning for tinyml systems," in *Proc. MLSys*, vol. 3, pp. 800–811, 2021.
[9] L. Lai *et al.*, "CMSIS-NN: Efficient neural network kernels for arm cortex-m CPUs," *arXiv*, 2018.
[10] P. Warden *et al.*, *Tinyml: Machine learning with tensorflow lite on arduino and ultra-low-power microcontrollers*. O'Reilly Media, 2019.
[11] C. Banbury *et al.*, "Mlperf tiny benchmark," *arXiv*, 2021.
[12] TensorFlowLite, *Deploy machine learning models on mobile and iot devices*. [Online]. Available: https://www.tensorflow.org/lite
[13] S. P. Baller *et al.*, "DeepEdgeBench: Benchmarking deep neural networks on edge devices," in *IC2E*. IEEE, 2021, pp. 20–30.
[14] S. Wang *et al.*, "Neural network inference on mobile SoCs," *IEEE Design & Test*, 2020, 37, 50-57.
[15] S. Wang *et al.*, "High-throughput CNN inference on embedded ARM Big. LITTLE multicore processors," in *T-CAD*, 2019, 39, 2254-2267.
[16] Arduino, *Arduino nano 33 ble sense*. [Online]. Available: https://store-usa.arduino.cc/products/arduino-nano-33-ble-sense
[17] Sparkfun, *Sparkfun edge development board - apollo3 blue*. [Online]. Available: https://www.sparkfun.com/products/15170
[18] STMicroelectronics, *Stm32f746zg: High-performance with fpu cortex-m7 mcu*. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f746zg.html
[19] STMicroelectronics, *Stm32l476rg: Ultra-low-power with fpu cortex-m4 mcu*. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32l476rg.html
[20] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proc. CVPR*, 2018, pp. 4510–4520.
[21] Y. Zhang *et al.*, "Hello edge: Keyword spotting on microcontrollers," *arXiv*, 2017.
[22] K. He *et al.*, "Deep residual learning for image recognition," in *Proc. CVPR*, 2016, pp. 770–778.
[23] ODROID, *Odroid smart power 2 with wifi connectivity*. [Online]. Available: https://www.odroid.co.uk/odroid-smart-power-2
[24] P. Warden, "Why gemm is at the heart of deep learning," *Peter Warden's Blog*, 2015.
[25] A. Capotondi *et al.*, "CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices," in *IEEE Trans. Circuits Syst. II Express Briefs IEEE T CIRCUITS-II*, vol. 67, no. 5, pp. 871–875, 2020.
[26] STMicroelectronics, *X-CUBE-AI: AI expansion pack for stm32cubemx*. [Online]. Available: http://www.st.com/en/embedded-software/x-cube-ai.html
[27] A. Garofalo *et al.*, "PULP-NN: accelerating quantized neural networks on parallel ultra-low-power risc-v processors," in *Philos. Trans. Royal Soc. A PHILOS T R SOC A*, vol. 378, no. 2164, p. 20190155, 2020.
[28] S. Williams *et al.*, "Roofline: an insightful visual performance model for multicore architectures," in *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.
[29] J. Zhang *et al.*, "High performance zero-memory overhead direct convolutions," in *ICML*. PMLR, 2018, pp. 5776–5785.
[30] A. Gural *et al.*, "Memory-optimal direct convolutions for maximizing classification accuracy in embedded applications," in *ICML*. PMLR, 2019, pp. 2515–2524.
[31] E. Liberis *et al.*, "Neural networks on microcontrollers: saving memory at inference via operator reordering," *arXiv*, 2019.
[32] H. Miao *et al.*, "Enabling large neural networks on tiny microcontrollers with swapping," *arXiv*, 2021.