

Exploiting Unused Storage Space for Ultra-Low-Power Neural Network Inference

Shiming Li[✉], Sebastian Åkerhielm, Alessio Petrucci, Xiaoyue Chen[✉],
Luca Mottola[✉], Yuan Yao[✉], Stefanos Kaxiras[✉], *Fellow, IEEE*

Abstract—Different from computation-heavy, vision-based tasks that require complex deep neural network (DNN) models to process, typical sensor data processing tasks in sensor networks or on-body devices can be handled with lightweight DNNs, which are usually compact in size, leaving storage space unused. We present a technique to generate C inference code from trained DNN models that utilizes the unused storage space for speedup and energy efficiency. We start with embedding weight values verbatim into the computation kernels' code, reducing loop and value loading overhead, and facilitating error-less, sub-neuron early exit opportunities. To keep the inference code within storage space budget, we then use a gain-cost estimation model to prioritize applying the aforementioned code transformations to the most profitable layers. Our technique allows generating efficient DNN inference code under a flexible extra storage space budget, and enables 25% time saving and 22% energy saving on a Cortex-M0+ microcontroller with 512 kB flash storage.

Index Terms—Deep neural networks, Microcontrollers

I. INTRODUCTION

NEURAL networks unleash the potential for complex data processing on devices at the network edge, such as energy-constrained microcontrollers (MCUs) or even batteryless devices [1]. Due to the tight energy budget of such devices, the efficiency of neural network inference is essential.

Lightweight networks leave storage space unused. Different from complex image or video processing tasks, the typical tasks for ultra-low-power MCUs, such as wildlife activity monitoring [2], sensor-based human activity recognition (HAR) [3] and low-resolution image classification [4], are usually processed with lightweight deep neural network (DNN) models that are free from the constraint of the storage space of modern ultra-low-power MCUs. In Table I, we show the estimated program memory utilization of typical lightweight DNN models. The flash memory is generally under-utilized.

We gain speedup by utilizing unused storage space. We present a technique to generate efficient C code from trained DNN models, by improving storage space utilization.

Received 17 August 2026; accepted 19 August 2026. Date of current version 20 August 2026. This work is supported by the Swedish Foundation for Strategic Research (SSF) grant FUS21-0067. This manuscript was recommended for publication by J. Lee. (Corresponding author: S. Li.)

S. Li, X. Chen, Y. Yao and S. Kaxiras are with Uppsala University (75105 Uppsala, Sweden) (email: {firstname.lastname}@it.uu.se).

S. Åkerhielm and A. Petrucci are independent researchers and have attended Uppsala University as master students (email: sebastian.akerhielm.8786@student.uu.se; alessio.petrucci@protonmail.ch).

L. Mottola is with Politecnico di Milano (20133 Milano, Italy), Research Institutes of Sweden (16440 Kista, Sweden), and Uppsala University (75103 Uppsala, Sweden) (email: luca.mottola@polimi.it).

Digital Object Identifier 10.xxxx/XXX.xxxx.xxxxxxx

TABLE I: Flash utilization of lightweight sensor data or low-resolution image processing DNN models from STM32 AI Model Zoo [5] on low-power STM32 MCU series. *Estimated from binary size of code generated by TinyEngine [6].*

Model Name	Flash Utilization Estimate (%)		
	STM32G0 Cortex-M0+, max 64 MHz, 512 kB Flash	STM32L4 Cortex-M4, max 80 MHz, 1024 kB Flash	STM32U5 Cortex-M33, max 160 MHz, 4096 kB Flash
Mini ResNet	30	15	4
Mini ResNet 2 stacks	91	46	11
Mini ResNet v2	30	15	4
Mini ResNet v2 2 stacks	92	46	11
ST HPR CNN	5	3	<1
GMP 24	5	2	<1
GMP 48	5	2	<1
IGN 24	5	3	<1
IGN 48	6	3	<1
ST MNIST CNN	8	4	1
ResNet 8	21	11	3
FdMobilenet	44	22	5
ST FdMobilenet	48	24	6
Mobilenet gray	54	27	7
Mobilenet	58	29	7

The first component of our technique is *storage-tradeoff code generation*. Instead of generating convolution and fully-connected layer code using loop-based, one-fits-all dot product functions, we generate dot product kernels where weights are written into the code line-by-line, reducing branch and flash access overhead. As this automatically allows arbitrary computation order in each dot product operation, this opens up opportunities to integrate *sub-neuron early exits* [7], [8]. Such code transformations exploit the property of the ReLU function, and by reordering the computations, allow early exiting when a neuron's value is deemed negative.

The second component of our technique is to efficiently use the *space budget* by selectively apply the aforementioned optimizations. We use a gain-cost estimation model to evaluate each layer's potential for speedup gain and the space overhead, and reduce the optimization to a 0/1 knapsack problem.

We visualize the workflow of our technique in Figure 1. Starting from a vanilla model, we evaluate the extra space cost and estimate the gains layer by layer. With a given extra storage space budget, we solve the optimization problem to find the DNN layers to apply storage-tradeoff code generation. The produced DNN inference code can stay within the given space budget after compilation, and produces the exact same result as the vanilla model, without value error.

Results. We integrate space-tradeoff code generation into a modified version of TinyEngine [6], [9], and implement model analysis and space budget planning in Python. We evaluate 15 trained DNN models from STM32 AI Model Zoo [5]. We

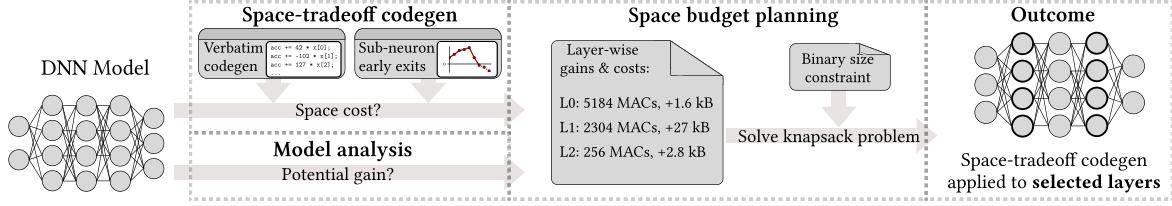


Fig. 1: The workflow to generate DNN inference code with a given space budget with our technique.

first evaluate with an arm-v6m simulator [10]. Our technique versatily exhibits efficiency under different space budget from 32 kB to 512 kB, showing up to 40% reduction in total executed instructions. We then evaluate with a STM32 NUCLEO-G0B1RE board equipped with a Cortex-M0+ MCU with 512 kB flash storage, and observe 25% time saving and 22% energy saving at 64 MHz clock speed.

II. BACKGROUND AND MOTIVATION

Lightweight DNN tasks on ultra-low-power MCUs. While there exist complex vision-based tasks that require huge DNN models, sensor data processing and low-resolution image processing tasks can be handled by lightweight DNN models. For example, a five-layer network is enough for decent accuracy for wild animal behavior prediction [2]. The HAR network [11] recognizes human activity from accelerometer data with a DNN of one convolutional, one pooling and two fully-connected layers, requiring a model of just 76 kB [5], which is further reduced to 7 kB after quantization. A three to four-layer DNN [4] can estimate water depth based on low resolution image from drones. An investigation in the flash utilization of lightweight DNNs on common ultra-low-power MCUs is presented in Table I. Such lightweight inference tasks often under-utilize the MCU’s storage space.

Sub-neuron early exits with code transformation. There exist techniques that exploits the ReLU activation function, common in MCU-level DNNs [5] due to its simplicity, for early-exiting during the computation of one neuron. Originally a DNN accelerator design, SnaPEA [7] forces the intermediate result to change monotonically when computing the value of an output neuron guarded by a ReLU function, so that when the intermediate result becomes negative and continues to decline, the final result is guarantee to be fixed to zero by ReLU. This is applicable to generating DNN inference code, provided that the input of a layer is all positive, and reordering the computations is possible. Without requirements on the input values, another technique, saturation-aware convolution [8], examines the possible result ranges of future computations, and determine whether the current intermediate result is extreme enough to early exit. We find these two methods can fit in our technique and provide performance benefits *without introducing value error*.

III. DESIGN

We first introduce our code generation process for dot product operations in convolution and fully connected layers, and then present a solution to selectively apply it to the most profitable layers under an extra storage space budget.

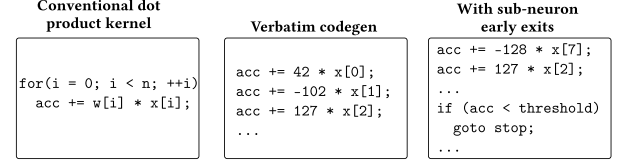


Fig. 2: Example of verbatim code generation and sub-neuron early exits on a dot product operation, compared with the conventional way. *The sub-neuron early exit example shows the general case regardless of the exact method: computations are reordered, and a threshold value controls the early exit.*



(a) Early exiting by reordering (b) Early exiting by saturation-awareness [8]. The error bars shows the mathematically possible range of the final result, providing an error-less threshold.

Fig. 3: Illustration of two kinds of sub-neuron early exits when computing a neuron value gated by a ReLU function, which fixes negative values to zero.

A. Space-Tradeoff Code Generation

Reducing overhead by generating verbatim code. We generate model-specific dot product kernels for convolutional and fully-connected layers where weights are written into the code line-by-line. The benefit is two-pronged. (i) *Reduced executed instructions.* We observe 12% to 15% of the executed instructions in DNN inference tasks is branch instruction, and can be removed. (ii) *Reduced pipeline bubbles.* Removing the branch instructions also reduces the branch taken penalties in the pipeline. Besides, when the MCU’s clock cycle is shorter than the flash memory’s reaction time, flash accesses need extra wait cycles. As verbatim code generation reduces weight array access, this also reduces pipeline bubbles by flash access.

Reducing computations with sub-neuron early exits. When generating the value in a neuron, verbatim code generation intrinsically allows arbitrary reordering of the computations, thus allowing optimizations reliant on computation order.

We create early exits based on the monotonic-value-change method in SnaPEA [7] for layers where the inputs are all positive. We generate the kernel code with an altered computation order prioritizing positive weights when computing one neuron’s value, so the intermediate result monotonically decreases towards the end of the computation, allowing early exits when the intermediate result is negative. For DNN layers whose inputs are not guaranteed to be all positive,

TABLE II: DNN models evaluated.

Model name	fflite Size (kB)	Input size (Bytes)	Note
Mini ResNet	140	64x50x1	Audio event detection (AED)
Mini ResNet 2 stacks	467	64x50x1	
Mini ResNet v2	142	64x50x1	
Mini ResNet v2 2 stacks	470	64x50x1	
ST HPR CNN	6.5	8x8x2	Hand posture recognition
GMP 24	5.6	24x3x1	
GMP 48	5.6	48x3x1	
IGN 24	6.7	24x3x1	
IGN 48	6.5	48x3x1	Human activity recognition (HAR)
ST MNIST CNN	19	28x28x1	
ResNet 8	94	32x32x3	
FdMobilenet	192	128x128x3	
ST FdMobilenet	217	128x128x3	Low-resolution image classification
Mobilenet gray	301	96x96x1	
Mobilenet	301	96x96x3	

we fall back to saturation-aware convolution [8]. Instead of solely using the lower boundary of ReLU function as the trigger to early exit, this considers the unfinished computations' theoretical maximum increase, given by $inc_max_i = \sum_{j=i+1}^{m-1} \begin{cases} 127 \cdot w_j, & \text{if } w_j \geq 0 \\ -128 \cdot w_j, & \text{if } w_j < 0 \end{cases}$, where w_j denotes the weights, i is the current computation step, and m is the total number of computations. Early exits are triggered if the final result definitely saturates the boundaries allowed by the output neuron. The computations are reordered by the absolute value of the weights involved, since this allows early exits to happen earlier [8]. In our implementation, we insert early exits in each convolution kernel and fully-connected layer channel at static positions decided at compile time, with 50%, 25% and 12.5% of all meaningful computations left. We exemplify the two sub-neuron early exit mechanisms [7], [8] in Figure 3, and the whole *space-tradeoff code generation* process in Figure 2.

B. Space Budget Planning

While space-tradeoff code generation enables speedup, we find applying it to the whole DNN incurs 0.8 to 9.1 times space overhead. Therefore, we selectively apply it to DNN layers where the potential gain is maximized under a storage space budget. We reduce this to a 0/1 knapsack problem. The storage space budget is the knapsack's size, and each DNN layer is one item, and we define their *values* and *costs* as below. As we notice a positive correlation between a layer's computation time and its speedup gain from *space-tradeoff codegen*, for simplicity, we use each layer's total amount of multiply-accumulate operations (MACs) as its *value*. For fully-connected layers, this is the multiply of the number of input and output neurons. For convolutional layers, this is $C_k \cdot H_k \cdot W_k \cdot C_o \cdot H_o \cdot W_o$, where C , H and W are the number of channels, height and width, respectively, and the subscripts k and o denote kernel and output. As for the *cost*, we define it as the space overhead in each layer, evaluated by simply measuring the binary size inflation after a test compilation. The knapsack problem can be solved by brute-force searches given the DNNs' layers are usually no more than several tens.

IV. EVALUATION

We implement space-tradeoff code generation on a modified version of the state-of-the-art code generation framework for DNN inference on MCUs, TinyEngine [6] [9]. We implement

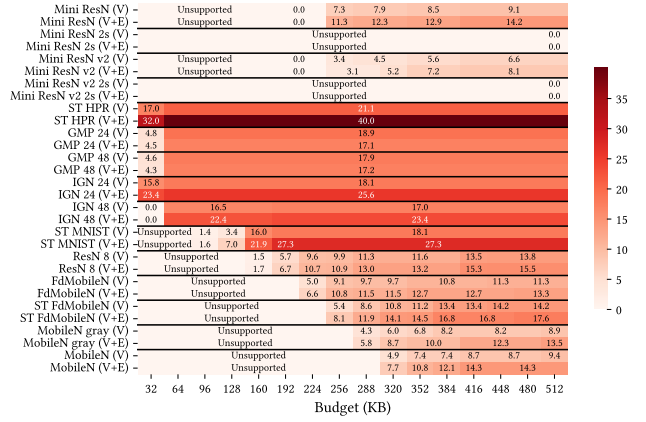


Fig. 4: Reduction of executed instructions (darker is better) with different binary size budgets. Tested with only *verbatim codegen enabled* (V), or together with *sub-neuron early exits* (V+E) enabled. “Unsupported” means the binary size budget is too small even for the baseline version.

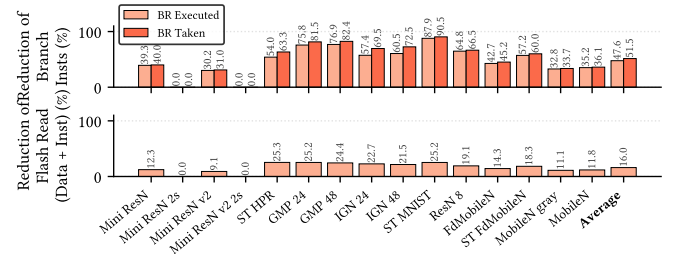


Fig. 5: Reduction of branches and flash reads at 512 kB budget.

the space cost evaluation, model analysis and space budget planning functionalities with Python. Generating C code with our technique takes from tens of minutes to several hours. We compile the C code with optimization level `-Ofast` since this option yields the best execution speed even for the baselines.

Evaluation methodology. We first evaluate our technique with an arm-v6m simulator [10], focusing on executed instructions and flash memory access. We span the evaluation to different binary size budgets to exhibit our technique's versatility, and turn on and off the sub-neuron early exits to verify its contribution to space-tradeoff code generation.

We also evaluate on a STM32 G0B1RE development board, which has a Cortex-M0+ SoC with 144 kB of RAM and 512 kB of flash memory. We use different clock frequencies: 16 MHz, 32 MHz and 64 MHz, corresponding to flash wait state 0, 1, and 2, respectively [12], and enable the ICACHE and instruction prefetch buffer in the flash controller to honestly reflect the effect of the flash access changes.

We use trained DNNs from STM32 AI Model Zoo [5] as benchmarks, detailed in Table II. We select models that can be fit onto the STM32 G0B1RE MCU. These models cover different typical task types handled with lightweight DNNs.

A. Simulator-Based Evaluation

Executed instructions reduced by up to 40%. We show an overview of executed instruction reduction in Figure 4. Our technique versatily exhibits performance gain under different storage space budgets from 32 kB to 512 kB. The

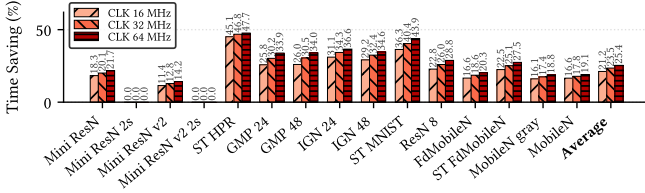


Fig. 6: Time saving on a Cortex-M0+ MCU (512 kB flash).

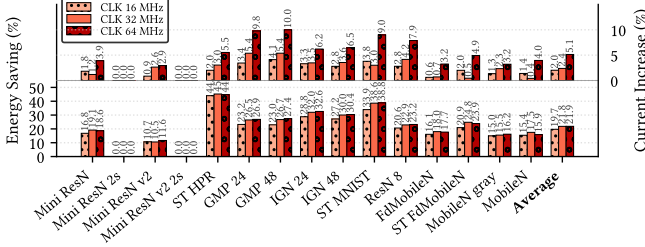


Fig. 7: Energy and current draw on a Cortex-M0+ MCU. only exceptions are the 2-stack versions of Mini Resnet v1 and v2, where the space budget planner cannot find a solution to apply our technique without exceeding the space limit. However, these two models are nearly 470 kB in the first place, leaving few free storage space for optimization.

Both verbatim codegen and early exits contribute. Models enjoy up to 21% reduction in executed instructions with solely verbatim codegen, marked as “V” in Figure 4, and this increases to 40% when sub-neuron early exits are enabled (marked as “V+E”). The only exceptions are two GMP models, displaying a slight performance degradation with early exits. We find these HAR models do not have ReLU activation functions at all, and the early exit thresholds are set to the data type’s lower boundary, and no early exits are triggered.

Branch instructions and flash reads reduced. We show results of reduction of branch instructions and flash access in Figure 5. The number of taken branches are halved, and the flash access is reduced by 16% on average.

B. Real Board Evaluation

Execution time reduced by 21% to 25%. We report the time saving on a Cortex-M0+ MCU with 512 kB flash in Figure 6. At the highest clock frequency, the average reduction of execution time is 25%. We see increasing time saving with 16 MHz, 32 MHz and 64 MHz clock frequency, because higher clock frequency induces more flash access latency (1 cycle at 32 MHz and 2 cycles at 64 MHz [12]), and our technique reduces overall flash access (Figure 5).

Working current increased, but overall energy cost is lower. We report the changes of working current and estimated inference energy cost in Figure 7. We see an increase in the working current draw by up to 10%, and we believe this is due to the reduction in pipeline bubbles pushing the dynamic power higher. The overall energy cost, despite the increase in working power, still decreases by 20% to 22% on average.

V. DISCUSSIONS

Generalizability. Our technique can be applied to more complex DNN models as long as they contain dot product

operations. Our technique is most effective on ultra-low-power MCUs whose ISA lacks SIMD capability. For SIMD-capable architectures, computation reordering may be more costly because this requires loading discontinuous input data into vectors. However, saturation-aware computation can still be applied without computation reordering.

Design of the gain-cost estimation model. For simplicity, the gain estimation only considers the number of MACs. Such estimation is based on the insight that the more time a layer takes, the more time it spends on overheads. The model does not reflect the precise measure of branch and flash access overhead, and it is possible that a better estimation method leads to more benefits. However, the purpose of this model is to ensure positive gain without exceeding the space budget, and a precise model is unnecessary.

VI. CONCLUSION

We exploit unused storage space for speedup and energy saving for lightweight DNNs on ultra-low-power MCUs. We generate code with space tradeoffs, with weight values embedded in the code and sub-neuron early exits inserted, and selectively apply the code transformation to the most profitable layers. This method versatily yields benefit with different storage space budgets, and gains 25% time saving and 22% energy saving on a Cortex-M0+ MCU with 512 kB flash.

VII. ACKNOWLEDGMENTS

AI tools were used to assist generating Python code for plotting Figure 4, 5, 6 and 7.

REFERENCES

- [1] H. R. Mendis, K. S. Yıldırım, M. Zimmerling, L. Mottola, and P.-C. Hsiu, “Special session-intermittent tinyml: Powering sustainable deep intelligence without batteries,” in *ACM SIGBED Intl. Conf. on Embedded Software (EMSOFT ’25)*, 2025.
- [2] R. Otsuka, N. Yoshimura, K. Tanigaki, S. Koyama, Y. Mizutani, K. Yoda, and T. Maekawa, “Exploring deep learning techniques for wild animal behaviour classification using animal-borne accelerometers,” *Methods in Ecology and Evolution*, vol. 15, no. 4, pp. 716–731, 2024.
- [3] K. Chen, D. Zhang, L. Yao, B. Guo, Z. Yu, and Y. Liu, “Deep Learning for Sensor-based Human Activity Recognition: Overview, Challenges, and Opportunities,” *ACM Comput. Surv.*, vol. 54, May 2021.
- [4] Q. Shen, K. Mahima, K. De Zoysa, L. Mottola, T. Voigt, and M. Flierl, “CNN-Based Estimation of Water Depth from Multispectral Drone Imagery for Mosquito Control,” in *2023 IEEE Intl. Conf. on Image Processing (ICIP ’23)*, 2023.
- [5] STMicroelectronics, “STM32 Model Zoo.” <https://github.com/STMicroelectronics/stm32ai-modelzoo>, 2023.
- [6] J. Lin, W.-M. Chen, Y. Lin, J. Cohn, C. Gan, and S. Han, “MCUNet: Tiny Deep Learning on IoT Devices,” in *Advances in Neural Information Processing Systems (NeurIPS ’20)*, pp. 11711–11722, 2020.
- [7] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, “SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks,” in *Proc. of the 45th Annual Intl. Symposium on Computer Architecture, ISCA ’18*, 2018.
- [8] S. Li, L. Mottola, Y. Yao, and S. Kaxiras, “Efficient CNN Inference on Ultra-Low-Power MCUs via Saturation-Aware Convolution,” in *Conference on Design, Automation & Test in Europe (DATE ’26)*, 2026.
- [9] S. Li, “TinyEngine-armv6m.” <https://github.com/shm-li/tinyengine-armv6m>, 2025.
- [10] M. Hicks, “Thumbulator: cycle-accurate arm-v6m simulator.” <https://github.com/impedimentToProgress/thumbulator>, 2018.
- [11] A. Ignatov, “Real-time human activity recognition from accelerometer data using Convolutional Neural Networks,” *Applied Soft Computing*, vol. 62, 2018.
- [12] STMicroelectronics, “STM32 G0 Flash.” https://www.st.com/resource/en/product_training/STM32G0-Memory-Flash-FLASH.pdf, 2020.