iMCU: A 102-µJ, 61-ms Digital In-Memory Computing-based Microcontroller Unit for Edge TinyML

Chuan-Tung Lin, Paul Xuanyuanliang Huang, Jonghyun Oh, Dewei Wang, Mingoo Seok

Columbia University

TinyML envisions performing a deep neural network (DNN)-based inference on an edge device, which makes it paramount to create a *neural* microcontroller unit (MCU). Toward this vision, some of the recent MCUs integrated in-memory computing (IMC) based accelerators [1-3]. However, they employ analog-mixed-signal (AMS) versions, exhibiting limited robustness over process, voltage, and temperature (PVT) variations [1-2]. They also employ a large amount of IMC hardware, which increases silicon area and cost. Also, they do not support a practical software dev framework [1-3] such as TensorFlow Lite for Microcontrollers (TFLite-micro) [5]. Because of this, those MCUs did not present the performance for the standard benchmark MLPerf-Tiny [6], which makes it difficult to evaluate them against the state-of-the-art neural MCUs. In this paper, we present *iMCU*, the IMC-based MCU in 28nm, which outperforms the current best neural MCU (SiLab's xG24-DK2601B [6]) by 88X in energy-delay product (EDP) while performing MLPerf-Tiny. Also, iMCU integrates a digital version of IMC hardware for maximal robustness. We also optimize the acceleration targets and the computation flow to employ the least amount of IMC hardware yet still enable significant acceleration. Hence, iMCU's total area is only 2.03 mm^2 while integrating 433KB SRAM and 32KB IMC SRAM.

Fig. 1 left illustrates the overall organization of iMCU, which consists of i) a 32b RISC-V CPU core (the host processor), ii) a digital IMC accelerator that contains the IMC cluster, iii) instruction memory (IMEM), iv) the main data memory (DMEM), v) direct memory access (DMA), vi) a universal asynchronous receiver-transmitter (UART), vii) a general-purpose IO (GPIO), and viii) a 32b ARM AHB/APB bus.

Fig. 1 right shows the matching software dev framework for iMCU, which is based on TFLite-micro. It starts with the training of an 8-b DNN model via TensorFlow, which produces a TF file. Then, we convert the TF file into the TFLite file by fusing a batch norm layer into a convolution layer. This helps to avoid adding explicit hardware support for batch-norm-related computation. The next step is to convert the TFLite file to the C header file model.cc. Then, finally, we compile the header file with the input data file (input.cc) and the TFLite-micro library file. The compilation produces the instruction and data hexadecimal files, which we store in IMEM and DMEM, respectively. Using the framework, we develop the software for the following DNN models, tiny-conv, tiny-embedding-conv (from [5]), and ResNetv1 (from [6]) (Fig. 2 top left).

We start our iMCU design by determining which layers are worth accelerating. The accelerator supports only those layers to reduce the area overhead. We profile the complexity of each layer using SPIKE [7]. The convolution layer is the most dominant, followed by the addition layer (Fig. 2 top right). If we accelerate convolution layers by 500X, we estimate the total cycle count will be reduced by 119X. If we also accelerate the addition layers, we can gain an additional 3.6X speed-up (total 434X) (Fig. 2 bottom left). All the other layers (pooling, fully connected, softmax) are not worth accelerating since it provides only a negligible cycle count reduction.

We then devise the computation flow (sequence) that requires the least amount of IMC hardware yet still provides a significant acceleration. Existing works employ arbitrarily large amounts of IMC hardware to store more than one (sometimes all) layers of weight data of a DNN model before starting computation [1-3]. Such architecture, however, severely increases area overhead. We devise an alternative computation flow where the main data memory (DMEM), implemented in dense foundry 6T bitcells, stores all the weights, and the IMC hardware buffers the weight data of only one layer right before the accelerator computes on the layer. While the proposed flow increases data movement cost between the main memory and the IMC accelerator, we found that the area savings largely outweigh: the IMC hardware's area reduces by 5X while the cycle count increases by only 23% (Fig. 2 bottom right). This is because we perform 100-10,000 VMMs for each layer and thereby amortizing the data movement cost over many VMMs.

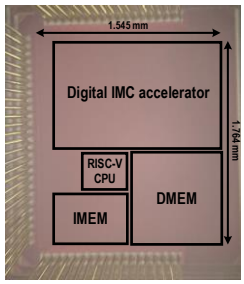

Fig. 7. Die micrograph.

Fig. 3 left shows the proposed flow for an end-to-end inference. The host starts the program, and as it reaches a convolution (or an addition) layer, it configures DMA to transfer the weight data of that layer from DMEM to the IMC cluster; and only for the input layer, DMA transfers the input data from DMEM to the scratchpad. Then, the host configures layer-related parameters such as dimensions of input, filter, and output, stride and padding sizes, input and output offsets, and the starting addresses of the input, weight, output data accesses, etc. Also, the host configures which digital IMC macros to use and clock-gate unused macros. Then, the accelerator starts to compute on the layer, which involves many iterations of three sub-tasks, namely input vector preparation, IMC operation, and output quantization. Finally, it stores the output of the layer in the scratchpad and then interrupts the host. The host resumes the program.

Based on the computation flow, we determine the sizes of the memory blocks and create the memory map (Fig. 4 bottom right). Again, the IMC accelerator requires to buffer only one layer at a time. Therefore, we can set the IMC size to 32KB, roughly matched to the largest layer of the target models. Also, the largest model has a total of 179KB of weight data. Thus, we set the main data memory (DMEM) size to 256KB. We also want the scratch pad in the accelerator to fully buffer the output of one layer such that it can use it as the input of the next layer. Therefore, we set the scratchpad size to 48KB, matched to the largest output data size. Also, we set the IMEM to 128KB to store the largest program. (Fig. 3 right).

We design the *fully-pipelined* IMC accelerator (Fig. 4 top). It has three stages, and each stage takes the same 64 cycles. The first stage (INVEC) prepares input vectors and feeds them to the next stage (IMC). It employs two 512B buffers operating in a ping-pong fashion to hide the latency. The second stage (IMC) performs VMM using the 4×4 IMC macro cluster. The cluster can complete one multiplication between an 8b 512d vector and an 8b 64×512d matrix in 64 cycles. The last stage (QUAN) performs the quantization. The IMC stage's result can have up to 25 bits, but we need to quantize to 8b before storing them in the scratchpad. Simply removing the LSBs is not optimal for inference accuracy. Instead, we support the quantization scheme of TFLite-micro [5], where the quantized value q is defined as $Q=2^n \cdot M_0 \cdot (r+Z)$, where n, M_0, Z are offline-computed hyper-parameters and r is the IMC stage's result. QUAN needs to quantize only one 64d vector in 64 cycles. Therefore, it employs only one 2-input 32b adder, one 2-input 64b multiplier, and one 32b shifter. Finally, the accelerator can still support a layer that is larger than the IMC cluster. It can produce partial sums with partial weight data and combine them to produce the final result.

We designed a *digital* IMC macro to maximize the robustness while trying to reduce the area overhead of digital circuits (Fig. 4 bottom left). To do so, we adopt and improve a time-sharing architecture [4], where the macro employs 128×128 compact 6T bitcells. Every eight bitcells time-share one multiplier, and every 128×8 bitcells time-share a set of compressors and an adder tree, which results in an excellent weight density of ~126 KB/mm^2. The macro achieves the excellent compute density of 1.25 TOPS/mm^2 at 1V and an energy efficiency of 40.16 TOPS/W at 0.6V with a 25% input toggle rate.

We prototyped iMCU in a 28nm. To evaluate against the state-of-the-art neural MCUs, we have iMCU to execute the standard benchmark, ResNetv1, from MLPerf-Tiny [6]. It takes 60.9 ms and consumes 102.18 µJ per inference (Fig. 5 top left). This marks 88X EDP improvement (22X in E and 3.94X in D) over the best neural MCU (SiLab's xG24-DK2601B [6]). Fig. 5 bottom shows the area, energy, and delay breakdown. iMCU is fully-digital hardware and always produces the correct computation results across PVT variations. The proposed optimizations reduce the silicon area of iMCU down to 2.73 mm^2. The on-chip 432KB foundry SRAM takes 0.678 mm^2 and the 32KB IMC SRAM takes 0.254 mm^2 (Fig. 7).

Acknowledgement: This work is supported by NSF (PFI-RP1919147). The authors gratefully thank Manho Kim and Prof. Hyuk-jae Lee for their valuable support and discussions.

2023 IEEE Custom Integrated Circuits Conference (CICC) | 979-8-3503-9948-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/CICC57935.2023.10121221

References: [1] Jia et al., JSSC, 2020. [2] Chang et al., ISSCC, 2022. [3] Wang et al., ISSCC, 2019. [4] B. Yan et al., ISSCC, 2022. [5] R. David et al., arXiv:2010.08678. [6] C. Banbury et al., arXiv:2106.07597. [7] Spike RISC-V ISA Simulator (link)

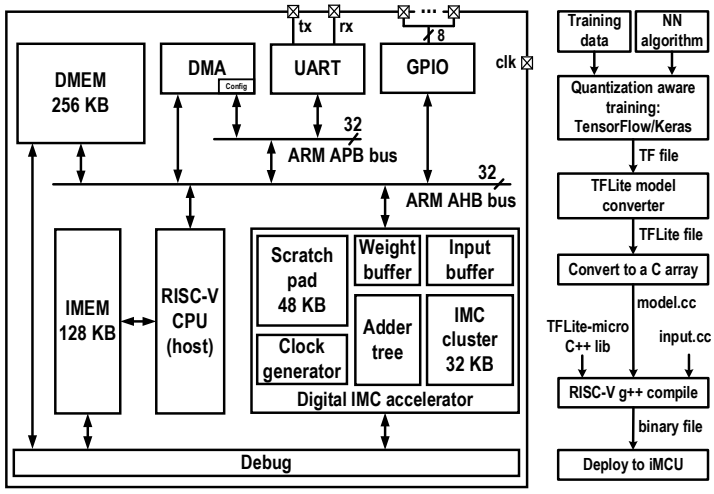

Fig. 1. Proposed iMCU chip architecture. The proposed software development framework for iMCU.

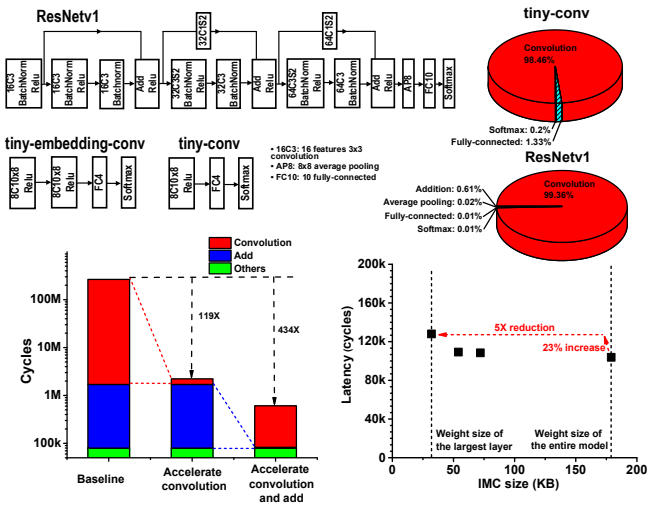

Fig. 2. Neural network models: ResNetv1, tiny-embedding-conv, and tiny-conv. Workload profiling results using SPIKE. The latency improvement estimation. The proposed computation flow enables 5X area reduction at the 23% latency penalty.

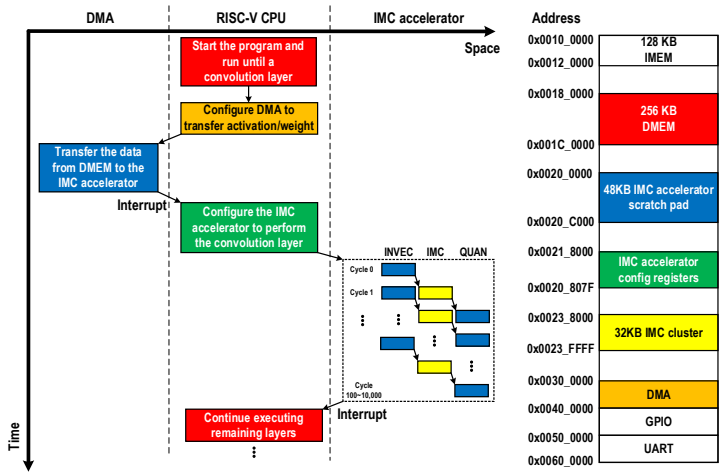

Fig. 3. The proposed computation flow and the memory map of iMCU.

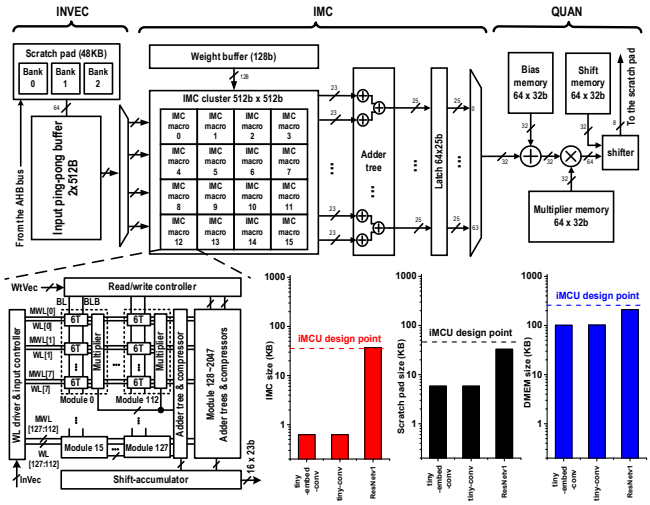

Fig. 4. IMC accelerator microarchitecture. A 128x128 digital IMC macro. Based on the proposed computation flow, we set the sizes of the IMC macro cluster, the in-accelerator scratch pad, and the DMEM to support the target workloads.

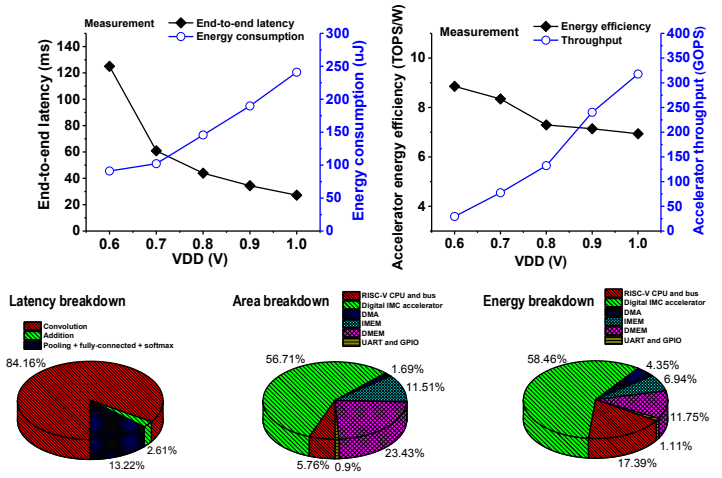

Fig. 5. Measurement results. Latency and energy consumption of iMCU over supply voltages. Energy efficiency and throughput of the IMC accelerator over supply voltages. Latency, area and energy consumption breakdowns.

		Can perform the standard CNN benchmark				Cannot perform	
		This work	xG24-DK2601B Silicon Labs [6]	NUCLEO-H7A3ZI-Q STMicroelectronics [6]	RX65N-Cloud-Kit Renesas [6]	JSSC20 [1]	ISSCC19 [3]
	Technology [nm]	28	n/a	n/a	n/a	65	28
Architectural features	Host processor	RISC-V 32b	Cortex-M33 32b	Cortex-M7 32b	Renesas RXv2 32b	RISC-V 32b	Cortex-M0 32b
	Accelerator	Digital IMC	Digital accelerator	n/a	n/a	Analog IMC	Digital IMC
	Activation precision [bit]	8	8	32	32	1-8	1-32
	Weight precision [bit]	8	8	32	32	1-8	1-32
	IMEM size	128KB	1.5MB[5]	2.06MB[5]	2MB[5]	128KB	16KB
	DMEM size	256KB	256KB	1.4MB	640KB	128KB	16KB
	IMC size	32KB	n/a	n/a	n/a	73.75KB	96KB
	In-accelerator scratch pad size	48KB	n/a	n/a	n/a	0KB	0KB
	Total SRAM size	464KB	256KB	1.4MB	640KB	329.75KB	128KB
	Total SRAM area [mm²]	0.933	n/a	n/a	n/a	4.626	1.225
	Total area [mm²]	2.03	n/a	n/a	n/a	8.56	1.85
	IMC density [KB/mm²] (IMC size/IMC area)	125.8	n/a	n/a	n/a	25.2	104.5
	SRAM density [KB/mm²] (Total SRAM size/total SRAM area)	497.42	n/a	n/a	n/a	71.28	104.49
Hardware performance	Supply voltage [V]	0.6-1	n/a	0.74-1.3	n/a	0.85-1.2	0.6-1.1
	Operating frequency [MHz]	6-35 (host) 29-310 (accelerator)	40-78	280	120	40-100	114-475
	Macro compute density [TOPS/mm²]	1.25 (1V,8b,8b)	n/a	n/a	n/a	0.0094 (1.2V,8b,8b)[2]	0.0287 (1.1V,8b,8b)
	Macro energy efficiency [TOPS/W]	40.16 (0.6V,8b,8b)[1]	n/a	n/a	n/a	6.25 (0.85V,8b,8b)[2]	0.56-5.27 (0.6V,8b,8b)
	Accelerator compute density [TOPS/mm²]	0.301 (1V,8b,8b)	n/a	n/a	n/a	0.0094 (1.2V,8b,8b)[2]	0.0287 (1.1V,8b,8b)
	Accelerator energy efficiency [TOPS/W]	8.86 (0.6V,8b,8b)	n/a	n/a	n/a	6.25 (0.85V,8b,8b)[2]	0.56-5.27 (0.6V,8b,8b)
	Accelerator throughput [TOPS]	0.318 (1V,8b,8b)	n/a	n/a	n/a	0.0341 (1.2V,8b,8b)[2]	0.0327 (1.1V,8b,8b)
MLPerf-tiny: ResNetv1 on CIFAR10[4]	Latency [ms]	60.9	239.98	158.13	289.35	n/a	n/a
	Energy consumption [uJ]	102.18	2248.02	4151.13	14350.89	n/a	n/a

1) Simulated. 2) Normalized to 8b weights and 8b activations. 3) n/a: not available. 4) The top-1 accuracy of all systems are above 85%, meeting the quality target in the benchmark suite. 5) Mostly implemented in embedded flash memory.

Fig. 6. Comparison table.