FP-IMC: A 28nm All-Digital Configurable Floating-Point In-Memory Computing Macro

Jyotishman Saikia, Amitesh Sridharan, Injune Yeo, Shreyas Venkataramanaiah, Deliang Fan, Jae-sun Seo
School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA
Email: jsaikia@asu.edu, jaesun.seo@asu.edu

***Abstract*—In-memory computing (IMC) provides energy-efficient solutions to deep neural networks (DNN). Most IMC designs for DNNs employ fixed-point precisions. However, floating-point precision is still required for DNN training and complex inference models to maintain high accuracy. There have not been float-point precision based IMC works in the literature where the float-point computation is immersed into the weight memory storage. In this work, we propose a novel floating-point precision IMC macro with a configurable architecture that supports both normal 8-bit floating point (FP8) and 8-bit block floating point (BF8) with a shared exponent. The proposed FP-IMC macro implemented in 28nm CMOS demonstrates 12.1 TOPS/W for FP8 precision and 66.6 TOPS/W for BF8 precision, improving energy-efficiency beyond the state-of-the-art FP IMC macros.**

***Index Terms*—Digital in-memory computing, floating-point acceleration.**

I. Introduction

State-of-the-art DNN algorithms achieve high accuracy for many practical tasks such computer vision, natural language processing, and autonomous driving, but require a large amount of computation and memory. DNN training workloads necessitate floating-point (FP) precision, while high training accuracy has been demonstrated down to 8-bit FP (FP8) precision [1]. For DNN inference workloads, fixed-point precision has been widely used, but [2] pointed out that complex inference tasks can also need floating-point precision to avoid accuracy loss. To resolve the computation and memory access bottleneck in conventional hardware accelerators, in-memory computing (IMC) has emerged as a promising technique to perform multiply-and-accumulate (MAC) computations inside the memory macro [3]–[6]. Many IMC macros presented in the literature mostly focused on fixed-point precision operations [3], [4]. FP engines were reported in [5] using near-memory computing units, without tighter integration of computation and memory. [6] reported a floating-point IMC macro design, but additional memory was dedicated to the mantissa product and exponent sum with 64% overhead. Recently, [7], [8] reported floating-point IMC macro for 16-bit brain FP precision, but suffers accuracy loss due to hybrid analog circuits [7] and approximate computing/quantization [8].

In this work, we propose a novel floating-point precision IMC (FP-IMC) macro where the float-pointing computation is immersed into the weight memory storage. The proposed FP-IMC supports two FP8 configurations of (a) normal 8-bit floating-point (FP8) precision with 1-bit sign, 5-bit exponent, and 2-bit mantissa (1-5-2) [1], and (b) 8-bit block floating-point (BF8) precision with a shared exponent [9] among a 64-element weight tensor (Fig. 1). Our proposed FP-IMC macro was implemented in a prototype chip in 28nm CMOS, and achieves a normalized throughput (per kb of macro) of 166.9/606.0 GOPS/kb and energy-efficiency of 12.1/66.6 TFLOPS/W for FP8 and BF8 precision modes, respectively, largely improving those of the state-of-the-art floating-point IMC works.

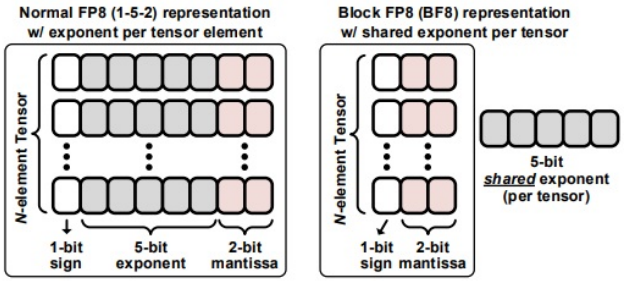


Fig. 1. Illustration of FP8 and BF8 schemes.

II. FP-IMC Architecture and Operation

Fig. 2 shows the overall architecture of the proposed FP-IMC macro. We implement a 64×64 (4kB) macro, which is divided into eight column groups of 64×8 sub-macro arrays. Each sub-macro receives the 512-bit input in parallel to output the 8-bit FP8 MAC output simultaneously. Each of the 64 rows in the sub-macro is dedicated to 8-bit of input. In one column group, each row consists of 8 SRAM bitcells (8-bit FP weight) that will undergo a FP MAC operation against an 8-bit FP input. To perform the FP multiply, the 8-bit SRAM bitcells along with the supporting exponent/mantissa handling modules and a normalization module, work in conjunction. This multiplied output is then handled in an adder tree to perform the accumulate portion of the MAC operation. The architecture supports FP8 as well as BF8, the ouptut of the rows are connected to either the FP8 or BF8 adder tree based on the active mode.

A. FP Multiply Operation:

FP8 (1-5-2) multiplication of input (IN) and weight (W) is:

$$IN \times W = (-1)^{IN[7]} \times e^{IN[6:2]} \times 1.(IN[1:0]) \times (-1)^{W[7]} \times e^{W[6:2]} \times 1.(W[1:0]), \quad (1)$$

$$= (-1)^{(IN[7] \oplus W[7])} \times e^{(IN[6:2]+W[6:2])} \times 1.(IN[1:0]) \times 1.(W[1:0]), \quad (2)$$

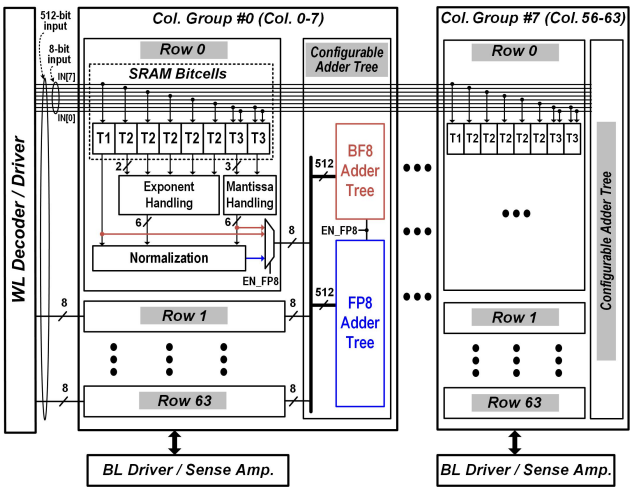


Fig. 2. Proposed FP-IMC macro and architecture design.

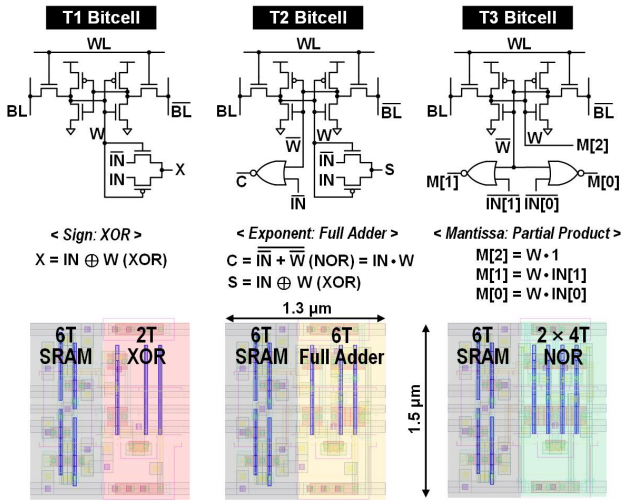


Fig. 3. T1/T2/T3 bitcell designs, compute functionality, and layout.

which can be simplified into three sets of sub-operations: (a) XOR for the sign bits (bit [7]), (b) addition of the exponent bits (bits [6:2]), and (c) multiplication of the mantissa bits (bits [1:0]).

To efficiently implement these sub-operations of FP8 multiplication, we designed the FP-IMC SRAM with three types of bitcells (T1/T2/T3 bitcells), as shown in Fig. 3. **T1** bitcell performs bit-wise XOR between the input sign bit and the weight sign bit. **T2** bitcell integrates a half adder to add the input exponent bit and weight exponent bit, and outputs sum and carry bits. Here we implement XOR function with pass-gate based 2T for the sum generation and perform the AND function for the carry with 4T NOR gate. For each mantissa bit of the weight (W_M), **T3** bitcell integrates two NOR gates, so that it can generate the 3-bit partial product of $M[2:0] = (1'b1, I[1:0] \times W_M)$. An 8-bit weight in this implementation is represented by one T1 bitcell (sign bit), five T2 bitcells (exponent bits), and two T3 bitcells (mantissa bits).

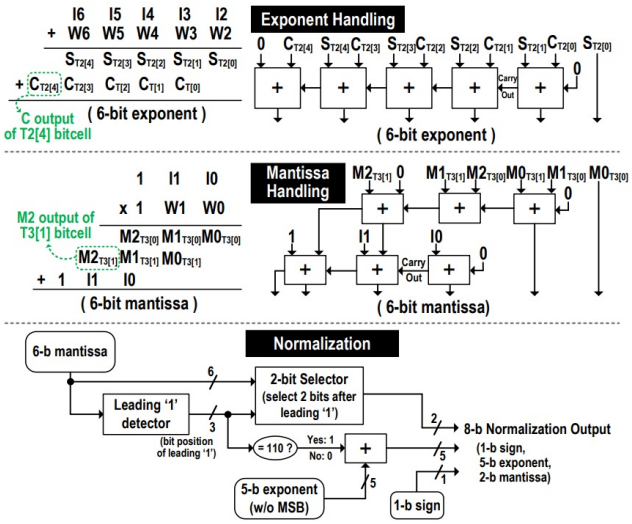


Fig. 4. Block diagram of exponent handling module (top), mantissa handling module (middle), and normalization module (bottom).

Each of the T1/T2/T3 bitcells consumes 1.3×1.5 μm^2 bitcell area for regularity.

After the sub-operations by T1/T2/T3 bitcells, ensuing computations are performed to complete the overall 8-bit floating-point multiply operation. In the **exponent handling module** shown in Fig. 4 (top), the sum and carry bits from the full adder in each of the five T2 bitcells are added through the area-efficient ripple carry adder in the exponent handling module, and outputs the 6-bit exponent. Essentially the five individual 1-bit sums are translated to a concrete 5-bit sum of the exponents. In the **mantissa handling module** shown in Fig. 4 (middle), the three partial products obtained from the T3 bitcells over 1-bit mantissa weights are shifted and accumulated further to obtain the 6-bit mantissa product. The exponent handling module is turned off in the BF8 mode, since the exponent is shared among the weights in a tensor group. In the **normalization module**, the outputs of the exponent/mantissa handling modules subsequently undergo the normalization operation (Fig. 4, bottom). The normalization module in each row detects the leading '1' in the 6-bit mantissa, based on which adjusts the 2-bit mantissa and 5-bit exponent, and outputs 8-bit (FP8) value to the adder tree.

Finally we have a multiplexer to clock-gate the submodules during the BF8 mode. The EN_FP8 signal is used to toggle between the two modes. The signal is responsible for turning on/off specific submodules and selecting a mode-specific adder tree.

B. FP Accumulate Operation:

The MAC output of a column group is formulated as:

$$Col_{OUT} = (IN_0 \times W_0) + ... + (IN_{63} \times W_{63}) \quad (3)$$

To complete the accumulation of 64 multiplication results, we have two mode-specific adder trees dedicated to the FP8 precision mode and the BF8 precision mode. While it is possible to use a generic adder tree supporting both modes,

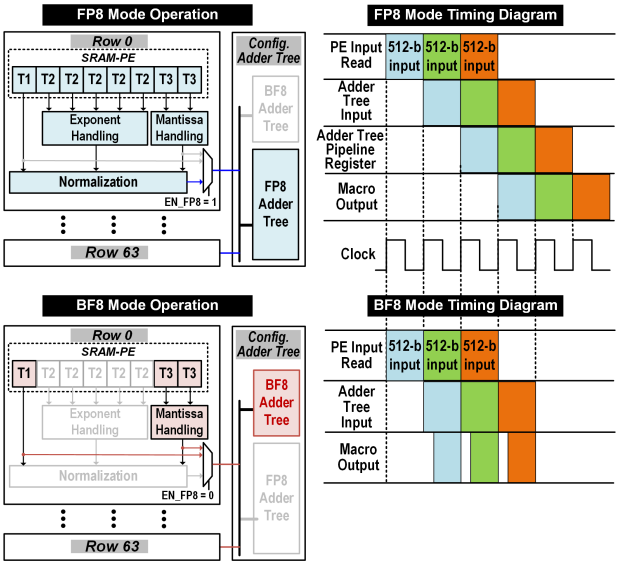


Fig. 6. (Left) Active submodules in FP8 and BF8 mode. (Right) Timing diagrams for FP8 and BF8 modes.

dedicating an adder tree to each mode allows us to fully exploit the benefits of BF8 mode by turning off the more complex FP8 adder portion of the architecture to lower the overall power consumption.

The BF8 adder tree employs a series of fixed-point precision adders as each multiplied output shares the same exponent and can be added directly, as long as the common exponent value is accounted for after the accumulation. The fixed-point point accumulated mantissa (12-bit) over the 64 rows is then passed to a normalization module, along with the shared 5-bit exponent to obtain the normalized 8-bit MAC output.

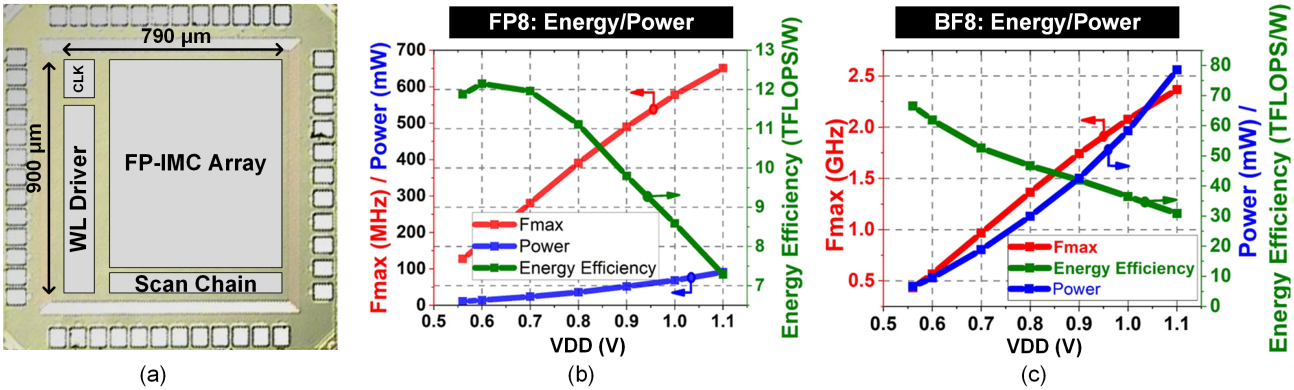
The FP8 adder tree is a series of FP16 adders used to accumulate the values over all the rows in a column group. We employ FP16 adders to ensure that there is no information loss because of overflow. To accumulate 64 FP8 multiplication results for FP8, we implement an adder tree with FP16 precision (1-5-10) in two pipeline stages. The first level consists of eight FP16 adder trees that return a normalized FP8 accumulated output. These are then connected to a single FP16 adder tree in the second level, which outputs the final FP8 MAC result. Once the accumulation within the adder tree is complete, the FP16 mantissa (8 bits) can now be adjusted to a FP8 mantissa (2 bits) by adjusting the exponent bits.

When the EN_FP8 signal is '1' ('0'), the FP8 mode is enabled and the FP8 (BF8) adder tree is turned on, and the BF8 (FP8) adder tree is completely clock-gated (Fig. 6, left). During the BF8 mode, all the rows share exponent, which makes the addition of exponent during FP multiply operation redundant. The T2 bitcells, the exponent handling block and the normalization modules can now be turned off. The expected exponent sum is directly accounted for inside the BF8 adder tree. This simpler BF8 mode logic allows the macro input-to-output latency to be a single cycle (Fig. 6, right). Applying consecutive inputs every cycle, the MAC output is obtained every cycle.

The FP8 adder tree is much more complex and results in a higher latency corresponding to the BF8 counterpart. To improve the throughput of this relatively complex FP8 mode logic, we pipelined the overall FP8 MAC operations in three stages. We place pipeline registers at the (1) outputs of SRAM bitcells and (2) inputs of adder tree. As shown in the timing diagram, we have the 3-cycle latency for FP8 and 1-cycle latency for BF8. With the 3-stage pipelining for FP8, consecutive inputs (512 bits for 64 rows) can be applied every cycle without any stall, and the 64×64 array can maintain a high throughput of 64×2×8=1024 FP8 operations per cycle.

III. Chip Measurements and Results

The FP-IMC macro design was implemented in a prototype chip in 28nm CMOS technology (Fig. 5(a)), where the core area consumes 0.71 mm^2. With dynamic voltage scaling experiments, the prototype chips were fully functional down to the 0.56V supply voltage at room temperature for both FP8 mode (Fig. 5(b)) and BF8 mode (Fig. 5(c)). For power measurements, the inputs were continuously fed into the macro with a pseudo-random linear-feedback shift register (LFSR). For weights, we randomly assigned non-zero weights between the minimum and maximum range of FP8.



(a) (b) (c)

Fig. 5. (a) Chip micrograph. (b) Energy-efficiency and power of FP-IMC in FP8 mode. (c) Energy-efficiency and power of FP-IMC in BF8 mode.

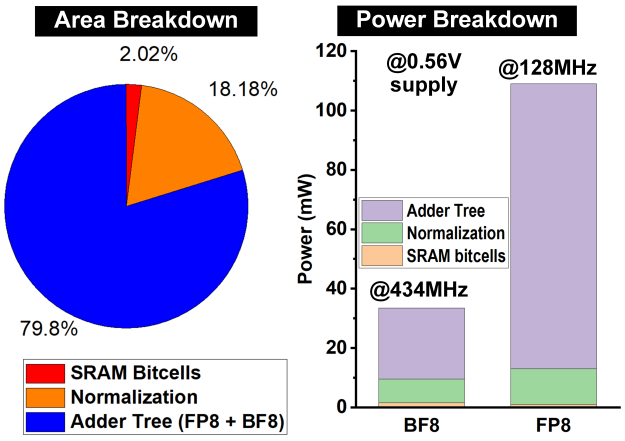


Fig. 7. Area and power breakdown of FP-IMC macro.

For FP8 mode, the highest energy-efficiency of 12.1 TFLOPS/W was obtained at 0.6V, and the highest throughput of 0.67 TFLOPS was obtained at 1.1V at 650 MHz. The BF8 mode demonstrated the highest energy-efficiency of 66.6 TFLOPS/W at 0.56V, and 2.42 TFLOPS throughput was achieved at 1.1V at 2.4 GHz.

Fig. 7 shows the area and power breakdown of the FP-IMC macro, where the total power is based on chip measurements and the module-level power breakdown is based on post-layout simulation. The adder tree with FP16 precision dominates both area and power consumption. For BF8 mode, the frequency is $>3\times$ higher than that of the FP8 mode. It should be noted that the multiplexer power and area are included with the normalization modules for the area and power breakdowns. Since the normalization modules are turned off in the BF8 mode, the reported normalization power in the BF8 mode is from the multiplexers.

To evaluate the energy-efficiency benefits of FP-IMC, we also implemented a conventional (non-IMC) 8-bit floating-point MAC engine with 64×64 off-the-shelf SRAM array generated by commercial memory compiler in 28nm CMOS. Compared to the non-IMC implementation that performs the same functionality (post-layout simulation), due to enhanced parallelism and tight integration of computation, the proposed FP-IMC macro achieves $\sim2.8\times$ higher energy-efficiency.

Table I compares the proposed FP-IMC work to prior floating-point IMC works. We achieve among the highest macro-level throughput normalized per IMC array size, due to pipelining and high operating frequency, while some works present higher level of system integration. For the same FP8 precision, our work achieves $7.4\times$ higher energy-efficiency than [6] at the macro level. BF8 mode energy-efficiency is observed to be the highest, which is $>2.2\times$ higher even than the energy-efficiency reported in [5].

IV. Conclusion

In this work, we explored the implementation of floating-point in-memory computing macro, resulting in a significant improvement in energy-efficiency. Our discussions have focused on the key challenge with implementing the MAC operations crucial to a DNN in IMC and we provide a novel solution to the same. This allows for a large improvement in throughput and energy-efficiency owing to the IMC while still maintaining a high accuracy with floating-point precision. We achieved 66.6 TFLOPS/W and 12.1 TFLOPS/W energy-efficiency for BF8 and FP8 modes, respectively.

TABLE I
Comparison with prior floating-point IMC works.

	ISSCC 2022 [5]	ESSCIRC 2022 [6]	ISSCC 2023 [7]	ISSCC 2023 [8]	This work
Technology	28nm	28nm	22nm	28nm	28nm
Memory	12Kb	16Kb	832Kb	64Kb	4Kb
CIM Bitcell	10T	8T/10T	6T	2x6T	8T/14T
Accuracy Loss	No	No	Yes (analog circuits)	Yes (approximate computing)	No
Voltage	0.6-1.0V	0.5-0.9V	0.6-0.8	0.6-0.9	0.55-1.2V
FP Precision	BF16	FP8-FP32	BF16	BF16	FP8/BF8
Frequency	50-220 MHz	53-403 MHz	151-156 MHz	71-147 MHz	650MHz /2.4GHz
Area (mm^2)	3.28x2.04	1.0x1.0	3x6	0.7x0.2	1.2x1.3
Normalized[a] Peak Throughput (GFLOPS/Kb)	90 (1.0V, BF16)	3.62 (0.9V, FP8)	1.54 (0.8V, BF16)	4.67 (0.9V, BF16)	166.9/606.0 (1.1V, FP8/BF8)
Energy Efficiency (TFLOPS/W)	29.2 (0.65V, BF16)	1.64 (0.5V, FP8)	17.6[b] (0.6V, BF16)	31.6 (0.6V, BF16)	12.1/66.6 (0.55V, FP8/BF8)

[a]Throughput normalized per Kb of CIM memory size.
[b]Average energy-efficiency reported in [7]. For 90% input sparsity, 70.2 TFLOPS/W reported.

V. Acknowledgements

This work is supported in part by NSF grants 1652866 and 2144751, and CoCoSys Center in JUMP 2.0, an SRC program sponsored by DARPA.

References

[1] N. Wang *et al.*, "Training deep neural networks with 8-bit floating point numbers," *NeurIPS*, 2018.
[2] N. P. Jouppi *et al.*, "Ten lessons from three generations shaped Google's TPUv4i: Industrial product," in *ACM/IEEE ISCA*, 2021.
[3] Y.-D. Chih *et al.*, "An 89 TOPS/W and 16.3 TOPS/mm^2 all-digital SRAM-based full-precision compute-in memory macro in 22nm for machine-learning edge applications," in *IEEE ISSCC*, 2021.
[4] B. Zhang *et al.*, "A 177 TOPS/W, capacitor-based in-memory computing SRAM macro with stepwise-charging/discharging DACs and sparsity-optimized bitcells for 4-bit deep convolutional neural networks," in *IEEE CICC*, 2022.
[5] F. Tu *et al.*, "A 28nm 29.2 TFLOPS/W BF16 and 36.5 TOPS/W INT8 reconfigurable digital CIM processor with unified FP/INT pipeline and bitwise in-memory booth multiplication for cloud deep learning acceleration," in *IEEE ISSCC*, 2022.
[6] S. Jeong *et al.*, "A 28nm 1.644 TFLOPS/W floating-point computation SRAM macro with variable precision for deep neural network inference and training," in *IEEE ESSCIRC*, 2022.
[7] P.-C. Wu *et al.*, "A 22nm 832Kb hybrid-domain floating-point SRAM in-memory-compute macro with 16.2-70.2 TFLOPS/W for high-accuracy AI-edge devices," in *IEEE ISSCC*, 2023.
[8] A. Guo *et al.*, "A 28nm 64-kb 31.6-TFLOPS/W digital-domain floating-point-computing-unit and double-bit 6T-SRAM computing-in-memory macro for floating-point CNNs," in *IEEE ISSCC*, 2023.
[9] S. Q. Zhang *et al.*, "Fast: DNN training under variable precision block floating point with stochastic rounding," in *IEEE HPCA*, 2022.