M3FPU: Multiformat Matrix Multiplication FPU Architectures for Neural Network Computations

Won Jeon, Yong Cheol Peter Cho, HyunMi Kim, Hyeji Kim, Jaehoon Chung,
Juyeob Kim, Miyoung Lee, Chun-Gi Lyuh, Jinho Han, and Youngsu Kwon
AI Processor Research Team, AI SoC Research Department
Electronics and Telecommunications Research Institute
Daejeon, Republic of Korea
Email: {jeonwon,cho,chaos0218,hyejikim,jchung,juyeob,sharav,cglyuh,soc,yskwon}@etri.re.kr

***Abstract*—Parallel computing performance on floating-point numbers is one of the most important factors in modern computer systems. The hardware components of floating-point units have the potential to improve parallel performance and resource utilization, however, the existing vector-type multiformat parallel floating-point units cannot take advantage of them. We propose M3FPU, a new matrix-type multiformat floating-point unit that applies an outer product matrix multiplication method to a multiplier tree of floating-point units to increase parallelism and resource utilization by the square. M3FPU utilizes the unused part of the multiplier tree of the existing floating-point unit that is filled with zeros. The proposed M3FPU is implemented on a 12nm silicon process and achieves a 44.17% smaller area compared to the state-of-the-art multiformat floating-point unit architecture when supporting the same number of 8-bit floating-point number parallel operations.**

Keywords—floating-point unit, multiformat arithmetic, matrix multiplication, machine learning

I. INTRODUCTION

With advances in machine learning and high-performance computing applications over the past decade, the importance of throughput processors, such as graphics processing units and neural processing units, has emerged. In particular, the parallel computing power of floating-point units (FPUs) determines the overall performance of such throughput processors. Furthermore, there have been extensive researches that employ floating-point formats smaller than the IEEE 754 32-bit number (FP32) [1][2]. The smaller formats such as FP16, brain floating-point (BF16), or FP8 numbers can improve computational and area efficiency compared to FP32 without significantly degrading the accuracy or performance of machine learning applications. In this context, FPU architectures that support the multiformat arithmetic have been proposed [3][4][5][6][7].

We first observe that the existing multiformat FPUs cannot fully utilize the hardware resources. The existing multiformat FPUs vectorize multiple inputs and perform parallel computation (e.g., two FP32 operations can be performed on one FP64 FPU). Though they achieved higher hardware utilization and throughput, there are underutilized resources such as a multiplier tree. The size of the multiplier tree in FPUs increases with the square of the one input width. When the input data width is halved, the utilization of the multiplier tree becomes quartered. For the general-purpose computing, vectorized element-wise computation has higher usability. Therefore, the part of the multiplier that generates unnecessary results is not used and is filled with zeros.

Second, we revisit matrix multiplication processes to find a suitable computation for the zero-filled part of the multiplier tree. Matrix multiplication is the most common operation in machine learning or neural network computations. When using the outer product method for matrix multiplication, each computation stage takes two N-sized vectors and generates an N^2-sized matrix as an output. We can observe that the output matrix is identical to the result of the multiformat FPU when including the unused part of the multiplier tree. Combining the two observations, we propose a new multiformat FPU that can perform outer product matrix multiplication inside of an FPU and fully utilize the hardware resources that increase by the square, named multiformat matrix multiplication FPU (*M3FPU*).

The primary contributions of this paper are as follows:

- Discovering the resource underutilization problem of the existing multiformat FPUs and the opportunity to improve it with matrix multiplication operations.
- Proposing a new FPU architecture, M3FPU, that can perform outer product matrix multiplications inside of an FPU and improve the hardware resource utilization.
- Implementing the proposed M3FPUs on a 12nm silicon process, integrating them with RISC-V cores that can control and execute the M3FPUs, and scaling them up for product-level throughput performance.

II. BACKGROUND AND MOTIVATION

A. Floating-Point Number Formats

The floating-point numbers represent real numbers with formulaic formats in computer systems. The most commonly used data formats are defined in the IEEE 754 standard. A floating-point number data comprises the sign (s), exponent (e), and mantissa (m). The real number (N) of a floating-point number is decoded using the following formula:

$$N = (-1)^s \times (1 + m) \times 2^{e - bias} \quad (1)$$

Generally, 32-bit single-precision floating-point numbers and 64-bit double-precision floating-point numbers are the most widely used formats in modern computer systems. For specific applications such as neural network computations 16-bit or 8-bit floating-point numbers are used. The formats of floating-point numbers are summarized in TABLE I.

TABLE I. FORMATS OF FLOATING-POINT NUMBERS

Format	Sign	Exponent	Mantissa	Bias
FP64	1	11	52	1023
FP32	1	8	23	127
FP16	1	5	10	15
BF16 [1]	1	8	7	127
FP8 [2]	1	4	3	7

B. Multiformat Floating-Point Number Multiplier

The multiplication process of two floating-point numbers roughly consists of three parts: exponent addition, mantissa multiplication, and normalization. The mantissa multiplier performs (1 + m) bit-wide integer multiplication owing to the implicit bit and usually requires large area resources. To reduce the redundant area overhead for multiple floating-point format computation, many previous works have proposed the mantissa multiplier that can be shared for computations of multiple smaller data formats [3][4][5][6][7].

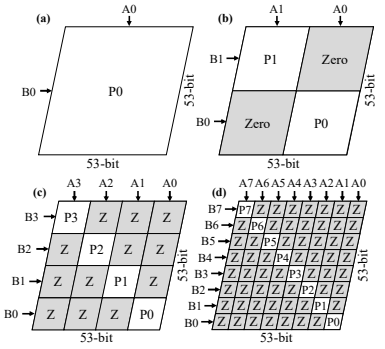

Fig. 1. Mantissa multiplier tree that can compute multiple mantissas of smaller data types in parallel. (a-d) show the FP64, FP32, FP16/BF16, and FP8 multiplication modes, respectively.

The previously proposed multiplier can perform parallel mantissa multiplication as illustrated in Fig. 1. The multiplier can perform one double-precision, two single-precision, or four half-precision floating-point mantissa multiplications per operation, making the FPU a kind of vector machine. For smaller data types except for double-precision, only diagonal parts of the multiplier are used, and the unused parts are filled with zeros to avoid corrupting the final multiplication results. The zero-filling method is highly suitable for vector processors. However, the resource utilization of the mantissa multiplier can be greatly improved if it can compute meaningful data using the currently unused part of the multiplier rather than filling zeros.

III. M3FPU Architecture

To maximize the resource utilization of the mantissa multiplier of FPUs, the first thing to be considered is which data to calculate using the limited input data path. We target the matrix multiplications in neural network applications as operations that utilize the proposed FPU architecture. The FPU can perform multiformat matrix multiplications that maximize the multiplier resource utilization, named multiformat matrix multiplication FPU (*M3FPU*).

A. Matrix Multiplication Method for M3FPU

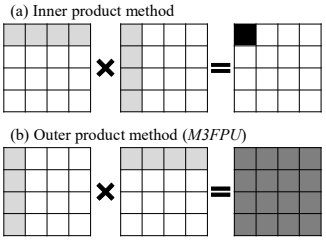

Fig. 2. Inner and outer product method to complete matrix multiplication. The black element in (a) is a final result and the dark gray elements in (b) are the partial sum of final values.

In general, matrix multiplication can be processed in two different ways as shown in Fig. 2 [9] [10]. While the inner product method requires a smaller on-chip memory size for result values, its input vector reusability for computation is lower. 16 iterations are required for final results in the case of the inner product method. On the other hand, the outer product method is required to hold a much larger size of partial result values. However, each input vector is only read once, making input vector reusability higher. Only 4 iterations are required to complete a matrix multiplication in the case of the outer product method. For M3FPU, we apply the outer product method for the matrix multiplication. The zero-filled part of the previously proposed multiformat parallel mantissa multiplier can be used for meaningful computations with the outer product matrix multiplication.

B. Basic Multiplier Structure: BFPU

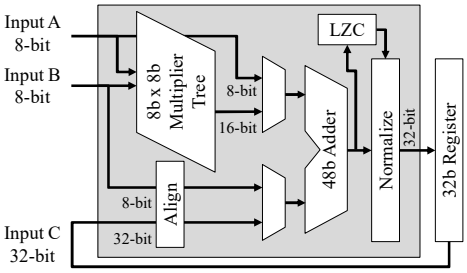

Fig. 3. A basic multiplier for M3FPU, FP8 multiplier. The BFPU can perform addition, subtraction, and multiplication of FP8 or INT8 and MAC operations with FP32 or INT32 for accumulation.

The most basic computing structure of the proposed M3FPU is a Byte FPU (*BFPU*) that can compute FP8 floating-point number operations. The BFPU consists of exponent handling and aligning path, multiplier tree, adder, leading-zero counter (LZC), and result normalization path as shown in Fig. 3. The BFPU can perform addition, subtraction, or multiplication operations of two FP8 floating-point or two 8-bit integer (INT8) inputs. Although multiplication of FP8 only requires a 4-bit mantissa multiplier, we employ an 8-bit signed multiplier to support INT8 multiplication and partial product of larger data types, such as FP64. To achieve a higher area-performance efficiency for matrix operations in neural network computations, the BFPU does not support floating-point number division or square root operations.

In neural network computations, especially for training processes, accumulating results in small-sized floating-point numbers often suffers from the truncation [8]. Because M3FPU mainly targets neural network computations, our BFPU produces the results of addition, subtraction, and multiplication in FP32 formats (or INT32 for INT8 computations) to avoid such issues. For that, the FPU consists of a 48-bit adder, LZC, and normalization path. Furthermore, because accumulation processes are essential parts of matrix multiplication, the FPU supports the multiply-accumulate (MAC) operation between the result of FP8 multiplication and FP32 value stored in the register (or MAC between INT8 and INT32).

C. Merging Multiple Small FPUs into a Larger FPU

Using the multiplier tree of the proposed BFPU and additional merging logics, we build a larger mantissa multiplier for FP16 and BF16 floating-point numbers. For example, the multiplicand of an FP16 number is an 11-bit

unsigned integer, including mantissa bits and an implicit bit. The multiplication of two 11-bit data can be separated and computed using the multiplication formula in Fig. 4. According to the formula, one large 11-bit multiplication can be separated into four 6-bit multiplication. Applying this method, we can build one mantissa multiplier for 16-bit floating-point numbers using four BFPUs and additional merging logics, such as shifter and adder. Note that the multiplier tree of BFPU is capable of multiplying two 8-bit signed integer numbers.

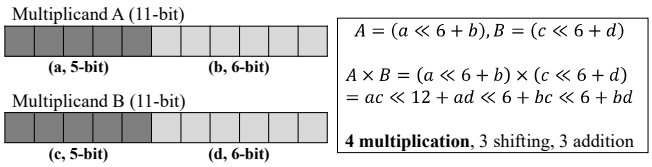

Fig. 4. Multiplying two FP16 mantissas by dividing each number into two smaller values and perform four INT8 multiplications.

The proposed FPU merging technique fits well with the multiplier's data input path and outer product method for the matrix multiplication. We place four BFPUs in square shape as shown in Fig. 5. In case the input data type is FP8 as in (a) in the figure, each BFPU performs independent four multiplications. P1(A1×B0) and P2(A0×B1) are usually unnecessary results in the case of the parallel vector multiplier presented in Fig. 1. However, the outer product method for the matrix multiplication makes the results essential. The FPUs can perform the multiplication of FP16 as in (b) in Fig. 5. Each BFPU works in INT8 mode to perform 4 multiplications in the formula of Fig. 4 in parallel. The partial products (i.e., *ac*, *bc*, *ad*, and *bd*) are then shifted and added to complete the FP16 mantissa multiplication.

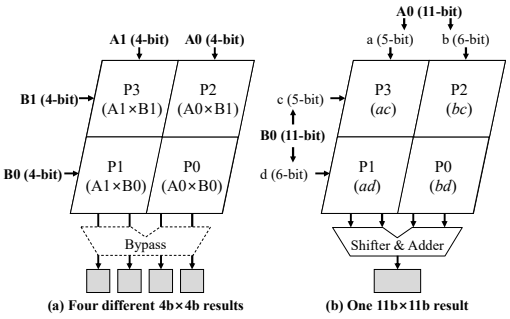

Fig. 5. Computing multiplication of (a) four FP8 mantissas or (b) one FP16 mantissa using fully shared hardware modules.

The FPU merging technique is recursively applied from FP8 to FP64, for example, an FP64 mantissa multiplication process requires four integer multiplication stages: INT8, INT16, INT32, and INT64. In addition to the BFPU, we define half-precision FPU (*HFPU*), single-precision FPU (*SFPU*), and double-precision FPU (*DFPU*). Because the largest supporting format of M3FPU is the FP64, a DFPU is equal to M3FPU. Four FPUs of each stage are merged to make one FPU of the next stage. When an FPU is used as a subset of a larger FPU, it works in integer multiplier mode, and logics for floating-point operations, such as exponent handler, LZC, and normalization path, are disabled.

TABLE II. describes input and output information of various FPU modules of M3FPU according to supporting operation modes. From the view of the M3FPU's I/O, it takes 2N inputs and generates N^2 outputs, where N is 1, 2, 4, 8 for FP64, FP32, FP16/BF16, and FP8, respectively. As addressed in Section II, existing parallel mantissa multipliers take 2N inputs and generate only N outputs. In summary, the proposed M3FPU generates more computation results and improves resource utilization by using the zero-filled areas of the existing FPUs.

TABLE II. M3FPU I/O FORMAT ACCORDING TO OPERATION MODE

Mode	I/O[a]	FPU Module			
		DFPU	**SFPU**	**HFPU**	**BFPU**
FP64	I	1x FP64 (53)	27/26	14/13/ 13/13	7/7/7/6/ 7/6/7/6
	O	**1x FP64**	4x INT64	16x INT32	64x INT16
FP32	I	2x FP32	24	12/12	6/6/6/6
	O	**4x FP32**	1x FP32	4x INT32	16x INT16
FP16	I	4x FP16	2x FP16	11	6/5
	O	**16x FP32**	4x FP32	1x FP32	4x INT16
BF16	I	4x BF16	2x BF16	8	4/4
	O	**16x FP32**	4x FP32	1x FP32	4x INT16
FP8	I	8x FP8	4x FP8	2x FP8	4
	O	**64x FP32**	16x FP32	4x FP32	1x FP32

[a.] Input row shows data format or mantissa slice for a single input operand.

IV. IMPLMENTATION RESULTS

A. Implementation

We fully design custom RTL modules and functions for M3FPU using SystemVerilog. The functionalities of M3FPU, such as arithmetic operations, rounding, special number handling, are extensively tested with RTL testbenches using Cadence Xcelium Simulator. To perform area comparisons with existing FPU designs, we synthesize and estimate the area of the proposed M3FPU designs and various state-of-the-art FPU designs (i.e., FPnew [7]) using Synopsys Design Compiler. In addition, to validate the compatibility with general processors and scalability of multiple M3FPUs, we integrate the proposed M3FPU RTL design with a custom RISC-V RV64G processor. We perform silicon virtual prototyping (SVP) of the extensive parallel computing system including multiple instances of M3FPU. Throughout the synthesis and SVP processes, we used TSMC 12-nm technology cell library at 0.72V global operating voltage. In our implementation, the maximum achievable frequencies of the basic modules of FPnew (FP8 FPU) and M3FPU (BFPU) are 497.51 MHz and 746.27 MHz, respectively.

B. Area Comparisons of M3FPU and FPnew

We implement various baseline FPU designs using the FPnew platform that provides implementations of custom multiformat FPU architectures. Fig. 6 presents the area evaluation results using the implementation environment described in the previous subsection. In the figure, the *FPnew Single* design does not support multiformat computations, thus only computes each data type. *FPnew Multiple* design simply includes multiple instances of each FPU and the numbers of each FPU are determined to match the parallelism level of an M3FPU. Lastly, *FPnew Vector* design is implemented using the multiformat (or transprecision) feature of the FPnew design platform. The list of numbers in the square bracket notation represents the number of parallel floating-point operations for FP64, FP32,

FP16, BF16, and FP8 in order. Note that when applied to each M3FPU design, the notations are as follows: 'BFPU [0, 0, 0, 0, 1]', 'HFPU [0, 0, 1, 1, 4]', 'SFPU [0, 1, 4, 4, 16]', 'DFPU [1, 4, 16, 16, 64]'.

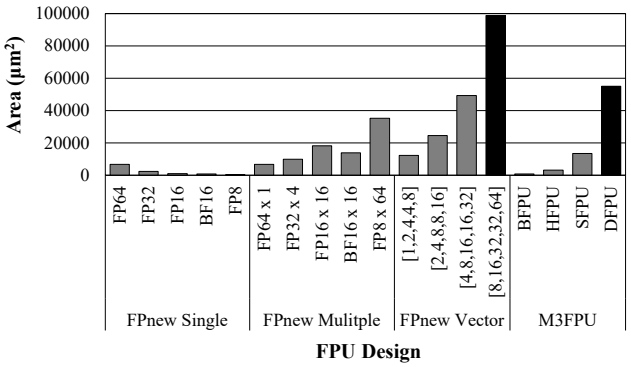

Fig. 6. Post-synthesis area results of the proposed M3FPU designs compared to various FPU designs generated using FPnew platform [7].

To provide 64 FP8 operations per cycle, a processor needs to have separate 64 FP8 FPUs (*FPnew Multiple*), 8 FP64 multiformat FPUs (*FPnew Vector*), or 1 DFPU (*M3FPU*). According to our area evaluation results, the three implementation candidates require an area of 35212.49 μm^2, 98757.18 μm^2, and 55141.36 μm^2, respectively. Note that despite the smallest area, using the naïve 64 FP8 FPUs does not support operations on BF16, FP16, FP32, and FP64. DFPU requires a 44.17% smaller area compared to 8 FP64 multiformat FPUs. Furthermore, when gathering each *FPnew Multiple* implementation to achieve [1, 4, 16, 16, 64] FPU configuration using the FPnew FPUs, it requires 87841.82 μm^2 of area, making the area of DFPU 34.23% smaller.

C. M3FPU Integration with RISC-V Cores

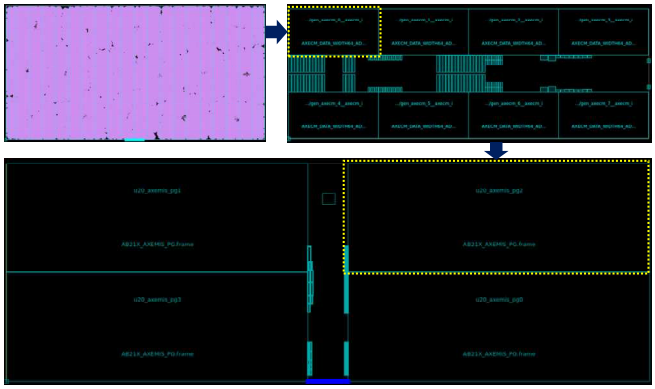

Fig. 7. SVP layouts of 4x4 M3FPUs with a single RISC-V core (upper left, 16 FPUs), eight instances of the integrated computing unit (upper right, 128 FPUs), and four instances of the cluster (lower, 512 FPUs).

We integrate the proposed M3FPU modules with a custom RISC-V core (i.e., arithmetic pipelines for RV64G, instruction, and data cache memories). In detail, a single RISC-V core includes 4x4 M3FPUs to perform a 4x4 FP64 outer product matrix multiplication in a single cycle. Using the matrix-type multiformat arithmetic of M3FPU, the 4x4 M3FPU module can perform matrix multiplications of 8x8 FP32, 16x16 FP16/BF16, 32x32 FP8 data in a single cycle. The RISC-V core executes custom instructions to control the M3FPUs and fetches data from memory spaces to M3FPUs. Using the integrated computing unit, we build a cluster of processors to verify the scalability to the high floating-point throughput required for neural network computations. We are currently working on building an FP8 Peta-FLOPS level processor with thousands of M3FPUs. Fig. 7 shows the SVP layout of a part of the processor which includes 512 M3FPUs.

V. Conclusion

In this paper, we observe that the existing multiformat FPUs are missing a significant potential for parallel performance and resource utilization due to the vector-type computations. We propose M3FPU architecture that can perform outer product matrix multiplications inside of an FPU, improve the parallel performance, and maximize the hardware resource utilization. The M3FPU performs computations for all fully-connected input combinations, not element-wise vector computations. Furthermore, we integrate multiple M3FPU instances with custom RISC-V cores to validate compatibility and scalability. With the new FPU merging technique and novel architecture, M3FPU achieves a 44.17% smaller area compared to the state-of-the-art multiformat floating-point unit architecture when supporting the same number of FP8 parallel operations.

Acknowledgment

This research was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2018-0-00195, Artificial Intelligence Processor Research Laboratory).

References

[1] Intel, "BFLOAT16 – Hardware Numerics Definition White Paper," November 2018.

[2] N. Wang, J. Choi, D. Brand, C. Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," In Proceedings of the 32nd International Conference on Neural Information Processing Systems, pp. 7686-7695, December 2018.

[3] H. Kaul, et al., "A 1.45 GHz 52-to-162GFLOPS/W variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm CMOS," In 2012 IEEE International Solid-State Circuits Conference, pp. 182-184, February 2012.

[4] K. Manolopoulos, D. Reisis, and V. A. Chouliaras, "An efficient multiple precision floating-point Multiply-Add Fused unit," Microelectronics journal, vol. 49, pp. 10-18, March 2016.

[5] V. Arunachalam, A. N. J. Raj, N. Hampannavar, and C. B. Bidul, "Efficient dual-precision floating-point fused-multiply-add architecture," Microprocessors and Microsystems, vol. 57, pp. 23-31, March 2018.

[6] H. Zhang, D. Chen, and S. B. Ko, "Efficient multiple-precision floating-point fused multiply-add with mixed-precision support," IEEE Transactions on Computers, vol. 68, no. 7, pp. 1035-1048, July 2019.

[7] S. Mach, F. Schuiki, F. Zaruba, and L. Benini, "Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 29, no. 4, pp. 774-787, December 2020.

[8] N. J. Higham, "The accuracy of floating point summation," SIAM Journal on Scientific Computing, vol. 14, no. 4, pp. 783-799, 1993.

[9] A. Buluc and J. R. Gilbert, "On the representation and multiplication of hypersparse matrices," In 2008 IEEE International Symposium on Parallel and Distributed Processing, pp. 1-11, April 2008.

[10] S. Pal, et al., "Outerspace: An outer product based sparse matrix multiplication accelerator," In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 724-736, February 2018.