FPonAP: Implementation of Floating Point Operations on Associative Processors

Walaa Amer, Mariam Rakka, *Graduate Student Member, IEEE*, and Fadi Kurdahi, *Fellow, IEEE*

***Abstract*—The associative processor (AP) is a processing in-memory (PIM) platform that avoids data movement between the memory and the processor by running computations directly in the memory. It is a parallel architecture based on content addressable memory (CAM), allowing it to address data by its content and thus accelerating search and pattern recognition tasks. APs are suggested as a promising solution to the memory wall caused by the data movement bottleneck in traditional Von-Neumann architectures for data-driven applications, such as machine learning. However, modern implementations of the AP still lack support for floating point (FP) operations that are heavily used in the target applications. In this letter, we present a novel implementation of FP operations on the AP and evaluate its performance on the levels of latency and energy, showing that the proposed solution outperforms parallel FP execution on central processing unit and even GPU for large vector sizes.**

***Index Terms*—Associative processor (AP), floating point (FP), processing in-memory (PIM).**

I. Introduction

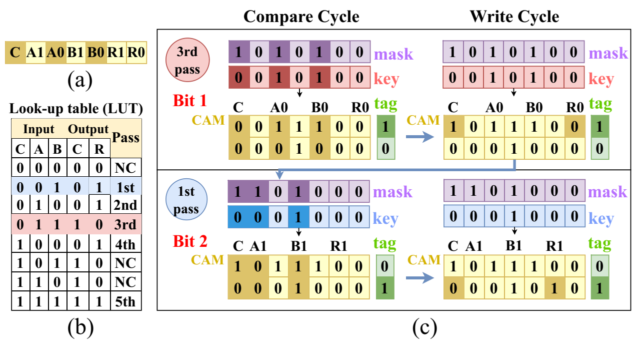
In conventional Von-Neumann systems, data are frequently transferred back and forth between the central processing unit (CPU) and memory, leading to significant delays and energy consumption. One of the promising technologies tackling this issue is processing-in-memory (PIM). This new computing paradigm integrates computational capabilities directly within the memory chips, enabling data to be processed where it resides and reducing the need for data transfer. This approach can dramatically enhance performance and efficiency, particularly for data-driven applications, such as machine learning, big data analytics, and real-time processing.

The associative processor (AP) is a content addressable memory (CAM)-based parallel computing architecture. CAMs are memory structures where data can be retrieved by its content instead of the address where the data are stored. The CAM backbone of the AP, designed to efficiently handle parallel search and pattern-matching operations, gives this architecture an advantage over conventional processors. It allows it to perform massively parallel look-up table (LUT)-based logical and arithmetic operations on stored input data across the entire memory array through a repetitive pattern of compare and write CAM cycles. However, until now, the proposed AP architectures can only run fixed point computations, due to the complexity of floating point (FP) operations.

Manuscript received 11 August 2024; accepted 12 August 2024. This work was supported in part by the U.S. National Science Foundation (NSF) under Grant CNS 2213404. This manuscript was recommended for publication by A. Shrivastava. *(Corresponding author: Walaa Amer.)*

The authors are with the Center for Embedded and Cyber-Physical Systems, University of California at Irvine, Irvine, CA 92697 USA (e-mail: walaaa@uci.edu; mrakka@uci.edu; kurdahi@uci.edu).

Digital Object Identifier 10.1109/LES.2024.3446912



Input			Output		Pass
C	A	B	C	R	
0	0	0	0	0	NC
0	0	1	0	1	1st
0	1	0	0	1	2nd
0	1	1	1	0	3rd
1	0	0	0	1	4th
1	0	1	1	0	NC
1	1	0	1	0	NC
1	1	1	1	1	5th

Fig. 1. (a) Data format in the AP. (b) Fixed-point addition LUT. (c) 2-bit fixed-point addition on the AP.

FP is a method of representing real numbers in a way that can accommodate a wide range of values. The IEEE 754 standard defines the most widely used FP representation, specifying formats for single-precision (32-bit) and double-precision (64-bit) numbers.

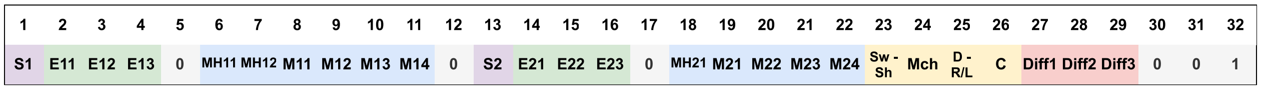
Unlike the fixed-point representation, the FP representation allows numbers to be expressed in a scientific notation-like format. The flexibility of the FP representation enables it to handle very large and very small numbers efficiently, making it essential for scientific computations. However, implementing such operations on the AP becomes challenging as the execution steps vary from one memory entry to the other and thus cannot be easily performed uniformly across all entries.

In this letter, we introduce an algorithmic implementation of FP addition, subtraction, multiplication, and division on the AP. Our contributions are summarized as follows.

1) We develop a novel format for representing data in the AP and novel LUTs for FP number comparison, normalization, and denormalization.
2) We propose algorithms for word-parallel FP operations tailored for the capabilities of the AP.
3) Our simulator evaluation shows that the proposed FP implementation outperforms the CPU and GPU on the level of latency and energy for large vector sizes, even in the worst-case scenario.

II. Background and Related Works

APs benefit from the search speed that the CAM offers by applying it to perform more complex word-parallel bit-serial computations. The operations are saved as LUTs. For every bit position, each entry in the LUT is passed as the key with



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
S1	E11	E12	E13	0	MH11	MH12	M11	M12	M13	M14	0	S2	E21	E22	E23	0	MH21	M21	M22	M23	M24	Sw - Sh	Mch	D - R/L	C	Diff1	Diff2	Diff3	0	0	1

Fig. 2. Proposed format.

the input bits in that position and needed bits for the operation masked. During the compare phase, the key is compared to the masked entries, and the tag is set for the rows where the key matches the content. This step is followed by the write phase where the corresponding output of this LUT entry is written for the rows with set tags. Fig. 1 shows an example of the fixed-point 2-bit addition on a word-parallel AP. The AP has only two rows for simplicity. The first step is to add the least significant bits (LSBs), so the bits in this position in A and B and the carry bit are masked. In the LUT, entries 1, 6, and 7 incur no change in the results, and thus are not compared against the input. During the first pass, the second entry of the LUT (001) is passed as the key representing "0" for C, "0" for A0, and "1" for B0. In this example, the keys of the first and second (010) passes do not match with any entry in the AP. However, the third one (011) matches with the first entry in the AP, so its tag is set to 1. During the write cycle, the result and updated carry value of that LUT pass are written to their corresponding bits R0 and C in the AP rows with set tags. The fourth and fifth passes do not match with any entry. The same process is repeated for bit 2, where the first LUT pass matches the second AP entry.

This AP architecture has been proposed many years ago. Reference [1] is one of the remarkable resources that first offered deep insight into the principles of content-addressable memory-based designs. Followed by this book, Scherson et al. published [2] where they proposed a novel architecture for logic and arithmetic operations on APs that enables the computations to be massively parallelized. They also demonstrate the potential of this architecture to run FP operations. Since then, researchers have further developed APs to use them for various applications, including heavily parallel computations, such as database analytic queries [3], convolution [4], and even deep learning inference [5]. Additional architectural optimizations to the AP have been also proposed. Golden et al. [6] presented a virtual vector instruction set on the AP. Zha and Li [7] presented an optimized version of the traditional AP with its own instruction set architecture (ISA) and micro architecture. However, all of these implementations still run all of the computations using fixed point.

Compared to fixed point, FP operations are much more complex. Specifically, FP addition requires first representing the two operands with the same exponent (by shifting the mantissa of the smaller number), then adding the mantissas, adjusting the exponent of the resulting value, and finally normalizing the result. FP multiplication is simpler but still is a multistep process. The exponents of the operands are added and their mantissas are multiplied and finally the result is normalized. The multiplication and normalization steps require by themselves a layered implementation with several passes over several LUTs. Implementing these steps on the AP becomes a challenge seen that the operations cannot be uniformly performed over all rows in the memory.

TABLE I
FUNCTION OF EVERY BIT IN PROPOSED FORMAT

Bit Position	Function
1&13	Sign bits of N1 and N2
2-4	Exponent Bits of N1
5&12&17	Zero bits used for shifting N1 right, N1 left, N2 right
6-7	Hidden mantissa bits of N1
8-11	Fraction mantissa bits of N1
14-16	Exponent bits of N2
18	Hidden mantissa bit of N2
19-22	Fraction mantissa bits of N2
23	Bit indicating a need for switching/shifting N1 and N2 in this row
24	Bit assisting correct parallel functional simulation
25	Bit indicating that comparing N1 and N2 is done or the direction of shifting (left or right)
26	Carry bit for addition and subtraction
27-29	Bits to store the difference E1-E2
30-32	Bits to store1 for incrementing/decrementing

Algorithm 1: FP Addition Algorithm on AP

```
Input  : the content addressable memory array $cam$
Output: $cam$
1  $cam \leftarrow$ Init()
2  for $i \leftarrow 0$ to $n-1$ do in parallel
3      $Sw \leftarrow$ Compare($N1, N2$)
4      ($N1, N2$) $\leftarrow$ Switch($N1, N2, switch$)
5      $Diff \leftarrow$ FindDiff($E1, E2$)
6      $M2 \leftarrow$ Denormalize($M2, Diff$)
7      $M1 \leftarrow$ AddMantissa($M1, M2$)
8      $Diff \leftarrow$ Clear($Diff$)
9      ($M1, Diff$) $\leftarrow$ Normalize($M1, R/L$)
10     $E1 \leftarrow$ AddSubExp($E1, Diff$)
11 end forpar
12 return $cam$
```

III. Proposed Solution

A. Floating Point Addition and Subtraction

FP Addition is a complex operation, mainly since it includes shifting operations that are needed for denormalizing the operands to add the mantissas and then normalizing the result. The amount of shifting is different for every entry in the AP, and thus implementing it as a parallel operation becomes challenging. To tackle this problem, we propose a novel format for correct word-parallel 8-bit FP addition/subtraction on the AP, shown in Fig. 2. We are assuming an E3M4 representation, where 1 bit is reserved for the sign, 3 bits for the exponent, and 4 bits for the mantissa. However, this format can be generalized to other data types, such as 8-bit E4M3, half-precision, full-precision, and double-precision. Table I elaborates on the use of every bit in the format.

Algorithm 1 explains the different steps followed to compute FP addition on the AP. Line 1 shows the first step which is to load the values into the AP in the correct format shown

TABLE II
PROPOSED LUTS. (A) LUT FOR COMPARING SIGNS. (B) LUT FOR COMPARING EXPONENTS OR MANTISSAS. (C) LUT FOR SHIFTING RIGHT. (D) LUT FOR SHIFTING LEFT

(a)

S1	S2	Sw	D	Note
0	0	0	0	NC[1]
0	1	0	1	
1	0	1	1	
1	1	0	0	NC

(b)

D	E1	E2	Sw	D	Note
0	0	0	0	0	NC
0	0	1	1	1	
0	1	0	0	1	
0	1	1	0	0	NC
1	x	x	0	0	NC

(c)

Sh	M1	M2	M2	Note
0	0	0	0	NC
0	0	1	0	
0	1	0	1	
0	1	1	1	NC
1	x	0	0	NC
1	x	1	1	NC

(d)

Sh	M1	M2	M1	Note
0	0	0	0	NC
0	0	1	1	
0	1	0	0	
0	1	1	1	NC
1	0	x	0	NC
1	1	x	1	NC

[1] NC = No Change.

in Fig. 2, including copying N1 and N2 for every row and initializing the values of the bits in yellow to zero, N1 and N2 being the two 8-bit FP operands. In line 3, the two operands are compared using the LUTs shown in Table II(a) and (b). For rows where N2 is larger than N1, the *Sw* flag in bit 23 is raised. In line 4, the switching operation is executed for the previously flagged rows. These two steps ensure that N1 is larger than N2 for all entries in the memory, allowing us to run the same following steps on all rows in parallel. Then, in line 5, the difference between the two exponents is computed and saved in *Diff* in order to identify the amount of shifting needed to denormalize N2 in line 6. This latter step requires the shifting operation that introduces variation across rows. To tackle this issue, we developed the LUTs shown in Table II(c) and (d) implementing a shift operation for both right and left, respectively. While denormalizing, the shifting left operation will take place *Diff* times. Before every iteration, *Diff* will be compared to zero, and if the condition is satisfied, the shift flag *Sh* will be set for that specific row. The *Sh* flag will then be included in the comparing operation as shown in the tables. If the flag is set, shifting is not performed and no change takes place. Now that M1 and M2 are aligned, they are added in line 7 and their result is stored in place of M1. The difference bits are then cleared in line 8 in order to reuse them as a counter to identify the amount of shifting needed to normalize the results. The result stored in M1 is then normalized in line 9 while keeping track of the number of shifts made in the counter. Finally, the value stored in the counter is added or subtracted from the exponent E1 in line 10 to find the final normalized format of the result. To implement subtraction, the mantissa addition in line 7 is replaced by mantissa subtraction.

The complexity of the FP addition is $O(m^2)$, m being the number of bits representing the mantissa, as shown in

$$\begin{aligned} T_{\text{FPAdd}} &= T_{\text{Comp}} + T_{Sw} + T_{\text{FindDiff}} + T_{\text{Denorm}} \\ &\quad + T_{\text{AddMant}} + T_{\text{Clr}} + T_{\text{Norm}} + T_{\text{AddSubExp}} \\ &= 2 * \Big(O(s+m+e) + O(e) + O\Big(m + m^2 + m * 2^e\Big)\Big) \\ &\quad + O(m) + O(1) \\ &= O\Big(m^2\Big). \qquad (1) \end{aligned}$$

Algorithm 2: FP Mult and Div Algorithms on AP

```
Input : the content addressable memory array cam
Output: cam
1 cam ← Init()
2 for i ← 0 to n − 1 do in parallel
3     E1 ← AddExp(E1, E2)
4     M1 ← MultMantissa(M1, M2)
5     (M1, Diff) ← Normalize(M1, R/L)
6     E1 ← AddSubExp(E1, Diff)
7 end forpar
8 return cam
```

B. Floating Point Multiplication

Algorithm 2 explains the different steps followed to compute FP multiplication on the AP. The process is initialized by loading the values into the AP in the format previously discussed in Fig. 2. For every row, the exponents are then added in place. Following that step, the mantissas are multiplied also in place. Multiplication is implemented using the shift method where the LSB of the multiplicand is checked at every iteration and its value is added to the accumulator if the LSB is set. The accumulator and multiplier are then shifted by 1 to the right. For the mantissa multiplication, this process is repeated m times. The final result is then normalized and the amount of shifting applied to reach the normalized format is saved in the variable *Diff*, which is finally used to adjust the exponent of the result saved in E1 to the correct value. To implement division, the divisor's exponent is subtracted from the dividend's exponent instead of adding the exponents in line 3, and the multiplication in line 4 is replaced by a regular division. The complexity of the FP multiplication is also $O(m^2)$ as shown in

$$\begin{aligned} T_{\text{FPMul}} &= T_{\text{AddExp}} + T_{\text{MultMant}} + T_{\text{Norm}} + T_{\text{Denorm}} \\ &\quad + T_{\text{AddSubExp}} \\ &= 2 * O(e) + O\Big(m^2\Big) + 2 * O\Big(m + m^2 + m * 2^e\Big) \\ &= O\Big(m^2\Big). \qquad (2) \end{aligned}$$

IV. EVALUATION

A. Experimental Setup

The correctness of the proposed implementations was validated using a functional simulator in MATLAB. The truth tables in Table II are implemented as LUTs which are then used to simulate the execution of Algorithm 1 on the AP in software. The latency and energy consumption values are estimated using an SRAM-based AP simulator modeling the performance of the operations on a word-parallel AP using a 16 nm technology [8], based on values for energy consumption of write and compare operations provided by [9], including

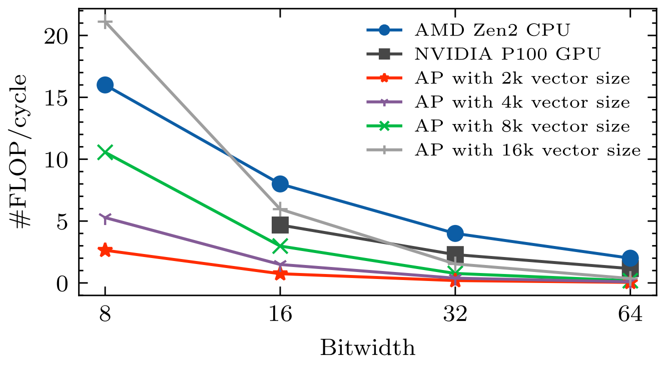


Fig. 3. Number of FP additions per cycle for different bit widths and memory sizes.

the data transfer overhead. The simulator uses the operation's trace recorded offline to estimate the energy and latency of executing the operation on the AP running at 1GHz. The traces used for this evaluation reflect the worst-case scenarios which include for example the need to use every bits to compare the two numbers, to always switch the two operands, and to shift the maximum amount for normalization in all memory entries.

B. Results

Fig. 3 demonstrates the number of operations executed per cycle for both the CPU, GPU, and the AP with different configurations. For fair comparison, we are assuming a single-core AMD Zen 2 CPU with an 1×128 b FPU and a single Nvidia P100 GPU core. For small bit widths and a large vector size, the AP is able to outperform the CPU. For an 8-bit input, the AP with 16 K rows can compute 1.32× more FLOPs/cycle than the CPU. However, as the bit width increases, the AP's performance decreases but still performs better than the GPU for larger memory sizes. At 16-bit width, the AP with 16 K vector size performs around 1.3× more FLOPs per cycle than the GPU core. It is safe to conclude that for highly parallelizable computations with lower bit precision, the AP can run faster computations than the traditional CPU and even the GPU. This is due to the AP's ability to run the operations in word-parallel bit-serial mode on all the entries in one shot.

In terms of energy, Fig. 4 shows its variation with respect to the size of the data for the AP, the Intel Xeon 5670 CPU and the NVIDIA K20c for 64-bit FP addition. We rely on the CPU energy consumption values provided by [10] per data transfer from DRAM and FP addition and multiply that by AP vector size. For the GPU energy values, we estimate the power consumption for data transfer per byte and multiply that by 256 then add it to energy consumed for fp64 operation then multiply that by the vector size. The plot shows that the AP consumed less energy than the CPU and GPU for all visualized memory sizes. For a memory size of 4096 rows, the AP is able to consume 1.72× and 21.4× less energy than the CPU and GPU, respectively. These savings are mainly due to the fact that data transfer is highly reduced.

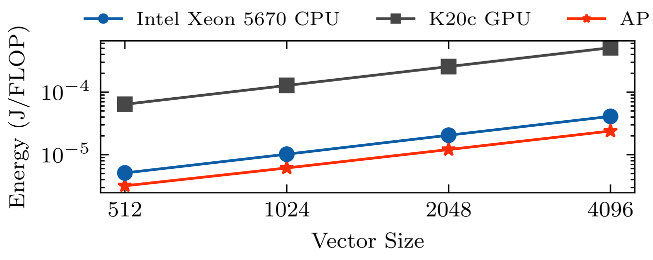


Fig. 4. Energy consumption (in J/FLOP) for executing FP addition on the AP, CPU, and GPU for different vector sizes.

V. Conclusion

In this letter, we propose a novel approach for implementing FP operations on the AP by designing a supporting format and LUTs. We propose algorithms for the FP addition, subtraction, multiplication, and division tailored for the capabilities of the AP. Results show that our implementation enables faster and more energy-efficient highly parallelizable and low-precision FP computations than the CPU and GPU.

Acknowledgment

The views, opinions, and/or findings contained in this letter are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the U.S. NSF.

References

[1] C. C. Foster, *Content Addressable Parallel Processors*. New York, NY, USA: Van Nostrand Reinhold Co., 1982.

[2] I. D. Scherson, D. A. Kramer, and B. D. Alleyne, "Bit-parallel arithmetic in a massively-parallel associative processor," *IEEE Trans. Comput.*, vol. 41, no. 10, pp. 1201–1210, Oct. 1992.

[3] H. Caminal, Y. Chronis, T. Wu, J. M. Patel, and J. F. Martínez, "Accelerating database analytic query workloads using an associative processor," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, 2022, pp. 623–637. [Online]. Available: https://doi.org/10.1145/3470496.3527435

[4] H. E. Yantır, A. M. Eltawil, and K. N. Salama, "IMCA: An efficient in-memory convolution accelerator," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 3, pp. 447–460, Mar. 2021.

[5] J. Silveira and L. Wanner, "Design and evaluation of associative processing kernels," in *Proc. IEEE 33rd Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, 2021, pp. 64–73.

[6] C. Golden, D. Ilan, C. Huang, N. Zhang, Z. Zhang, and C. Batten, "Supporting a virtual vector instruction set on a commercial compute-in-SRAM accelerator," *IEEE Comput. Archit. Lett.*, vol. 23, no. 1, pp. 29–32, Jan.–Jun. 2024.

[7] Y. Zha and J. Li, "Hyper-ap: Enhancing associative processing through a full-stack optimization," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2020, pp. 846–859.

[8] M. Rakka, M. E. Fouda, R. Kanj, A. Eltawil, and F. J. Kurdahi, "Design exploration of sensing techniques in 2T-2R resistive ternary CAMs," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 68, no. 2, pp. 762–766, Feb. 2021.

[9] H. E. Yantır, A. M. Eltawil, and F. J. Kurdahi, "A two-dimensional associative processor," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 9, pp. 1659–1670, Sep. 2018.

[10] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Characterizing the energy consumption of data transfers and arithmetic operations on x86-64 processors," in *Proc. Int. Conf. Green Comput.*, 2010, pp. 123–133.