FloatAP: Supporting High-Performance Floating-Point Arithmetic in Associative Processors

Kailin Yang
Computer Systems Laboratory
*Cornell University**
Ithaca, NY, USA
ky362@cornell.edu

José F. Martínez
Computer Systems Laboratory
Cornell University
Ithaca, NY, USA
martinez@cornell.edu

***Abstract*—Associative Processors (AP) enable in-situ, data-parallel computation in content-addressable memories (CAM). In particular, arithmetic operations are accomplished via bit-serial sequences of bulk search and update primitives. The data-parallel nature of AP-based computation, whereby thousands of operations can take place concurrently, more than makes up for the slowness of its bit-serial arithmetic; as a result, APs can deliver very high performance. Indeed, recent AP proposals have shown promising results across various application domains, primarily for integer codes. However, floating-point (FP) support is crucial for many critical AI/ML and scientific workloads.**

This paper introduces FloatAP, the first highly programmable AP architecture tailored for high-performance FP arithmetic. We show that the straightforward application of AP-based bit-serial arithmetic to FP leads to insufficient computational throughput. With FloatAP, we propose a bit- and component-parallel vector mantissa alignment method that significantly improves the throughput of vector FP addition and reduction. We also repurpose the existing adder trees in modern APs to efficiently implement multiplication and dot product operations, markedly outperforming AP's traditional bit-serial multiplication.

FloatAP is flexible enough to accommodate different FP formats while maintaining high utilization and integer compatibility. It is programmable using RISC-V 'V' vector extension and easily accommodates domain-specific languages and ML frameworks. We evaluate FloatAP using contemporary ML workloads, including CNN-, RNN-, and LLM-based parallel codes. The results show that, on average, FloatAP achieves $2.7\times$ higher inference throughput compared to an area-equivalent NVIDIA A100 Tensor Core GPU for single-precision FP ($1.5\times$ for bfloat16) and $2.4\times$ higher energy efficiency ($1.6\times$ for bfloat16).

***Index Terms*—floating-point, processing-in-memory, associative processor, content-addressable memory**

I. Introduction

Associative Processing (AP) is a decades-old processing-in-memory (PIM) paradigm [16], [37], [38] that utilizes sequences of parallel search and bulk update primitives on content-addressable memory (CAM) to implement data-parallel in-situ computation. Recently, AP has regained attention from researchers thanks to today's prevailing parallel applications, the need to address the von Neumann bottleneck, and advances in CAM bit-cell technologies [11], [12], [17], [19], [21], [25], [26], [29], [35], [36], [50]–[55]. AP architectures have achieved promising results for integer-based general-purpose codes [12], [50], [55] and for multiple domains such as databases [11], deep learning [19], and genomics [25], [29].

Nevertheless, prior AP designs lack efficient support for Floating-Point (FP) arithmetic, a crucial component that underpins contemporary AI/ML and scientific applications. FP workloads typically consume massive data and exhibit abundant data-level parallelism (DLP), demanding high computational throughput. Unfortunately, as we will show, the straightforward extension of AP to support FP arithmetic leads to insufficient computational throughput due to the bit-serial nature of AP operations, most notably multiplication, which takes more than 4,000 cycles for single-precision values [19].

In this work, we aim to design the first highly programmable AP architecture tailored for high-performance FP arithmetic. We exploit three key properties of modern APs to fully unleash the computing power. First, AP excels in parallel search operations. This capability allows efficient searching of values across exponent fields in FP vectors, which is a critical step in FP mantissa alignment. Second, some contemporary AP designs feature a bit-sliced data layout [11], [12], [35], [50], where each bit of an operand is stored in a different memory subarray. This design enables bit-parallel operations across subarrays (e.g., in integer-based APs, it enables bit-parallel logical operations) and, as we will show, allows concurrency between the computation in mantissa and exponent fields. Third, APs often include a reduction tree alongside the CAM array to calculate the match count of a search [12], [51], [54], which we hypothesize can be repurposed for FP multiplications.

Inspired by these observations, we introduce FloatAP. This novel high-performance AP architecture capitalizes on recent AP designs [12], [35], [50] based on compact 6T push-rule SRAM cells and compatible with RISC-V standard vector extensions. We propose a bit-parallel, component-parallel vector mantissa alignment method that significantly improves the throughput of vector FP addition and reduction. We also deconstruct AP multiplication/dot-product operations, carefully mapping different stages to the AP's CAM and its reduction tree to attain high performance. FloatAP can accommodate

This work is supported by the Semiconductor Research Corporation (SRC) through the Center for Research on Intelligent Storage and Processing-in-memory (CRISP) and ACE Center for Evolvable Computing (ACE).

*Kailin Yang now works at ByteDance Ltd. in San José, CA, USA.

different FP formats while maintaining high utilization and integer compatibility. Moreover, FloatAP remains highly programmable using RISC-V 'V' vector extensions and easily integrates with domain-specific languages and ML frameworks.

We evaluate FloatAP using contemporary ML workloads, including CNN-, RNN-, and LLM-based parallel codes. The results show that, on average, FloatAP achieves 2.7× higher inference throughput compared to an area-equivalent NVIDIA A100 Tensor Core GPU for single-precision FP (1.5× for bfloat16) and 2.4× higher energy efficiency (1.6× for bfloat16).

The contributions of this paper are as follows:

- To the best of our knowledge, this is the first published work that discusses associative FP arithmetic in detail, including how to realize FP vector operations using CAM arrays, including addition, maximum, reduction, multiplication, dot-product, and conversion between two's complement and signed format (needed to support IEEE 754). Our approach does not rely on precalculated LuTs, approximation, or quantization.
- We leverage existing bit-sliced AP designs to realize fast mantissa alignment, which helps reduce the time complexity of FP add and reduction from $O(m^2)$ to $O(m)$, where m is the mantissa bit width.
- We design novel AP multiplication and dot-product operations, which utilize the CAM and reduction tree collaboratively to deliver high performance.
- We propose architectural support to allow FloatAP to accommodate different FP formats while maintaining high utilization. We also show mechanisms to enable concurrency between exponent and mantissa fields.

II. Background and Motivation

A. Floating-Point Basics

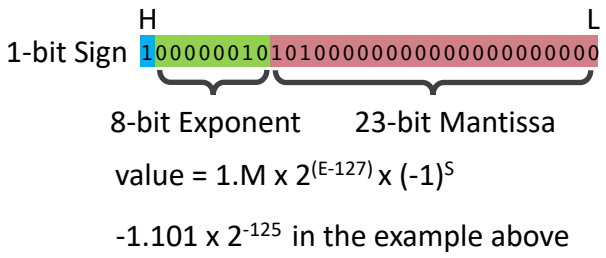


Fig. 1. An illustrative example of an IEEE754 [42] 32-bit single-precision floating-point number.

Floating-point representation (FP) is a high-precision, high-range data type used to encode real numbers in a computer system. The absolute value of an FP number equals a variable fractional number called *mantissa* multiplied by a power-of-two scaling factor. The index of the scaling factor is given by an integer *exponent*. The number's *sign* is typically represented by a separate bit. Figure 1 shows an illustrative example of a single-precision (32-bit) FP number using IEEE 754 encoding [42][1]. From low- to high-order bits, it consists of 23 bits of mantissa, 8 bits of exponent, and the sign bit. Conventionally, the mantissa only physically stores the fractional part to the right of the radix point; the 0/1 value to the left of the radix point always equals '1' and is implicit. Therefore, the 23-bit mantissa actually represents 24 bits. In Figure 1, the mantissa field is '101', so the mantissa equals to 1.101 in base 2. Note that the mantissa itself is positive and not represented using two's complement.

Value	Description	Representation
Normal Value	$1.M\times2^{E-127}\times(-1)^S$	0<E<255, M ≠ 0
±∞	Infinity, e.g., -7÷0, ∞+5	E = 255, M = 0
NaN	Not a number, e.g., 0÷0, ∞×0	E = 255, M ≠ 0
±0	Zero with sign	E = 0, M = 0
Denormal Value	$0.M\times2^{-126}\times(-1)^S$	E = 0, M ≠ 0

TABLE I: IEEE 754 single-precision FP values and representation.

The 8-bit exponent field is an unsigned integer; the actual index value equals the exponent minus an implicit constant bias. The bias equals (2^e-1) if the exponent field has e bits (e.g., 127 for an 8-bit exponent). In this way, the unsigned exponent field can represent a symmetric range of signed integers centered at zero, making it possible to scale mantissa to both very large and close-to-zero values. Meanwhile, the unsigned format makes the comparison between two values easier than the two's complement. In Figure 1, the exponent field is '10', so the index equals -125, and the exponentiation factor of this number is 2^{-125}. The value of the sample equals -1.101×2^{-125}.

Importantly, when the exponent field is all zeros and the mantissa is not zero, the implicit value to the left of the radix point is assumed to be 0 instead of 1. In addition, the scaling factor will equal $2^{1-\text{bias}}$ instead of $2^{-\text{bias}}$. Those FP numbers are called *denormalized* or *subnormal* values. The purpose is to be able to represent very small values. Moreover, IEEE 754 standard reserves certain encodings for special values, including ±0, ±∞, NaN (not-a-number). Table I summarizes all the possible value types and their representation. Specifically, the division by zero of a non-zero value will lead to infinity, whereas mathematically undefined behavior, e.g., zero divided by zero, will result in a Not-a-Number (NaN). The special values can be propagated quietly through arithmetic operations, although some systems and standards may choose to signal an exception for certain types of NaNs, for example.

FP multiplication involves adding the exponents, multiplying the mantissas, and XORing the sign bits. FP addition requires two steps in series: before adding the mantissas, the operand with the smaller exponent must first right-shift its mantissa and increment its exponent as many times as needed to equalize both exponents, after which the mantissas can be added. As a result of this normalization, some precision may

[1]Without loss of generality, in this paper, we use 32-bit single-precision IEEE 754 FP encoding in all the examples unless otherwise specified.

be lost.

B. Associative Processor Architecture

Associative Processors (APs) [11], [12], [16], [17], [19], [21], [26], [27], [36]–[38], [51]–[55] realize in-situ data-parallel computation using the parallel *search* and bulk-wise *update* abilities of content-addressable memories (CAMs). Specifically, a search operation can search an n-bit pattern across all the rows in n columns.[2] At each row, a *tag bit* is set to '1' when the entire n-bit region in that row matches the searched pattern. Some bits in the search key can be masked. Similar to search, an update operation can write an n-bit pattern to n columns across all the rows in parallel, or selectively overwrite the rows whose tag bits are set. Both search and update are parallel and take a fixed latency (typically one cycle) regardless of row count and pattern width. Therefore, AP architectures exhibit orders of magnitude higher DLP than area-equivalent traditional compute engines.

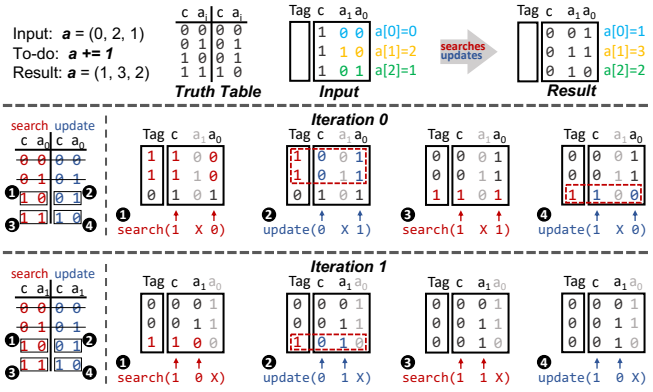


Fig. 2. Using an AP to increase the components of a vector by one. In the example, ***a*** is a vector with three 2-bit elements. ***c*** is a metadata column reserved for the carry bit, which is initialized to one in this example. Some entries in the truth table are stroke through because their input and output patterns are exactly the same.

APs can implement vector arithmetic/logic instructions using sequences of searches and updates. Fig. 2 provides a logical example of how to realize a vector increment-by-one instruction. As the top region of Fig. 2 shows, assuming the input vector ***a*** consists of 2-bit elements, each element is stored in a different row of the CAM array. Each column stores one bit of the elements, from the most (left) to the least (right) significant ones.

AP adopts a bit-serial approach to realize the increment-by-one. First, it initializes a carry bit column ***c*** to '1.' Then, it adds ***c*** to the least significant bit of ***a***, and stores the newly generated carry in ***c***. Afterward, it moves to the next bit of ***a*** and repeats the same process. Thus, for n-bit wide data type, this bit-serial increment function takes n iterations.

Within each iteration, the AP implements ***a+=c*** using the search and update operations of the CAM array. To do so, it generates search-update pairs based on the truth table (TT) of the half-adder. For example, in iteration 0 in Fig. 2, the AP first searches the first input pattern '10' in the TT across ***c*** and $\boldsymbol{a}_0$ (❶). The tag bits record the search result. Then, for the matched rows, the AP updates the output pattern '01' in the corresponding columns(❷). The AP repeats this search-update steps for every entry in the TT, and skips the entries where the input and output patterns are exactly the same (the rows being stricken through in Fig.2). In this way, it performs the one-bit half addition without reading out the data or using ALUs. Since the TT in this example has four effective entries, every iteration takes four operations. The increment-by-one function on AP takes $O(4n)$ operations, where n is the bit-width of the input vector's data type. Nevertheless, because CAM array can finish a search or update operation within one cycle no matter how many rows the array has, the AP can mitigate the long latency using tremendous DLP, as in reality the CAM array can have tens of thousands of rows and each row stores a different word.[3]

Using the above method, AP can realize any desirable functionality according to its TT, and any function is essentially a microprogram of search/update operations, which is called *Associative Algorithm (AA)*. Recent AP designs such as CAPE and PUMICE [11], [12], [35], [50] microprogram AAs into the vector instructions from the standard RISC-V 'V' vector extension [14]. As a result, AP becomes a general-purpose and programmable (very long) vector architecture that hides the search/update operations from the programmer's view. Fig. 3(a) shows its block diagram. The key component is the CAM-based Compute-Storage Block (CSB), which functions as a unified vector register file and ALU tightly coupled to a Control Processor (CP). The CSB implements all the element-wise vector-vector operations using search/updates. In addition, APs often include an adder tree alongside the CSB to calculate vector reduction sum [12], [51], [54] (the triangle in Fig. 3(a)). The CP is an in-order RISC-V core that fetches and decodes instructions from the program, executes scalar and control flow instructions itself, and sends vector instructions to the CSB to execute. To drive the CSB, the CP utilizes a Vector Control Unit (VCU), which stores the AAs of all the supported vector instructions in a tiny read-only AA storage. Upon receiving a vector instruction, the VCU loads the corresponding AA into a buffer and generates search/update signals via a sequencer to drive the CSB to execute the AA. There is no cache hierarchy between CSB and DRAM, and the CSB also does not function as a cache. Instead, the programmer can access CSB using vector names (e.g., `v0`), and transfer data between CSB and DRAM using vector load/store (e.g., `vld v0, 0xAB`). A Vector Memory Unit (VMU) interfaces the CSB and DRAM. Additionally, PUMICE [50] incorporates a small load-store buffer for the CSB and enables decoupled vector arithmetic and load/store operations on AP.

[2]Note that in this paper, the "column" and "row" are logical views for readability and consistency. Physically, a "column" may correspond to either a bitline or wordline, depending on the circuit-level implementation of the CAM.

[3]The array here is also a logical concept. Physically, it is typically implemented using thousands of small subarrays.

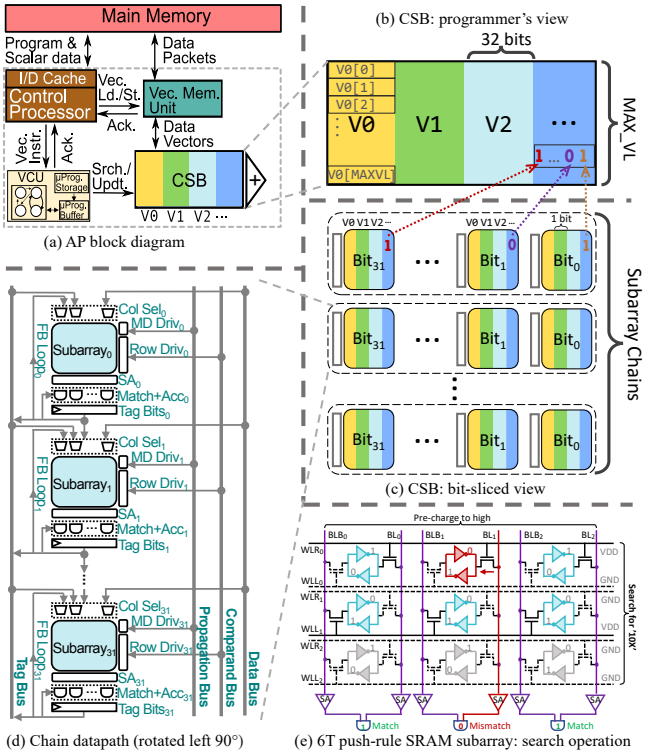


Fig. 3. The top-down view of bit-sliced AP architecture following CAPE's [12] and PUMICE's [50] framework (a). The key component is a compute-storage block (CSB) built from CAM arrays. CSB serves as a unified vector register and ALU from the programmer's view (b) and has mega-bytes of capacity. Physically, it consists of multiple 32-subarray chains (c). Each chain adopts a bit-sliced data layout and has peripheral logic for search/update and buses to support carry propagation and command/data distribution (d). Every subarray is a 36-by-32 6T push-rule SRAM array [31] (e), which can search a vertical pattern along the direction of the wordlines. So the physical orientation of the chain is rotated left 90°from the logical view.

Bit-Sliced Associative Processing – Many arithmetic operations are carried out bit-serially in APs; still, a bit-sliced data layout across memory subarrays [12], [20], [35], [45] (i.e., store different bits of a word in different subarrays) makes steps involving same-order bits of multiple operands local (e.g., each step of a carry-propagate addition). It also enables bit-parallel operations when appropriate (e.g., bitwise OR). Fig. 3(b)(c) shows how CAPE architecture implements its CSB using numerous physical subarrays. In the CSB, every 32 subarrays form a *chain*. Every 32-bit integer word is sliced and scattered across the subarrays in a chain. The bits from the same word are stored at exactly the same row and column in the 32 subarrays in the same chain (the arrowed example in Fig. 3(b) and (c)). From left to right, each subarray in a chain stores the most to the least significant bit only. And the k^{th} column of any subarray always store the bit from the vector register $\mathrm{V}k$. Every subarray also reserves several metadata columns [12] for intermediate results generated within the lifecycle of a vector instruction (e.g., carry). They are invisible to the ISA and programmer. Different elements from the same vector are stored on different rows. Thus, the maximum vector length of each vector register is determined by the row count of each chain and the total number of chains.

Bit slicing can largely reduce the latency of AP vector instructions because all subarrays in a chain can perform searches/updates in parallel. For logical operations that do not generate carry or carry-save addition, the CSB can execute them in a bit-parallel, word-parallel manner, $32\times$ faster for integer words. Nevertheless, for arithmetic operations, the carry generated by subarray i has to be propagated to subarray $i+1$. Fig. 3(d) shows the datapath of a chain, where neighbor subarrays are connected to support carry propagation. Additionally, a tag bus allows any subarray to send its tag bits to all the other subarrays.

Note that the physical view of a chain in Fig. 3(d) is rotated 90°from the logical view in (b) because CAPE implements its subarrays using dense 6T push-rule SRAM bitcells [31] and a logical column corresponds to a physical wordline in this type of CAM. Fig. 3(e) depicts how a subarray performs a search.

C. Limitations and Motivation

Existing APs lack efficient support for FP arithmetic. While integer APs can handle FP multiplication algorithmically, the throughput is limited by the bit-serial nature of the operations– the best existing approach takes more than 4,000 cycles [19]. Moreover, to our knowledge, prior research does not discuss FP addition via AP. Existing FP addition algorithms in other digital PIM architectures require $O(m^2)$ time complexity [18], [28], where m is the mantissa bit width. For those architectures and AP, at every cycle, the memory array has to perform exactly the same operation to all the active elements. Yet FP addition involves a data-dependent mantissa alignment process (see Sec. II-A). Thus, those algorithms must complete one FP vector-vector addition in m iterations. During each iteration, only operand pairs with the same exponent differences are added. Since bit-serial add already takes $O(m)$, the overall complexity becomes quadratic.

Nevertheless, as we will show shortly, the bit-sliced AP shows great potential to overcome the above limits. First, the AP could be made to efficiently search for values across exponent fields in FP vectors, which is a critical step in FP mantissa alignment. Second, the mantissa alignment does not involve inter-bit data dependencies, and thus can be potentially realized in a bit-parallel manner, reducing the FP add complexity to linear. Third, the adder tree alongside the CSB, previously reserved for reduction sum, can be repurposed for multiplication calculation. Fourth, the mantissa and exponent are stored in different subarrays in a bit-sliced data layout, so AP can potentially overlap the operations on the two fields. Motivated by the above observations, we introduce FloatAP next–a novel high-performance AP architecture that capitalizes on recent bit-sliced AP designs [12], [35], [50]. FloatAP co-designs the associative algorithm and the AP architecture. We discuss them in Sec. III and IV respectively. Then we show our evaluation methodology and results in Sec. V and VI.

III. FloatAP Associative Algorithms

In this section, we introduce FP associative algorithms (AAs) that can fully harness the bit-sliced subarray chain to maximize performance. We use m and e to denote the bit width of the mantissa and exponent fields, respectively. We assume that, within each chain, the sign is stored in the most significant subarray, the next e subarrays store the exponent, and the last m subarrays store the mantissa. First, we introduce an efficient AA that can align the mantissa field of an FP vector based on the exponent in $O(m)$ cycles. Then, we discuss how FloatAP implements reduction sum and addition using this AA, which results in m times fewer cycles than the existing approach [18], [28]. Next, we show a hybrid AA that realizes multiplication within $O(m)$ cycles by a shift-and-accumulate approach. It performs in-situ shifting via search/update and then calculates the accumulation with the adder tree. Finally, by combining the insights of FloatAP mantissa alignment and hybrid multiplication, we introduce a fast dot-product that provides $6\times$ higher throughput than back-to-back multiplication and reduction.

A. Mantissa Alignment

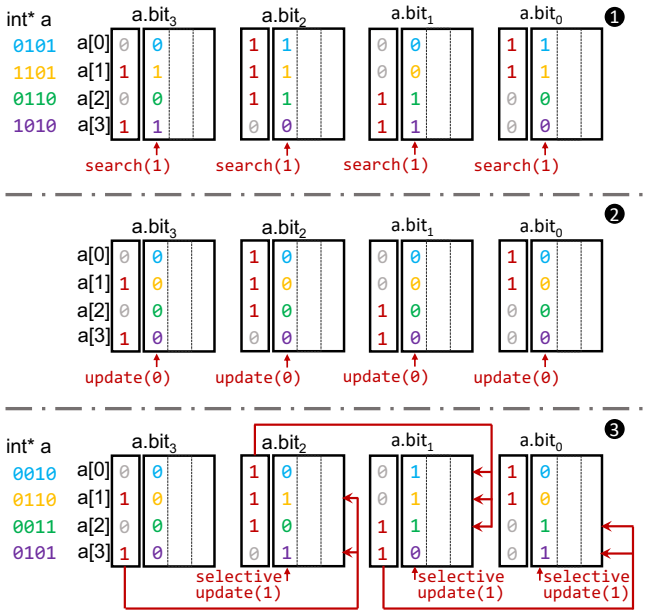


Fig. 4. A word-parallel, bit-parallel, one-bit right shift (*1brsh*) takes three cycles. The four subarrays store bit3 to bit0 from left to right, respectively.

One-Bit Shift (*1bsh*) – FloatAP implements mantissa alignment using one-bit right shift (*1brsh*) operations. We propose a bit-parallel *1brsh*, illustrated in Fig. 4, which completes in three cycles. In Fig. 4, ***a*** is a mantissa vector with four elements (each element in a different color), and each element has four bits sliced across four subarrays from left to right. When stored in this way, *1brsh* is essentially moving the data in a particular column to the same column in the right neighboring subarray. To do so, all subarrays first simultaneously search for 1 in the column where ***a*** is stored. After this step, if a particular cell in the column stores 1, the corresponding row in the tag is marked as 1. Essentially, this search copies the target column into the tag column. In the second cycle, all subarrays overwrite the original value in the target column as 0 using a bulk update. Last, the subarrays use the left neighbor's tag bits as masks to selectively update 1 to the target column (shown as the red path in Fig. 4). Only the rows marked as 1 by the left neighbor's tag bits will be updated as 1, and the remaining rows will stay 0. Finally, the *1brsh* of vector ***a*** completes. We also implement *1blsh* in the same fashion, which is necessary for our hybrid multiplication AA. Using *1brsh* and *1blsh*, FloatAP can realize any n-bit shift one hop after another in $3n$ cycles. Note that the example above shows a destructive implementation–i.e., the original value is overwritten. FloatAP also supports a non-destructive version with the same latency by allocating either a new vector register or metadata column for the result. Additionally, rotation and bit-width upgrading are both supported with the help of the tag bus in Fig. 3(d). Using the left shift with bit-width upgrading as an example. The overflowed most significant bit will be stored in a metadata column (invisible to the programmer) in the rightmost subarray. This twist over *1blsh* does not impact cycle count. Specifically, in cycle two, the rightmost subarray will update 0 to both target and metadata columns. In cycle three, while other subarrays selectively update the target column based on the right neighbor's tag, the rightmost subarray updates the metadata column based on the leftmost subarray's tag, transmitted by the tag bus. Moreover, for vectors in two's complement representation, FloatAP also supports the arithmetic right shift. The only difference is that, in cycle 2, the leftmost subarray selectively updates 1 to its target column based on its own tag bits instead of unconditionally updating 0. Similarly, for the implicit-one representation in Fig. 1, the leftmost subarray in cycle two unconditionally updates 1 instead of 0.

Mantissa Alignment (*align*) – To align the mantissas of an FP vector, we right-shift each element's mantissa so that its exponent matches the maximum exponent value in the vector, $E_{\max}$. Specifically, an element with an exponent E right-shifts its mantissa by $E_{\max} - E$ bits as shown below:

$$2^E \cdot M = 2^{E_{\max}} \cdot M \cdot 2^{E-E_{\max}} = 2^{E_{\max}} \cdot M >> (E_{\max} - E)$$

Because the mantissa bit-width is m, if E is less than $E_{\max} - m + 1$, the mantissa will become zero after shifting. Thus, the *align* algorithm focuses on elements with the top m largest exponent values, from $E_{\max} - m + 1$ to $E_{\max}$. Accordingly, *align* takes m iterations. In iteration i, the CSB searches for the elements whose exponents fall into the range of $(E_{\max} - m, E_{\max} - m + i]$, and right-shifts their mantissa by *one* bit using *1brsh*. In this way, those whose exponents are equal to E will only appear in the search results of the last $E_{\max} - E$ iterations. And the CSB will right-shift their mantissa by exactly $E_{\max} - E$ bits. FloatAP relies on the existing OR gate at each tag bit [12], [55] to complete the range search in each iteration within one cycle. Initially, it searches $E_{\max} - m + 1$ across the exponent field and saves the result in the tag bits of

the exponent subarray. From the second iteration, it searches $E_{\max}-m+i$, combines the search result with the saved tag bits from the previous iteration using the OR gates, and saves the new results. Therefore, the tag bits of the exponent subarray keep the accumulative results of all the searches in the previous iterations, which is exactly $(E_{\max}-m, E_{\max}-m+i]$. At last, the tag bits mark all the elements with an exponent in $(E_{\max}-m, E_{\max}]$. We can then utilize them to update all the unselected rows' mantissa to zero as their exponents are smaller than $E_{\max}-m+1$. Using our approach, each iteration takes 4 cycles (1 for search and 3 for *1brsh*). And the *align* takes $(4m+4)$ cycles in total. Note that searching for specific values across a vector's exponent field requires all the exponent bits stored in the same subarray, which contradicts bit-slicing. We address this challenge in IV-A. Also, before *align*, the AP must first find $E_{\max}$. This vector maximum operation involves searching for 1 from the most to the least significant bit, which finishes in e cycles for e-bit wide exponent field.

B. Addition and Reduction Sum

With the help of the *align*, FloatAP can efficiently implement addition instructions. Nevertheless, FP mantissa uses a sign bit and has an implicit one, whereas integers adopt two's complement representation, so FloatAP first needs to convert mantissas into integers before performing addition.

Mantissa-Integer Conversion – The subarrays first perform a *1brsh* to all the mantissa and set the most significant bit to 1 (the implicit one). The least significant bit is temporarily rotated to a metadata column in the most significant subarray. In the meantime, all the exponents are increased by one due to the right shift of the mantissa. Next, all negative mantissa are flipped and increased by one from the least significant bit. At this point, all mantissa are in integer. The rotated least significant bit can now be dropped so that the width of the mantissa remains as m. In total, the conversion takes **(4e+4m+7)** cycles. Note that in practice, we convert mantissa to integer as soon as a vector is loaded into CSB, and they remain in integer form until stored back in the main memory so that the conversion overhead is minimized.

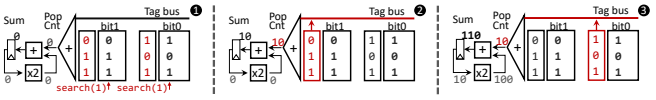


Fig. 5. Mantissa reduction sum using the bit-serial adder tree proposed by CAPE [12] takes $O(m)$ cycles. The input vector in this example is (1,2,3) and adopts a bit-sliced layout (2-bit wide). The sum equals 6.

Reduction Sum (*vfredsum.vs*) – FloatAP utilizes the bit-serial integer adder tree proposed by CAPE [12] to calculate the reduction sum of the mantissa. Fig. 5 shows how it works under the bit-sliced design. First, all the mantissa subarrays simultaneously search 1 across the column where the mantissa vector is stored. This will copy the vector operand into the tag bits (❶). Then, at every cycle, one of the subarrays sends its tag bits to the tree via the tag bus (❷ to ❸). This happens from the leftmost subarray (most significant) to the right (least significant). Every time, the tree will reduce the tag bits into an integer population count (popcnt). Meanwhile, the sum from the previous cycle is multiplied by 2 and accumulated with the new popcnt. This process is pipelined so after exactly m cycles, we get the reduction sum of the mantissa vector. And the result's exponent equals the $E_{\max}$ found during *align* algorithm. In total, the FP *vfredsum* takes **(e+5m+9)** cycles, including a 5-cycle adder tree latency when the vector length is large.

```
# to-do: v0[i] = v1[i] + v2[i]
def vfadd.vv(v0, v1, v2):
    # step1: calculate exponent difference
    ΔE = vsub.vv(v1.Exp, v2.Exp)

    # step2: swap some operand pairs so that v2[i]
    # always have a smaller exponent than v1[i]
    for i in range(n):
        if ΔE[i] < 0:
            swap(v1[i], v2[i])

    # step3: align v2's mantissa based on ΔE
    ΔE = |ΔE|
    align(v2.Man, ΔE)

    # step4: add the mantissa and copy v1's exponent
    vadd.vv(v0.Man, v1.Man, v2.Man)
    v0.Exp = v1.Exp
```

Listing 1. Pseudocode of FloatAP's *vfadd.vv* algorithm inspired by [18], [28]. With bit-sliced data layout and our *align* algorithm, FloatAP achieves linear time complexity, whereas in both [18], [28] *vfadd* requires quadratic complexity.

Vector-Vector Addition (*vfadd.vv*) – Other than *vfredsum*, FloatAP also supports element-wise *vfadd.vv*. By exploiting the bit-sliced data layout and the *align* algorithm, FloatAP significantly optimizes the latency of FP *vfadd* of other digital PIM architectures [18], [28]. As shown in Listing 1, *vfadd* follows four steps. First, it calculates the difference between the exponent fields of the two input vectors. Next, it swaps every pair of $\boldsymbol{v1}[i]$ and $\boldsymbol{v2}[i]$ if $\boldsymbol{v1}[i]$ has smaller exponent. After this step, $\boldsymbol{v1}$ always stores the values with the larger exponent in every pair of $\boldsymbol{v1}[i]$ and $\boldsymbol{v2}[i]$. FloatAP can efficiently achieve this step using two `vmerge.vv` operations that are already supported by CAPE [12]. In step 3, *align* is performed to $\boldsymbol{v2}$'s mantissas so that every pair of $\boldsymbol{v2}[i]$ and $\boldsymbol{v1}[i]$ have the same exponent. Unlike in *vfredsum*, the *align* in *vfadd* uses the $\boldsymbol{\Delta E}$ calculated in step1 as the reference exponent vector instead of the exponent of $\boldsymbol{v2}$ itself. Finally, in step 4, the result's mantissa vector is calculated by adding $\boldsymbol{v1}$ and $\boldsymbol{v2}$'s mantissa using integer *vadd*. The result's exponent equals $\boldsymbol{v1}$'s exponent.

FloatAP *vfadd* is extremely efficient ($O(e+m)$) because all four steps only require linear or constant time complexity. In comparison, other digital PIM architectures can only achieve $O(e+m^2)$ [18], [28] because they adopt a contiguous data layout and, therefore, cannot support bit-parallel *align*. Thus, their *vfadd* has to iterate step 4 by m times, resulting in quadratic time complexity. In addition, we also optimize the original AP integer *vadd* to further accelerate steps 1 and 4. The original *vadd* does not take advantage of bit-slicing and

takes $8n$ cycles [11], [12]. In FloatAP, we first calculate carry-save addition in a fully bit-parallel manner and then perform a bit-serial carry propagation similar to Fig 2. This optimization largely reduces the *vadd* time complexity to $4n+2$ cycles. As a result, FloatAP *vfadd* in total costs **(4e+9m+16)** cycles.

C. Multiplication and Dot-Product

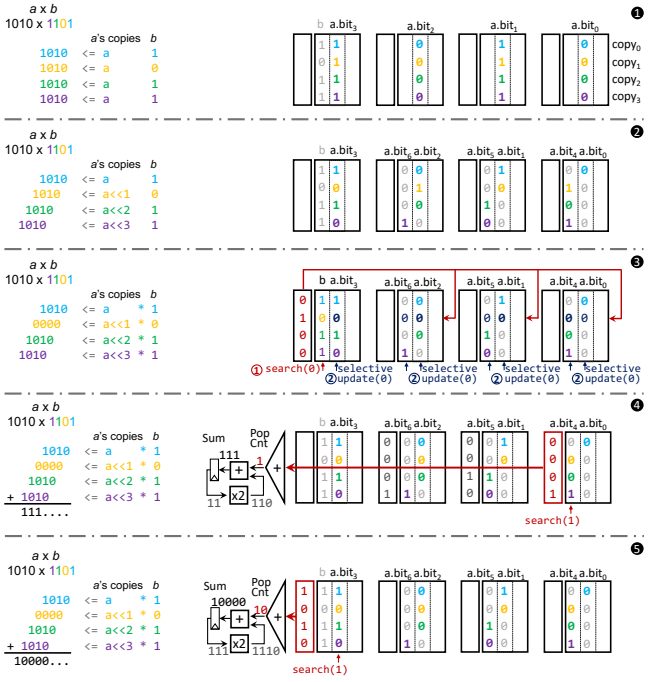


Fig. 6. A propagation chain can implement *vfmul.vv* with $O(m)$ time complexity by utilizing the reduction tree.

FP multiplication contains two steps: exponent addition and mantissa multiplication. Due to the inevitable bit-serial carry propagation, the complexity of FP multiplication is at least $O(e + m^2)$. Furthermore, the inner loop of the associative algorithm involves multiple searches/updates, rendering even the most efficient design to approximately $6m^2$ cycles. Consequently, relying entirely on search/update for multiplication is inefficient. Instead, we observe that multiplication is essentially a shift-and-sum operation, as shown in Equation (1). Thus, FloatAP can utilize the reduction logic in Fig. 5 for multiplication and dot-product.

$$a \cdot b = \sum_{i=0}^{m-1} (a \cdot 2^i) \cdot b.bit_i \tag{1}$$

Vector-Vector Multiplication (*vfmul.vv*) – As Equation (1) shows, assuming a and b are a pair of scalar m-bit mantissa, their product is the sum of m copies of a, with each copy left-shifted by 0 to $(m-1)$ bits and multiplied by the corresponding bit of b. Accordingly, FloatAP executes the shift and 1-bit multiply using search/update and calculates the sum using the reduction tree. This hybrid *vfmul* combines the strength of both subarrays and ALUs and realizes linear time complexity and higher throughput with no additional hardware overhead as existing integer AP already equips every chain with the reduction logic (Fig. 5).

$$\begin{aligned} \boldsymbol{M_1} \cdot \boldsymbol{M_2} &= \sum_{i=0}^{vlen} \boldsymbol{M_1}[i] \cdot \boldsymbol{M_2}[i] \\ &= \sum_{i=0}^{vlen} \sum_{j=0}^{m-1} (\boldsymbol{M_1}[i] \cdot 2^j) \cdot \boldsymbol{M_2}[i].bit_j \\ &= \sum_{j=0}^{m-1} \sum_{i=0}^{vlen} (\boldsymbol{M_1}[i] \cdot 2^j) \cdot \boldsymbol{M_2}[i].bit_j \\ &= \sum_{j=0}^{m-1} (\sum_{i=0}^{vlen} \boldsymbol{M_1}[i] \cdot \boldsymbol{M_2}[i].bit_j) \cdot 2^j \end{aligned} \tag{2}$$

```
# input: v1, v2 are two FP vectors
# to-do: result = v1 · v2
def vfdot.vv(result, v1, v2):
    # step1: add the exponent fields of input
    vectors
    E_sum = vadd.vv(v1.Exp, v2.Exp)

    # step2: align v1's mantissa based on E_sum
    align(v1.Man, E_sum)

    # step3: calculate the dot-product of mantissa
    fields of input vectors based on Equation (2)
    r = 0
    for (j=m-1; j>=0; j--):
        v3 = vand.vv(v1.Man, v2.Man.bit_j)
        r += vredsum.vs(v3)
        r *= 2

    result.Man = r
    result.Exp = vmaxu.vx(E_sum) # Max value in E_sum
```

Listing 2. Pseudocode of calculating the dot-product of two FP vectors (vfdot.vv), which implements dot-product of mantissa vectors based on Equation (2).

Fig. 6 explains how our *vfmul* works. The example only shows one chain due to the page limit, which performs a *scalar* multiplication. In reality, the CSB has thousands of chains performing the operation in Fig. 6 simultaneously. In step 1, operand a is duplicated for m times after being loaded from memory and stored in m rows in a chain (❶). The load and replicate operation is already supported by the original integer AP [12]. Next, each copy is left-shifted by 0 to $m-1$ bits, respectively (❷). FloatAP implements this step similarly as *align* and only takes $3m$ cycles. Note that for the m-bit mantissa multiplication, $2m$ bits are needed for the shifted copies of a. We utilize a metadata column from each subarray to store the higher m bits. In this way, both bit i and $(i+m)$ are stored in subarray i as shown in Fig. 6 step ❷. After step 2, a's copies are shifted into the correct position. Before performing the reduction sum, we will multiply each copy with bit i of b, i.e., updating copy i to zero if the i^{th} bit of b is zero. As shown in ❸, we store b in the contiguous data-layout (the natural form in DRAM and cache) in a column in one of a subarray, instead of bit-slicing. This allows us to implement step three in only two cycles: one search and one

update. Finally, the multiply result is calculated in the same way as Fig. 5 for both the upper and lower m bits (❹ and ❺ in Fig. 6). In total, the latency of *vfmul* is only **(5m+4e+4)** cycles, including the time of adding the exponent field.

Since a needs to be duplicated and occupies m different rows in a chain, the next element in the input vector will occupy a different chain. Thus, the total vector length of the entire memory array is reduced by m times. Yet even in terms of throughput, our *vfmul* is still 20% better than the bit-serial approach in [12], as the latter spends more than $6m^2$ cycles for FP values. Moreover, when the physical vector length of the memory array is much longer than the vector length of the data structure, our *vfmul* can achieve more than $m\times$ throughput improvement.

Dot-Product (*vfdot.vv*) – Similar as *vfmul*, FloatAP can realize the *vfdot.vv*, the most critical operation for tensor-based applications, by simultaneously leveraging the associative memory and the adder tree. We do not adopt the straightforward implementation of *vfdot* with back-to-back *vfmul* and *vfredsum*. Instead, we exploit the commutativity of addition to reorder the sum in *vfredsum* and in *vfmul* based on Equation (2) when calculating the dot-product of the mantissa vectors, like in other tree-based dot-product proposals for integers [15], [49]. In this way, FloatAP does not need to replicate and shift the mantissa of $\boldsymbol{v1}$ like in *vfmul*. Instead, every element in $\boldsymbol{v1}$ and $\boldsymbol{v2}$ occupies a different row in a chain like in other vector operations such as *vfadd*. As a result, the *vfdot* can achieve much higher FLOPS than *vfmul*.

Listing 2 shows the pseudo-code of *vfdot*. Before calculating the dot-product of the mantissa fields, FloatAP first needs to add $\boldsymbol{v1}$ and $\boldsymbol{v2}$'s exponent vectors (line 5 in Listing 2), and align $\boldsymbol{M_1}$ based on this E_{sum} vector (line 8). Those two steps require $(5e+4m+6)$ cycles. Then, it calculates the dot-product of the mantissa fields in m iterations. In each iteration, FloatAP calculates $\sum_{i=0}^{\mathrm{vlen}} \boldsymbol{M_1}[i] * \boldsymbol{M_2}[i].\mathrm{bit}_j$ for a different j value. Specifically, the $\boldsymbol{M_1}[i] * \boldsymbol{M_2}[i].\mathrm{bit}_j$ in Equation (2) is realized by a vector AND operation (line 13). FloatAP can efficiently implement it using one search and two updates, similar to ❸ in Fig. 6. The inner sum $\sum_{i=0}^{\mathrm{vlen}}$ is then calculated using the same method as Fig. 5, taking m cycles (line 14 in Listing 2). As for the outer sum $\sum_{j=0}^{m-1}$ and the 2^j item, FloatAP amortizes them across the m iterations. In each iteration, it only needs to calculate a scalar add-accumulation (line 14) and multiply-accumulation (line 15) using the $\times 2$ and accumulate units at the root of the adder tree. The latency is completely hidden behind the *vredsum* operation. In addition, as Fig. 5 shows that, during *vredsum*, the chain does not perform any search/update in all the m cycles except for the first one. Therefore, FloatAP can overlap line 14 in iteration $(j-1)$ with line 13 in iteration j. Thus, the entire step 3 in Listing 2 only takes (m^2+6) cycles, including a five-cycle latency to fill the pipelined adder tree. In total, the *vfdot* operation takes **(5e+m^2+4m+12)** cycles. Compared to a back-to-back *vfmul* and *vfredsum*, our *vfdot* algorithm achieves more than $4\times$ throughput improvement.

D. Handling Special Values

FloatAP can efficiently handle all the special values listed in Table I. Since FloatAP can search a pattern across a vector in one cycle, it can quickly locate special values and handle them before performing computation on normal values. This entire handling process is done using parallel search/update, eliminating the need for additional exception-handling logic.

Specifically, the denormal values are handled during the mantissa-integer conversion discussed in Sec. III-B. FloatAP first searches all zeros across the exponent field to locate denormal values in a vector. These values are then masked out when resolving the hidden-1 for normal FP numbers. After that, both normal and denormal values' mantissas are converted to two's complement. Finally, FloatAP updates the exponent of denormals from zero to one in parallel so that they can be interpreted as normal values in future computations. The overall overhead is merely several cycles.

FloatAP supports both quiet and signaling modes for Infs and NaNs. Before an arithmetic operation, FloatAP first searches for and locates them. In signaling mode, it will directly trigger an invalid operation exception. In quiet mode, FloatAP will visit all the propagation rules (e.g., ∞+x=∞, ∞*0=NaN), search for the input combinations across input vectors, and update the output of the matched lanes according to the rules. Even if 0, Inf, and NaN all exist, the worst-case overhead is merely 8 cycles. Moreover, if the arithmetic behavior and dataset are both trustworthy, FloatAP can disable the special value handling to avoid the overhead or perform sanity checks periodically using search operations.

E. Other Functionalities

Using addition and multiplication as basic operations, FloatAP can efficiently implement other more complex functionalities, such as MAC, division, and non-linear activation functions (e.g., GELU or Sigmoid), with the Newton-Raphson method or Taylor-Maclaurin series. This way, FloatAP can perform any operations without assistance from other compute engines. Any frequently used function can be implemented as a single vector instruction by directly microprogramming its AA into a single instruction and hard-coding it into the VCU's AA storage. In our evaluation, however, the non-linear functions are implemented as a RISC-V program using multiple *vfmul* and *vfadd* instructions to avoid introducing non-standard instructions to the ISA. Note that while FloatAP is compatible with a lookup table (LuT)-based activation approximation [32], [33] that other PIM designs tend to use, we choose not to do so because storing the pre-calculated values in SRAM consumes too much area, whereas if in DRAM it entails extra data movement.

IV. FloatAP Architecture

A. Propagation Chain Microarchitecture

We build FloatAP on top of the CAPE framework [11], [12], [35], [50] in Fig. 3, the state-of-the-art SRAM-based bit-sliced AP. Originally, CAPE's chain was designed for 32-bit

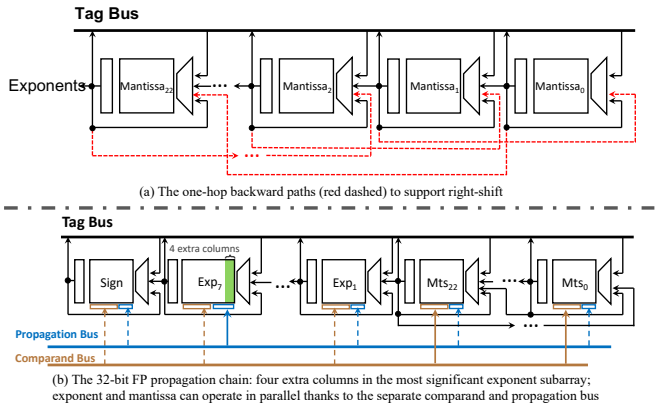


Fig. 7. FloatAP's modifications to the chain microarchitecture.

TABLE II: The time complexity of FloatAP vs. CAPE. Original CAPE does not support FP, so we map the best existing PIM-FP algorithm [28] to it.

RISC-V Inst.	FloatAP	CAPE-FP
vfredsum.vs	e+4m+10	e+1.5m^2+0.5m+9
vfadd.vv	4e+7m+18	4e+1.5m^2+8.5m+4
vfmul.vv	5m+2	4e+6m^2+11m
vfdot.vv	5e+m^2+3m+13	9e+7.5m^2+11.5m+9

integers and every chain contains 32 subarrays. Thus, single-precision FP values (FP32) can be naturally bit-sliced and stored in the CAPE chain. The sign will be stored in Bit_{31}, the 8-bit exponent in Bit_{30} to Bit_{23}, and mantissa in Bit_{22} to Bit_0. 16-bit formats such as half-precision (FP16, 5-bit exponent, 10-bit mantissa) and bfloat16 (BF16, 8-bit exponent, 7-bit mantissa) [48] can also reside in the chain despite only utilizing the second half of the chain (we address this later). For double-precision (FP64), FloatAP can accommodate it using two vector registers, each storing the higher and lower 32 bits. We adopt the following modifications to the integer chain to support the FP operations described in Sec. III.

1) Mantissa Subarrays: 1brsh requires a subarray to selectively update rows based on the tag bits of its left neighbor. So we add an extra one-hop rightward tag path for every mantissa subarray, shown as red dashed lines in Fig. 7(a), which feeds the tag bits of a subarray to its right neighbor's mux.

2) Exponent Subarrays: align entails searching a specific 8-bit key across the entire exponent field. For each row, the tag bit is marked as 1 only if all bits match. However, since the exponent is bit-sliced, this operation will necessitate one of the exponent subarrays temporarily storing a copy of the searched exponent in a contiguous data layout. Thus, we widen one of the exponent subarrays by several columns, i.e., the green region in Fig. 7(b). Every subarray already has 4 metadata columns [12], all free during exponent search. So we only need 4 extra columns because we focus on FP32 or narrower formats in this work. For FP64 it will need 7 extra columns.

Apart from the above two mandatory modifications, we propose two optional enhancements that can fully exploit the concurrency among subarrays to boost the performance.

Chain Splitting for 16 bits – BF16 and FP16 are prevalent in ML inference. Thus, FloatAP supports chain splitting to utilize the second half of the chain for FP16 and BF16. Specifically, the CSB's maximum vector length is doubled for 16-bit formats. The second half of the vector will be bit-sliced and stored in the second half of the chain. Both half-chains can then simultaneously perform the same operation on the first and second half of the vector. To support chain splitting, Subarray_{14} needs four extra columns for the exponent search in the second half-chain. Subarray_{25} to Subarray_{23}, which store the exponent for FP32 and BF16, are repurposed for mantissa for FP16. Thus, they need the one-hop rightward tag path as well. Tri-state buffers are needed to separate the tag bus at the midpoint so that both half-chains can utilize the tag bus simultaneously. However, there is no need to duplicate chain controllers or command buses since the two half-chains perform exactly the same operation at every cycle. Last, we incorporate an extra reduction tree for the second half-chain to maximize the *vfdot* and *vfmul* performance.

Exponent-Mantissa Concurrency – As discussed in III-A, each iteration in *align* searches a specific value across exponent and performs *1brsh* to mantissa. Thus, the exponent search can be overlapped with the last mantissa update in the previous iteration. As we store the contiguous copy of the exponent in metadata columns, this enhancement involves no hardware cost because, as Fig. 7(b) shows, the original CAPE chain already separates the command buses for the metadata (blue) and the data columns (brown). This optimization can reduce the mantissa alignment time complexity to $O(3m)$ cycles. Table II shows that after this optimization, the time complexity of *vfredsum*, *vfadd*, and *vfdot* becomes **(e+4m+10)**, **(4e+7m+18)**, and **(5e+m^2+3m+13)**, respectively.

B. FloatAP Architecture

A FloatAP core adopts the same organization as CAPE, shown in Fig. 3(a). One can instantiate FloatAP cores and integrate them into a system in various ways. In our evaluation, we envision FloatAP as a loosely coupled accelerator with a large number of cores and its own DRAM system. Each AP core is an independent PE and works on an exclusive partition of the problem. FloatAP cores do not have caches for the CSBs, nor do CSBs act like caches.[4] Instead, they are the end consumers and producers of the data and access DRAM directly using vector loads/stores. For the frequently-used data, such as weights, the program can load them into CSB once and keep them there for an extended period of time.

ISA and Software Stack – We program FloatAP using the FP instructions in RISC-V's standard vector extension [14]. Table II summarizes the corresponding RISC-V vector instructions of the operations we introduced in Sec. III. As we mentioned, the mantissa-to-integer conversion is performed right after a vector is loaded. Note that, apart from the ones in the table, FloatAP also supports all the non-FP vector instructions of CAPE. In this work, we manually implement

[4]The CP of each FloatAP core does have a tiny private I/D cache. However, it does not share data with CSB.

TABLE III: Specifications of one FloatAP core.

Control processor	2-issue 5-stage in-order, 2.7GHz, 7nm, 0.22mm^2,0.16W 32KB/32KB I/D cache, 8-way, LRU, 512B block
CSB – Physical	9MB data capacity (10MB total), 7nm, 7mm^2, 3W 2,304 chains, 32 subarrays per chain 32×36 cells per subarray
CSB – Logical	32 data vector registers, 4 metadata vectors 72K FP32 elements per vector register 0.6/4.8/6.6 TFLOPS for FP32/FP16/BF16

TABLE IV: Specs of area-equivalent FloatAP and A100 Tensor Core GPU [6].

	FloatAP
Overall	7nm, 2.7GHz, 815mm^2, 352W
AP	110 FloatAP cores, 990MB data capacity 68/525/717 TFLOPS for FP32/FP16/BF16
Memory	No cache; 80GB HBM2e, 2TB/s

	A100 Tensor Core GPU
Overall	7nm, 826mm^2, 400W TDP
SM	108 SMs, 4 Tensor Cores, 64K regs, 192KB L1 per SM Base: 19.5/78/39 TFLOPS for FP32/FP16/BF16 Tensor Core: 312 TFLOPS for FP16/BF16
Memory	40MB L2; 80GB HBM2e, 2TB/s

kernel functions such as *GeMM* and convolution using C language with inline RISC-V vector instructions and compile them using RISC-V toolchain [8]. One can implement BLAS in the same manner that can be directly called by the domain-specific languages such as NumPy or Julia, or ML frameworks like PyTorch. We leave this to the future work.

V. Experimental Setup

To assess the performance of FloatAP, we compare it against both CAPE and NVIDIA's A100 Tensor Core GPU [7]. All three architectures are based on 7 nm technology. The A100 GPU features 826 mm^2 area and 400 W TDP and consists of 108 Symmetric Multiprocessors (SMs) (Table IV). Thus, we set the area of one SM (7.6 mm^2) as the target design point of one FloatAP core, and the overall area of A100 for a large-scale FloatAP configuration. Detailed specifications of FloatAP and an individual AP core are available in Table IV and III, respectively.

A. Circuit-Level Modeling

Chain and CSB – We adopt the same methodology as CAPE for circuit-level modeling [12]. A full-custom subarray is simulated with the ASAP 7nm PDK [46]. The chain with 32 subarrays is then synthesized and place-and-route using Synopsys DC [4] and Cadence Innovus [5]. A 7 nm adder tree is also synthesized for the entire CSB. Due to the prevalence of 3×3 filter size in modern CNNs, we make the maximum vector length of each register divisible by 9. As a result, a CSB comparable to an SM in size has 2,304 chains and can hold 32 vectors, each with 72K lanes for FP32 (9 MB capacity). It consumes 7 mm^2 area and 3 W power. Compared to CAPE CSB with the same chain count, FloatAP CSB is 1.11× larger. The extra area is attributed to the wider exponent subarrays, the extra one-hop tag path for the mantissa subarrays, and the extra reduction tree for FP16 and BF16.

CP, VCU, and VMU We model the CP using an in-order dual-issue ARM Cortex-A53 core [3] with L1 I/D caches. We do not incorporate L2 or LLC for the CP as CSB handles all the data-intensive tasks. As the Cortex-A53 is in 16 nm, we scale it down based on the area ratio between 16 nm and 7 nm HD SRAM bitcell [1], [2]. Thus, the CP consumes 0.22 mm^2 area and 0.16 W power. For VCU and VMU, we model their AA storage, AA buffer, and load/store buffer using the area of a 7 nm High-Performance SRAM bitcell, totaling only 0.003 mm^2. We also factor into an extra Cortex core to account for the area and power of the sequencer in VCU. In total, the CP, VCU, and VMU contribute 0.44 mm^2 area, a 6% overhead compared to CSB.

B. Performance Modeling and Benchmarks

We model a single FloatAP core in Table III using gem5 simulator [10]. The latency of each vector instruction is calculated according to Table II, which already factors in the adder tree latency. We also accurately account for the command distribution latency from VCU to subarrays based on the delay of an H-Tree on Metal 4. Apart from FloatAP, we model a CAPE core with FP extension (CAPE-FP) that has the same CSB size as the baseline. As the original CAPE does not support FP, we implement the best existing PIM-FP algorithm [28] using search/update and map to CAPE. The complexity can also be found in Table II. We evaluate the performance of a FloatAP core using a series of kernel functions in Table V. They fall into three categories: the backbone of the ML model (GeMM and Conv); common post-processing and activation functions (LayerNorm, Sigmoid, and Softmax); plus some traditional data analysis applications (PCA, K-Means, Lreg) that were studied in prior AP works [12], [50], [55]. The programs are manually vectorized and compiled using RISC-V toolchain [8]. We compare the result of one FloatAP core with both CAPE-FP and area-equivalent A100 GPU SM. For the SM, we report the theoretical best performance.

Additionally, we assess the inference throughput of three real-world ML models on FloatAP with 110 AP cores. The evaluated models include a CNN (VGG16 [41]), an RNN (ResNet18 [24]), and an LLM (OPT-13B [56]).[5] As gem5 simulation is time-consuming, we developed an in-house simulator for the full-model experiments that accurately models FloatAP's latency in all aspects, including vector instructions, CP pipeline, data load/store, and intra-CSB data movement. We compare FloatAP's results with the theoretical best throughput of the area-equivalent A100 GPU. Finally, we conduct an energy efficiency study.

TABLE V: The evaluated kernel functions and input dimensions.

GeMM	Conv	LayerNorm	Sigmoid	Softmax	PCA	K-means	Lreg
1K×5K×5K	56-256-256, 3×3	1K×5K	1K×5K	1K×1K	1.5K×1.5K	100K	500MB

[5] We select 13B-parameter as its the largest configuration that can fit into the main memory (80 GB) after factoring in the weights and KV-matrices. The large models such as GPT-4 will require scaling out the system, which we leave for future study.

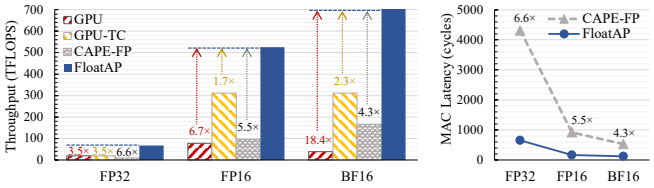


Fig. 8. Left: Peak dot-product throughput of FloatAP vs. area-equivalent A100 GPU and CAPE-FP. "GPU" gives A100's throughput with classic FMA operations whereas "GPU-TC" shows its throughput using Tensor Cores. Right: Latency reduction of FloatAP's *vfdot* operation over CAPE's search/update based *vfdot*.

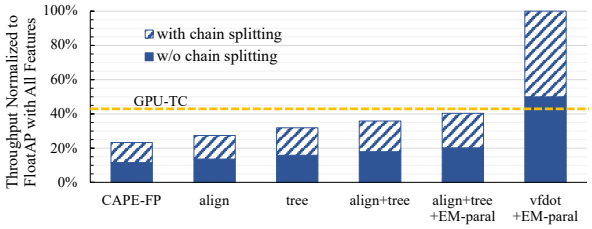


Fig. 9. The ablation study of the proposed techniques. The y-axis shows the BF16 MAC throughput normalized to FloatAP with all features enabled (the right-most bar in Fig. 8. The yellow dashed line gives the GPU Tensor Core performance. The striped region on each solid bar shows the performance improvement after chain splitting is enabled.

VI. Evaluation

A. Latency and Peak Throughput

The bar plot in Fig. 8 compares the peak dot-product throughput (TFLOPS) of FloatAP to area-equivalent A100 GPU and CAPE-FP implementations. The numbers on each bar indicate the relative slowdown compared to FloatAP. FloatAP achieves 67.8, 525, and 717 TFLOPS for FP32, FP16, and BF16 on average, respectively–between $3.5\times$ to $18.4\times$ higher than GPU and 4.3 to $6.6\times$ than CAPE-FP for all three formats. Notably, even though the state-of-the-art Tesor Core technology [7] ("GPU-TC") already boosts GPU's 16-bit performance by 4 to $8\times$, FloatAP can still outperform it by 1.7 and $2.3\times$ for FP16 and BF16, respectively. Without the contributions of this work, however, the best possible AP performance ("CAPE-FP") cannot compete with Tensor Cores. As the plot on the right shows, this drastic enhancement of CAPE's throughput is credited to our efficient hybrid *vfdot* algorithm, which reduces the latency of search/update-based dot-product by 4.3 to $6.6\times$.

For FP16, FloatAP achieves $7.7\times$ higher throughput than FP32. This improvement is ascribed to two factors. First, the time complexity of *vfdot* is quadratic to the mantissa's bit width (Table II), so FloatAP's FP16 *vfdot* latency is about $4\times$ shorter than FP32. Second, each 32-subarray chain can execute two FP16 *vfdot*s in flight by utilizing chain splitting, further improving the throughput by $2\times$. BF16 also benefits from these two facts. Moreover, compared to FP16, it has fewer mantissa bits (7 bits vs. 10 bits). Thus, FloatAP's BF16 throughput is $1.4\times$ higher than FP16. The wider exponent field of BF16 does not counteract the benefit of narrower mantissa because the exponent width only contributes linearly to the *vfdot* time complexity.

Fig. 9 shows an ablation study of the proposed techniques based on the MAC throughput of BF16, normalized to the throughput with all features enabled. The left-most bar represents the baseline CAPE-FP that calculates dot-product with bit-serial multiplication and quadratic reduction sum. From left to right, we show the effect when gradually enabling each proposed technique, one after another. The striped region shows the effect of chain splitting, which can always double the throughput of a 16-bit datatype.

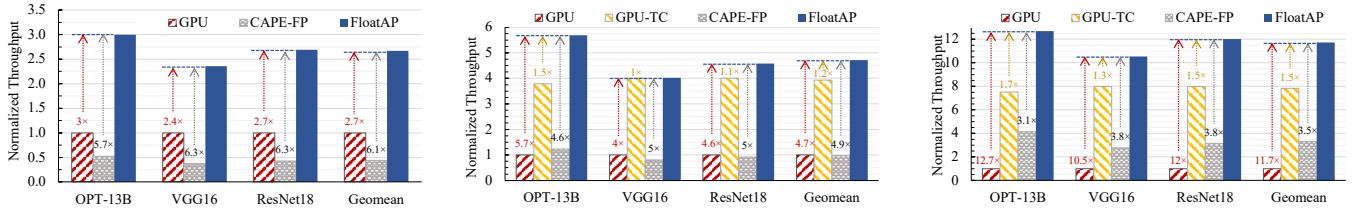
align represents the throughput when only parallel mantissa alignment is enabled while still using bit-serial multiplication. In contrast, *tree* stands for enabling only adder tree-based multiplication, with mantissa alignment remaining quadratic. *Align+tree* combines both techniques. Compared to the baseline, each of these two techniques individually improves the CAPE-FP throughput by $1.2\times$ and $1.4\times$, respectively, and by $1.6\times$ when combined. *align+tree+EM-paral* shows that exponent-mantissa parallelism can further enhance the throughput by an additional $1.1\times$. Finally, the right-most bar demonstrates that by replacing back-to-back *vfmul* and *vfredsum* operations with our *vfdot* algorithm, FloatAP achieves a $2.5\times$ performance gain. Note that both *align* and *tree* are prerequisites for *vfdot*. The ablation study confirms that all of our proposed techniques are effective in pursuing high-performance FP arithmetic.

B. ML Inference Performance

Fig. 10 shows FloatAP's inference throughput for three commonly used ML models compared with baselines. For OPT-13B, we choose a batch size of 256 and utilize sparse attention [47], [57] so that the model is not memory-bound for both baselines and FloatAP. The prompt and generation lengths are both set to 1,024. For VGG and ResNet, we adopt a weight stationary mapping on FloatAP. We distribute layers to AP cores based on the operation count of each layer to ensure every core has a similar workload.

FloatAP outperforms A100 GPU across all models and FP formats. The highest speedup over GPU is with BF16 because BF16 has the most narrow mantissa width. Even with the help of Tensor Cores, the A100 GPU is still on average $1.5\times$ ($1.2\times$) slower than FloatAP for BF16 (FP16). And for single-precision, where A100 cannot benefit from Tensor Cores, FloatAP achieves an even larger $2.7\times$ speedup. The only occasion where both FloatAP and A100 have similar performance is VGG16 for FP16. For this model, the early layers have a short channel count, resulting in a low utilization of vector length on FloatAP. Even so, FloatAP still delivers a similar overall performance as A100. Therefore, we conclude that FloatAP is an efficient solution for today's ML workloads. In contrast, CAPE-FP only has performance merits over GPU for BF16 data type and can never outperform Tensor Cores.

Note that FloatAP achieves higher speedups for OPT than VGG and ResNet because the last two require feature map



(a) FP32 (b) FP16 (c) BF16

Fig. 10. The inference throughput of FloatAP and baselines normalized to A100 GPU.

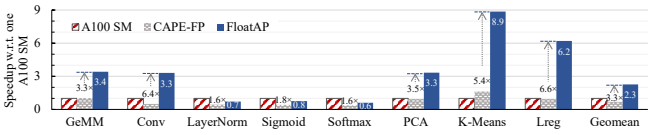
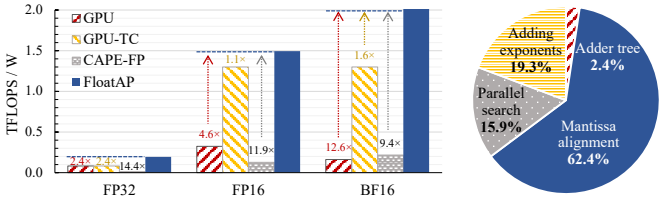


Fig. 11. The speedup of area-equivalent one FloatAP core with respect to a single A100 Symmetric Multiprocessor (SM) and CAPE-FP.



(a) Energy efficiency of FloatAP vs. baselines (b) Energy breakdown

Fig. 12. The energy efficiency and FP32 *vfdot* energy breakdown of FloatAP.

shifting within each AP core, which introduces extra communication overhead. Moreover, for CNNs, it is harder to balance the load among AP cores perfectly compared to the GeMM in OPT. Therefore, FloatAP's performance of VGG16 and ResNet18 is bottlenecked by the AP core with the largest amount of workload.

C. Kernel Function Performance

To better understand FloatAP's performance in different scenarios, we evaluate FloatAP over a set of kernel functions. The dimensions of the input dataset (# of FP32 values) are shown in table V. As the problem size is much smaller than a full ML model, in this experiment, we compare an individual FloatAP core with an area-equivalent A100 SM. Fig. 11 gives the FP32 performance. The other two formats have similar results.

On average, FloatAP achieves 2.3× speedup over A100 SM and 3.6× over CAPE-FP. For the apps majorly comprising dot-products (GeMM, Conv, and PCA), the speedup over GPU is similar to the peak throughput improvement. For K-Means and Lreg, FloatAP performs even better because these two apps also involve a considerable amount of *vfadd*, which is extremely efficient on FloatAP since our *vfadd* only has linear complexity. In contrast, CAPE-FP cannot run these two apps quite as fast because its *vfadd* is quadratic with the bit width. Nevertheless, neither FloatAP nor CAPE-FP can outperform GPU SM for activation functions, since the calculation of the Taylor series requires frequent *vfmul*. As explained in Sec. III-C and Table II, although the latency of *vfmul* on FloatAP only takes $O(5m)$, it requires m times more vector lanes than the actual vector length. Thus, from the throughput aspect, FloatAP *vfmul* is five times lower than the *vfdot*. Even though still 1.2× faster than CAPE-FP and other AP proposals, FloatAP cannot match GPU's *vfmul* performance. However, in ML models, those functions typically only take 1% of the overall execution time, and FloatAP is only 1.3× to 1.6× slower than GPU, so they will not harm the overall performance. In addition, we believe that supporting the execution of activation functions locally on FloatAP is still a good design choice, as it eliminates the need to move data between FloatAP and other processing engines in the system or load look-up tables from DRAM.

D. Energy Efficiency

In Fig. 12a, we assess the energy efficiency of FloatAP relative to the other architectures. The values are shown in TFLOPS/W (i.e., TOps/J). For A100, based on its thermal behavior study [58], we conservatively assume its power consumption is only 60% of the reported 400 W TDP. FloatAP achieves 0.2, 1.5 and 2 TFLOPS/W for FP32, FP16, and BF16, respectively, 2.4× to 12.6× higher than A100. When compared to Tensor Core throughput for 16-bit formats, FloatAP is still 1.1× to 1.6× more efficient. Moreover, compared to CAPE-FP, FloatAP's energy efficiency is an order of magnitude higher, even though both architectures are AP-based. The reason is that the associative primitives, i.e., searches and updates, are much more energy-consuming than the adder tree. As shown in Fig. 12b, even though FloatAP performs the major part of *vfdot* and *vfmul* on the adder trees, their energy consumption only constitutes 2.4% of the entire operation. CAPE-FP, on the other hand, implements all phases of multiplication using searches and updates. As a result, the mantissa multiplication phase of CAPE-FP consumes more than twice as much energy as the whole *vfdot* on FloatAP. In conclusion, FloatAP largely improves the energy efficiency of AP by utilizing the reduction tree for multiplication, which makes AP a more economical solution than commercial GPU for today's FP-based ML workloads.

VII. Related Work

Associative Processor – AP designs have existed for almost half a century [16], [37], [38]. Many modern ar-

chitectures have been proposed following the philosophy of AP [11], [12], [17], [19], [21], [25]–[27], [29], [35], [36], [50]–[55]. Among them, [12], [19], [36], [50]–[54] utilize the quadratic bit-serial *vmul* or *vfmul*, which takes thousands of cycles, resulting in much lower throughput and longer latency than FloatAP's hybrid *vfmul* and dot-product. Moreover, none of them introduce FP *vfadd* and *vfredsum*. [27] and [26] implement associative arithmetic operations by storing the pre-computed results of the *whole operands* in the lookup tables. Thus, their LuTs can only store the combinations of certain frequent operands to keep the size under control. So [27] and [26] can only perform approximate computing combined with quantization. Other APs only adopt integer or fixed-point.

Other PIM Architectures for FP – PIM designs have been proven to be a good fit for modern ML workloads [23], [30], especially neural networks. Nevertheless, many existing works rely on quantization and only support fixed-point arithmetic. The common practice to realize in-memory FP operations is to leverage the conductance properties of resistive memory cells (ReRAM) for *analog* dot-product operations [40], [43], [44]. As the functionality is limited, those analog architectures are less flexible than the digital PIM proposals. Also, the ADC/DAC introduces a significant area and power overhead (e.g., more than 30% area and 50% power for ISAAC [23], [40]). Moreover, all those proposals are affected by the endurance and process variation problem of ReRAM cells. Duality Cache [18] and FloatPIM [28] are two fully digital PIM architectures that support accurate FP arithmetic. Duality Cache realizes in-SRAM computation by implementing bitline-ALUs at each sense amplifier. FloatPIM leverages triple-cell NOR operation on ReRAM crossbars to perform in-situ add and multiply operations. However, they both adopt contiguous data layouts. Their FP add and multiply are bit-serial with quadratic time complexity. Moreover, neither work efficiently supports reduction or dot-product operations, which are critical in ML workloads. Other in-DRAM [22], [34], [39] and in-SRAM [9], [13], [15] digital PIM proposals do not support FP arithmetic.

VIII. Conclusion

In conclusion, FloatAP marks a significant advancement in highly programmable APs for high-performance FP arithmetic. By leveraging novel bit-parallel associative algorithms and repurposing existing reduction trees for multiplication and dot-product, FloatAP drastically increases the FLOPS of APs at minimal hardware expense while retaining RISC-V programmability. The microarchitectural enhancements further reduce the instruction latency and boost the utilization of FP formats with different widths. The experiments show that FloatAP outperforms the area-equivalent A100 GPU in both inference throughput and energy efficiency, making FloatAP a compelling solution for contemporary AI/ML applications.

Acknowledgment

We would like to thank Helena Caminal, Socrates Wong, and Cecilio Tamarit for the early discussions and support, Christopher Batten and Christopher De Sa for their advice and feedback, and Michael Woodson and the rest of the CoE-CIS IT team for their technical support.

References

[1] "16 nm lithography process," https://en.wikichip.org/wiki/16_nm_lithography_process, accessed: 2021-04-14.

[2] "7 nm lithography process," https://en.wikichip.org/wiki/7_nm_lithography_process, accessed: 2021-04-14.

[3] "Cortex-a53," https://developer.arm.com/ip-products/processors/cortex-a/cortex-a53, accessed: 2021-04-14.

[4] "Dc ultra: Concurrent timing, area, power, and test optimization," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html, accessed: 2020-04-13.

[5] "Innovus implementation system," https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html, accessed: 2020-04-13.

[6] "Nvidia a100," , accessed: 2024-03-04.

[7] "Nvidia a100 tensor core gpu architecture," , accessed: 2024-03-28.

[8] "Risc-v gnu toolchain," https://github.com/riscv/riscv-gnu-toolchain, accessed: 2021-04-07.

[9] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das, "Compute caches," in *2017 IEEE International Symposium on High Performance Computer Architecture*, 2017.

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

[11] H. Caminal, Y. Chronis, T. Wu, J. M. Patel, and J. F. Martínez, "Accelerating database analytic query workloads using an associative processor," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 623–637. [Online]. Available: https://doi.org/10.1145/3470496.3527435

[12] H. Caminal, K. Yang, S. Srinivasa, A. K. Ramanathan, K. Al-Hawaj, T. Wu, V. Narayanan, C. Batten, and J. F. Martínez, "Cape: A content-addressable processing engine," *The 2021 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2021.

[13] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018.

[14] A. A. et al., "Risc-v "v" vector extension," https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc, accessed: 2023-03-14.

[15] R. Fan, Y. Cui, Q. Chen, M. Wang, Y. Zhang, W. Zheng, and Z. Li, "Maicc: A lightweight many-core architecture with in-cache computing for multi-dnn parallel inference," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 411–423.

[16] C. C. Foster, *Content Addressable Parallel Processors*. John Wiley & Sons, Inc., 1976.

[17] M. E. Fouda, H. E. Yantır, A. M. Eltawil, and F. Kurdahi, "In-memory associative processors: Tutorial, potential, and challenges," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2022.

[18] D. Fujiki, S. Mahlke, and R. Das, "Duality cache for data parallel acceleration," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[19] E. Garzón, A. Teman, M. Lanuzza, and L. Yavits, "Aida: Associative in-memory deep learning accelerator," *IEEE Micro*, vol. 42, no. 6, pp. 67–75, 2022.

[20] 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 Computer Architecture Letters*, vol. 23, no. 1, pp. 29–32, 2024.

[21] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman, "Ac-dimm: Associative computing with stt-mram," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 189–200. [Online]. Available: https://doi.org/10.1145/2485922.2485939

[22] N. Hajinazar, G. F. Oliveira, S. Gregorio, J. D. Ferreira, N. M. Ghiasi, M. Patel, M. Alser, S. Ghose, J. Gómez-Luna, and O. Mutlu, "Simdram: A framework for bit-serial simd processing using dram," 2020.

[23] M. Hassanpour, M. Riera, and A. González, "A survey of near-data processing architectures for neural networks," *Machine Learning and Knowledge Extraction*, vol. 4, no. 1, pp. 66–102, 2022.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[25] Y. Huang, L. Kong, D. Chen, Z. Chen, X. Kong, J. Zhu, K. Mamouras, S. Wei, K. Yang, and L. Liu, "Casa: An energy-efficient and high-speed cam-based smem seeding accelerator for genome alignment," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1423–1436.

[26] M. Imani, S. Patil, and T. Šimunić Rosing, "Approximate computing using multiple-access single-charge associative memory," *IEEE Transactions on Emerging Topics in Computing*, vol. 6, no. 3, pp. 305–316, 2018.

[27] M. Imani, M. Samragh Razlighi, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, "Deep learning acceleration with neuron-to-memory transformation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 1–14.

[28] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "Floatpim: In-memory acceleration of deep neural network training with high precision," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 802–815. [Online]. Available: https://doi.org/10.1145/3307650.3322237

[29] Z. Jahshan, I. Merlin, E. Garzón, and L. Yavits, "Dash-cam: Dynamic approximate search content addressable memory for genome classification," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 1453–1465.

[30] J.-H. Jang, J. Shin, J.-T. Park, I.-S. Hwang, and H. Kim, "In-depth survey of processing-in-memory architectures for deep neural networks," *JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE*, vol. 23, no. 5, pp. 322–339, 2023.

[31] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw, "A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, pp. 1009–1021, April 2016.

[32] D. Kwon, S. Lee, K. Kim, S. Oh, J. Park, G.-M. Hong, D. Ka, K. Hwang, J. Park, K. Kang *et al.*, "A 1ynm 1.25 v 8gb 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep learning application," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 1, pp. 291–302, 2022.

[33] Y. Kwon, K. Vladimir, N. Kim, W. Shin, J. Won, M. Lee, H. Joo, H. Choi, G. Kim, B. An *et al.*, "System architecture and software stack for gddr6-aim," in *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE, 2022, pp. 1–25.

[34] S. Li, D. Niu, K. T. Malladi, H. Zheng, B. Brennan, and Y. Xie, "Drisa: A dram-based reconfigurable in-situ accelerator," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[35] J. F. Martínez, H. Caminal, K. Yang, K. Al-Hawaj, and C. Batten, "Content-addressable processing engine," Oct. 4 2022, uS Patent 11,461,097.

[36] A. Morad, L. Yavits, S. Kvatinsky, and R. Ginosar, "Resistive gp-simd processing-in-memory," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2845084

[37] J. Potter, J. Baker, S. Scott, A. Bansal, C. Leangsuksun, and C. Asthagiri, "Asc: an associative-computing paradigm," *Computer*, vol. 27, no. 11, pp. 19–25, 1994.

[38] G. E. Sayre, "Staran: An associative approach to multiprocessor architecture," in *Computer Architecture.Springer Berlin Heidelberg*, 1976.

[39] V. Seshadri, D. Lee, T. Mullins, H. Hassan, A. Boroumand, J. Kim, M. A. Kozuch, O. Mutlu, P. B. Gibbons, and T. C. Mowry, "Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.

[40] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 14–26. [Online]. Available: https://doi.org/10.1109/ISCA.2016.12

[41] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015.

[42] I. C. Society, "Ieee standard for floating-point arithmetic," *IEEE STD 754-2019*, pp. 1–84, 2019.

[43] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 541–552.

[44] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "Graphr: Accelerating graph processing using reram," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 531–543.

[45] M. S. Q. Truong, E. Chen, D. Su, L. Shen, A. Glass, L. R. Carley, J. A. Bain, and S. Ghose, "Racer: Bit-pipelined processing using resistive memory," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 100–116. [Online]. Available: https://doi.org/10.1145/3466752.3480071

[46] V. Vashishtha, M. Vangala, and L. T. Clark, "Asap7 predictive design kit development and cell design technology co-optimization: Invited paper," in *2017 IEEE/ACM International Conference on Computer-Aided Design*, 2017.

[47] H. Wang, Z. Zhang, and S. Han, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, mar 2021, pp. 97–110. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/HPCA51647.2021.00018

[48] S. Wang and P. Kanwar, "Bfloat16: The secret to high performance on cloud tpus," https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus, accessed: 2024-01-18.

[49] S. Wimer and I. Koren, "Energy efficient deeply fused dot-product multiplication architecture," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2016, pp. 115–122.

[50] S. S. Wong, C. C. Tamarit, and J. F. Martínez, "Pumice: Processing-using-memory integration with a scalar pipeline for symbiotic execution," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, 2023, pp. 1–6.

[51] L. Yavits, A. Morad, and R. Ginosar, "Computer architecture with associative processor replacing last-level cache and simd accelerator," *IEEE Transactions on Computers*, 2015.

[52] L. Yavits, A. Morad, and R. Ginosar, "Sparse matrix multiplication on an associative processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 11, pp. 3175–3183, 2015.

[53] L. Yavits, R. Kaplan, and R. Ginosar, "Giraf: General purpose in-storage resistive associative framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 2, pp. 276–287, 2021.

[54] L. Yavits, S. Kvatinsky, A. Morad, and R. Ginosar, "Resistive associative processor," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 148–151, 2015.

[55] Y. Zha and J. Li, "Hyper-AP: Enhancing associative processing through a full-stack optimization," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture*, 2020.

[56] S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer, "Opt: Open pre-trained transformer language models," 2022.

[57] Z. Zhang, Y. Sheng, T. Zhou, T. Chen, L. Zheng, R. Cai, Z. Song, Y. Tian, C. Ré, C. Barrett, Z. Wang, and B. Chen, "H_2o: Heavy-hitter oracle for efficient generative inference of large language models," 2023.

[58] M. Špeťko, O. Vysocký, B. Jansík, and L. Říha, "Dgx-a100 face to face dgx-2—performance, power and thermal behavior evaluation," *Energies*, vol. 14, no. 2, 2021. [Online]. Available: https://www.mdpi.com/1996-1073/14/2/376