Appears in the Proceedings of the 29th IEEE Int'l Symp. on High-Performance Computer Architecture (HPCA'29), Feb 2023

EVE: Ephemeral Vector Engines

Khalid Al-Hawaj, Tuan Ta, Nick Cebry, Shady Agwa, Olalekan Afuye, Eric Hall, Courtney Golden,
Alyssa B. Apsel, and Christopher Batten

School of Electrical and Computer Engineering,
Cornell University, Ithaca, NY
{ka429,qtt2,nfc35,sr972,ota2,ewh73,ckg35,aba25,cbatten}@cornell.edu

Abstract—**There has been a resurgence of interest in vector architectures evident by recent adoption of vector extensions in mainstream instruction set architectures. Traditionally, vector engines leverage this abstraction by exploiting its inherent regularity to increase performance and efficiency. Recent work on SRAM-based compute-in-memory has shown promise in reducing the area overhead of these engines. In this work, we propose ephemeral vector engines (EVE) where we leverage SRAM-based compute-in-memory techniques as well as bit-peripheral computations to facilitate efficient vector execution. EVE uses a novel approach of bit-hybrid execution, striking a balance between throughput and latency. Evaluated on the Rodinia and RiVEC benchmark suites, EVE achieves almost 8× speed-up compared to an out-of-order processor and 4.59× compared to an integrated vector unit. EVE achieves speed-ups comparable to an aggressive decoupled vector unit and increases the area-normalized performance by over 2×. By repurposing SRAM arrays in the L2 cache to create ephemeral vector execution units, EVE is able to efficiently achieve high performance while incurring as little as 11.7% area overhead.**

I. Introduction

As technology scaling fails to provide regular improvements in transistor performance and efficiency [15, 19], there is a resurgence of interest in vector architectures demonstrated by ARM SVE [56, 57] and RISC-V RVV [43]. Traditionally, vector execution has been achieved either through simplified subword packed SIMD units or through aggressive long-vector engines [16, 32, 45, 60]. There is an emerging trend towards next-generation vector architectures, which provide unified abstractions suitable for a variety of different micro-architectures and implementations (see Table I). To implement these next-generation vector architectures, one can either use an integrated vector unit (IV) or a decoupled vector engine (DV). Integrated vector units are typically tightly coupled into the pipeline of the control processor. These units incur lower area overhead as they reuse the control processor execution hardware and often support short hardware vector lengths. Decoupled vector engines, on the other hand, are often loosely coupled with the control processor and incur higher area overhead as they use aggressive execution hardware and support long hardware vector lengths. There is a fundamental tension between performance and area among these next-generation vector micro-architectures: (1) integrated vector units achieve modest performance in accelerating regular data-parallel workloads while costing modest area overhead; (2) decoupled vector engines have significantly better performance at significantly higher area overhead. This paper seeks to address this tension: **Is it possible to achieve the performance of decoupled vector engines with the area overhead of integrated vector units?**

Compute-in-memory (CIM) is a novel approach to reduce the area overhead associated with accelerating data-parallel kernels, offering a promising path to solving this tension. There are multiple flavors of CIM targeting different technologies: DRAM-based compute-in-memory (D-CIM) [28, 33, 34, 50], RRAM-based compute-in-memory (R-CIM) [10, 13, 14, 35, 41, 54, 55], and SRAM-based compute-in-memory (S-CIM) [2–4, 17, 23, 30, 51, 52, 61]. This work focuses on S-CIM since it can be readily implemented in current state-of-the-art processes and enables closer integration with general-purpose processors. Prior work on S-CIM leverages bit-line computation [30] to perform simple bit-wise logical operations through a single read of a traditional SRAM. Complex integer, fixed-point, and floating-point operations can be executed in-situ by additional peripheral hardware. There are two key challenges when leveraging S-CIM to accelerate data-parallel workloads: (1) **S-CIM Programming:** offering a compelling abstraction enabling wider applicability and flexible programmability, and (2) **S-CIM Serialization Latency:** mitigating the latency overhead incurred by the serialized nature of S-CIM in both compute and memory operations.

TABLE I. A Summary of Vector Architectures.

Attribute	Packed SIMD	Long Vector	Next Generation
Length	fixed, short	scalable, long	scalable
Element Width	variable	fixed	variable
Predication	limited	full	full
Cross-Element Ops	full	limited	full
Memory Gather/Scatter	limited	full	full
Integration	integrated	decoupled	either
Speculative Execution	yes	no	either
Compute Pipeline	integrated	decoupled	either
Memory Bandwidth	modest	large	either
Memory Latency	low	high	either

Duality cache [23] is a recent work that leverages S-CIM techniques to accelerate data-parallel workloads in a coarse-grain fashion. Duality cache addresses the S-CIM programming challenge by adopting a SIMT abstraction. The shared last-level cache (LLC) in a chip multi-processor (CMP) can be reconfigured to create a large SIMT-style execution engine on demand. Leveraging S-CIM techniques, memory arrays in the LLC are used to create a large number of bit-serial arithmetic and logical units (ALUs). Due to the use of bit-serial execution, duality cache suffers from high latencies (i.e., thousands of cycles) in arithmetic operations. Incoming data need to be transposed to fit the required layout for bit-serial execution, incurring area overhead, additional latency, and reduced memory bandwidth. The fully-transposed data along with the requirement for all registers of a given thread to exist in the same column forces duality cache to allocate architectural registers to neighboring SRAM arrays. As a result, extra move operations are required to execute instructions between registers in different SRAM arrays. Duality cache requires extreme levels of parallelism to achieve compelling speed-ups; thus requiring most of the last-level cache for execution. As a result, these limitations constrain the use of duality cache to a coarse grain offloading model.

Though most of the prior work on S-CIM uses bit-serial execution, VRAM [6] proposes bit-parallel as well as bit-serial execution. VRAM shows that bit-parallel execution incurs similar area overhead to that of bit-serial. VRAM attempts to address the S-CIM serialization latency challenge by using bit-parallel execution. Whereas a bit-serial approach targets higher throughput, a bit-parallel approach can achieve lower latencies in compute and memory operations. CRAM [61] is another work on S-CIM that addresses the S-CIM serialization latency challenge in bit-serial execution by utilizing 8T-SRAM bit cells; thus, lowering the serialization overhead. The compute operations in CRAM still require high latencies due to their bit-serial nature. Due to the use of 8T-SRAM bit cells, CRAM also incurs a high area overhead.

In this paper, we propose ephemeral vector engines (EVE). EVE leverages prior work on S-CIM to build efficient next-generation vector accelerators with support for all 32-bit integer instructions in the RISC-V vector extension [43]. By way-partitioning a private L2 cache, each core in a CMP can dynamically create an ephemeral private vector engine to execute data-parallel workloads efficiently. While state-of-the-art S-CIM leverages bit-serial execution, EVE opts for a novel bit-hybrid approach that balances the throughput and latency of different vector instructions. Elements are broken down into *n*-bit segments that are computed in bit-parallel fashion, while the segments themselves are computed serially. SRAM arrays in the partitioned cache ways are replaced with EVE SRAM, which is a 6T-SRAM capable of bit-line computation with novel bit-peripheral circuits added to enable complex bit-hybrid operations. Additional units are added to: control instruction execution, handle memory instructions, and perform reduction/cross-element instructions.

We use a vertically integrated research methodology to evaluate EVE on two important metrics: area and performance. We generated a layout of a simplified EVE SRAM through a modified version of OpenRAM [27], which is a Python-based open-source SRAM generator. The layout is composed of a simplified version of the EVE circuits added to a bit-line compute capable 6T-SRAM. We used this layout to estimate the area overhead of EVE circuits as well as the cycle-time penalty over a traditional SRAM array. For performance evaluation, we built a cycle-approximate model of EVE in gem5 [7] and simulated its performance on vectorized implementations of applications from the Rodinia [11] and RiVEC [42] benchmark suites. To quantify EVE's performance, we built a cycle-approximate model of: (1) a high-performance decoupled vector engine loosely based on Tarantula [20], and (2) an integrated vector unit with hardware vector length matching conventional SIMD width [44, 58]. EVE is able to achieve comparable speedups to a decoupled vector engine, and a 3.5 × speedup over an integrated vector unit with as little as 5% area overhead.

Our main contributions are: (1) a novel bit-hybrid execution approach for SRAM-based compute-in-memory (S-CIM); to our knowledge, this is the first work that proposes using bit-hybrid approach for S-CIM to balance throughput and latency; (2) a template for EVE circuits that enables building EVE-*n* SRAM capable of executing vector operations targeting *n*-bit-hybrid execution; (3) an exploration of the cycle-level impact and trade-offs of targeting different *n*-bit-hybrid execution configurations (i.e., n = 1, 2, 4, 8, 16, 32); (4) the novel EVE micro-architecture that enables transforming private L2 cache ways into ephemeral vector engines; (5) a detailed evaluation of different EVE design points exploring the trade-offs and impact of various design parameters.

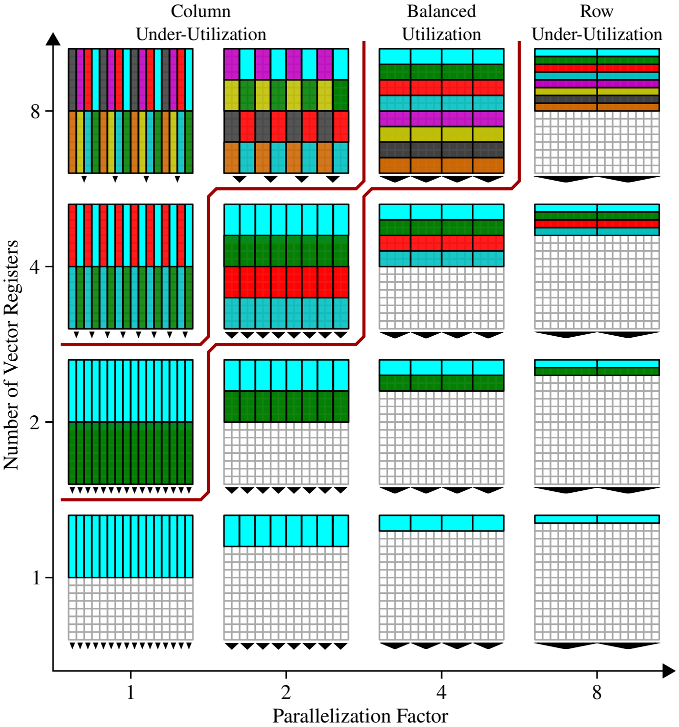


Figure 1. Data Organization in S-CIM SRAM Array – varying number of supported vector registers and the parallelization factor. Grey boxes represent bit-cells. Each vector register is assigned a unique color. Elements belonging to the same vector register are shown as boxes with the same color. Triangles at the periphery represent in-situ ALUs.

II. Taxonomy of Vector S-CIM

Previous work has examined two different design points for a S-CIM vector engine: bit-serial [6, 17, 61] and bit-parallel [6]. In bit-serial execution, the S-CIM vector engine breaks down each 32-bit element into one-bit segments (i.e., each element consists of 32 segments). The engine processes one segment of a given element in a cycle, and processes the rest of the segments serially. In bit-parallel execution, the S-CIM vector engine processes each 32-bit element as one segment, where the segment size is 32 bits. We define the *parallelization factor* of a S-CIM vector engine as the width (in bits) of a segment from an element that the engine can process in parallel. Using this generalization, bit-serial designs are expressed as S-CIM vector engines with a parallelization factor of one. Bit-parallel designs are considered to be S-CIM vector engines with a parallelization factor of *n*, where *n* is the element size supported.

By varying the parallelization factor, there is a spectrum that describes different S-CIM vector engines. On one end of the spectrum, S-CIM vector engines with a parallelization factor of one (i.e., bit-serial) achieve higher throughput at the cost of higher latency. On the other end of the spectrum, S-CIM vector engines with a parallelization factor of 32 (i.e., bit-parallel, assuming 32-bit elements) achieve lower latency but lower throughput as well. To explore this spectrum, we construct an analytical model that calculates the throughput and latency for a vector addition and multiplication. As one of the principles of S-CIM is to have all input elements for an operation in the same column, the model requires all vector registers, with all their elements, to be stored in the same SRAM array.

Element Layout & Available In-Situ ALUs – Figure 1 shows the layout of vector registers holding 8-bit elements in a 16 × 16 SRAM while varying the parallelization factor. Considering an ISA that supports one vector register, with parallelization factor of one, each element occupies a single column in the SRAM. As a result, half the SRAM is occupied providing storage for 16 elements. By increasing the parallelization factor, the segment size increases while the number of segments decreases. As each segment occupies a row, the number of elements for the vector register decreases; thus, lowering the number of available in-situ ALUs.

Starting with a parallelization factor of one, as the number of supported vector registers increases, the utilization of the SRAM grows as well. Each column in the SRAM can be used as an ALU performing in-situ computation of corresponding elements from different vector registers. As Figure 1 shows, the SRAM reaches balanced utilization with two vector registers. However, to support more vector registers, some of the columns are repurposed to hold the additional registers, reducing the number of in-situ ALUs. With higher parallelization factor, the SRAM can support more vector registers without reaching column under-utilization, as each vector register is composed of fewer segments. But, high parallelization factors struggle to increase the row utilization of the SRAM unless provided with more vector registers. Figure 2 shows the latency and throughput for vector addition and multiplication as the parallelization factor increases for a S-CIM vector engine implemented using a 256 × 256 SRAM with 32 vector register support.

Latency – Figure 2 shows the latency for a vector addition and multiplication normalized to that of parallelization factor of one. As the parallelization factor increases, the number of segments decreases accordingly; because the number of cycles required to perform a vector addition and multiplication correlates to the number of segments, the latency of these operations decreases as well. However, one can observe from Figure 2 that the latency is not linearly correlated with the number of segments. The insight behind this is the control overhead induced by initializing counters and control branching to process segments serially.

Throughput – Figure 2 shows the throughput for a vector addition and multiplication. Starting with a parallelization factor of one, the S-CIM vector engine experiences column under-utilization (as previously exemplified in Figure 1). As the parallelization factor increases, the elements are composed of fewer segments; as a result, the column under-utilization is lessened and the throughput increases despite having the same number of in-situ ALUs (since the latency of the in-stu ALUs decreases). The throughput peaks when the parallelization factor reaches four, achieving balanced utilization for the S-CIM SRAM. Beyond balanced utilization, the S-CIM vector engine experiences row under-utilization and the number of in-situ ALUs is decreased causing a drop in the throughput. Although, as discussed previously, the latency will decrease further as the parallelization factor increases, the decrease is not enough to compensate for the reduction in the number of in-situ ALUs.

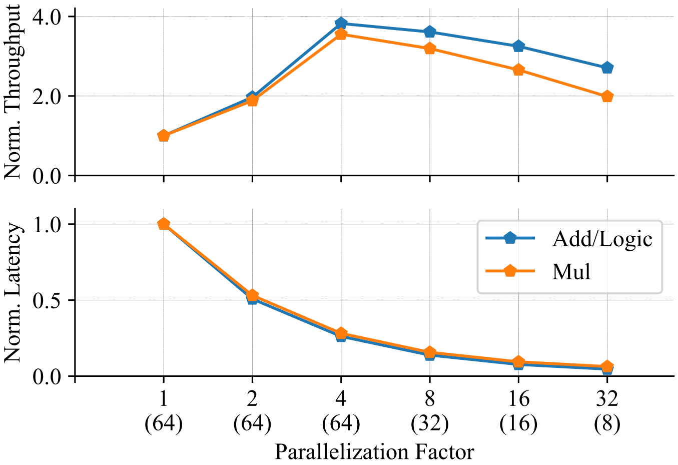


Figure 2. Latency and Throughput of Add/Logic and Multiply vs. Parallelization Factor – achieved by a 256 × 256 S-CIM SRAM assuming 32 vector register support normalized to latency and throughput of parallelization factor of 1. The number of in-situ ALUs for each parallelization factor is shown between parentheses in the X-axis.

Key Insights – This section shows that both bit-serial and bit-parallel are sub-optimal and induce problems such as row and columnunder-utilization; moreover, the taxonomy shows that targeting bit-hybrid, which is neither bit-serial nor bit-parallel is the optimal design decision. *To our knowledge, EVE is the first to make this observation and explore bit-hybrid design space.* Previous work has predominantly explored either bit-serial or bit-parallel. Duality cache, which leverages bit-serial execution, tried to mitigate column under-utilization by dividing vector registers into four banks and introducing explicit move instructions inserted automatically through compiler analysis. As a result, duality cache binaries are not optimal nor portable across designs with different sub-array sizes. EVE's elegant bit-hybrid approach alleviates column under-utilization without the need for compiler solutions, making its binaries optimal and portable regardless of underlying hardware micro-architecture.

III. EVE Circuits

EVE transforms traditional 6T-SRAM into a vector execution unit with additional peripheral hardware. EVE leverages the circuit design from prior work on VRAM [6] as the peripheral hardware with additional modifications to expand its functionality, as well as proposing a new circuit design for supporting bit-hybrid execution. Considering a 32-bit precision, EVE-1 utilizes a bit-serial execution approach, and EVE-32 utilizes a bit-parallel execution approach, while EVE-*n* targets an *n*-bit-hybrid execution approach.

The EVE circuits are composed of different stacks of logic and take as an input the outcome of a bit-line compute operation. To perform bit-line compute, the differential sense-amplifiers in the traditional 6T-SRAM are modified to support reconfigurable differential and single-ended modes. An extra address decoder is added to allow the selection of two wordlines simultaneously. When performing bit-line compute operation, the two operands (i.e., wordlines) to the operation are selected simultaneously and the sense-amplifiers are set in the single-ended mode. As a result, the sense-amplifiers compute four bit-wise logical operations: `and`, `nand`, `or`, and `nor`. The circuit for an EVE design is a stack composed of a mixture from seven different layers of logic: bus logic, XOR/XNOR logic, add logic, XRegister, mask logic, constant shifter, and spare shifter. Due to the bit-wise nature of the bus logic and XOR/XNOR logic, these layers are the same for the different EVE designs. The bus logic amplifies and selects one of the values computed by the circuit for it to be written back to the SRAM. Meanwhile, the XOR/XNOR logic uses the `nand` and `or` values to compute the `xor` and `xnor` of the operands.

The remainder of the section discusses the design of the rest of the stack for each EVE design. The first, second, and third subsections discuss EVE-1 circuit, EVE-32 circuit, and EVE-*n* circuit, respectively.

A. EVE-1 Bit-Serial Circuit

For bit-serial execution, EVE adopts the same circuit from BS-VRAM [6]. Although BS-VRAM does not support variable shift/rotation, the functionality can be supported through programming the circuits to perform multiple reads/writes to shift the values down the rows without any changes to the circuit. The circuit, shown in Figure 3(c), is composed of five different layers of logic: bus logic, XOR/XNOR logic, add logic, XRegister, and mask logic. Each column of the circuit operates independently to process 32-bit values over the span of multiple cycles. The add logic in each column employs a single block of a Manchester carry chain to perform a one-bit full addition and uses the `xor` and `xnor` of the operands along with the carry-in to calculate the sum and the carry-out of the addition. The XRegister stores the carry bit from the add logic facilitating a bit-serial addition. The mask logic contains a single latch to store the masking value for each column. The input to the latch can either be one of the values computed by the circuit or an input mask provided to the SRAM.

B. EVE-32 Bit-Parallel Circuit

The bit-parallel circuit in EVE heavily leverages the BP-VRAM [6]. However, as BP-VRAM does not support variable shifts and rotations, extra layers have been added to the stack to facilitate shift/rotation support. The EVE-32 bit-parallel circuit, Figure 3(d), is composed of six layers: bus logic, XOR/XNOR logic, add logic, XRegister, constant shifter, and mask logic. To support bit-parallel execution, sets of 32 columns are grouped together to process 32-bit values in parallel. For the add logic, a Manchester carry chain is used to propagate the carry of the addition and calculate the sum value. Each column contains a block of the Manchester carry chain. The XRegister is configured as a shift-right register spanning the 32 columns. This shift register aids in executing complex operations such as multiplication, division, and shift. The mask logic contains a single latch to store the masking value. As with bit-serial circuit, the input to the latch can either be the values computed by the circuit or an input mask provided to the SRAM through port.

In contrast to BP-VRAM, the EVE-32 bit-parallel circuit contains constant shifter logic, which can be used to execute variable shifts and rotations. After the 32-bit value is loaded, the shifter supports conditional one-bit shift-left and shift-right. The condition for the constant shifter is set to the mask of the column. By leveraging conditional one-bit shifts, a variable shift larger than one can be computed through binary decomposition of the shift amount. For each bit of the shift amount at index i, the constant shifter can perform multiple one-bit shifts adding up to 2^i. By iterating through the shift amount, the conditional one-bit shifts will eventually add-up to the shift amount over time.

C. EVE-n Bit-Hybrid Circuit

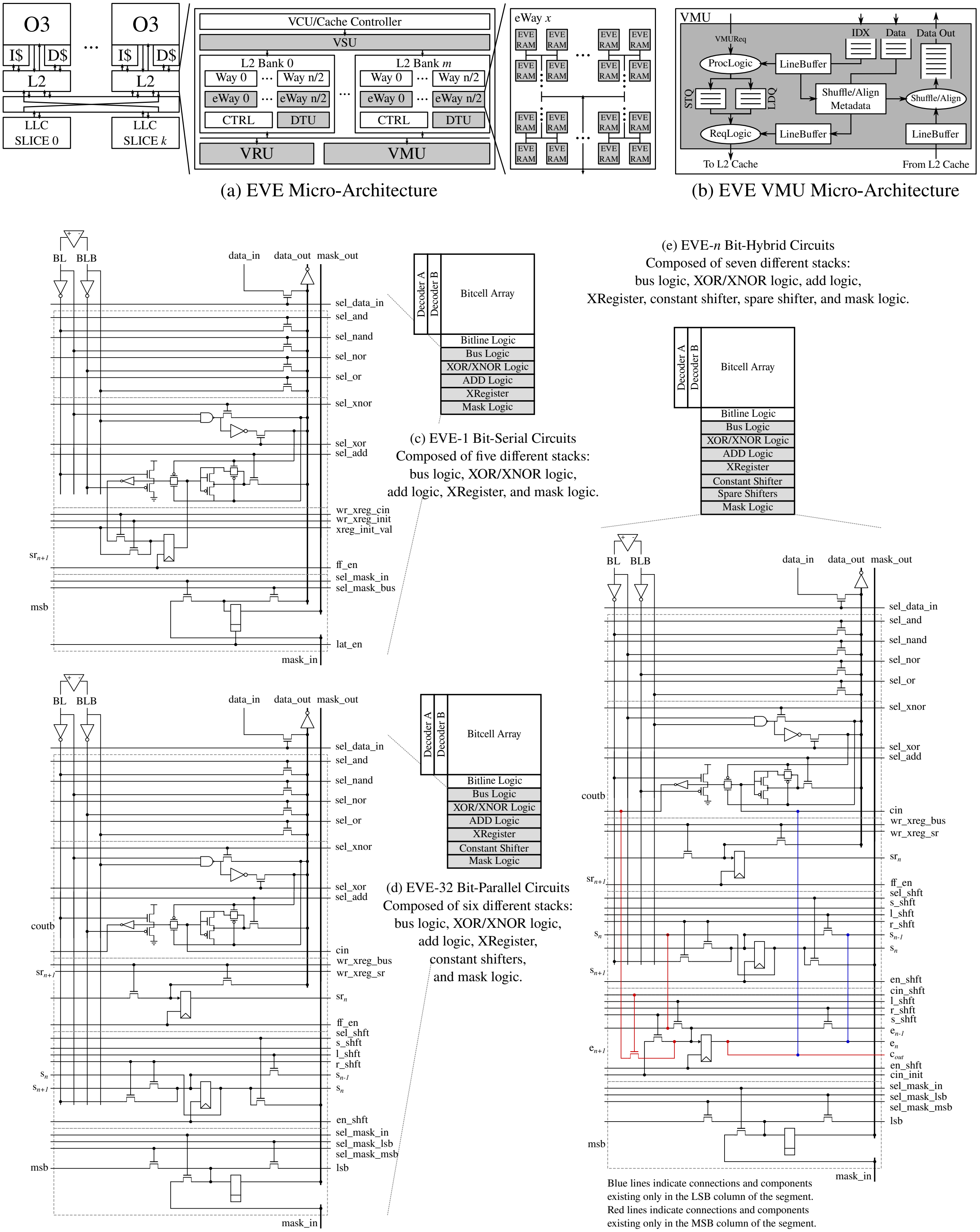
The circuit for EVE-*n* is set at design-time to target a fixed parallelization factor of n (where n is between 2 and 16). The EVE-*n* bit-hybrid circuit, shown in Figure 3(e), is composed of seven layers: bus logic, XOR/XNOR logic, add logic, XRegister, constant shifter, spare shifter, and mask logic. The circuit processes one n-bit segment from an element in parallel and processes each of the n-bit segments serially. Every n columns of the SRAM are grouped together to form an n-bit-hybrid execution hardware. The mask logic contains a single latch used to store a mask for a given column. The mask can be set to the XRegister value of either the most-significant column or the least-significant column of the segment. The mask latch can also be loaded with either a value computed by the circuit or an input mask.

For the add logic, an n-bit Manchester carry chain is used to perform full n-bit addition between two segments. The carry is then stored in one of the unused flip-flops in the spare shifter. Then, the flip-flop is wired to provide the stored carry as the carry-in for the Manchester carry chain when adding subsequent segments. Unlike the bit-serial circuit, the XRegister is no longer used to store the carry. As a result, the XRegister can be used as a shift-right register, similar to the XRegister in the bit-parallel circuit, to implement complex operations such as multiplication, division, and shift.

For shifts, the bit-hybrid circuit employs a constant shifter, similar to the bit-parallel circuit. However, the constant shifter processes one n-bit segment. To be able to perform shifts in a bit-hybrid fashion across multiple segments, the bit-hybrid circuit utilizes a newly introduced layer called the spare shifter. As the constant shifter performs either a left or right shift, the spare shifter performs either a right or left shift, respectively. As a result, the spare shifter can store the bits being shifted across different segments. A binary decomposition approach similar to that of the bit-parallel circuit can be used to perform the shifts efficiently. However, indices in the shift amount greater than or equal to $log_2(n)$ implies shifts that are integer multiples of the segment size. These shifts can be performed much more efficiently in the bit-hybrid approach than the bit-parallel approach by conditionally shifting a whole n-bit segment rather than performing conditional one-bit shifts.

IV. EVE Micro-Programming

To control the circuits detailed in Section III, EVE employs a micro-operation (μop) abstraction that enables writing micro-programs to achieve elaborate and more complex operations.



(a) EVE Micro-Architecture

(b) EVE VMU Micro-Architecture

(c) EVE-1 Bit-Serial Circuits Composed of five different stacks: bus logic, XOR/XNOR logic, add logic, XRegister, and mask logic.

(d) EVE-32 Bit-Parallel Circuits Composed of six different stacks: bus logic, XOR/XNOR logic, add logic, XRegister, constant shifters, and mask logic.

(e) EVE-*n* Bit-Hybrid Circuits Composed of seven different stacks: bus logic, XOR/XNOR logic, add logic, XRegister, constant shifter, spare shifter, and mask logic.

Blue lines indicate connections and components existing only in the LSB column of the segment. Red lines indicate connections and components existing only in the MSB column of the segment.

Figure 3. EVE General Overview.

TABLE II. SUPPORTED EVE MICRO-OPERATIONS

μOperation	Syntax	Description
read	rd a, src	read a into src
write	wr d, src	write src into d
blc	blc a, b	Bit-line compute of a and b
lshift	lshft	1-bit shift left
rshift	rshft	1-bit shift right
lrotate	lrot	1-bit rotate left
rrotate	rrot	1-bit rotate right
mask shft	m_shft	1-bit shift right the XRegister
cnt_init	init cnt, val	Initialize cnt to val
cnt_decr	decr cnt	Decrement cnt by one
bnz	bnz cnt, l	Branch to l if cnt is not zero
bnd	bnd cnt, l	Branch to l if cnt is a decade
ret	ret	Conclude execution

m = {msb, lsb, none}. *src* = {(n)and, (n)or, x(n)or, add, shift, data_in}.

Each μop takes a single cycle to execute and has well-defined inputs, outputs, and side-effects. This section discusses the micro-programming aspect of EVE. The remainder of the section is organized as follows: the first subsection details the different μops implemented by EVE; the second subsection explains how different macro-operations are implemented by sequencing μops.

A. Micro-Operations

Table II lists all the μops supported by the different EVE-*n* types. There are three types of μops: arithmetic, control, and counter. Arithmetic μops are executed by the circuits detailed in Section III. The other two categories are executed by the control logic. The arithmetic μops are as follows:

Read/Write (rd/wr a, [dst/src], mask) – These two μops represent the native SRAM read and write. The μops take as an operand a destination to store resulting read; or a source to the data for the write. Both operations include a mask as an operand to specify which columns in the SRAM are active or inactive to during the read/write execution.

Bit-Line Compute (blc a, b) – This μop performs a bit-line compute operation between wordline a and wordline b. It starts by enabling the two decoders in the EVE SRAM at the same time selecting wordline a and wordline b. Then, it reconfigures the sense-amplifiers in the single-ended mode. The sense-amplifiers will compute the bit-wise logical operations and feed these values to the rest of the EVE circuitry (as detailed in Section III).

Shift Left/Right ({l,r}shift) – This μop shifts left or right the content of the constant shifter by one bit. For EVE-32, only the constant shifter is shifted. However, for EVE-*n*, a spare shifter is shifted along side the constant shifter; whereas constant shifter is shifted left or right, the spare shifter is shifted in the opposite direction (i.e., right or left, respectively).

Writeback (wb d, src, mask) – After executing a blc, this μop reads the value computed by src and writes it back to the EVE SRAM at wordline d. The destination of the write back can also be specified as the mask registers (XRegister in bit-parallel and bit-hybrid; mask logic in bit-serial).

Mask Shift (mask_shift) – Conditionally executing μops is an essential functionality to implement more complex operations. To this end, for bit-hybrid and bit-parallel, the mask_shift μop allows manipulating a masking value through left shifts without the need for additional SRAM reads. The mask registers can be loaded through the writeback μop.

Each μop is used as op c, a, b: a and b are inputs and result is stored in c. addr_X is the row address of X. Labels are at beginning of line with colon. Control μops use labels as destination to redirect execution flow. <(X): indicates setting the data_in port to X. (;): indicates a mini-op composed of 2-3 μops.

```
 loop:
1 blc     addr_a, addr_b
  ; decr  seg_cnt[0]
2 wb      addr_c, add
  ; bnz   seg_cnt[0], loop
```

(a) add

```
1 rd      addr_b, mask
 iter:
2 blc     addr_c, addr_a
  ; decr  seg_cnt[0]
3 wb      addr_c, add, mask
4 rd      addr_c
5 wb      addr_c, add
  ; bnz   seg_cnt[0], iter
6 mask_shift
  ; decr  bit_cnt[0]
  ; bnz   bit_cnt[0], iter
7 rd      addr_b, mask
  ; decr  seg_cnt[1]
  ; bnz.r seg_cnt[1], iter
```

(b) mul

Figure 4. add and mul Macro-Operations.

EVE also includes 12 counters shared by all EVE SRAMs. These 12 counters are grouped in three groups: segment counters (seg_cnt[0-3]), bit counters (bit_cnt[0-3]), and array counters (arr_cnt[0-3]). Each group is initialized to a specific value, which is inferred from the EVE configuration and execution state: segment counters are initialized to number of segments; bit counters are initialized to the size of a segment; and array counters are initialized to the number of active arrays in EVE. When the value in any counter is decremented to zero, the counter is reset to its initial value. There are two sets of flags that track the state of each counter: zero flags, which track whether each counter has reached zero and was reset to its initial value; and binary decade flags, which record whether a counter has reached a binary decade. Counter μops are executed by a unified control logic to manipulate the counters. The first counter μop is: incr/decr cnt, which increments or decrements the counter specified by cnt. The second μop is: init_cnt, val, which forces an initialization of the counter cnt to the value val—a specified value inferred from the EVE configuration and execution state (e.g., number of segments, segment size, or active SRAM arrays).

To control which arithmetic μop is executed, a unified control logic in EVE executes control μops to manipulate a micro-program counter (μpc). There are two classes of control μops: conditional and non-conditional branches. Conditional branches inspect the corresponding flags for a specified counter and redirect the μpc depending on the condition (i.e., whether the counter has reached zero or a binary decade). The inspected flag for the conditional branch is reset when the branch is taken. Non-conditional branches change the μpc unconditionally. Control μops include a ret flag, which indicates whether the current micro-program is terminating execution and yielding to the next micro-program.

B. Macro-Operations

Incoming vector instructions are broken down into one or multiple macro-operations. Any of these macro-operations to be executed on the EVE SRAM are implemented as a micro-program (μprogram), which is a sequence of micro-operation tuples with a micro-program counter (μpc). Each tuple is composed of three μops, one from each of the three categories outlined in Section IV-A. These tuples are executed in the following order: counter μop, arithmetic μop, then control μop.

Figure 4 shows the bit-hybrid implementation of integer addition and multiplication. Integer addition, shown in Figure 4(a), is executed by performing bit-line computation between corresponding segments of the two input elements and then writing the results back into the destination. The carry between the segments is propagated through the XRegister in the circuits. The control required for the addition is straight-forward requiring a simple count-down loop that iterates over the segments. Integer multiplication, shown in Figure 4(b), requires more elaborate control with nested loops and different bounds for each loop. An outer loop (`iter`) handles predicated summation for calculating the multiplication (executed for N iterations, where N is the number of segments). The inner loop (i.e., `iter_add`) will perform a single addition (executed for n iterations).

V. EVE Micro-Architecture

This section introduces the micro-architecture of EVE, shown in Figure 3(a). To facilitate vector execution in EVE SRAM, a vector control unit (VCU) converts incoming vector instructions into one or multiple macro-operations. Depending on the type of the vector instruction, these macro-operations are broken down into a sequence of micro-operations by the help of either the vector sequencing unit (VSU), vector memory unit (VMU), and/or vector reduction unit (VRU). The stream of micro-operations are executed by EVE SRAMs. The VCU coordinates execution and communitcation between these units and the control processor.

A. *Vector Control Unit (VCU)*

When the control processor encounters a vector instruction, it is inserted in a special queue waiting to be sent to EVE. As EVE does not support precise exceptions, the vector instructions are *only* sent to EVE when they are ready to be committed. Once sent, the vector instructions are committed to unblock the commit logic. Then, these vector instructions are sent to the retire stage awaiting a response confirming their execution from EVE. If a writeback is to be expected from a vector instruction (e.g., `vmv.x.s`), the instruction instead stalls the commit logic awaiting a response from EVE with the value to be written back. To manage scalar-vector memory dependency and synchronization, a new vector memory fence instruction (`vmfence`) is introduced. The `vmfence` stalls the load-store-queue (LSQ) in the control processor from executing subsequent memory instructions until the vector fence is committed. Once at the commit stage and all pending scalar stores are performed, the `vmfence` is sent to EVE and stalls awaiting a response.

Once a vector instruction is received, depending on its type, the VCU creates one or more macro-operations. For non-memory and non-cross-element vector instructions, the VCU creates one macro-operation that is executed by the VSU. Once the macro-operation is sent to the VSU, the vector instruction is marked as performed and a response is sent back to the control processor. For memory and reduction/cross-element vector instructions, two macro-operations are created: one macro-operation is sent to either the VMU to initialize the memory operation or the VRU to initialize the reduction operation; the second macro-operation is sent to the VSU to perform the necessary reads and writes from and to EVE SRAMs. For vector memory fences, the VCU stalls waiting for the VMU to drain all pending loads and stores; then, the VCU executes the fence by simply sending a response to the control processor.

B. *Vector Sequencing Unit (VSU)*

The vector sequencing unit (VSU) decodes macro-operations into a sequence of μops primitives as explained in Section IV. The arithmetic μops are sent to the EVE SRAMs for execution as detailed by Section III, while the control μops are executed by the VSU. The VSU consists of a micro-program counter (μPC), a ROM containing the implementation of various macro-operations, and a set of counters. The μPC points to the current μop tuple being executed in the ROM. The set of counters help in executing the various control μops (e.g., `bnz` and `bnd`). The VSU adapts a VLIW-style encoding for the μops. Each cycle the VSU fetches a tuple consisting of three different μops: arithmetic μop, counters μop, and control μop. These micro-operations can be executed simultaneously. Executing control μops with the `ret` flag set will cause the VSU to stop executing the current vector instruction and return control back to the VCU.

C. *Vector Memory Unit (VMU)*

Executing memory vector instructions requires upwards of three macro-operations. One macro-operation is for the vector memory unit (VMU), shown in Figure 3(b), to generate the appropriate requests to the memory sub-system. Another one or two macro-operations are for the VSU to read indices required for the memory requests and to perform a read/write for the data from/to the EVE SRAMs. The VMU guarantees cache-line alignment for the generated requests. As for memory gather instructions (i.e., indexed loads), the VMU generates a request for each element and then performs the required gather operation to form a single line to be written back into the EVE SRAMs. An extra port in the TLB of the control processor is dedicated for the VMU to translate virtual addresses for incoming requests.

D. *Vector Reduction Unit (VRU)*

The vector reduction unit handles reduction instructions (e.g., `vredsum.vs`) as well as vector cross-element instructions (e.g., `vrgather`). Assuming `B` bits as the read/write bandwidth of the EVE SRAM, the VSU is able to stream $\frac{B}{n}$ elements (i.e., `E`) into the VRU for a given n-bit-hybrid configuration. The elements will be streamed one segment at a time over the span of $\frac{32}{n}$ for 32-bit element precision. The VRU contains `E` ports with each port having a fast lightweight detranspose logic. After all the segments are received, the VRU begins performing a dot-operation between the received `E` elements and the previously reduced `E` elements. Finally, once all elements have been streamed and the dot-operation between the different element groups has finished, the VRU starts a linear reduction operation. Then, the VSU issues a read μop to read the calculated reduction and either write it to a destination vector register or back to the control processor.

E. *Reconfigurability*

To support EVE, half of the private L2 cache ways are designed with EVE SRAMs instead of vanilla SRAMs. These ways operate as normal SRAMs performing reads and writes for L2. To spawn EVE, the associativity for the private L2 cache is reduced by half and the L2 cache is way-partitioned into: cache ways, and EVE ways. The cache ways constitute the new L2 and continue to service the core with no impact, aside from the halved associativity. The overhead of setting up EVE consists of reconfiguring the remaining EVE ways through invalidating cache lines residing in these ways. For dirty cache lines, the invalidation causes a writeback to the LLC. For clean cache lines, the invalidation causes the number of sharers to be decremented in the LLC, depending on the cache protocol. Since the cache hierarchy is inclusive, the invalidation and write-back to LLC should scale linearly with the number of cache lines (i.e., each cache line should incur constant number of cycles to invalidate in L1 and write-back to the LLC). To achieve this, a simple FSM machine and a counter can iterate through the reconfigured EVE ways causing the L2 to stall. However, the core can continue to be serviced from L1 as long as no misses are encountered. To reconfigure EVE ways back to L2 cache, there is no overhead and simply the cache associativity can be increased with all the cache lines returned to the L2 cache initialized as invalid.

VI. Circuits Evaluation

This section discusses the evaluation methodology and results of EVE's circuits.

A. *Methodology*

To evaluate the circuit of the different EVE designs with varying parallelization factors, we leveraged OpenRAM [27], which is an open-source Python-based memory generator. OpenRAM was modified to generate layout of an SRAM memory capable of bit-line computation on a 28nm technology node. Bit-line computation required an additional decoder to the SRAM as well as modifying the sense-amplifiers to support reconfigurable modes: differential and single-ended. A simplified version of the EVE-1 and EVE-32 circuits were implemented and used as an estimate for the proposed circuits. The simplified version lacked support for constant shifting; but, otherwise, matched the rest of the proposed circuits. The simplified circuits were implemented and laid out on the aforementioned 28nm technology node and they passed DRC and LVS checks. After the simplified circuits were verified through SPICE simulations, they were added as modules to OpenRAM. Then, OpenRAM was modified to include these circuits as part of the peripheral hardware for EVE SRAM. By leveraging the flexibility of OpenRAM, once OpenRAM was capable of generating EVE-1 and EVE-32 SRAMs, multiple designs of bit-parallel EVE SRAM (i.e., EVE-32) with varying bit-precision (b) were generated effortlessly (i.e., b = {2, 4, 8, 16, 32}). Each b bit-parallel EVE SRAM was used as proxy to analyze b-bit-hybrid EVE SRAMs as both designs have the same critical combinational path (i.e., carry propagation). Cycle-time for the EVE-n SRAM was estimated through the extracted netlist for the SRAM generated by OpenRAM. Area overhead was estimated by inspecting the layout of the simplified circuits adding estimated area from the missing stacks.

B. *Results*

OpenRAM was used to generate the layout of a 256 × 128 simplified EVE SRAM, which includes the simplified EVE circuit in its peripheral hardware. The layout passed the DRC and LVS checks for the 28nm technology node. Schematic circuits for the full bit-serial, bit-parallel, and bit-hybrid EVE configuration was built and used to verify correct functionality.

Table III. Simulated Systems

IO	Single-issue in-order RV64GC core: • L1I: 1-cycle-hit 4-way 32KB, 16 MSHRs • L1D: 2-cycle-hit 4-way 32KB, 16 MSHRs • L2: 8-way 8-bank 8-cycle-hit 512KB, 32 MSHRs
O3	Out-of-order 8-way RV64GC core: • Same L1I, L1D, and L2 as *IO*
O3+IV	Small vector unit integrated into O3: • Same L1I, L1D, and L2 as *O3* • IV: 4-element VL, out-of-order issue, 3 exec pipes
O3+DV	Decoupled vector engine connected to O3: • Same L1I, L1D, and L2 as *O3* • DV: 64-element VL, in-order issue, 4 exec pipes
O3+EVE	EVE engine connected to O3: • Same L1I and L1D as *O3* • L2: 4-way 8-bank 8-cycle-hit 256KB in vector mode • EVE-x: in-order issue, 1 exec pipe • VL (elements): • EVE-{1,2,4}=2048 • EVE-8=1024 • EVE-16=512 • EVE-32=256
LLC	Same for all systems: 16-way, 12-cycle-hit & 2MB, 32 MSHRs
Memory	Same for all systems: single-channel DDR4-2400

According to the layout, the 256 × 128 simplified EVE SRAM incurs 8.2% area overhead compared to a vanilla 28nm SRAM generated by OpenRAM. By estimating the additional stacks for the different designs, EVE-1 incurs 9.0% area overhead, EVE-x (bit-hybrid) incurs 15.6% area overhead, and EVE-32 incurs 12.6% area overhead. An EVE SRAM is composed of two banked 256 × 128 sub-arrays, further reducing the area overheads by half: EVE-1 (4.5%), EVE-x (7.8%), and EVE-32 (6.3%). As for the cycle-time, the baseline vanilla SRAM has a cycle-time of 1.025ns with the read logic being the critical timing path. By estimating n-bit-hybrid circuit through bit-parallel circuit with n-bit precision, our analysis shows that n-bit-hybrid (with $n <= 8$) has a cycle-time equivalent to that of the baseline with no penalty. However, targeting 16-bit-hybrid incurs a cycle-time penalty of about 15% (cycle time of 1.175ns). 32-bit-hybrid configuration further increases the overhead to 51% (cycle-time of 1.55ns).

Vanilla SRAM supports two basic operations: read, and write. EVE SRAM supports read and write basic operations as well as extra operations (shown in Table II). According to power analysis conducted on the extracted netlist from the layout, EVE SRAM energy for the basic operations (i.e., read and write) are similar to the vanilla SRAM. Other than bit-line compute operation (blc), the extra operations supported by EVE SRAMs incur much lower energy since no sense-amplifiers nor bit-line pre-charging is involved. Bit-line compute (blc) in EVE SRAM incur around 20% higher energy when compared to a read (the most energy-expensive operation in the vanilla SRAM). Despite the energy increase, EVE is energy-efficient since no energy-expensive data movements are needed through the H-tree.

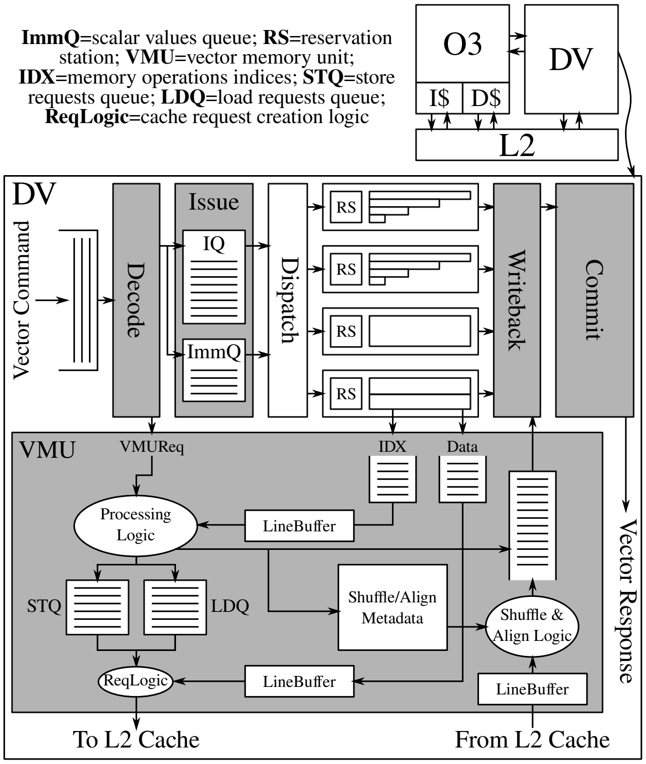


Figure 5. Overview of Decoupled Vector Engine (O3+DV).

VII. Architecture Evaluation

This section discusses the performance methodology and evaluation of the different EVE designs.

A. Methodology

To model the performance of the different EVE designs, we leveraged gem5 [7,36,59], which is a cycle-approximate simulator. We built cycle-approximate models for EVE-*n* and other baseline systems in gem5 and verified their correctness. In our modeling, we used two core types as the scalar baselines: out-of-order (O3) and in-order (IO). For the O3 core, we leveraged gem5's out-of-order core. As for IO core, we developed our own single-issue, in-order core. For the memory sub-system, we used ARM's CHI cache coherency model [25]. We modified ARM CHI to include special ports to connect vector units to either the L2 cache or the LLC, while modeling arbitration between the L1 and the vector unit.

To quantify the performance of EVE, we leveraged two scalar baselines and two vector baselines, detailed in Table III. The two scalar baselines are in-order core (IO) and out-of-order core (O3).The two vector baselines are: integrated vector unit added to an out-of-order core (O3+IV) (loosely based on [44, 58]) and decoupled vector unit (shown in Figure 5) added to an out-of-order core (O3+DV) (loosely based on [20]). IV shares three execution pipes with the control core (two floating-point/SIMD pipes, and memory execution pipe) and shares the load-store queue. Constant strides and indexed memory operations are decomposed to micro-operations and handled as scalar loads/stores by the load-store queue. As for DV, it has four execution pipes: simple integer, pipelined complex integer, iterative complex integer/cross-element, and memory execution pipe. DV includes a detailed model of a vector memory unit (VMU) capable of performing unit-stride, strided, and indexed memory operations. The VMU uses its TLB port to translate addresses for each generated cacheline memory request. Our model accounts for the request generation and address translation with one cycle and it assumes translated addresses always hit in the TLB.

We built a flexible cycle-approximate model of EVE in gem5 to calculate the performance of the different EVE-*n* designs. The model contains four units: vector control unit (VCU), vector sequencing unit (VSU), vector reduction unit (VRU), and vector memory unit (VMU). The control processor sends a vector request to EVE upon encountering a vector instruction in commit stage. The VCU handles the incoming vector requests as well as the outgoing vector responses. The VCU forwards the requests in the order received to the VSU and the VMU. The VSU performs a micro-decoding and starts executing the micro-program for the vector request. Each generated µops from the VSU is sent to the EVE SRAMs in one cycle. The VSU stalls special read/write µops from/to the DTU until the data is ready. Our model performs a separation between execution and timing by using the µops to estimate timing while execution for the vector instructions happens functionally.

We evaluated the performance of the different EVE designs on applications form Rodinia [11] and RiVEC [42] benchmark suites (shown in Table IV). The applications were vectorized manually using vector intrinsics in LLVM 13. The characterization of the scalar and vector versions of the applications is detailed in Table IV.

B. Results

Figure 6 shows the simulated performance (normalized to IO) of the different EVE designs against other baselines. Table IV details the performance of the different EVE designs normalized to different baselines. O3+DV achieves the best performance on the chosen data-parallel workloads, as expected. On average, O3+DV achieves 21.58× speedup over the in-order scalar baseline (IO). O3+IV achieves a speedup of 5.58× over IO and 1.74× speedup over out-of-order scalar baseline (O3). Among all the different EVE designs, EVE-8 achieves the best performance of 25.60× over the IO baseline, which is comparable to O3+DV performance. EVE-16 achieves the best next performance among EVE designs; however, EVE-16 suffers from a cycle-time penalty that affects its scalar performance. As a result, EVE-8 is a very compelling design point incurring modest overhead, yet achieving performance comparable to a decoupled vector unit.

EVE incurs a modest area overhead making it area-comparable to O3+IV. The VMU employed by EVE is similar and sized equivalently to O3+IV's VMU. EVE requires only eight data transpose units (DTUs). Each DTU is equivalent in size to half a sub-array. The ROM required for to implement the macro-operations is equivalent to one sub-array. In total, EVE incurs a 7.8% increase in the number of sub-arrays in the L2 cache (which contains 64 sub-arrays). Considering EVE-8, the circuits incur 7.8% area overhead, but since only half the SRAMs are EVE SRAMs, the circuit overhead is 3.9% . Overall, EVE-8 incurs a total area overhead of 11.7% . EVE in its best configuration can achieve 4.59× performance speedups over

TABLE IV. BENCHMARK APPLICATIONS

Name	Suite	Input	Scalar DIns	Scalar IOc	Vector (VL=64) DIns	VI%	ctrl	ialu	imul	xe	us	st	idx	prd	DOp	VO%	VPar	WInf	ArInt	Speedup vs. O3+IV DV	E-1	E-2	E-4	E-8	E-16	E-32	E-8 vs. E-1	E-32
vvadd	k	8.388M	75.5M	205M	1.6M	42%	20	20	0	0	7	0	0	0	35.5M	96%	22.6	0.47	0.33	3.64	3.19	3.23	3.24	3.28	3.23	3.38	1.03	0.97
mmult	k	1024	8.60B	48.5B	151M	44%	25	25	25	0	25	0	0	0	3.35B	97%	22.2	0.39	2.00	4.42	0.93	1.84	3.55	5.34	5.29	4.60	5.71	1.16
k-means	ro	10Kx34	2.11B	3.59B	51.5M	46%	1	52	18	∼0	∼0	10	7	1	1.53B	98%	29.8	0.72	2.44	2.28	1.22	1.32	1.35	1.86	1.82	1.51	1.53	1.24
pathfinder	ro	5Mx10	1.08B	2.43B	22.5M	50%	31	37	0	0	16	0	0	25	513M	97%	22.8	0.48	1.20	8.11	5.37	6.27	6.33	6.30	6.30	6.20	1.17	1.02
jacobi-2d	rv	2Kx10	1.59B	8.50B	35.4M	44%	8	50	8	17	7	0	0	0	940M	98%	26.6	0.59	4.50	6.36	6.18	9.50	11.30	13.49	13.50	12.69	2.18	1.06
backprop	ro	524K	1.17B	15.7B	21.5M	39%	13	19	25	∼0	5	12	0	0	488M	96%	22.7	0.42	1.00	2.14	2.01	2.05	2.07	2.07	2.06	2.06	1.03	1.01
sw	g	2070	1.38B	1.69B	20.4M	39%	10	55	0	11	10	14	0	10	433M	97%	21.2	0.31	2.75	3.44	2.43	3.97	5.32	6.21	6.14	5.08	2.55	1.22
geomean																				3.87	2.88	3.64	4.03	4.59	4.55	4.16	1.59	1.10

k = kernel, ro = rodinia, rv = RiVEC, g = genomics, DIns = number of dynamic instructions in ROI, IOc = number of cycles to run on an in-order core, VI% = percent of dynamic instructions that are of vector type, ctrl = vector control instructions, ialu = vector integer alu instructions, imul = vector integer multiplication and division instructions, xe = vector cross-element instructions, us = vector unit stride memory instructions, st = vector constant stride memory instructions, idx = vector indexed memory instructions, prd = predicated vector instructions, DOps = total number of operations (scalar instructions + vector instructions × active vector length), VO% = percent of operations performed by vector unit, VPar = logical parallelism (total ops / dynamic instructions in vectorized program), WInf = work inflation (total ops in vectorized program / dynamic instructions in scalar program), ArInt = arithmetic intensity (mathematical operations / memory operations) for vector unit, DV = O3+DV, E-x = EVE-x, geomean = calculated for: {k-means, pathfinder, jacobi-2d, backprop, sw}.

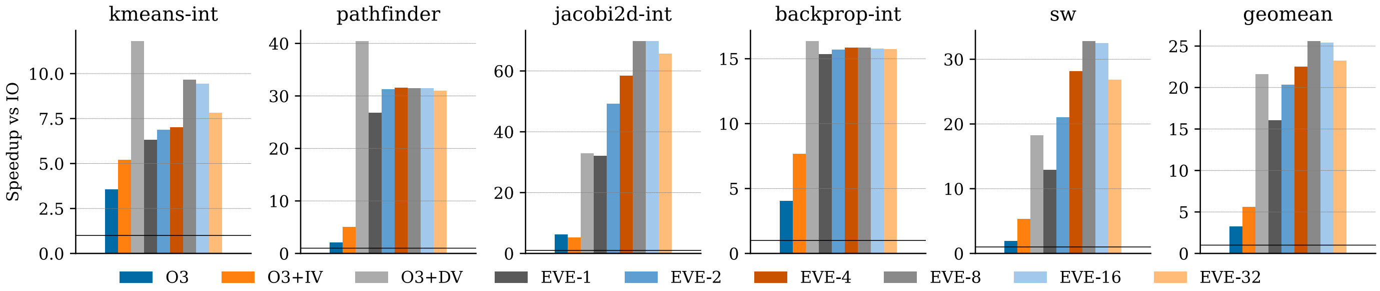


Figure 6. Performance Evaluation – performance of the different EVE configuration on applications from Rodinia and RiVEC benchmark suite normalized to the performance of an in-order core.

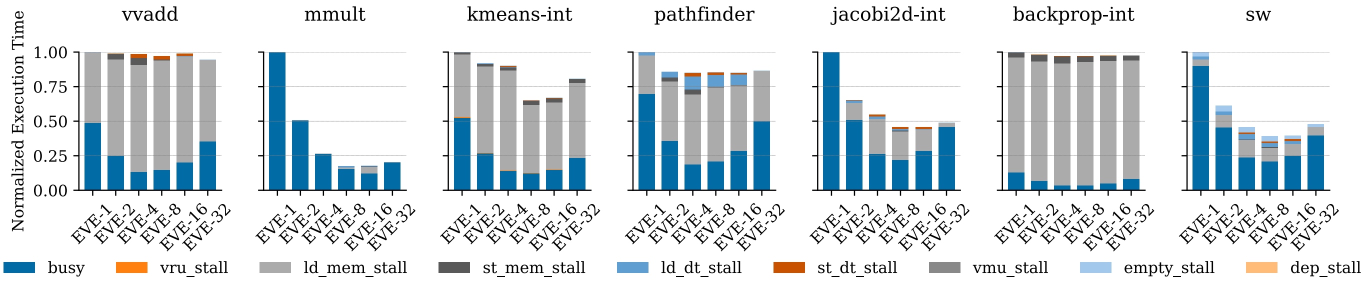


Figure 7. Execution Breakdown – profiling execution breakdown between different categories (normalized to EVE-1 execution time); busy=executing useful work; vru_stall=VRU structural hazard stall; ld_mem_stall=load memory stall; st_mem_stall=store memory stall; ld_dt_stall=load transposing stall; st_dt_stall=store detransposing stall; vmu_stall=VMU structural hazard stall; empty_stall=empty cycle stall; dep_stall=register dependency stall.

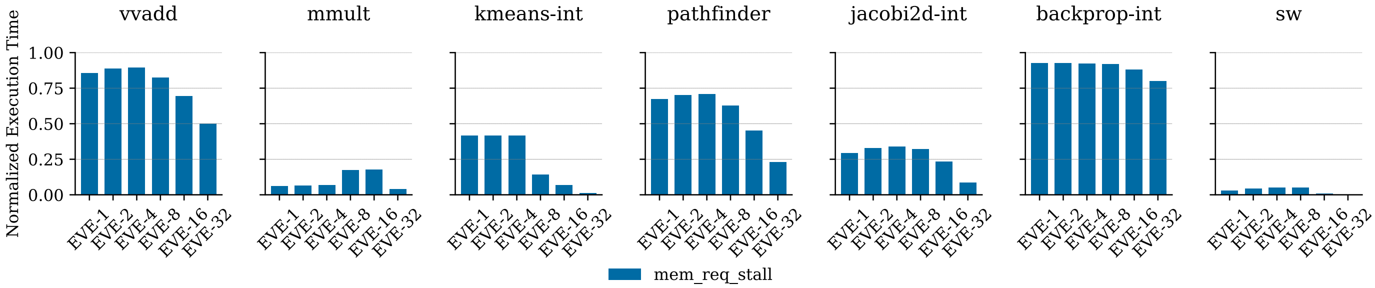


Figure 8. Cache-Induced Stalls in the VMU – showing the percentage of time the VMU is faced with a stall when sending a request to the LLC. These stalls do not necessarily cause a bubble/stall in execution as they can be hidden by overlapping outstanding compute in EVE.

the O3+IV. By achieving higher performance at a similar area overhead, EVE is more area-efficient than O3+IV.

Compute Throughput – Figure 7 shows the execution breakdown of different EVE designs running several applications. The execution breakdown, normalized to EVE-1, clearly shows the column under-utilization effect in S-CIM: *the percentage of time where EVE is busy executing useful work goes down as the parallelization factor increases until balanced utilization (i.e., EVE-4), then it starts to increase due to row under-utilization and slower clock*. The reason behind this behavior is: despite the hardware vector length being constant, the latency of the in-situ ALUs decreases going from EVE-1 to EVE-4 (with EVE-4 being balanced utilization), giving EVE higher compute throughput and thus requiring less time to perform the same amount of work; beyond EVE-4, row under-utilization coupled with slower clock (for EVE-16, and EVE-32 only) causes lower compute throughput and thus EVE requires more time to perform the same amount of work.

For applications that are memory-bound (i.e., *vvadd*, *pathfinder*, *backprop-int*), the execution time for the different parallelization factors is dominated by memory stalls. Therefore, compute throughput have a negligible impact on the overall performance. *vvadd* is inherently memory bound, while the others (as will be shown later) are faced with hardware limitations from the memory sub-system.

Transpose/Detranspose Overhead – Although EVE employs a very conservative transpose units, the transpose/detranspose overhead is minimal in most applications, as shown in Figure 7. One of the applications that experience measurable overhead from transpose/detranspose is *vvadd*, which is heavily memory bound. In *vvadd*, the bandwidth of transpose/detranspose in EVE fails to match the bandwidth of the data streaming from the cache sub-system. As a result, the transpose units start to induce negligible stalls. *pathfinder* is a the only application that show significant transpose/detranspose overheads. EVE struggles to overlap compute with transpose/detranspose and as a result, the execution stalls waiting for the data.

In EVE-1, the transpose/detranspose overhead is insignificant regardless of the application. Due to its low compute throughput, EVE-1 is able to overlap execution with the transpose/detranspose operation. This effect is more prominent in *mmult*, which is compute-bound. As the compute throughput increases starting from EVE-1 till EVE-4, the transpose/detranspose overhead increases while memory stalls are non-existent; EVE-32, however, requires no transpose/detranspose and, therefore, has higher performance despite the increase in its compute time. For *pathfinder* (which has a significant transpose/detranspose overhead), EVE-32 achieves higher performance despite its lowered compute throughput, also. In *sw*, the lack of transpose/detranspose operation for EVE-32 allows it to match EVE-16 performance while still having lower compute throughput and frequency.

Limited MSHR Effect – For applications dominated by memory stalls, as explained before, some are inherently memory-bound, while others are limited by the number of available MSHRs in the LLC. Figure 8 shows the percentage of execution time in which the VMU experiences a stall in issuing a cache request. Execution breakdown for *kmeans-int* indicates a significant amount of memory stalls; these stalls are explained by Figure 8 as it shows that the VMU experiences cache-induced stalls for almost half of the execution time between EVE-1 and EVE-4. In EVE-8, however, due to row under-utilization, the hardware vector length is halved and, thus, the required number of MSHRs is also halved. Indeed Figure 8 shows that the cache-induced stalls are reduced by more than half. As the hardware vector length is halved further for EVE-16 and EVE-32, the cache request stalls in the VMU keep on decreasing further. Although similar behavior is observed in *pathfinder* enabling EVE-8 in achieving the highest performance, the transpose/detranspose overhead is a bottleneck in achieving higher performance. Finally, *backprop-int* performs strided-memory operations with a very large stride. As a result, no two elements in these operations would reside in the same cacheline, and thus this application requires significantly more MSHRs than available. This is further shown in Figure 8 where the VMU experiences cache-induced stalls for more than 90% of the time while executing *backprop-int*. While these stalls are lowered as the parallelization factor increases and the hardware vector length is halved, the decrease in these stalls is very minimal and indicates a significant limitation on the number of available MSHRs.

Energy/Power Analysis – According our evaluation of EVE circuits, the peak power consumption of the SRAM arrays is expected to increase by 20% . Although this figure might look concerning, performing vector operations in EVE requires a mixture of multiple μops, and as such the expected power consumption overhead is lower than 20% . Previous work [2, 17, 23] have shown the energy efficiency of S-CIM execution, and more recent work [6] have shown that different execution paradigms have comparable energy efficiency. EVE leverages these energy efficient S-CIM execution techniques with bit-hybrid execution. Moreover, vector execution in EVE is energy-efficient as it eliminates the need for highly multi-ported vector register files that incur high access energy. Also, since the compute is fused with storage, there is no need for expensive redundant data movement from private L2 caches (through the costly H-tree) to these highly multi-ported vector register files, only to be read again to the functional units.

Area Efficiency Analysis – The best design point of EVE achieves comparable performance to O3+DV while incurring an area overhead equivalent to O3+IV. Compared to O3 core, the O3+IV baseline area is estimated to be 1.10× , while the O3+DV baseline area is estimated to be 2.00× . As for EVE design points, EVE-1 has an area estimate of 1.10× , EVE-2 through EVE-16 has an area overhead of 1.12× , and EVE-32 has an area overhead of 1.11× . The best EVE best design point, EVE-8, is able to increase performance by 4.59× at comparable area overhead to the O3+IV. EVE-8 also achieves over twice the area-normalized performance compared to O3+DV. EVE-8 is able to attain higher area-normalized performance than O3+DV by achieving comparable performance at a much lower area-overhead.

VIII. Related Work

Conventional vector processing architectures use dedicated long-vector engines to accelerate data-parallel computation [1, 16, 20, 60]. These machines achieve incredibly high performance on data parallel workloads at the expense of a significant cost in area. Subword packed SIMD is an attempt to reap some of the same benefits as conventional vector processing without paying the same area overhead. These designs re-use many of the scalar components of a processor to perform vector operations [29,40,53]. However, their performance is limited by the lack of available hardware units and limited instruction sets. Recently, next-generation scalable vector length ISA extensions which are flexible enough to support both of the traditional hardware paradigms have been developed [9, 12, 43, 46, 58]. These vector extensions also have the flexibility to support new developments in the vector processing space, like EVE.

Traditional processing-in-memory (PIM) techniques have primarily involved implementing computation logic in DRAM chips [18, 38, 39]. The goal of these implementations is to address the memory bandwidth wall by reducing the amount of data transfer across the system memory bus. Some recent variations have leveraged 3D manufacturing techniques to reduce the amount of change required to the physical design of the memory circuits [5, 21]. These designs aim to address the fact that modifying DRAM chips is a difficult proposition due to the incredibly high circuit density and desire for wide compatibility by implementing the compute circuitry on separate chips that can be assembled with 3D manufacturing techniques.

While traditional processing-in-memory (PIM) focuses on bringing compute closer to data, in-situ processing proposes to reconfigure the memory into compute engines achieving higher efficiency and performance. Rowclone [49] is among the earliest work to explore transforming DRAM into a compute engine capable of in-DRAM row cloning. Ambit [50] and DRISA [34] proposed transforming the DRAM into an accelerator capable of massive bit-wise logical operations between multiple rows. ComputeDRAM [24] explores implementing this technique with entirely stock DRAM chips.

While the work on in-situ processing-in-DRAM seems promising and can be explored beyond rudimentary bit-wise logical operations, special technology considerations and reduced margins-of-error make DRAM a difficult choice. Instead, SRAM is a more promising venue to explore in-situ processing. Jeloka et. al. [30, 31] introduces the bit-line compute technique, which performs digital bit-wise logical operations between SRAM rows. Bit-line compute constitutes an important precursor to subsequent processing-in-SRAM work. Compute caches [2] uses bit-line compute to transform the caches in a chip multi-processor into bit-wise logical compute engines.

Further work based on bit-line compute explored extending its functionality by utilizing a bit-serial approach to complex integer and floating-point operations. This work has demonstrated the ability to use bit-line compute to transform caches in a CMP into fixed-function accelerators for neural networks [17] and single-instruction-multiple-threads (SIMT) engine [23]. To mitigate transposing data, Wang et. al. [61] explored adding bit-line compute to an 8T-SRAM allowing the computation to be performed horizontally in the compute-bitline (CBL), while data read and writes are performed on the vertical bitlines. The 8T bitcell hinders its use in traditional caches due to low density. EVE leverages traditional 6T-SRAM, thus retaining high density and can be used in traditional caches. VRAM [6] explored utilizing bit-serial and bit-parallel execution paradigms to extend the functionality of bit-line compute. As bit-serial execution achieves high throughput but high latency and bit-parallel execution achieves lower latency at the expense of lower throughput [6], EVE proposes bit-hybrid execution to balance throughput and latency.

Another line of work [8, 26, 37, 62–64] on in-situ compute-in-memory explores leveraging associative compute abstraction [22, 47, 48]. Some work in this space proposes utilizing analog compute on emerging resistive technology [26,62,64] to perform the computations. However, analog compute accuracy suffers due to process variation. Other work utilizes CAM search/multi-write [8,37,63] to perform the computations, incurring extremely long latencies for the operations.

IX. Conclusion

This paper have demonstrated the ability of EVE to resolve the tension between performance and area in next-generation vector architectures. By employing S-CIM techniques, EVE is able to reconfigure private L2 caches into ephemeral vector engines with comparable performance to decoupled vector engines. EVE addresses the S-CIM programming challenge by adopting next-generation vector architecture. For the S-CIM serialization latency challenge, EVE mitigates the serialization overhead by proposing a novel bit-hybrid execution mechanism facilitating lower compute and memory latency. EVE serves as a motivation for future research in leveraging S-CIM techniques to alleviate the area-overhead associated with next-generation vector architectures.

Future research can explore techniques to further increase the utilization of the S-CIM arrays (e.g., dynamic micro-operation scheduling) with the help of an out-of-order core. Since EVE is an example of a new breed of very long next-generation vector machines, another line of research is to address the limited MSHRs efficiently to enable EVE to utilize memory bandwidth more effectively. EVE have shown the ability of bit-hybrid execution paradigm in balancing latency and throughput with focus on integer operations, future research can explore using bit-hybrid execution to balance latency and throughput for floating-point operations.

Acknowledgment

This work was supported in part by NSF PPoSS Award #2118709, NSF SHF Award #2008471, NSF E2CDA Award #1740136, the Semiconductor Research Corporation (SRC) as nCORE task 2758.002 and 2758.004, and the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a SRC program co-sponsored by DARPA, as well as equipment, tool, and/or physical IP donations from Intel, Synopsys, Cadence, and ARM. The authors acknowledge and thank Al Molnar for his advice on SRAM design, and José Martínez and Helena Caminal for useful discussion on various processing-in-memory architectures. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of any funding agency.

References

[1] D. Abts, A. Bataineh, S. Scott, G. Faanes, J. Schwarzmeier, E. Lundberg, T. Johnson, M. Bye, and G. Schwoerer. The Cray BlackWidow: A Highly Scalable Vector Multiprocessor. *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Nov 2007.

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. Compute Caches. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2017.

[3] A. Agrawal, A. Jaiswal, C. Lee, and K. Roy. X-SRAM: Enabling In-Memory Boolean Computations in CMOS Static Random Access Memories. *IEEE Trans. on Circuits and Systems I*, Jul 2018.

[4] A. Agrawal, A. Jaiswal, D. Roy, B. Han, G. Srinivasan, A. Ankit, and K. Roy. Xcel-RAM: Accelerating Binary Neural Networks in High-Throughput SRAM Compute Arrays. *IEEE Trans. on Circuits and Systems I*, Apr 2019.

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. *Int'l Symp. on Computer Architecture*, Jun 2015.

[6] K. Al-Hawaj, O. Afuye, S. Agwa, A. Apsel, and C. Batten. Towards a Reconfigurable Bit-Serial/Bit-Parallel Vector Accelerator using In-Situ Processing-In-SRAM. *Int'l Conf. on Circuits and Systems (ISCAS)*, pages 1–5, Oct 2020.

[7] 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*, 39(2):1–7, Aug 2011.

[8] 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. *Int'l Symp. on High-Performance Computer Architecture*, pages 557–569, 2021.

[9] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini. Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor with Multi-Precision Floating-Point Support in 22nm FD-SOI. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[10] N. Challapalle, S. Rampalli, L. Song, N. Chandramoorthy, K. Swaminathan, J. Sampson, Y. Chen, and V. Narayanan. GaaS-X: Graph Analytics Accelerator Supporting Sparse Data Representation using Crossbar Architectures. *Int'l Symp. on Computer Architecture*, May/Jun 2020.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, , and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. *Int'l Symp. on Workload Characterization*, Oct 2009.

[12] C. Chen, X. Xiang, C. Liu, Y. Shang, R. Guo, D. Liu, Y. Lu, Z. Hao, J. Luo, Z. Chen, C. Li, Y. Pu, J. Meng, X. Yan, Y. Xie, and X. Qi. Xuantie-910: A Commercial Multi-core 12-stage Pipeline Out-of-order 64-bit High Performance RISC-V Processor with Vector Extension: Industrial Product. *Int'l Symp. on Computer Architecture*, 2020.

[13] M. Cheng, L. Xia, Z. Zhu, Y. Cai, Y. Xie, Y. Wang, and H. Yang. TIME: A training-in-memory architecture for memristor-based deep neural networks. *Design Automation Conf.*, Jun 2017.

[14] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory. *Int'l Symp. on Computer Architecture*, Jun 2016.

[15] T. M. Conte, E. P. DeBenedictis, P. A. Garhini, and E. Track. Rebooting Computing: The Road Ahead. *IEEE Computer*, 50(1):20–29, Jan 2017.

[16] T. Dunigan, J. Vetter, J. White, and P. Worley. Performance Evaluation of the Cray X1 Distributed Shared-Memory Architecture. *IEEE Micro*, 25(1):30–40, Jan/Feb 2005.

[17] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer†, D. Sylvester, D. Blaauw, and R. Das. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. *Int'l Symp. on Computer Architecture*, Jul 2018.

[18] D. Elliot, M. Stumm, W. Snelgrove, C. Cojocaru, and R. McKenzie. Computational RAM: Implementing Processors in Memory. *IEEE Design and Test of Computers*, Jan-Mar 1999.

[19] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. *Int'l Symp. on Computer Architecture*, Jun 2011.

[20] R. Espasa, F. Ardanaz, J. Emer, S. Felix, J. Gago, R. Gramunt, I. Hernandez, T. Juan, G. Lowney, M. Mattina, and A. Seznec. Tarantula: A Vector Extension to the Alpha Architecture. *Int'l Symp. on Computer Architecture*, Jun 2002.

[21] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim. NDA: Near-DRAM Acceleration Architecture Leveraging Commodity DRAM Devices and Standard Memory Modules. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2015.

[22] C. C. Foster, editor. *Content Addressable Parallel Processors*. Van Nostrand Reinhold, 1976.

[23] D. Fujiki, S. Mahlke, and R. Das. Duality Cache for Data Parallel Acceleration. *Int'l Symp. on Computer Architecture*, Jun 2019.

[24] F. Gao, G. Tziantzioulis, and D. Wentzlaff. ComputeDRAM: In-Memory Compute Using Off-the-Sdelf DRAMs. *Int'l Symp. on Microarchitecture*, Oct 2019.

[25] Arm's AMBA 5 CHI Ruby Model in gem5. Online Webpage, accessed Nov 20, 2021. `https://www.gem5.org/documentation/general_docs/ruby/CHI/`.

[26] Q. Guo, X. Guo, R. Patel, E. Ipek, and E. G. Friedman. AC-DIMM: Associative Computing with STT-MRAM. *Int'l Symp. on Computer Architecture*, 41(3):189–200, 2013.

[27] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar. OpenRAM: An Open-Source Memory Compiler. *Int'l Conf. on Computer-Aided Design (ICCAD)*, Jan 2016.

[28] M. Imani, S. Gupta, Y. Kim, and T. Rosing. FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision. *Int'l Symp. on Computer Architecture*, Jun 2019.

[29] Intel SSE4 Programming Reference. Intel Reference Manual, 2007. `http://software.intel.com/file/18187`.

[30] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw. A Configurable TCAM/BCAM/SRAM Using 28nm Push-Rule 6T Bit Cell. *Symp. on Very Large-Scale Integration Circuits (VLSIC)*, Jun 2015.

[31] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T BitCell Enabling Logic-in-Memory. *IEEE Journal of Solid-State Circuits*, Apr 2016.

[32] K. Kitagawa, S. Tagaya, Y. Hagihara, and Y. Kanoh. A Hardware Overview of SX-6 and SX-7 Supercomputer. *NEC Research & Development Journal*, 44(1):2–7, Jan 2003.

[33] G. Li, G. Dai, S. Li, Y. Wang, and Y. Xie. GraphIA: an In-Situ Accelerator for Large-Scale Graph Processing. *Int'l Symp. on Memory Systems*, Oct 2018.

[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 IEEE/ACM 50th International Symposium on Microarchitecture MICRO, Boston, MA, USA*, pages 288–301, Oct 2017.

[35] W. Li, P. Xu, y. Zhao, H. Li, Y. Xie, and Y. Lin. Timely: Pushing Data Movements And Interfaces In Pim Accelerators Towards Local And In Time Domain. *Int'l Symp. on Computer Architecture*, May/Jun 2020.

[36] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, et al. The gem5 Simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152*, 2020.

[37] A. Morad, L. Yavits, S. Kvatinsky, and R. Ginosar. Resistive GP-SIMD Processing-In-Memory. *ACM Trans. on Architecture and Code Optimization*, 12(4):1–22, 2016.

[38] M. Oskin, F. T. Chong, and T. Sherwood. Active Pages: A Computation Model for Intelligent Memory. *Int'l Symp. on Computer Architecture*, Jun 1998.

[39] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, Mar 1997.

[40] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *Int'l Symp. on Microarchitecture*, Aug 1996.

[41] K. Qiu, N. Jao, M. Zhao, C. S. Mishra, G. Gudukbay, S. Jose, J. Sampson, M. T. Kandemir, and V. Narayanan. ResiRCA: A Resilient Energy Harvesting ReRAM Crossbar-Based Accelerator for Intelligent Embedded Processors. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2020.

[42] C. Ramírez, C. A. Hernández, O. Palomar, O. Unsal, M. A. Ramírez, and A. Cristal. A RISC-V Simulator and Benchmark Suite for Designing and Evaluating Vector Architectures. *ACM Trans. on Architecture and Code Optimization*, 17(4):30, Nov 2020.

[43] RISC-V Foundation. RISC-V "V" Vector Extension. *https://github.com/riscv/riscv-v-spec/releases/download/0.7.1/riscv-v-spec-0.7.1.pdf*, Jun 2019.

[44] J. Rupley, B. Burgess, B. Grayson, and G. D. Zuraski. Samsung M3 Processor. *IEEE Micro*, 39(2):37–44, 2019.

[45] R. M. Russel. The Cray-1 Computer System. *Communications of the ACM*, 21(1):63–72, Jan 1978.

[46] M. Sato. The Supercomputer "Fugaku" and Arm-SVE Enabled A64FX Processor for Energy Efficiency and Sustained Application Performance. *19th International Symposium on Parallel and Distributed Computing (ISPDC)*, 2020.

[47] G. E. Sayre. Staran: An Associative Approach to Multiprocessor Architecture. *Computer Architecture*, 4(2):199–221, 1976.

[48] G. E. Sayre. ASC: An Associative-Computing Paradigm. *Computer*, 27(11):19–25, 1994.

[49] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization. *Int'l Symp. on Microarchitecture*, Dec 2013.

[50] 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 IEEE/ACM 50th International Symposium on Microarchitecture MICRO, Boston, MA, USA*, pages 273–287, Oct 2017.

[51] X. Si, Y.-N. Tu, W.-H. Huang, J.-W. Su, P.-J. Lu, J.-H. Wang, T.-W. Liu, S.-Y. Wu, R. Liu, Y.-C. Chou, Y.-L. Chung, W. Shih, C.-C. Lo, R.-S. Liu, C.-C. Hsieh, K.-T. Tang, N.-C. Lien, W.-C. Shih, Y. He, Q. Li, and M.-F. Chang. A Local Computing Cell and 6T SRAM-Based Computing-in-Memory Macro With 8-b MAC Operation for Edge AI Chips. *IEEE Journal of Solid-State Circuits*, Apr 2021.

[52] W. A. Simon, Y. M. Qureshi, M. Rios, A. Levisse, M. Zapater, and D. Atienza. BLADE: An in-Cache Computing Architecture for Edge Devices. *IEEE Trans. on Computers (TOC)*, Feb 2020.

[53] A. Sodani, R. Gramunt, J. Corbal, H.-S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y.-C. Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, 2016.

[54] L. Song, X. Qian, H. Li, and Y. Chen. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2017.

[55] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen. GraphR: Accelerating Graph Processing Using ReRAM. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2018.

[56] N. Stephens. ARMv8-A Next-Generation Vector Architecture for HPC. *Symp. on High Performance Chips (Hot Chips)*, Aug 2016.

[57] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The ARM Scalable Vector Extension. *IEEE Micro*, Mar 2017.

[58] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker. The ARM Scalable Vector Extension. *IEEE Micro*, 37(2), 2017.

[59] T. Ta, L. Cheng, and C. Batten. Simulating Multi-Core RISC-V Systems in gem5. *Workshop on Computer Architecture Research with RISC-V*, 2018.

[60] S. Tagaya, M. Nishida, T. Hagiwara, T. Yanagawa, Y. Yokoya, H. Takahara, J. Stadler, M. Galle, and W. Bez. The NEC SX-8 Vector Supercomputer System. *High Performance Computing on Vector Systems*, May 2006.

[61] J. Wang, X. Wang, C. Eckert, A. Subramaniyan, R. Das, D. Blaauw, and D. Sylvester. A Compute SRAM with Bit-Serial Integer/Floating-Point Operations for Programmable In-Memory Vector Acceleration. *Int'l Solid-State Circuits Conf.*, Feb 2019.

[62] H. E. Yantır, A. M. Eltawil, and F. J. Kurdahi. Low-Power Resistive Associative Processor Implementation Through the Multi-Compare. *Int'l Conf. on Electronics, Circuits, and Systems (ICECS)*, pages 165–168, 2018.

[63] L. Yavits, A. Morad, and R. Ginosar. Computer Architecture with Associative Processor Replacing Last-Level Cache and SIMD Accelerator. *IEEE Trans. on Computers (TOC)*, 64(2):368–381, 2015.

[64] Y. Zha and J. Li. Hyper-Ap: Enhancing Associative Processing Through A Full-Stack Optimization. *Int'l Symp. on Computer Architecture*, pages 846–859, 2020.