CAPE: A Content-Addressable Processing Engine

Helena Caminal
Cornell University
hc922@cornell.edu

Kailin Yang
Cornell University
ky362@cornell.edu

Srivatsa Srinivasa
*Intel Labs**
srivatsa.rs@intel.com

Akshay Krishna Ramanathan
Pennsylvania State University
axr499@psu.edu

Khalid Al-Hawaj
Cornell University
ka429@cornell.edu

Tianshu Wu
Cornell University
tw387@cornell.edu

Vijaykrishnan Narayanan
Pennsylvania State University
vxn9@psu.edu

Christopher Batten
Cornell University
cbatten@cornell.edu

José F. Martínez
Cornell University
martinez@cornell.edu

***Abstract*—Processing-in-memory (PIM) architectures attempt to overcome the von Neumann bottleneck by combining computation and storage logic into a single component. The content-addressable parallel processing paradigm (CAPP) from the seventies is an in-situ PIM architecture that leverages content-addressable memories to realize bit-serial arithmetic and logic operations, via sequences of search and update operations over multiple memory rows in parallel. In this paper, we set out to investigate whether the concepts behind classic CAPP can be used successfully to build an entirely CMOS-based, general-purpose microarchitecture that can deliver manyfold speedups while remaining highly programmable. We conduct a full-stack design of a *Content-Addressable Processing Engine* (CAPE), built out of dense push-rule 6T SRAM arrays. CAPE is programmable using the RISC-V ISA with standard vector extensions. Our experiments show that CAPE achieves an average speedup of 14 (up to 254) over an area-equivalent (slightly under 9 mm^2 at 7 nm) out-of-order processor core with three levels of caches.**

***Index Terms*—Associative processing, associative memory, vector processors**

I. Introduction

Processing-in-memory (PIM) architecture proposals attempt to overcome the *von Neumann bottleneck* by combining computation and storage logic into a single component [15], [17], [18], [22], [29], [31], [34], [36], [44], [50]. In particular, *in-situ* PIM architectures leverage low-level computational abilities in the memory array itself [12], [16], [21], [30], [40], [42]. Content-addressable memories (CAMs) arguably constitute the first in-situ PIM architectures, as they have been around for more than 60 years [41]. They are equipped with additional logic per bitcell to perform *searches* to many cells simultaneously [35]. Content-addressable parallel processor (CAPP) designs from the seventies [20], [37], [39] extend CAMs with the ability to *update* multiple rows in parallel. By sequencing such search/update operations, CAPP designs can also perform a variety of arithmetic and logic operations (called *associative algorithms*) in a massively parallel and bit-serial fashion [20].

Recently, some interesting proposals have emerged that advocate for leveraging the foundations of CAPP in modern microarchitectures [33], [49], [51]. However, the proposed solutions entail emerging memory technology [33], [51] or expensive 12T memory bitcells [49], and they require either low-level programming [33], [49] or a restrictive programming language with a custom compilation flow [51].

In this paper, we set out to investigate whether the concepts behind classic CAPP architectures can be used successfully to build an entirely CMOS-based, general-purpose microarchitecture that can deliver manyfold speedups while remaining highly programmable. We conduct a full-stack design of a *Content-Addressable Processing Engine* (CAPE), built out of dense push-rule 6T SRAM arrays. CAPE is programmable using the RISC-V ISA with standard vector extensions [46].

We envision CAPE to be a standalone core that specializes in associative computing, and that can be integrated in a tiled multicore chip alongside other types of compute engines. Our experiments show that CAPE achieves an average speedups of 14 (up to 254) over an area-equivalent (slightly under 9 mm^2 at 7 nm) out-of-order processor core tile with three levels of caches accross a diverse set of representative applications.

The contributions of this paper include:

- A CMOS-based implementation of an associative-compute-capable engine based on dense 6T SRAM arrays.
- An optimized data layout on these SRAM arrays that maximizes operand locality.
- A microarchitecture organization that can perform data-parallel computations on tens of thousands of vector elements.
- A system organization able to perform efficient data transfers to maintain the benefits of its inherent massively parallel computational power.
- A mapping of the standard RISC-V ISA to this microarchitecture, which allows for generality, high programmability, and compatibility with existing compilation flows.

II. Associative Computing

An associative computing engine [20]: 1) stores data in vector form, 2) can compare a key against all vector elements in parallel (*search*), and 3) can update all matching elements in bulk with a new value (*update*). These operations are typically arranged in search-update pairs, and they are bit-serial, element-parallel—i.e., a search-update pair operates on

*Work done in part while a Ph.D. student at the Pennsylvania State University.

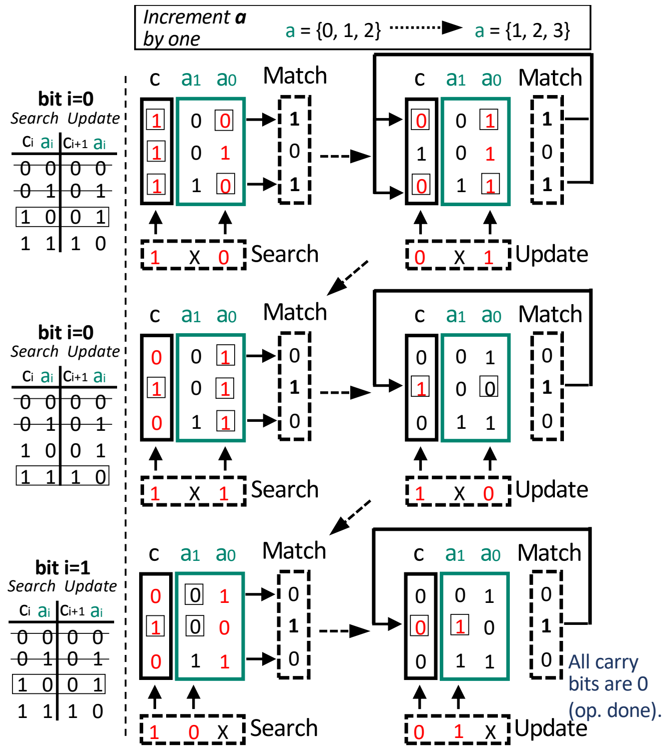

Fig. 1: Associative increment instruction as a bit-serial sequence of search-update operations to a vector of three two-bit elements. Carry bit column c is initialized to 1.

the same bit of all the elements of a vector (or several vectors). The sequence of search-update pairs that operate sequentially on all the bits of each vector value constitute basically an *instruction* in this associative computing paradigm. *Associative algorithms* are thus simply sequences of such instructions, much like a regular program. Consider a vector increment (Figure 1)—that is, all vector elements go up in value by one. An associative computing engine would first add 1 to the least significant bit of all vector elements and remember any carry. Then, for each element, it would add the corresponding carry to the next bit; and so forth. Of course, an associative computing engine cannot "add" bits per se. Instead, it implements bitwise addition through a sequence of search-update pairs that essentially follow the truth tables for a half adder, one bit combination at a time: 1) Search vector elements for which the i^{th} bit is 0 and the running carry for that element (an additional bit of storage) is 1, then bulk-update the i^{th} bit of matching elements to 1 and their running carry to 0. 2) Search vector elements whose i^{th} bit is 1 and the running carry for that element is also 1, then bulk-update the i^{th} bit of matching elements to 0 and the running carry to 1.

Note that, in the example above, we do not bother with search-update pairs for the two cases where the output is the same as the input—neither the element's bit nor the running carry flip as a result of applying the half adder truth table (crossed-out entries in the truth tables of Fig. 1). Note also that some additional support beyond search/update would be needed, namely: 1) We need two bits of additional storage per vector element: One bit to serve as the running carry (initialized to 1 at the beginning of the instruction with a single bulk-update), and one bit to "tag" matching elements (*Match*) in each of the two searches. Fortunately, these extra bits can be reused across vector element's bits (in fact, they can be reused across instructions, even if the vector names change). 2) In order to constrain searches and updates to the i^{th} bit of each element, we must be able to *mask out* the other bits. 3) The sequence of operations that implements the increment instruction needs to be "stored" somewhere (e.g., the micro-memory of a sequencer).

Associative computing was originally proposed as a bit-serial paradigm: for each bit it requires multiple search and update operations. Already for a relatively simple increment instruction on a 32-bit value this would represent over one hundred such operations. However, the key is to realize that this can be done *simultaneously to an extremely large number of vector elements*, and therein lies the power of associative computing. As our results will show, such vector-level parallelism more than makes up for the bit-serial nature of these operations. And our array organization enables, in some cases, bit-parallel associative instructions.

In addition to the massive data level parallelism, associative computing offers unique trade-offs compared to (more traditional) very long vector architectures. The associative computing paradigm evidently accelerates search and update operations commonly found in databases and text-based search applications (e.g., word count). Moreover, algorithms can be tuned to align to the strengths of associative engines. An illustrative example of this is the histogram benchmark (used in Section VI-E), which builds a histogram from pixel values found in an image. The thread-parallel code splits an image and distributes it to different threads that update a shared data structure for each pixel value. In turn, our vector code leverages very efficient searches to perform a brute-force sequence of searches for every possible pixel value (0 to 255), which leads to a speedup of 13× over an area-comparable baseline.

III. Overview of CAPE

Our goal is to leverage associative computing to deliver manyfold speedups while remaining highly programmable and general. We propose a *Content-Addressable Processing Engine (CAPE)*, an implementation of associative computing as an in-situ processor-in-memory (PIM) core that uses state-of-the-art CMOS technology, adopts a contemporary ISA abstraction, and can be readily integrated into a tiled architecture.

CAPE's architecture comprises four main blocks (Figure 2). The Control Processor (CP) is a small in-order core that runs standard RISC-V code with vector extensions [46]. It processes scalar instructions locally, and offloads vector instructions to the Compute-Storage Block (CSB), which acts as a coprocessor and is CAPE's associative computing engine. A vector instruction commits in the CP only after it completes in the CSB. In the shadow of an outstanding vector instruction, subsequent scalar logic/arithmetic ALU instructions may issue and execute (if not data-dependent with the vector instruction),

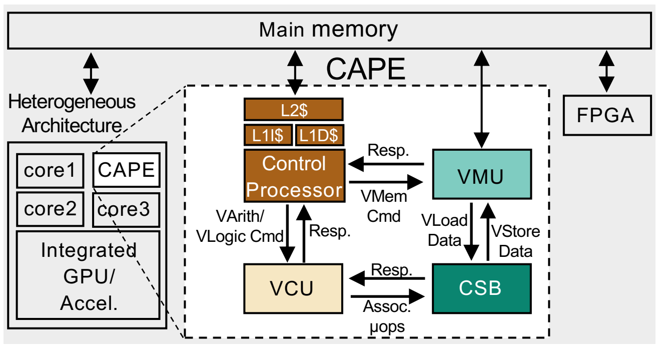

Fig. 2: CAPE System: Control Processor, Compute-Storage Block (CSB), Vector Memory Unit (VMU), and Vector Control Unit (VCU).

but not commit.[1] Subsequent vector instructions, however, stall at issue until the outstanding vector instruction commits.

Load/store vector instructions en route to the CSB go through the Vector Memory Unit (VMU). Other vector instructions go through the Vector Control Unit (VCU), which generates microcode sequences to drive the CSB and carry out the appropriate operations. VMU and VCU generate/transfer control/data signals to the CSB. The RISC-V vector register names in each instruction are used to index the appropriate vector operands within the CSB. These ultra-long vectors (order of 10^4 vector elements) are CAPE's primary source of parallelism.

The CSB is composed of tens of thousands of associative subarrays which can perform massively parallel operations. Each subarray is made up of 6T bitcells that can readily support the four microoperations used in CAPE's computational model: single-element reads and writes, as well as highly-efficient multi-element (vector) searches and updates. The RISC-V variable-length vector support allows for the very long vectors found in CAPE to be seamlessly leveraged by the applications.

IV. CAPE's Compute-Storage Block

In this section, we describe the low-level organization of CAPE's compute-storage block (CSB). First, we describe CSB's memory cell, a binary CAM which leverages a recent dense push-rule 6T SRAM design [26]. Then, we explain how these cells and data are arranged to optimize for the in-situ searches and updates that constitute the basis of associative computing. Finally, we describe CSB's support for reduction operations, which are a staple of many modern vector ISAs.

A. Cell and Subarray

Compared to standard 6T SRAM cells, traditional CAM cells require extra transistors and wires to enable content search [35]. Recently, however, Jeloka et al. propose a binary CAM (BCAM) based on push-rule 6T SRAM cells, which is able to perform reads, writes, and searches while maintaining the density of conventional SRAM [26]. The key difference

[1]As in many vector processing memory models, CAPE does not support store-load or store-store memory disambiguation between vector and scalar instructions—the compiler or programmer needs to insert memory fences as needed.

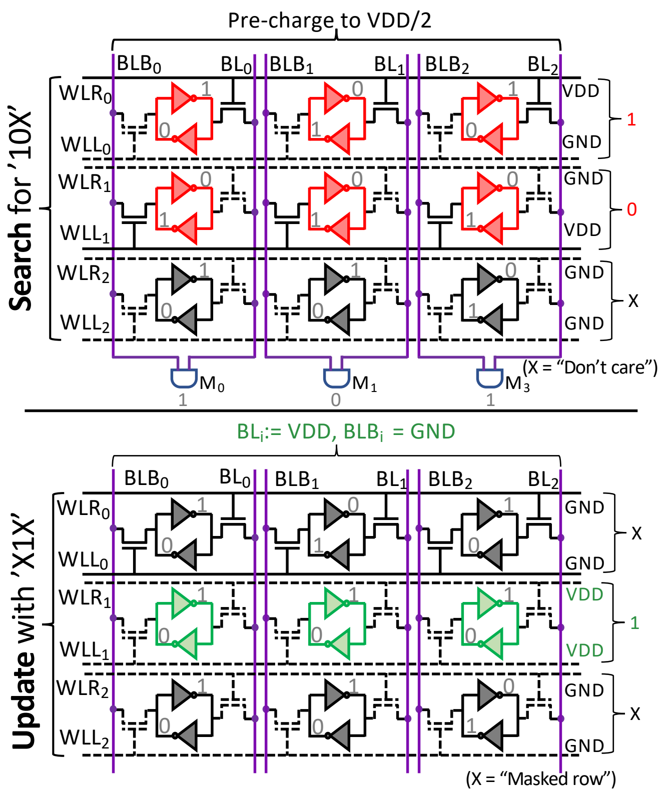

Fig. 3: Three-by-three illustrative 6-T SRAM array performing search (top) and update (bottom).

between this design and a conventional SRAM cell is that each row has two separate wordlines—wordline right (WLR) and wordline left (WLL)—, each connected to one of the access transistors of a cell. The design reuses the already existing wordlines as searchlines, and the bitlines as matchlines (the latter requires an AND gate per column).

Figure 3 is an illustrative example of a three-by-three 6T SRAM array with split wordlines performing search and update operations (read and write work as expected for a conventional SRAM). For a particular vector, it stores vector elements across columns; thus, different rows mean different bits of a vector element.

A search operation will look for matches in every column at the same time. In order to search for a 1, we set WLR to VDD and WLL to GND. To search for a 0, we set WLR to GND and WLL to VDD. To exclude a row from a search ("don't care"), we set both WLR and WLL to GND. At each column, ANDing bitlines BL and BLB yields the outcome of the search for each column: 1 for a full match, or 0 for at least one bit mismatch. To perform a bulk update across all columns, we assert both WLR and WLL of the active rows to be updated, and set all BL/BLB to VDD/GND or GND/VDD to write 1s or 0s, respectively.

B. Data Layout

We organize the CSB in subarrays of 32 by 32 cells (plus some peripheral logic, as we will see later). Further, we bit-slice each vector element across CSB subarrays of the same

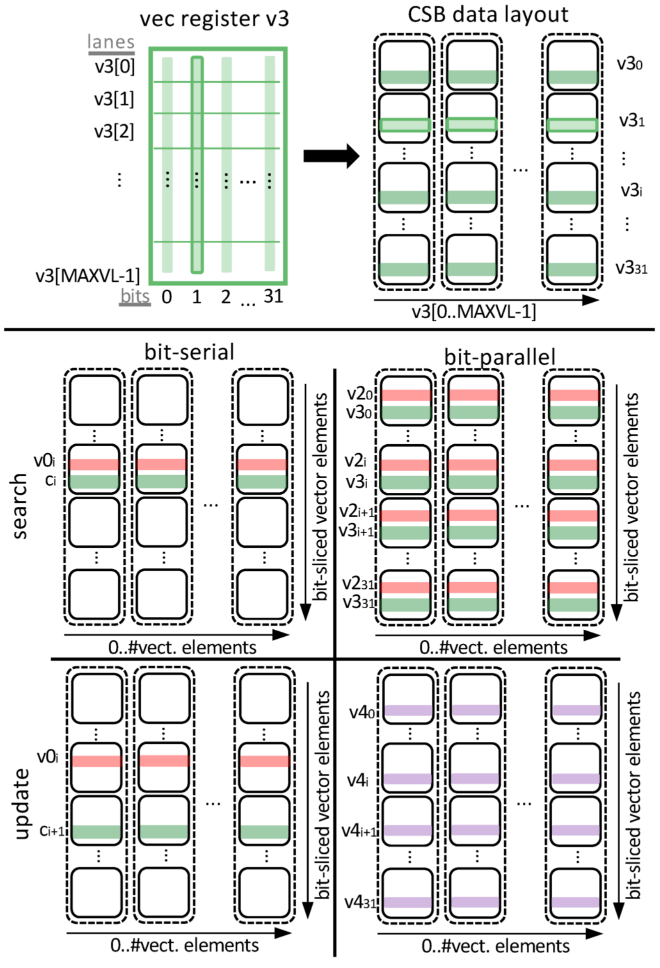

Fig. 4: Mapping of logical vector register to the CSB subarrays (top). Bit-serial (left half) and bit-parallel (right half) search and update bit-vector active operands on the Compute-Storage Block (bottom).

column, such that subarray i will store the i^{th} bit of the vector elements of *all* 32 RISC-V vector names for that column (Figure 4, top). Thus, each 32×32 subarray contains the i^{th} bit for 32 contiguous vector elements of all vector names. For example, subarray S_{ki} contains the i^{th} bit of $v0-31[32\cdot k]$, $v0-31[32\cdot k+1]$, ..., $v0-31[32\cdot k+31]$. The total number of subarrays in the CSB is the number of vector elements in a vector, times the bit width of each vector element, divided by 32.

The 32 by 32 geometry, combined with the bit-sliced data layout, allows CAPE to be clocked fast and minimize data movement: 1) We keep the access latency of a subarray low. 2) Further, a search-update pair that is part of a bit-serial instruction can be performed locally by the subarrays that contain the i^{th} bit of all the vector elements involved, and the other subarrays can be in sleep mode. Note that this minimizes data movement by ensuring operand locality across all vector names in the RISC-V alphabet, to the best of our knowledge, for the first time, in associative computing. 3) Finally, logic instructions (e.g., bitwise XOR) can be carried out in a bit-parallel fashion, thus involving all subarrays simultaneously.

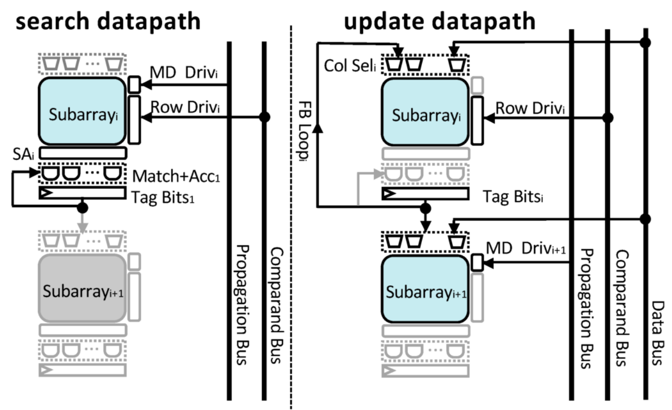

Fig. 5: Search (left) and update (right) datapaths in a cross-section of a CSB chain comprising two subarrays.

Figure 4 (bottom) is a simplified example of CAPE's CSB structure. The left half is performing a search-update pair as part of the increment instruction example of Section II. Each vector element is laid out vertically in a bit-sliced fashion, and for each vector its vector elements reside in different bit columns (some in different bit columns of the same subarray, and some in different subarrays). In the top-left quadrant, the search operation looks for a particular combination of bits $v0_i$ (data) and c_i (carry) on every vector element of $v0$ and c, respectively. Once the matching vector elements have been identified (which is recorded using tag bits, not shown), a bulk update (bottom) simultaneously updates bits $v0_i$ and c_{i+1} of every matching vector element. At each step, the subarrays not involved in the operation can potentially be placed in sleep mode. The right half of the figure shows another example involving a logic operation (e.g., $v4 = v2 \wedge v3$). As indicated before, logic operations can be carried out in a bit-parallel fashion, and thus all subarrays are involved.

C. Peripheral Logic

Each subarray contains the following peripheral logic (Figure 5): *Match generator (Match):* One AND gate per column to generate the match/mismatch signal (similar to Jeloka et al. [26]). *Tag bits:* One flip-flop per column, to store the output of the Match generator. *Tag bit accumulator (Accum):* One OR gate per column, to accumulate searches that update with the same values, as proposed by previous works [51]. *Feedback loop (FB Loop):* Used during updates, to transfer the match/mismatch mask generated by searches to the input of its own column driver (BL/BLB).

D. Propagation Chain

Typically, bit-serial instructions carry over information from one step to the next (e.g., carry in a bit-serial increment). Because we bit-slice vector elements, we need to support communication of such metadata vertically across consecutive subarrays, and the subarrays of a column thus form a *propagation chain*. In general, a chain will have as many subarrays as the bit width of a vector element. To support this, we add

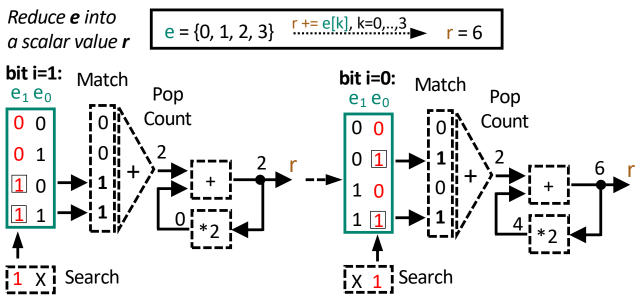

Fig. 6: Associative (bit-serial) reduction sum of a four-element two-bit vector. Bit-vectors are echoed through the tag bits by searching for value 1 to each column, from the most to the least significant bit. Then, the population count (pop count) of the tag bits is shifted by 2 (multiplied by 2) and accumulated at each step.

logic to optionally allow the tag bits of subarray i to select the columns of subarray $i+1$ that should be updated (Figure 5, right). This is how, in the increment example on Figure 1, the tag bits generated in the search can be used to select the vector elements to be updated for *both* subarray i (to update $v0_i$) and subarray $i+1$ (to update c_{i+1}) of every chain.

E. Supporting Reduction Sum Operations

Finally, we discuss CSB's support for reduction sum (redsum) operations, which aggregates the elements of a vector by adding them to produce a scalar result (Figure 6). This algorithm flows from the most to the least significant bits of the input, and the steps for each bit are: 1) search for '1' on each bit i (mask the rest); 2) the tag bits are reduced into an integer value (population count); 3) the output of the pop count is accumulated and multiplied by 2 at each step. CAPE supports redsum operations across chains, using external logic composed by: one pop count per chain, a left shift block (to multiply by two), an adder, and a register to store the scalar result. In Section VI, we give details on a specific redsum logic implementation used on a system made up of thousands of chains.

V. CAPE ARCHITECTURE

In this section, we describe the mapping of the RISC-V vector abstraction to the CSB (Section IV). We also describe the micro-architecture of the VCU (Section V-D) and VMU (Section V-E), which generate control commands for the CSB and enable efficient data transfers between the CSB and the main memory, respectively.

A. ISA

Vector architectures have been around for decades [19], and code vectorization is a well understood way to express data parallelism. This suggests that a vector ISA abstraction of the CAPE architecture is an attractive way to make CAPE highly programmable and versatile. Recently, the RISC-V Foundation has released a specification for RISC-V vector extensions [46]. Because of its increasing popularity, free availability, and support for vector length agnostic (VLA) instructions, we choose RISC-V as the ISA abstraction for our CAPE architecture.

	RISC-V vector Inst	Truth Table Ent.	Active Rows/Sub Srch	Active Rows/Sub Upd	Red Cycles (n bits)	Total Cycles (n bits)	Per-lane E (pJ)
Arith.	**vadd.vv**	5	3	1	0	$8n+2$	8.4
	vsub.vv	5	3	1	0	$8n+2$	8.4
	vmul.vv	4	4	1	0	$4n^2-4n$	99.9
	vredsum.vs	1	1	0	n	$\sim n$	0.4
Logic	**vand.vv**	1	2	1	0	3	0.4
	vor.vv	1	2	1	0	3	0.4
	vxor.vv	2	2	1	0	4	0.5
Comp.	**vmseq.vx**	1	1	0	n	$n+1$	0.4
	vmseq.vv	2	2	1	n	$n+4$	0.5
	vmslt.vv	5	2	1	0	$3n+6$	3.2
Other	**vmerge.vv**	4	3	1	0	4	0.5

TABLE I: Metrics of an illustrative subset of RISC-V vector instructions supported by CAPE. Left to right: instruction's mnemonic and mode, truth table entry count, active rows per subarray for search and update, reduction cycle count, total cycle count, and energy per vector lane (pJ).

RISC-V vector names map to the appropriate CAPE locations transparently through the VCU; the programmer never sees CAPE's CSB as addressable memory (although CAPE can be configured alternatively to be used as a memory-only tile by the chip, which we briefly address in Section VII). RISC-V's VLA support [46], whereby vector length is programmable, is easily supported in CAPE, by simply masking out the unused CSB columns or turning off entire chains. The flexibility that VLA support provides is actually key to CAPE's ability to accommodate a variety of applications with different amount of data-level parallelism.

Table I shows relevant metrics of an illustrative subset of RISC-V instructions supported by CAPE. Note that logic instructions are very efficient, because their execution is bit-parallel. Generally, arithmetic instructions are bit-serial due to the need to propagate carry/borrow information. Equality-comparison instructions map directly to CAPE's bit-parallel search operation (Figure 4, top-right). However, since each vector element is bit-sliced, there needs to be a bit-serial post-processing of each the tag bits in order to generate a single match/mismatch value. The maximum number of active rows/subarray during update and search illustrates that our circuits need only be able to search to at most four rows and to update to one row.[2] Note that arithmetic instructions (i.e. `vadd.vv`) will update to two subarrays simultaneously, but to only one row/subarray. The truth table entry count corresponds to the number of search-update pairs needed to execute per bit of the input operands; it is an estimation of the instruction's complexity. While some instructions have smaller truth tables than others, they may traverse them multiple times (for example, `vmul.vv` traverses its truth table a quadratically number of times, compared to `vadd.vv`).

While CAPE could support floating point with some modifications to the data layout and microcode sequences, we leave it for future work. With the current chain organization, CAPE could also handle element types smaller than 32 bits relatively easily, by configuring the microcode to handle sequences under

[2]This also the case for the RISC-V vector instructions not shown in Table I.

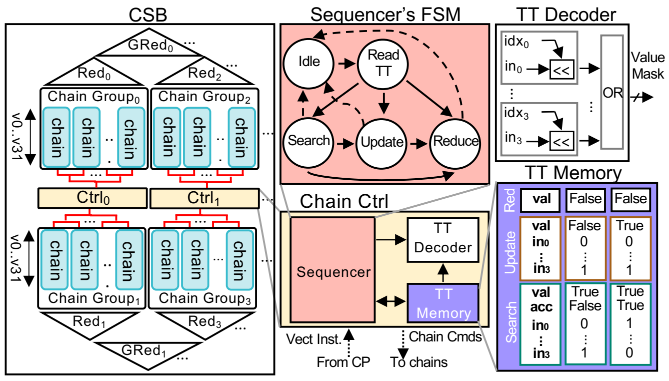

Fig. 7: CSB architecture (left), chain controller (bottom-middle), finite-state machine (FSM) (top-middle), truth-table (TT) decoder (top-right), and TT Memory (bottom-right).

32 bits, with support for padding/sign extension in some cases (also handled by the microcode).

B. CAPE Micro-architecture

Recall that the CAPE system is organized into four blocks (Figure 2) the control processor (CP), the vector control unit (VCU), the vector memory unit (VMU) and the compute-storage block (CSB). The CSB is made up of CAPE chains which have already been described in Section IV-D. In Sections V-D and V-E, we describe in detail the VCU and the VMU.

C. Exception Handling

Vector instructions are issued to the VCU/VMU only when they are committed at the end of the Control Processor's pipeline, and thus they are not subject to rollback due to exceptions triggered by earlier instructions. As for exceptions triggered by a vector instruction itself, a) load/store operations can be restarted at the index where a page fault occurred[3] and b) an arithmetic/logic exception would probably be handled imprecisely, consistently with the ISA specification for vector instructions.

D. Vector Control Unit

The vector control unit (VCU) breaks down each vector instruction into a sequence of commands. Commands include the four CAPE microoperations (read, write, search and update), as well as reconfiguration commands (e.g., to reconfigure the vector length). We implement a distributed design of the VCU, built from multiple chain controllers, shared across chain groups (Figure 7, left).

A global control unit maintains a programmable truth table memory and a set of control status registers (CSRs). When the VCU receives a vector instruction, it propagates the truth table data of the corresponding associative algorithm to each of the chain controllers which store it in a small, dedicated CAM (*global command distribution*).

[3]Vector-indexed loads/stores are more challenging and left for future work, and consequently, they have not been used in this work. *Software restart markers* may be a solution to this issue at a minimal performance overhead compared to imprecise solutions [24].

The chain controllers then distribute the commands to the apropriate subarray(s) in the chain (*local command distribution*). The chain controller (Figure 7, center) is composed of a sequencer, a *truth table memory (TTM)* and a *truth table decoder*. Each TTM entry corresponds to one search-update-reduce data *pack*, encoded efficiently to only store values for the bits involved in the operations. The entries in the TTM use a standard format to represent any associative algorithm's truth table. Four additional bits per TTM entry (valid bits and accumulator enable) are used to indicate if a search (with/without accumulation) or update operations are active, and if the reduction logic is going to be used.

The sequencer implements a simple FSM with five states (Figure 7, top center): (1) Idle, (2) Read TTM, (3) Generate comparand and mask for search, (4) Generate data and mask for update, and (5) reduce. The controller is by default in the idle state. Once the Control Processor sends a new request, the sequencer transitions into state (2). The controller keeps track of one counter, μpc, which helps navigate the entries in the TTM, and another counter, *bit*, to keep track of the bit we are operating on and generating the apropriate *idx* and *subarray select* signal for the chain controller. The counters are initialized appropriately: μpc=0 every truth table (TT) loop, and *bit* is set to either MSB or LSB, depending on the operation, given an operand size.

The truth table decoder (Figure 7, top-right) produces the search and update data and masks, from the values stored in the TTM by shifting them by the appropriate amount and ORing them to generate a single digital word to used by the subarray row and column drivers. This approach is similar to a vertical micro-code scheme. On a 32-bit configuration, the chain controllers distribute 143 bits of commands through the chain command buses, as shown in Figure 7.

E. Vector Memory Unit

CAPE communicates with the memory system via the Vector memory Unit (VMU). When receiving a vector memory instruction from the control processor, the VMU will break it into a series of sub-requests to main memory. Each sub-request accesses a block of memory of the memory system's data bus packet size. When the sub-request is served to the VMU, the CSB consumes it as follows: Similar to the byte interleaving scheme across different chips of a DRAM DIMM for optimal throughput, CAPE stores adjacent vector elements in different chains, which have the ability to perform the transfer independently, in a single cycle. This allows for the vector loads and stores to complete a full sub-request transfer in a single cycle. We design our system in order to ensure that the sub-request size is smaller than the total number of chains, so that sub-requests do not need to be buffered in the VMU. CSB reads and writes are concurrent to the main memory data transfers.

CAPE's CSB is cacheless: Due to the large footprint of the vector memory request and the limited temporal locality, it is not beneficial to have a data cache between CAPE and the main memory. As a result, the VMU is directly connected to the memory bus, and follows the same cache coherence protocol as the control processor's caches. Nonetheless, the cache coherence introduces very trivial performance overhead, since the CSB and the control processor share small amounts of data.

F. Reconfigurable Active Window

Set vector length: Variable-length vectors allow for applications to request a desired amount of data parallelism. In order to modify the vector length (`vl`), programmers can use the standard RISC-V instructions `vsetvl` or `vsetvli`, which will return the maximum amount of lanes supported by the hardware (`MAX_VL`) or the exact amount requested, if it is smaller than `MAX_VL`. In CAPE, that translates into using more or fewer columns, or even full chains. As supported by the RISC-V standard documentation, the elements in any destination vector register with indices greater than vl (tail elements) remain unchanged [46].

Set vector start: Similarly to `MAX_VL`, RISC-V's standard CSR `vstart` is used to specify the index of the first active element in a vector instruction.

CAPE support for the active window: Setting a `vl` smaller than its hardware limit `MAX_VL` will mask columns that are stored in different chains. To implement that, each chain controller locally computes a mask given its *chain ID*, the `vstart` value, the `vl` value. The mask is used in updates to generate the column signal: the address bus signals will contain zeroes on the masked columns. If all elements in a chain are masked, the chain controller can power-gate its peripherals while still maintaining the data stored unchanged.

G. Vectorizing for CAPE

Programmers can use RISC-V assembly, vector intrinsics or a vectorizing compiler to map well-structured data-parallel code to the CAPE instruction set. Many classic vector optimization techniques will directly apply to CAPE including loop reordering, loop restructuring, and memory access transformations [13], [32]. In this section, we discuss two CAPE-specific optimizations that can improve performance when compared to traditional vector architectures.

Vertical vs. Horizontal Operations – Traditional vector architectures discourage horizontal (i.e., cross-lane) operations since they are usually implemented using expensive and slow reduction trees. CAPE's horizontal operations use a combination of an intra-chain reduction sum (redsum) primitive and a modest global bit-serial reduction tree (see Section IV-E). The ability to bit-serially reduce all rows of all chains simultaneously results in performance roughly proportional to the bitwidth (we give implementation details of the reduction tree for a system of 1,024 chains in Section VI-C). A vector redsum instruction is thus eight times faster than an element-wise vector addition. This trade-off opens new algorithmic optimizations that favor using vector redsum instructions when possible.

Replica Vector Load – It can be challenging to fully utilize CAPE's long vector registers when applications operate over

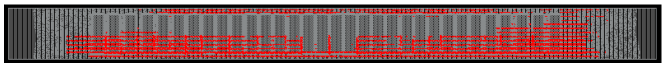
Fig. 8: Layout of one CAPE chain made up of 32 subarrays and its peripherals. Area is 13x175μm^2.

	Read	Write	Search 4 Rows	Update w/o Prop	Update w/ Prop	Red
D (ps)	237	181	227	209	209	217
BS E (pJ)	-	-	1.0	1.2	1.2	-
BP E (pJ)	2.8	2.4	5.7	3.8	-	8.9

TABLE II: Delay (D) and dynamic energy of bit-serial (BS E) and bit-parallel (BP E) microoperations executed by one chain. Note that read requires bidirectional communication.

matrices with a modest number of elements in each dimension. CAPE includes a new *replica vector load* instruction (`vlrw.v v1, r1, r2`) which loads a chunk of `r2` contiguous values, starting from the address in `r1`, and replicates them along the vector register `v1`. Replica vector loads are particularly useful when vectorizing dense matrix multiplication in three steps: (1) a unit-stride vector load reads multiple rows from the first matrix into one vector register; (2) a replica vector load reads a single row from the (transposed) second matrix and replicates this row into a second vector register; and (3) iterate over the rows and use `vmul` and `vredsum` to efficiently calculate the partial product.

VI. Evaluation

In this section, we discuss our circuit, instruction, and system modeling. Subsection A, *Microoperation modeling* provides delay and energy estimates for each CAPE microoperation on one chain. Subsection B, *Instruction modeling* combines these circuit-level estimates with an associative behavioral emulator to estimate the delay and energy for each vector instruction. Subsection C, *System modeling* integrates these instruction-level estimates into a cycle-approximate gem5-based [14] simulation model capable of executing binaries for both micro-benchmarks and complete applications. We use this multi-level modeling approach to explore system-level trade-offs.

A. Microoperation Modeling

We simulate a memory subarray of 32 columns by 36 rows (32 rows: 1 row/vector name, and 4 additional rows for metadata) based on the 6T bicell from Jeloka et al.'s design with split wordlines [26] (Figure 3). A CAPE subarray consists of SRAM bitcells, precharge circuitry, write drivers, search AND gates, tag bit accumulator and tag bits. All of these are designed using ASAP 7nm PDK circuit simulation libraries [45]. The latency and energy results incorporate wordline, bitline resistance and capacitances. We then model this subarray as a black box and instantiate it in the synthesized chain design using Synopsys DC compiler [7]. Synthesis results are further fed into an auto-place and route tool [9] for floorplan and placement to generate a chain layout (Figure 8). The control signals are routed to all the subarrays which are driven by wire repeaters to reduce the overall delay.

Delay of CAPE Primitives – Conventional wisdom might suggest that parallel microoperations (i.e. search and update) should be significantly slower (potentially 32 times since they might operate on 32 elements per chain) than reads or writes. In CAPE, both the circuit design and data layout enable very efficient searches and udpates, since they are done across columns (with their own independent circuitry) and not rows. Searches are only done to at most four rows simultaneously, which speeds up the sensing of the search outcome. Updates write to at most one row per subarray, which essentially turns them into single-row conventional writes. In addition, updates do not use a (priority) encoder or address decoder, but rather re-use the outcome of searches (stored in the tag bits) to conditionally update columns. Overall, CAPE's microoperation delays are balanced and range between 181 and 237 picoseconds (Table II). The reduced size of the SRAM arrays enables very fast accesses (90 ps). For that reason microoperation delays are largely dominated by the peripheral logic (i.e. AND and OR gates, flip-flop) and the local command distribution delay of the control signals (55 ps). Read is the slowest microoperation (Table II), explained by the round-trip wire delay: once to transfer the control signals to all subarrays, and another one to transfer back the data read to the controller.

Energy of CAPE Primitives – CAPE's operand bit-slicing across the subarrays in a chain forces reads and writes to access a single bitcell (same row and column) of all subarrays in a chain. In turn, the same data layout allows for search and updates to maintain most subarrays in a chain idle, reducing the dynamic energy. For searches, only one subarray/chain will be active (because of operand locality); and for update, only one or two (if propagation is needed) subarrays/chain will be active. We show dynamic energy estimates of a single chain in Table II, which include local command distribution of the 184 bits to all subarrays, array access, as well as peripheral logic energy consumption. We show estimates for dynamic energy of the bit-serial (*BS E*) and bit-parallel (*BP E*) flavours of each microoperation. Note that bit-parallel microoperations are very energy-efficient given the shared control logic and command distribution.

B. Instruction Modeling

We use the chain layout, delay, and energy modeling from the previous section and combine them with the associative behavioral emulator to derive detailed ISA instruction-level energy and delay modeling for an entire chain.

Delay of CAPE Instructions – The associative emulator models the associative behavior of subarrays with read, write, search and update capability. We implement the associative algorithms required for each vector instruction and extract microoperation mix count for a configuration of 32-bit operand.

Energy of CAPE Instructions – We combine the associative emulator's microoperation statistics with the microoperation energy modeling in Table II to estimate the energy of each CAPE instruction executing on a single chain.

In Table I we can see the energy spent for each vector instruction per *scalar* operation (that is, per vector lane). As expected, arithmetic instructions are the most energy consuming

	Baseline Core	**CAPE's Ctrl Processor**
System configuration	out-of-order core, 3.6GHz	in-order core, 2.7GHz
	32 kB/32kB/1MB L1D/L1I/L2	32 kB/32kB/1MB L1D/L1I/L2
	5.5MB L3 (shared), 512B LL cache line	512B L2 cache line
Core configuration	8-issue, 224 ROB, 72 LQ, 56 SQ	2-issue in-order, 5 LSQ
	4/4/4/3/1 IntAdd/IntMul/FP/Mem/Br units	4/1/1/1 Int/FP/Mem/Br units
	TournamentBP, 4,096 BTB, 16 RAS	TournamentBP, 4,096 BTB, 16 RAS
L1 D/I cache	8-way, LRU,MESI, 2 tag/data latency	8-way, LRU, 2 tag/data latency
L2 cache	16-way,LRU,MESI,14 tag/data latency	16-way,LRU,14 tag/data latency
L3 cache	11-way, LRU, 50 tag/data latency, shared	N./A.
Main memory	4H HBM, 8 channels, 16GBps/512MB per channel	

TABLE III: Experimental setup

explained by their large cycle count. Vector multiplication is clearly the most energy expensive instruction, it performs more than 3,000 searches and udpates, combined. Logic instructions (`vand`, `vor`, `vxor`) are very efficient, since they perform very few (bit-parallel) microoperations. `vredsum` includes the energy consumed in doing the bit-parallel search, 3.0 pJ, as well as the energy consumed by the reduction logic, 8.9 pJ.

CAPE Cycle Time – The system's critical path is 237 ps (4.22 GHz), wich corresponds to the slowest microoperation (read). We conservatively reduce by 65% the maximum CAPE frequency to 2.7 GHz to account for clock skew and uncertainty.

C. System Modeling

We use the modeling from the previous sections to derive our global reduction logic and command distribution models as well as our system-level simulation framework.

Reduction Logic – We have synthesized the global reduction logic described in Section IV-E for a system of 1,024 chains. The global reduction is pipelined into 5 stages with a critical path of 217 ps. We estimate the number of stages to model different CSB capacities by replicating or removing the different pipeline stages.

Global Command Distribution – It includes the delay between the VCU and each of the chain controllers, and it is estimated using a first-order approximation of wire delay on Metal 4 of an H-Tree that distributes the VCU signals control to each of the chain controllers, using wire repeaters to improve the delay. The global command distribution is pipelined and is not included as part of the cycle time: it adds a constant number of cycles of overhead per vector instruction.

System Methodology – We model the CAPE System by extending the gem5 cycle-approximate simulator framework [14]. The control processor is modeled using the RISC-V RV64G MinorCPU and is configured as a dual-issue, in-order, five-stage pipeline. We modified the MinorCPU to send commands to the VMU or VCU. The simulator accurately models the global reduction tree and command distribution delays. We have developed detailed timing models of CAPE's VMU, VCU, and CSB. The VMU is connected to an HBM memory system [27] to perform data transfers to/from the CSB. We model the CSB delays of each vector instruction as described in Section VI-B.

Area Reference – We want to be able to make area-equivalent comparisons. To that end, we estimate the area of our baseline out-of-order CPU based on a high-end Intel Skylake processor in 14 nm technology. Each Skylake tile contains a processor, more than 1MB of private caches, and

1.375 MB of shared LLC [11]. To scale down the tile area to 7 nm, we apply an estimated scaling factor of 1.8× based on the area ratio between 14 nm and 7 nm High-Density SRAM bitcells [1], [3]. Furthermore, we subtract the area for AVX and floating-point support. (Later in the section, we assess the impact of adding an aggressive SIMD engine to the baseline, using a commercial-grade model and assuming no extra area overhead.) As a result, we estimate that one tile's area is about 8.8 mm^2.

CAPE32k and CAPE131k – We choose two design points of CAPE – CAPE32k and CAPE131K, corresponding to two different available vector length MAX_VL: 32,768 lanes and 131,072 lanes. Their CSBs have 1,024 and 4,096 chains respectively, with 4.5MB and 18MB of capacity. From the microoperation modeling (Section VI-A), we know one chain in CAPE takes 2,434. Therefore, the CSB of CAPE32k including the pipelined reduction tree takes 2.8 mm^2, whereas the area of CAPE131k's CSB is 11.3 mm^2. We then estimate the area of CP based on an in-order ARM Cortex-A53 core [6]. One such core built in 16 nm takes 0.6 mm^2 whereas 512kB L2 takes 0.7 mm^2. We scale the area down to 7 nm by 2.74× based on 16 nm and 7 nm HD SRAM bitcell area [2], [3]. In total, the area of the CP with 1MB L2 is around 0.73 mm^2. Besides, we estimate the total area of micro-memory in VCU based on the truth table entry count of all the vector instructions that we support. Each entry requires twelve 7 nm SRAM bitcells (Figure 7). We find the total area requirement of VCU's TT memory is merely 0.002 and 0.007 mm^2 for CAPE32k and CAPE131k. Since the total area of CAPE32k's CSB, CP and TT memory is much smaller than one (area-reference) tile– 3.5 vs. 8.8 mm^2, we pessimistically assume that the difference in area is taken up completely by the VMU and VCU's sequencer and TT decoder. We apply the same area budget as in CAPE32k for the VMU and VCU ($8.8 - 3.5\,mm^2$) to similarly estimate the area of CAPE131k, and the total is below the area of two CPU tiles with caches (17.3 mm^2). We manually vectorize the applications using RISC-V vector intrinsics, which are ran for different CSB capacities (MAX_VL) without any code modifications.

Baselines – We choose our baselines to be general-purpose, area comparable to CAPE32k (and CAPE131k), processing engines: one (and two) RISC-V RV64G out-of-order and 8-issue cores (connected to the same HBM memory system [27] as CAPE) running sequential (and pthreads) versions of the apps. When running a parallel (pthreads) version, we indicate in our plots how many cores we use (up to three, based on the area study above). When we run sequential codes of the benchmarks on the same multicore machine, we utilizes the extended shared cache capacity of two other (idle) cores. Table III summarizes the architectural configuration.

Thermal Design Power – We derive the thermal design power (TDP) of CAPE32k using the results from the SPICE simulation and CACTI. For each step in the CAPE primitives, e.g., WL drive, BL precharge, etc, we calculate the power using its delay and total energy (dynamic plus leakage). The associative algorithms used by RISC-V vector instructions need to update 3 columns, search 32 columns, or read/write 32 columns. We find the peak power is achieved when writing to 32 columns (accounting 15.85 mW per chain). Thus, the TDP of CAPE32k with 1,024 chains is 16.23 W, including both leakage and dynamic power. Besides, according to the result from CACTI, the total leakage power of CAPE32k is 0.48 W. The total power of the reduction logic is 32.12 mW, and is power gated when performing a write operation (hence it does not contribute to the TDP). In order to estimate the peak power of the CAPE control processor, we reference to the TDP of a state-of-the-art processor based on 7nm technology that operates as the similar frequency as CAPE control CPU [4]. The processor consists of 32 out-of-order superscalar cores and three level of caches, and has 180 W TDP. We pessimistically estimate the TPD of the CP the same as one core, i.e., 5.63 W. We also pessimistically assume the VMU and VCU are both as power-hungry as the control processor. In total, the TDP of CAPE32k is 33.12 W.

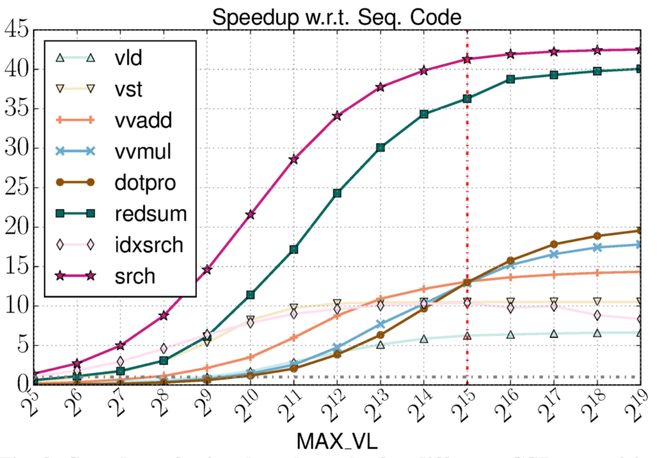

Fig. 9: Speedup of microbenchmarks for different CSB capacities. Vertical red line corresponds to CAPE32k.

D. Microbenchmarks

We first evaluate CAPE using eight microbenchmarks. Each benchmark loads one/two vectors with 524,288 32-bit elements each (which fits in the baseline's L3 cache). In Figure 9, we show the performance of CAPE for different CSB capacities (MAX_VL), normalized to the multicore (Table III) running a non-parallel sequential version of the microbenchmarks. For the baseline experiments, we warm up the caches before we start measuring performance.

Scalability Study– Memory-intensive benchmarks (*vld* and *vst*) show CAPE's ability to move data in and out of the CSB at different capacity design points. CAPE can achieve a speedup of 6.6 to 10.5 by efficiently moving large blocks of data from DRAM into the CSB with a single vector instruction, while the sequential baseline requires additional loop overhead and address calculation. In addition, the CPU still needs to serve requests across different levels of the cache hierarchy, even if L3 is warmed up.

Search-based benchmarks (*srch* and *idxsrch*) are representative operations of database workloads and text-parsing (i.e. word count) applications. Both perform constant-vector comparisons (`vmseq.vx`) to search a key in a vector. In addition, *idxsrch* performs a sequential post-processing for every matching element, with the intention to mimic the behavior of the text-parsing phoenix apps shown in Section VI-E. CAPE's ability to search efficiently enables a 42.5 speedup for *srch*, whereas *idxsrch*'s performance is eventually dominated by the sequential part of the algorithm for larger CSB capacities, achieving a speedup of 10× at MAX_VL=32k.

Arithmetic-intensive benchmarks (*vvadd*, *vvmul*, *dotpro*, and *redsum*) perform vector-vector addition, multiplication, multiply-accumulate, and reduction sum, respectively (besides loading/storing the input/output data). Their performance suggests that for moderate CSB capacities, CAPE's large data-parallelism is able to compensate for the bit-serial latencies. For very large CSB capacities ($\geq 2^{16}$), however, the overhead of global command distribution and data transfers limit their performance.

CAPE Roofline Model– To characterize CAPE's computational capabilities and scalability, we constructed a Roofline model [47] of CAPE at various CSB capacities (Figure 10), simbolized by different MAX_VL values. We adapt the traditional Roofline model metrics to capture CAPE's peculiarities. In the context of CAPE, a unit of work is defined as a vector element micro-operation (either a search or an update) performed on an element of a vector. The x-axis displays the element micro-operational intensity in vector element microoperations per byte of memory traffic between DRAM and the VMU, and the y-axis displays the attainable vector element micro-operational throughput in giga micro-operations per second. Higher intensity leads to higher utilization of data loaded into the CSB, and higher throughput suggests that the CSB is able to perform element micro-operations at a higher rate on average. The system's memory bandwidth is dominated by HBM's theoretical peak (128 GB/s). CAPE's theoretical maximum througput is obtained from a case that would execute 1 μop/cycle, without CP, VCU, and global command distribution overheads.

We can classify the microbenchmarks into two groups (Figure 10): *constant-intensity* and *increasing-intensity*, with increasing CSB capacity (MAX_VL). Most benchmarks fall in the first category, explained by the linear decrease in vector instructions as MAX_VL scales up. The second category only contains *idxsrch*, which still performs a serialized post-processing of each of the matches generated by the parallel search. As MAX_VL increases, the constant-intensity apps move from the compute-bound region to the memory bound region, and their throughput approaches the memory-bound roofline. This behavior suggests that constant-intensity apps are able to efficiently utilize CAPE's increasing computational cabilities. Ultimately, the speedup plateaus due to the limit of the peak memory BW. This phenomenon demonstrates CAPE's need for a high throughput memory system for large CSB capacities, justifying our use of HBM. In contrast, *idxsrch* remains in the memory-bound region for all MAX_VL, but far from the roofline peak throughput, indicating that it cannot fully utilize CAPE's increasing computational capacity. This matches the underwhelming scalability of *idxsrch* at larger MAX_VL (Figure 9).

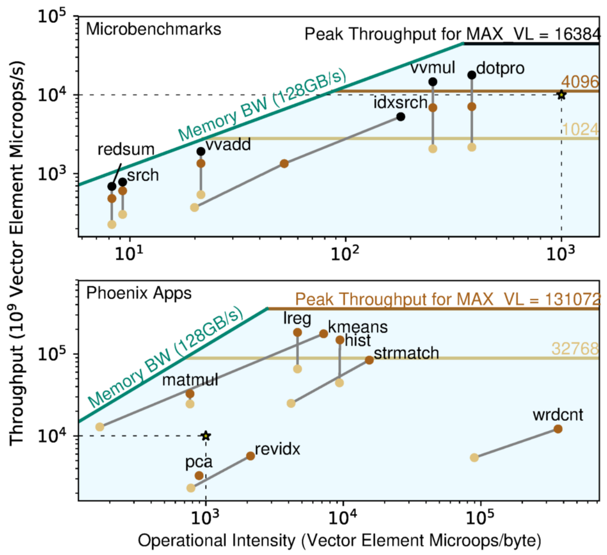

Fig. 10: Roofline plots of microbenchmarks and Phoenix apps for CAPE at various CSB capacities. The star is a random reference point to help reconcile the different axis scales across the two plots.

Application	Input Size	#Cyc. on One O3CPU	#Inst. of Seq. Code
Linear Regression	500MB	4.4 billion	3.8 billion
Histogram	1.4GB	13.6 billion	13.1 billion
Kmeans	100k	5.0 billion	6.6 billion
Matrix Multiply	1,000×1,000	7.0 billion	11.0 billion
PCA	1,500×1,500	16.6 billion	15.2 billion
String Match	500 MB	68.3 billion	52.0 billion
Word Count	10MB	4.9 billion	4.1 billion
Reverse Index	100MB	0.6 billion	0.9 billion

TABLE IV: Statistics of the Phoenix Benchmark Suite apps.

E. Phoenix Benchmarks

We use all the applications in the *Phoenix Benchmark Suite* [38] to evaluate CAPE's performance. Table IV shows the properties of each application.

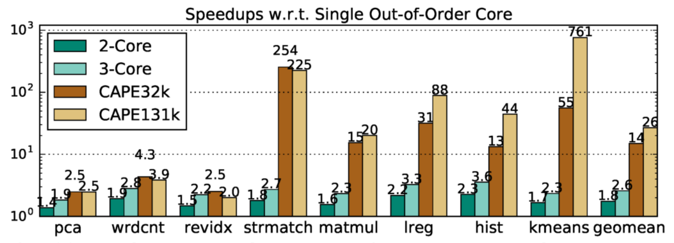

Fig. 11: Performance of the Phoenix Benchmarks for two- and three-core CPUs, CAPE32k and CAPE131k, normalized to a single CPU core. Single- and two-core CPUs are roughly area-equivalent to CAPE32k and CAPE131k, respectively.

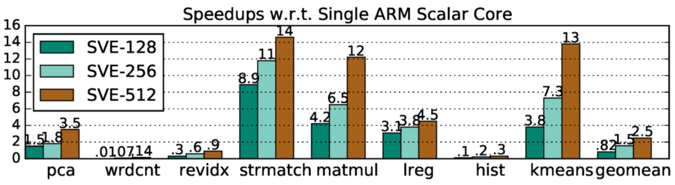

Fig. 12: Performance of the Phoenix Benchmarks for ARM SVE SIMD implementations of 128-, 256-, and 512-bit vectors, normalized to a single-core running ARM scalar code.

Results– In Figure 11 we show speedup of CAPE32k and CAPE131k, which have comparable area to one and two out-of-order cores with their caches (see baselines in Section VI-C). We also show performance of a three-core system, for reference. We see that CAPE32k accelerates all applications by 14× on average, compared to one core, at a similar area design point. Both matrix multiply (*matmul*) and PCA (*pca*) are matrix-based applications with relatively small input sizes. However, the for-loop inter-iteration dependencies found in pca prevented us from using the CAPE-specific instruction `vldr` (Section V-G) that increases CAPE's vector utilization, enabling a significant increase in parallelism necessary to compensate the bit-serial costly `vmul.vv` instruction.

CAPE131k accelerates the apps by 14.4× on average, compared to two cores, at a similar area design point. String match (*strmatch*), word count (*wrdcnt*) and reverse index (*revidx*) show worse performance, compared to CAPE32k. This scalability bottleneck is explained by the the sequential traversing of the input file, as well as the serialized post-processing of each match (similar to *idxsrch* of Section VI-D). In turn, the dramatic increase in peformance for Kmeans (*kmeans*) is due to its algorithmic nature. For CAPE32k, Kmean's dataset does not fit in the CSB, which results in having to load it multiple times. Instead, Kmean's dataset fits in CAPE131k's CSB, which translates into having to load it one single time and reuse it until the solution converges. In addition, the number of vector instructions inside the for-loops in the program is minimized due to the possibility to fully unroll all the iterations.

Comparison with SIMD baseline– One could argue that CAPE is a vector-first compute core, where non-vector instructions are supported by an adjoining scalar engine (CAPE's small Control Processor, already included in the area estimation and the simulations). In contrast, today's CPUs are typically scalar-first compute cores, where vector instructions may be supported by an adjoining vector engine (e.g., Intel AVX [10] or ARM SVE [43]). To tease out whether CAPE indeed constitutes an attractive compute tile for vectorizable code, we conduct an additional simulation experiment using a commercial-grade model of an ARM core with SVE support [43]. Specifically, we use ARM's upstream gem5 model [8], configured to match our RISC-V out-of-order baseline's size and latency (Table III), and equipped with four SIMD ALUs. We manually vectorize the applications using SVE intrinsics [5]. Although the standalone core in the ARM configuration is similar to that of our baseline, direct quantitative comparisons are tricky because of the different ISAs and compilation flows. Nevertheless, the results in Figure 12 shows speedups for the Phoenix Benchmarks running on the three SIMD configurations, normalized to a scalar-only run. The results in Figure 11 (CAPE32k vs RISC-V baseline) and Figure 12 (ARM+SVE vs ARM baseline) suggest that CAPE32k can achieve, on average, more than five times as much performance as the 512-bit SVE configuration (comparable to Intel's most aggressive SIMD implementation, AVX-512).

Roofline Study– Similar to the microbenchmarks section, we also plot the Phoenix apps using the Roofline model. By looking at Figure 11 and Figure 10, we can extract that the speedups of constant-intensity applications (*matmul, lreg, hist, kmeans*) improve from CAPE32k to CAPE131k. However, the speedup of variable-intensity applications (*wrdcnt, revidx, strmatch*) worsens; an exception is *pca*, whose speedup remained unchanged for reasons discussed in the last section and is reflected in its fixed position on the Roofline plot. *Kmeans*' change in intensity is explained by its algorithmic nature, previously discussed in the Results section. Unlike the other variable-intensity applications, its throughput on CAPE131k is much larger and closer to the compute-bound roofline, which leads to a dramatic increase in speedup: 426× with respect to an area comparable multicore system.

As expected, the throughput of constant-intensity apps shifts from compute-bound towards memory bound, explained by their vertical movement in the roofline space, with increased CSB capacity. This indicates that these apps are able to effectively utilize the increased computational capabilities of CAPE, until they are limited by the memory, which highlithgts CAPE's need for a high throughput memory system like HBM.

In contrast, the throughput of variable-intensity applications remain far from the memory-bound roofline, suggesting that they take advantage of the increased computational capability to a much lesser degree. Like the microbenchmark *idxsrch*, these text-based applications must sequentially traverse through the matches of parallel searches, and perform actions that are difficult to vectorize. As a result of Amdahl's law, any speedup from the vectorized regions is overshadowed by the cost of sequential regions, causing overall speedup to plateau. Coupled with increasing command distribution, the speedup in fact decreases as CAPE scales up.

VII. Memory-Only Mode

Although CAPE's primary mission is to implement a RISC-V vector ISA efficiently as an associate computing tile, CAPE's CSB could alternatively be reconfigured as storage by the chip whenever it may be more advantageous. In this section, we briefly outline three examples of CAPE's use as a memory-only tile; a detailed description of these or other possible mechanisms is out of the scope of the paper. In general, some additional support is needed to accept external requests.

Scratchpad: A scratchpad is simply a block of physical memory which can be typically accessed directly using ordinary loads and stores (i.e., mapped into the virtual addressing space). In a multicore chip, a scratchpad may be useful, for example, to store private data or to exchange noncacheable data across cores. To support this mode, all is needed is for the VMU to be able to take in memory requests from remote nodes through the system interconnect and perform the appropriate physical address indexing.

Key-value storage: The scratchpad above can be further customized to operate as key-value storage, which is simply a repository of key-value pairs, where a value can be read from or written to by first finding its unique key (or, if it is not found, by first allocating a new key-value pair). Because the CSB is content-addressable, it naturally supports this mode. Assuming, for example, that both key and value are 32-bit wide, and that each CSB chain is made up of 32 subarrays, then a chain can store $16 \times 32 = 512$ key-value pairs (that's about half a million key-value pairs in the smaller CAPE configuration of our evaluation, CAPE32k). Again, as in the case of the scratchpad, the VMU should be able to take in key-value requests from the system, and contain the appropriate indexing logic. To insert new key-value pairs, the VCU may assist by running a microprogram that scans the CSB looking for free entries, and/or the control processor may execute a small program that maintains a free list.

Cache: The CSB can leverage key-value storage functionality to work as a shared victim cache of the L2 caches, an additional slice of the LLC, etc. To do this, the control processor and the VCU should be programmed to work closely with the controller of the cache it is augmenting (e.g., on a miss, an L2 cache controller sends a message to the CAPE tile to check if the block is present in the victim cache CAPE is emulating, concurrently to initiating an LLC access). In one possible implementation, each cache line (tag and data) are stored row-wise (since cache blocks can be fairly large); neither tag nor data are bit-sliced. Jeloka et al.'s row read/write operations [26] take one/two cycles, respectively. Since the CSB has 32 rows of subarrays, and each subarray has 32 rows of bitcells, CAPE as a cache can support up to ten index bits in the address (1,024 rows). An access to the CAPE cache can be carried out with a few microinstructions that search for a tag match among a set of rows and, if a hit is found, command the VMU to deliver the data block.

VIII. Related Work

PIM architectures bring logic close or into memory to mitigate the effects of data movement [15], [17], [18], [22], [29], [31], [34], [36], [44], [50]. Recent in-situ PIM solutions are based on bitline computation, enabled either on DRAM [30], [40] or SRAM [12], [16], [21] memory technology. CAPE's associative processing provides an alternative to bitline computation. Most in-situ PIM proposals require a custom ISA or ISA extensions [12], [16], [40], but Fujiki et al. proposes to use SIMT as programming abstraction [21]. Our system provides a *direct* map to standard RISC-V vector code [46], able to reuse existing compilation flows.

Recently, some proposals [23], [33], [49], [51] have emerged proposing to utilize the foundations of traditional associative processing [20], [37], [39] for modern microarchitectures. However, these solutions require emergent technologies [23], [33], [51], custom compilation flows [23], [51], expensive 12T memory bitcells [49], or data copies to ensure operand locality in the associative arrays [51]. Moreover, [23] does not perform in-situ associative processing, like CAPE. Other emergent resistive devices with support for searches do not leverage associative computing [25], require sophisticated technology scaling techniques to preserve the accuracy [48], or represent an alternative architectural design point as processing-in-storage devices. [28].

Zha et al. propose an associative processor based on RRAM crossbar memory arrays [51]. Their approach is fairly different from CAPE. First, their algorithmical optimization relies on a TCAM array, which has a reduced capacity. While Zha et al.'s compiler is constrained by common SIMD vectorization limitations (e.g. they need to know the unrolling factor at compile time, no divergence support), CAPE reuses the standard RISC-V's vector abstraction resulting in a clean interface compatible with existing code and compilation flows (An example of it is its vector length agnostic paradigm, which allows the unrolling factor to be determined at runtime.). Finally, Zha et al.'s performance evaluation uses a simplistic fixed-latency timing model. This methodology fails to capture the interactions between the memory system or the task dispatching from the host.

IX. Conclusions

In this paper, we explored whether the concepts behind classic associative processing can guide the design of a next-generation, general-purpose CMOS microarchitecture that can deliver order-of-magnitude speedups while remaining highly programmable.

To do that, we conducted a full-stack design of a CMOS-based implementation of an associative engine based on dense 6T push-rule SRAM arrays. The resulting CAPE microarchitecture is scalable to data-level parallel computations on tens of thousands of vector elements, and it is fully programmable via a RISC-V ISA with standard vector instructions. Our evaluation shows that CAPE achieves average speedup of 14 (up to 254) across eight diverse applications, relative to an area-equivalent (slightly under $9\,\mathrm{mm}^2$ at 7 nm) out-of-order processor core with three levels of caches.

Acknowledgment

This work was supported in part by the Semiconductor Research Corporation (SRC) through the Center for Research on Intelligent Storage and Processing-in-memory (CRISP) and the Center for Applications Driving Architectures (ADA), two of six centers of the JUMP program co-sponsored by DARPA; by SRC and NSF through an E2CDA NSF Award #1740136, by NSF Award #2008365, and by NSF SHF Award #2008471. We thank Olalekan Afuye and Alyssa Apsel for early discussions and help with circuit design; Angela Jin, Ysabel Tan, and Socrates Wong for their help with experiments; Giacomo Gabrielli and Giacomo Travaglini for their help with some of the Arm tools; Michael Woodson for his technical support; and Ameen Akel and Sean Eilert from Micron Technology for their advice.

References

[1] "14 nm lithography process," https://en.wikichip.org/wiki/14_nm_lithography_process.
[2] "16 nm lithography process," https://en.wikichip.org/wiki/16_nm_lithography_process.
[3] "7 nm lithography process," https://en.wikichip.org/wiki/7_nm_lithography_process.

[4] “Amd epyc 7502 processor,” https://www.amd.com/en/products/cpu/amd-epyc-7502.

[5] “ARM C Language Extensions for SVE,” https://static.docs.arm.com/100987/0000/acle_sve_100987_0000_00_en.pdf.

[6] “Cortex-A53 - Microarchitectures - ARM,” https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a53.

[7] “DC Ultra: Concurrent Timing, Area, Power, and Test Optimization,” https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[8] “Git repositories on gem5,” https://gem5.googlesource.com.

[9] “Innovus Implementation System,” https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/innovus-implementation-system.html.

[10] “Intel 64 and ia-32 architectures software developer’s manual volume 2a: Instruction set reference.” Intel Corporation. 2015.

[11] “Skylake (server) Microarchitectures,” https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)#Extreme_Core_Count_.28XCC.29.

[12] 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.

[13] D. F. Bacon, S. L. Graham, and O. J. Sharp, “Compiler transformations for high-performance computing,” *ACM Computing Surveys*, 1994.

[14] 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.

[15] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, and G. Daglikoca, “The Architecture of the DIVA Processing-in-memory Chip,” in *Proceedings of the 16th International Conference on Supercomputing*, 2002.

[16] 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.

[17] D. G. Elliott, W. M. Snelgrove, and M. Stumm, “Computational RAM: A Memory-SIMD Hybrid and its Application to DSP,” in *1992 Proceedings of the IEEE Custom Integrated Circuits Conference*, 1992.

[18] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru, and R. Mckenzie, “Computational RAM: implementing processors in memory,” *IEEE Design Test of Computers*, 1999.

[19] R. Espasa, M. Valero, and J. E. Smith, “Vector Architectures: Past, present and future,” in *Proceedings of the 12th International Conference on Supercomputing*, 1998.

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

[21] D. Fujiki, S. Mahlke, and R. Das, “Duality Cache for Data Parallel Acceleration,” in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[22] M. Gokhale, B. Holmes, and K. Iobst, “Processing in memory: the Terasys massively parallel PIM array,” *Computer*, 1995.

[23] 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*, 2013.

[24] M. Hampton and K. Asanović, “Implementing virtual memory in a vector processor with software restart markers,” in *Proceedings of the 20th Annual International Conference on Supercomputing*. Association for Computing Machinery, 2006.

[25] M. Imani, S. Gupta, Y. Kim, and T. Rosing, “FloatPIM: In-memory acceleration of deep neural network training with high precision,” in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.

[26] 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*, 2016.

[27] Joonyoung Kim and Younsu Kim, “HBM: Memory solution for bandwidth-hungry processors,” in *2014 IEEE Hot Chips 26 Symposium (HCS)*, 2014.

[28] R. Kaplan, L. Yavits, and R. Ginosar, “PRINS: Processing-in-storage acceleration of machine learning,” *IEEE Transactions on Nanotechnology*, 2018.

[29] P. M. Kogge, “EXECUBE- A new architecture for scaleable MPPs,” in *Proceedings of the 1994 International Conference on Parallel Processing*, 1994.

[30] 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.

[31] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, “Smart Memories: a modular reconfigurable architecture,” in *Proceedings of 27th International Symposium on Computer Architecture*, 2000.

[32] S. Maleki, Y. Gao, M. Garzaran, T. Wong, and D. Padua, “An Evaluation of Vectorizing Compilers,” in *Proceedings of the 28th Int’l Conf. on Parallel Architectures and Compilation Techniques*, 2011.

[33] A. Morad, L. Yavits, S. Kvatinsky, and R. Ginosar, “Resistive GP-SIMD processing-in-memory,” *ACM Trans. Archit. Code Optim.*, 2016.

[34] M. Oskin, F. T. Chong, and T. Sherwood, “Active Pages: a computation model for intelligent memory,” in *Proceedings. 25th Annual International Symposium on Computer Architecture*, 1998.

[35] K. Pagiamtzis and A. Sheikholeslami, “Content-addressable memory (CAM) circuits and architectures: a tutorial and survey,” *IEEE Journal of Solid-State Circuits*, 2006.

[36] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick, “A case for intelligent RAM,” *IEEE Micro*, 1997.

[37] J. Potter, J. Baker, S. Scott, A. Bansal, C. Leangsuksun, and C. Asthagiri, “ASC: an associative-computing paradigm,” *Computer*, 1994.

[38] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, “Evaluating MapReduce for multi-core and multiprocessor systems,” in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

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

[40] 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.

[41] A. E. Slade and H. O. McMahon, “A cryotron catalog memory system,” in *Eastern Joint Computer Conference: New Developments in Computers*, 1957.

[42] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag, “PROMISE: An end-to-end design of a programmable mixed-signal accelerator for machine-learning algorithms,” in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture*, 2018.

[43] 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*, 2017.

[44] H. S. Stone, “A logic-in-memory computer,” *IEEE Transactions on Computers*, 1970.

[45] 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.

[46] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, “The RISC-V instruction set manual. Volume I User-level ISA,” https://www.amd.com/en/products/cpu/amd-epyc-7502, 2014.

[47] S. Williams, A. Waterman, and D. Patterson, “Roofline: an insightful visual performance model for multicore architectures,” *Communications of the ACM*, 2009.

[48] H. E. Yantır, A. M. Eltawil, and F. J. Kurdahi, “Low-power resistive associative processor implementation through the multi-compare,” in *25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2018.

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

[50] Yi Kang, Wei Huang, Seung-Moon Yoo, D. Keen, Zhenzhou Ge, V. Lam, P. Pattnaik, and J. Torrellas, “FlexRAM: toward an advanced intelligent memory system,” in *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, 1999.

[51] 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.