Supporting a Virtual Vector Instruction Set on a Commercial Compute-in-SRAM Accelerator

Courtney Golden, Dan Ilan, Caroline Huang, Niansong Zhang, Zhiru Zhang, and Christopher Batten

***Abstract*—Recent work has explored compute-in-SRAM as a promising approach to overcome the traditional processor-memory performance gap. The recently released Associative Processing Unit (APU) from GSI Technology is, to our knowledge, the first commercial compute-in-SRAM accelerator. Prior work on this platform has focused on domain-specific acceleration using direct microcode programming and/or specialized libraries. In this letter, we demonstrate the potential for supporting a more general-purpose vector abstraction on the APU. We implement a virtual vector instruction set based on the recently proposed RISC-V Vector (RVV) extensions, analyze tradeoffs in instruction implementations, and perform detailed instruction microbenchmarking to identify performance benefits and overheads. This work is a first step towards general-purpose computing on domain-specific compute-in-SRAM accelerators.**

***Index Terms*—In-memory computing, hardware/software interfaces**

I. Introduction

PROCESSOR logic scaling has outpaced that of memory technologies, creating a processor-memory performance gap in von Neumann architectures. Compute-in-memory systems address this gap by computing on memory bitlines and in peripheral circuitry. Unlike compute-in-DRAM, compute-in-SRAM is able to leverage logic technology scaling and enables monolithic solutions with low area- and energy-overheads. Much of prior work on compute-in-SRAM is based on small-scale simulations and/or academic prototypes [1], [2], [3], [4], [6], [8], [9], [12], [13], [14]. The recently released Associative Processing Unit (APU) from GSI Technology is, to our knowledge, the first commercial compute-in-SRAM chip [7]. The APU includes large SRAM arrays with support for computing on the bitlines and in the peripheral logic.

There are currently two ways to program the APU. The first is using direct microcode sequences, which involves writing custom bitwise operations for each application. The company also provides a higher-level abstraction, which facilitates easier programming but still includes many specialized operations targeting specific domains. As a result, prior work has used the APU for a small number of specific workload types, like similarity search, hash functions, and synthetic-aperture radar [7], [11].

Manuscript received 29 September 2023; revised 6 November 2023; accepted 1 December 2023. Date of publication 11 December 2023; date of current version 26 February 2024. This work was supported in part by NSF PPoSS Award #2118709; NSF SHF Award #2008471; and ACE, one of seven centers in JUMP 2.0, an SRC program sponsored by DARPA. *(Corresponding author: Courtney Golden.)*

Courtney Golden, Caroline Huang, Niansong Zhang, Zhiru Zhang, and Christopher Batten are with Cornell Univeristy, Ithaca NY 14850 USA (e-mail: ckg35@cornell.edu; lh494@cornell.edu; nz264@cornell.edu; zhiruz@cornell.edu; cbatten@cornell.edu).

Dan Ilan is with GSI Technology Inc., Tel Aviv Israel 39986, Israel (e-mail: dilan@gsitechnology.com).

Digital Object Identifier 10.1109/LCA.2023.3341389

In this letter, we explore generalizing beyond these domains by implementing a virtual vector instruction set on the APU. We implement a subset of the RISC-V Vector (RVV) extensions and analyze instruction-level performance using detailed microbenchmarking. We also use a K-nearest neighbor case study to gain further insight into the tradeoffs involved in adopting a virtual vector instruction set. The primary contributions of this paper are: (1) a description of the APU's architecture and microcode programming model; (2) a demonstration of implementing a virtual vector instruction set on the APU; and (3) detailed quantitative microbenchmarking of this virtual vector instruction set.

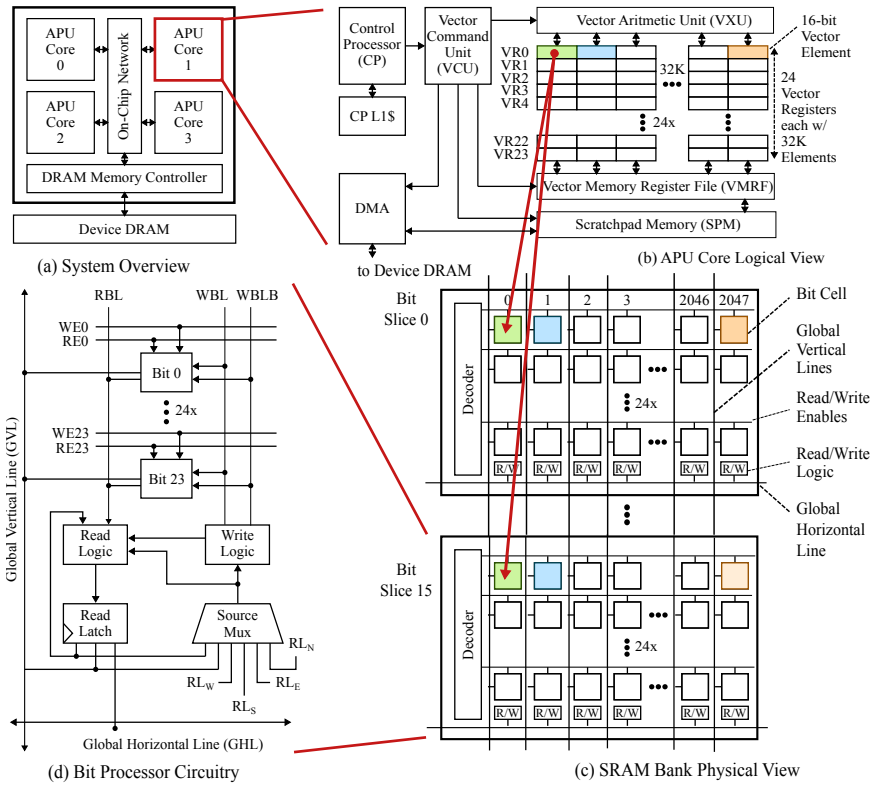
II. APU Architecture

In this section, we present a simplified view of the architecture and microarchitecture of the APU, as there is currently no detailed public description with a microcode-level view. We detail the main features that affect microcode programming while abstracting away additional functionalities. See [7] for details. [1]

As shown in Fig. 1(a), the APU platform consists of a standard x86-64 host CPU and a four-core APU chip, which communicate over PCIe and share a DDR4 DRAM. Logically, each APU core can be viewed as a traditional vector engine that operates on 32k-element vectors of 16-bit data values (Fig. 1(b)). Instruction distribution starts in the control processor (CP), which executes scalar code and issues vector commands to the vector command unit (VCU). The VCU decodes instructions into microcode operations that directly control the hardware at a cycle-level granularity. To operate on data from the shared device DRAM, the CP first uses DMA to move data to local 64KB scratchpad memories (SPM) in each core. Microcode operations then transfer these vectors to a 3MB vector memory register file (VMRF), which serves as a set of 48 "background" registers. Ultimately, the main unit of local storage is a compute-enabled 1.5MB vector register file (VRF) containing 24 vector registers, with elements striped across 16 banks.

Within one bank of the VRF (Fig. 1(c)), each of the 2048 columns stores the 16 bits of an element. Data is stored in a bit-sliced fashion, where corresponding bits of each VR are stored together as one bit-slice. For example, in column zero, bit-slice i contains bit i of element zero of all 24 vector registers, bit-slice $i+1$ contains bit $i+1$ of element zero from all 24 vector registers, etc. Each column of each bit-slice contains a single-bit read latch and associated logic. Data can also be stored in many other formats, such as storing multiple narrower elements in a single column and increasing the effective vector length. The architecture also contains global

[1] We have adopted slightly different terminology than GSI to provide a more intuitive description of the APU microarchitecture.



APU Architecture – (a) System Overview, (b) APU Core Logical View, (c) Bank Physical View, (d) Bit Processor Circuitry. CP = control processor, VCU = vector command unit, VXU = vector execution unit, VRF = vector register file, VMRF = vector memory register file, SPM = scratchpad memory, GVL = global vertical latch, R/W = read/write logic, RBL = read bitline, WBL = write bitline, WBLB = write bitline bar, REx = read-enable for bit x, WEx = write-enable for bit x, RL_N = north read latch. Note: exact bit-slice organization is not published by GSI.

Microarchitectural State

`RL`	read latch
`GVL`	global vertical latch
`GHL`	global horizontal latch
`VRF[i]`	vector register source *i*

Bit Masking

bm : *stmt*	16-bit mask (*bm*) activates bit-slices
(*bm*<<*imm*) : *stmt*	*bm* can be bitwise shifted by immediate (*imm*)

Operations on State

`RL = VRF[vrs0]`	read VR from VRF
`RL = VRF[vrs0,vrs1]`	read and bitwise AND of two VRs
`RL = L`	read value from a source latch
`RL = VRF[vrs0] op L`	operate on a VR and a latch
`RL` *op*`= VRF[vrs0]`	operate on current RL and a VR
`RL` *op*`= L`	operate on current RL and a latch
`RL` *op*`= VRF[vrs0] op L`	operate on RL, a VR, and a latch
`VRF[vrs0] = L`	write to VRF from source latch

RL = read latch; VR = vector register; VRF = vector regfile; *L* = latch (i.e., `RL`, `GVL`, `GHL`, RL_N, RL_S, RL_E, RL_W) or complement of latch (i.e., `~RL`, `~GVL`, `~GHL`, $\sim RL_N$, $\sim RL_S$, $\sim RL_E$, $\sim RL_W$)

(e) Microcode Semantics

```
(f) APL_FRAG vor_vv(vrd, vrs0, vrs1):
      0xFFFF: RL = VRF[vrs0];
      0xFFFF: RL |= VRF[vrs1];
      0xFFFF: VRF[vrd] = RL;

(g) APL_FRAG vmseq_vx(vrd_m, vrs, rs):
      0xFFFF: RL = VRF[vrs];
      ~rs: RL = ~RL;
      0xFFFF: GVL = RL;
      vrd_m: VRF[MASK_REG] = GVL;

(h) APL_FRAG vmv_vx(vrd, in_value):
      0xFFFF: RL = 0;
      in_value: RL = 1;
      0xFFFF: VRF[dst] = RL;

(i) APL_FRAG vsll(vrs):
      0xFFFF: RL = VRF[src];
      0xFFFF: VRF[src] = NRL;

(j) APL_FRAG vadd_vv(vrd, vrs0, vrs1):
      // ---- bit 0 ----
      // vrd = vrs0 XOR vrs1
      0x0001:  RL = VRF[vrs0];
      0x0001:  RL ^= VRF[vrs1];
      0x0001:  VRF[vrd] = RL;
      // cout = vrs0 AND vrs1
      0x0001:  RL = VRF[vrs0];
      0x0001:  RL &= VRF[vrs1];

      // ---- bit 1 ----
      // vrd = a ^ b ^ cin
      (0x0001<<1):  RL = VRF[vrs0];
      (0x0001<<1):  RL ^= VRF[vrs1];
      (0x0001<<1):  RL ^= RL_N;
      (0x0001<<1):  VRF[vrd] = RL;
      // cout = a*b + b*cin + a*cin
      (0x0001<<1):  RL = VRF[vrs0];
      (0x0001<<1):  RL &= VRF[vrs1];
      (0x0001<<1):  VRF[temp_0] = RL;
      (0x0001<<1):  RL = VRF[vrs1];
      (0x0001<<1):  RL &= RL_N;
      (0x0001<<1):  VRF[temp_1] = RL;
      (0x0001<<1):  RL = VRF[vrs0];
      (0x0001<<1):  RL &= RL_N;
      (0x0001<<1):  RL |= VRF[temp_0];
      (0x0001<<1):  RL |= VRF[temp_1];
      ...
```

Microcode Fragments for RVV Instructions: (f) vector or; (g) vector set-equals; (h) vector set-scalar; (i) left bit-shift; (j) vector addition. `MASK_REG` = VR storing masks; `vrd_m` = one-hot encoding indicating desired mask.

Fig. 1: APU Architecture, Microcode Semantics, and Microcode Fragments

vertical latches (GVL) and global horizontal latches (GHL). The GVL connects all cells in a column and can compute a logical AND. While the implementation of the GHL is quite complex, functionally, it ORs together the read latches of the elements in each bit-slice.

Each column of each bit-slice stores 24 bits in custom 12T SRAM cells and contains a small collection of logic gates ("R/W Logic" in Fig. 1(d); functionally equivalent to the VXU in Fig. 1(b)). To operate on any given bit, data values are read from memory cells using a single common read bit-line (RBL) and stored in a 1-bit read latch (RL). If multiple values are read simultaneously, the logical AND appears on RBL. The cell's read logic can perform AND, OR, and XOR on two or more operands, including data from the VRF, RL, GVL, or the RLs of bit processors to its north, south, east, or west (i.e. RL_N, RL_S, RL_E, RL_W). Write operands modify the VRF using the write bit-line (WBL) and its negation (WBLB). By default, identical operations are performed simultaneously on all 16 bit-slices and 2048 columns. However, a 16-bit bit-mask can be used to perform different operations on different bit-slices at once.

The APU's microinstructions control read and write logic, global structures (GVL and GHL), and data transfer between memory layers (VRF, VMRF, and SPM). Fig. 1(e) gives an overview of the syntax and semantics of the subset of microcode operations we use in this letter. Each operation on microarchitectural state can be expressed as a line of microcode with the inclusion of a bit-mask. Additionally, although not used in this letter, up to four micro-operations can be combined into a single VLIW instruction. It is the

TABLE I: Implemented RISC-V Vector Instructions

Instruction	Description	Execution Time (cycles) Theoretical	Meas.
`vor_vv`	bit-wise or	$6+e$	15
`vadd_vv`	element-wise addition	$14b-9$	215
`vsub_vx`	vector-scalar subtraction	$15b+5$	245
`vmul_vv`	element-wise multiplication	$27b+222$	438
`vmseq_vx`	set-if-equal	$7+e$	13
`vmseq_vx_m`	set-if-equal (masked)	$9+e$	15
`vredmin_vs`	find min element	$(11+C_1)b+38+C_2$	6849
`vredmin_vs_m`	find min element (masked)	$(11+C_1)b+39+C_2$	6970
`vfirst`	index of first set mask bit	$222+C_1$	6396
`vmsof`	set only first set mask bit	$191+C_1$	6229
`vmset`	set all elements of mask	3	10
`vmnot`	invert mask	7	12
`vmand`	bitwise-and of two masks	9	13
`vmv_vx`	broadcast scalar to vector	$5+e$	12
`vmv_xs`	extract element 0 of vector	$12+C_1$	895
`vmv_vv`	copy vector	$4+e$	11
`vmv_m`	copy mask	4	11
`vsll`	bitwise left-shift logical	3	13
`vsrl`	bitwise right-shift logical	3	13
`vsetvl`	set application vector length	$248+C_1$	225
`vload`	loads vector from DRAM	n/a	21943
`vstore`	stores vector to DRAM	n/a	21363

b = element bit-width. e is zero if $b = 16$ and one otherwise. C_1 and C_2 indicate cycles due to control processor instructions contained within our vector implementations; C_1 = cycles that would vary with bit-width; C_2 = cycles that are independent of bit-width. Rightmost column indicates average measured number of cycles for 16-bit elements.

responsibility of the programmer to avoid structural hazards when writing microcode sequences.

III. RVV Vector Abstraction Implementation

The well-known, open-source RISC-V ISA has been augmented in recent years with an ISA extension for vector instructions, the RISC-V Vector Extension (RVV) [5]. Here, we demonstrate implementing a subset of RVV-like instructions on this architecture, as a first step towards enabling more widespread use of the APU for general-purpose applications.

We implemented the 22 instructions shown in Table I, which model very closely the RVV ISA. Examples of microcode implementations are in Fig. 1(f)-(j). We implement a representative selection of instructions: (1) element-wise operations, like bitwise Boolean functions and element-wise addition; (2) cross-element operations, like set-equals and reductions; (3) vector mask operations, like inverting a mask; (4) vector memory operations, including loads and stores; and (5) a vector configuration operation to set the desired vector length. Element-wise operations often involve either simple bit-parallel operations using the read logic (Fig. 1(f)) or bit-serial algorithms (Fig. 1(j), more detail in Sec. IV-A); left bitwise shift (Fig. 1(i)) leverages connections between neighboring bit processors to read from RL_N. Cross-element operations often use data values as bit-masks to operate on certain bits of all elements in parallel (Fig. 1(g,h)). We support all instructions on unsigned 16-bit integers for vector lengths up to 32K, with some limitations as described below.

IV. Microbenchmarking Results

To microbenchmark the vector operations in our RVV abstraction, we profiled each instruction by measuring control-processor cycles for 10,000 calls to the same operation and averaging over ten trials. In the following section, we describe microbenchmarking results, various operation implementations, and the tradeoffs of adopting a virtual vector instruction set in terms of both arithmetic computation and memory operations.

TABLE II: Bit-Packing and Data Transfer

packed operation	avg # cycles	ops/cycle
one 16-bit ADD	215	152
one 8-bit ADD	93	352
one 4-bit ADD	37	886
one 2-bit ADD	13	2497
two 8-bit ADDs	93	705
four 4-bit ADDs	37	3542
eight 2-bit ADDs	13	20010
one 8-bit MUL	1835	18
one 4-bit MUL	450	73
one 2-bit MUL	126	260
two 4-bit MULs	456	144
four 2-bit MULs	143	917
Data Transfer	**avg # cycles**	**elements/cycle**
DRAM → SPM	21,575	1.5
SPM → VMRF	470	69.7
VMRF → VRF	21	1568.9

A. Microbenchmarking Narrow-Bitwidth Vector Operations

Due to its massive parallelism and bit-level control, the APU offers especially high throughput when performing vector operations on low-precision data. When operating on narrow-bitwidth values, many operations (e.g. bit-serial operations) can exit early. Furthermore, multiple data values can then be packed into a single element without needing additional cycles.

To demonstrate this, we implemented element-wise vector addition for 32k-element vectors with value bitwidths swept over $b = \{2, 4, 8, 16\}$ (code in Fig. 1(j)). We take a bit-serial approach to addition: for each bit, we compute and store a result bit by using bitlines and peripheral logic to implement the truth table for a full adder. A carry-out is then computed and propagated to the next bit-slice. The code for bit one is replicated, shifting its bit-mask by an additional bit each time, for all remaining bits. Our proof-of-concept implementation makes the simplifying assumption that source operands are distinct, but identical ones could be easily supported. Because this approach is bit-serial, smaller-bitwidth vectors can exit early from the computation and have lower latencies that scale linearly with b (see Table II). The second set of results in Table II then shows how multiple data values can be packed into a single element width, further increasing the number of operations per cycle.

To explore the usefulness of narrow bitwidths in quadratic-time operations, we also benchmarked vector multiplication. As seen in Table II, multiplying two four-bit values in each element (to produce an eight-bit result) takes fewer cycles due to early exit than multiplying two eight-bit values. Then, packing can be leveraged to double the number of operations done in each vector-multiply instruction, although a small amount of extra overhead is added due to structural hazards between packed operands. The combination of these two optimizations yields significantly higher throughput. Results

for both four-bit and two-bit vector multiplications can be seen in Table II.

B. Microbenchmarking Vector Multiply Instructions

Programmer control over bit-level operations provides a rich opportunity to exploit more efficient algorithms for basic operations without any additional hardware. We explore this through vector multiplication. The above quadratic-time algorithm iterates over standard carry-propagate additions. Implementing a more efficient algorithm like carry-save multiplication (linear time complexity in the operand bitwidth) usually requires more complex hardware (both full adders and carry-propagate adders). On the APU, however, carry-save multiplication requires just simple changes to the microcode, using an integer carry-save adder with a bit-parallel binary adder and our bitwise left-shift. To demonstrate this, we implemented 8-bit carry-save multiplication and found that it takes an average of 438 cycles per 32k-element multiplication with linear scaling. This is 8.3× faster than our initial ripple-carry-based implementation.

C. Microbenchmarking Overheads

Adopting a vector abstraction incurs a large performance overhead from data movement, as the abstraction limits the benefits that can be leveraged from the APU's layers of memory hierarchy. Specifically, the general-purpose vector load instruction in RVV loads data from main memory directly into a VR. Modeling this as closely as possible, our `vload` moves data directly from DRAM through the SPM and VMRF to the VRF. This is expensive, with a single load (of 32K 16-bit elements) requiring roughly 22,000 cycles. Profiling individual steps of this transfer (shown in Table II) reveals that interacting with DRAM is the most expensive step, consuming 21,575 cycles to transfer 32K 16-bit elements, or a transfer rate of 1.52 elements per cycle. If an application has large datasets and low arithmetic intensity, this data movement can become a bottleneck.

Our experiments also reveal an important overhead inherent in the APU architecture that arises when adhering to a virtual vector instruction set. Specifically, the control processor requires a minimum number of cycles for each group of microcode instructions it issues (called a 'microcode fragment'). For simple operations, written as microcode fragments with few instructions, this causes the measured cycle count to exceed the theoretical value. For example, for some short and simple operations in Table I, the last column shows an average measured cycle count that exceeds the theoretical value by between four and ten cycles. For optimal performance, microcode operations should be fused into larger fragments whose cycle counts exceed this threshold.

D. KNN Case Study

To gain further insight into the tradeoffs of adopting a virtual vector instruction set, we implement and evaluate K-nearest neighbors (KNN) using RVV. KNN takes a list of data points and a query point, and finds the k points with the smallest distances to the query. The computation involves (1) calculating the squared Euclidean distance between each data point and the query, and (2) searching the computed distances to find the minimum k times. We use the ANN_SIFT10 dataset with 32K data points but could expand this using strip-mining. Our RVV-like KNN implementation achieves a throughput of approximately 89 searches per second. Comparing to other platforms (i.e. CPU, GPU, and custom accelerators [10]) is left to future work.

Supporting such a general-purpose vector abstraction comes at the cost of unrealized performance. To quantify this tradeoff, we implemented three iterative optimizations that break the vector abstraction. First, we fused together some vector instructions into microcode segments that skip computing unnecessary intermediate values, which increased throughput to 97 searches/sec. In other cases, fusing vector operations allowed for reuse of temporary values without redundantly computing them in separate RVV instructions, which yielded 112 searches/sec. Finally, we broke the vector abstraction in memory instructions by implementing an asynchronous partial load from the DRAM to the VMRF to pipeline computation and data movement. This gave a total improved throughput of 129 searches per second.

V. Conclusion

In this letter, we explored the potential for supporting a virtual vector instruction set on a commercial compute-in-SRAM accelerator. This general-purpose vector abstraction can potentially serve as a target for standard auto-vectorizing compilers. Our analysis also motivates future work on performance and energy comparisons to CPUs and GPUs and explorations of new workload domains that are well-suited for this architecture.

References

[1] S. Aga et al. Compute Caches. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2017.

[2] K. Al-Hawaj et al. EVE: Ephemeral Vector Engines. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2023.

[3] H. Caminal et al. CAPE: A Content-Addressable Processing Engine. *Int'l Symp. on High-Performance Computer Architecture*, Feb 2021.

[4] C. Eckert et al. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. *Int'l Symp. on Computer Architecture*, Jul 2018.

[5] 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.

[6] D. Fujiki et al. Duality Cache for Data Parallel Acceleration. *Int'l Symp. on Computer Architecture*, Jun 2019.

[7] L. Gwennap. GSI Offers In-Memory Computing. *Microprocessor Report*, Jul 2020. *The Linley Group*.

[8] S. Jeloka et al. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T BitCell Enabling Logic-in-Memory. *IEEE Journal of Solid-State Circuits (JSSC)*, Apr 2016.

[9] M. Kooli et al. Towards a Truly Integrated Vector Processing Unit for Memory-Bound Applications Based on a Cost-Competitive Computational SRAM Design Solution. *J. Emerg. Technol. Comput. Syst.*, Apr 2022.

[10] J. Saikia et al. K-Nearest Neighbor Hardware Accelerator Using In-Memory Computing SRAM. *Int'l Symp. on Low Power Electronics and Design (ISLPED)*, Jul 2019.

[11] GSI. Technology. APU SAR Capabilities. https://ir.gsitechnology.com/node/11431/pdf, Sep 2022.

[12] J. Wang et al. A Compute SRAM with Bit-Serial Integer/Floating-Point Operations for Programmable In-Memory Vector Acceleration. *Int'l Solid-State Circuits Conf. (ISSCC)*, Feb 2019.

[13] J. Wang et al. A 28-nm Compute SRAM With Bit-Serial Logic/Arithmetic Operations for Programmable In-Memory Vector Computing. *IEEE Journal of Solid-State Circuits (JSSC)*, Jan 2020.

[14] J. Zhang et al. BP-NTT: Fast and Compact in-SRAM Number Theoretic Transform with Bit-Parallel Modular Multiplication. *Design Automation Conf. (DAC)*, Jul 2023.