Characterizing and Optimizing Realistic Workloads on a Commercial Compute-in-SRAM Device

Niansong Zhang
Cornell University
Ithaca, NY, USA
nz264@cornell.edu

Wenbo Zhu*
University of Southern California
Los Angeles, CA, USA
wenbozhu@usc.edu

Courtney Golden
MIT
Cambridge, MA, USA
cgolden@csail.mit.edu

Dan Ilan
GSI Technology Inc.
Tel Aviv, Israel
dilan@gsitechnology.com

Hongzheng Chen
Cornell University
Ithaca, NY, USA
hzchen@cs.cornell.edu

Christopher Batten
Cornell University
Ithaca, NY, USA
cbatten@cornell.edu

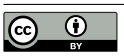
Zhiru Zhang
Cornell University
Ithaca, NY, USA
zhiruz@cornell.edu

Abstract

Compute-in-SRAM architectures offer a promising approach to achieving higher performance and energy efficiency across a range of data-intensive applications. However, prior evaluations have largely relied on simulators or small prototypes, limiting the understanding of their real-world potential. In this work, we present a comprehensive performance and energy characterization of a commercial compute-in-SRAM device, the GSI APU, under realistic workloads. We compare the GSI APU against established architectures, including CPUs and GPUs, to quantify its energy efficiency and performance potential. We introduce an analytical framework for general-purpose compute-in-SRAM devices that reveals fundamental optimization principles by modeling performance trade-offs, thereby guiding program optimizations.

Exploiting the fine-grained parallelism of tightly integrated memory-compute architectures requires careful data management. We address this by proposing three optimizations: communication-aware reduction mapping, coalesced DMA, and broadcast-friendly data layouts. When applied to retrieval-augmented generation (RAG) over large corpora (10GB–200GB), these optimizations enable our compute-in-SRAM system to accelerate retrieval by 4.8×–6.6× over an optimized CPU baseline, improving end-to-end RAG latency by 1.1×–1.8×. The shared off-chip memory bandwidth is modeled using a simulated HBM, while all other components are measured on the real compute-in-SRAM device. Critically, this system matches the performance of an NVIDIA A6000 GPU for RAG while being significantly more energy-efficient (54.4×-117.9× reduction). These findings validate the viability of compute-in-SRAM for complex, real-world applications and provide guidance for advancing the technology.

*This work was done during an internship at Cornell University.



MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/2025/10
https://doi.org/10.1145/3725843.3756132

CCS Concepts

• **Hardware → Emerging architectures**; • **General and reference → Evaluation**; • **Computing methodologies →** *Modeling methodologies.*

Keywords

Compute-in-SRAM, analytical modeling, energy efficiency, retrieval-augmented generation (RAG)

ACM Reference Format:
Niansong Zhang, Wenbo Zhu, Courtney Golden, Dan Ilan, Hongzheng Chen, Christopher Batten, and Zhiru Zhang. 2025. Characterizing and Optimizing Realistic Workloads on a Commercial Compute-in-SRAM Device. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756132

1 Introduction

Compute-in-memory (CIM) holds the promise of being a highly energy-efficient approach to accelerating data-intensive applications by reducing memory access overhead through the integration of compute units within or near memory arrays. Among CIM approaches, compute-in-SRAM stands out for its compatibility with standard CMOS technology and potential to achieve high memory bandwidth. The architectural and full-stack optimization of compute-in-SRAM systems continues to attract significant research interest. Recent works propose diverse designs: Compute Caches [1] repurpose cache elements as vector units using bit-line SRAM, delivering 1.9× speedup and 2.4× energy savings over a 32-byte SIMD CPU; EVE [7] uses a bit-hybrid execution mechanism to accelerate vector operations by nearly 8× versus an out-of-order CPU; and CAPE [11] offers a programmable CMOS associative engine, averaging 14× speedup with peaks up to 254× over an area-equivalent CPU. Specialized accelerators such as iMTransformer [30], TranCIM [45], PICMA [48], and iMCAT [29] target deep neural networks (DNNs) and transformer models, highlighting the potential for domain-specific acceleration.

Despite promising results, these architectures are primarily evaluated through instruction modeling and simulation [1, 29, 30, 47, 48] or small-scale prototypes [45], limiting insights into their practical, real-world effectiveness. This gap underscores the need for performance characterization of commercial compute-in-SRAM devices

Table 1: Comparison of GSI APU [22, 44], Intel Xeon 8280, NVIDIA A100, and Graphcore IPU.

	GSI APU	Xeon 8280	NVIDIA A100	Graphcore
Compute Cores	2 million×1 bit	28×2×512 bits	104×4,096 bits	1,216×64 bits
Tech node	28 nm	14 nm	7 nm	7 nm
Compute Speed	500 MHz	2.7 GHz	1.4 GHz	1.6 GHz
Peak Compute	25 TOPS	10 TOPS	75 TOPS	16 TOPS
On-Chip Memory	12MB L1	38.5MB L3	40MB L2	300MB L1
Mem. Bandwidth	26 TB/s	1 TB/s	7 TB/s	16 TB/s
Power	60W TDP	205W TDP	400W TDP	150W TDP

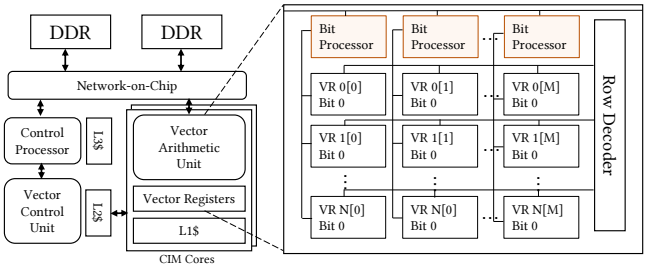


Figure 1: General-purpose compute-in-SRAM system model – CAPE [11], and GSI APU [22, 44] follow the same system abstraction, with different vector arithmetic unit and SRAM cell implementations.

under realistic workloads, bridging the divide between theoretical promise and practical feasibility.

General-purpose compute-in-SRAM systems typically employ a SIMD vector processor abstraction [1, 11, 47]. As illustrated in Fig. 1, a common compute-in-SRAM architecture—adopted by systems like CAPE—abstracts the computation-enabled SRAM array as a vector processing engine. The GSI APU [22, 44] aligns with this same abstraction, representing a commercial instance that provides a unique opportunity to evaluate the potential of compute-in-SRAM systems under realistic workloads and applications.

The GSI APU integrates 2 million bit processors at 500 MHz, delivering up to 25 TOPs for 8-bit addition [44]. Table 1 compares its compute capacity, memory bandwidth, and power efficiency against CPUs, GPUs, and ASIC accelerators, showing strong potential for data-intensive workloads. Fully exploiting this is difficult: the APU uses a 32,768-element vector processor with column-wise integrated compute and storage, offering TB/s on-chip bandwidth but limiting memory access within a vector register (VR). For instance, reductions across a VR are unsupported, and intra-VR group operations are about 10× slower than inter-VR operations.

Figure 2 shows a roofline model of different matrix multiplication kernels on the GSI APU[1]. The baseline approach, implementing a vectorized inner-product algorithm, does not account for data movement or layout overheads, resulting in suboptimal performance. However, with tailored data optimizations, performance approaches the compute roof with higher operational intensity.

This observation highlights a broader insight about compute-in-SRAM devices: despite performing computation directly within memory, these systems can still be bottlenecked by memory bandwidth if data movement is not carefully managed. To further analyze

[1]The peak computational bound is profiled for 16-bit unsigned multiplication and accumulation operations.

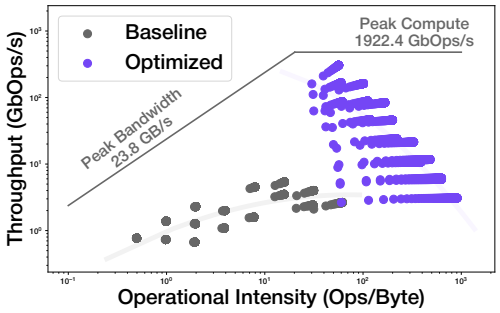


Figure 2: Performance of various matrix multiplication kernels on GSI APU, w/o data movement and data layout optimizations.

this issue, we develop an analytical framework that exposes the underlying performance limits. In this work, we propose three key optimizations to realize the potential of the compute-in-SRAM systems: communication-aware reduction mapping, coalesced DMA operations, and broadcast-friendly data layouts.

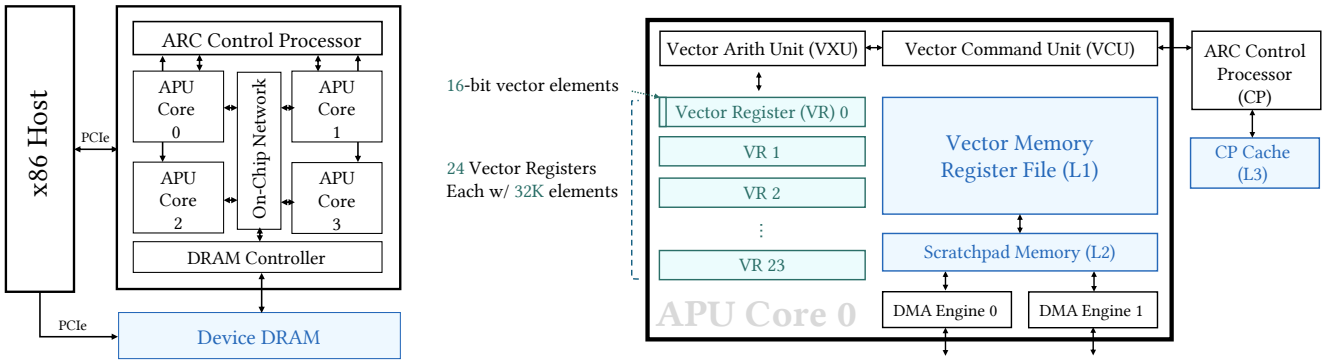
Furthermore, we evaluate compute-in-SRAM systems for Retrieval-Augmented Generation (RAG) in large language model (LLM) inference, demonstrating its suitability for this workload. We also compare its performance and energy efficiency against CPU and GPU to highlight its advantages. Our contributions are as follows:

- We present the first comprehensive evaluation of a commercial compute-in-SRAM device using realistic workloads. Specifically, we assess the GSI APU—a commercial instance of a general-purpose compute-in-SRAM device—using the Phoenix benchmark, matrix multiplication, and retrieval-augmented generation (RAG) workloads. We compare its performance and energy efficiency against established architectures, including an Intel Xeon Gold CPU and an NVIDIA A6000 GPU.
- We develop a flexible analytical framework that identifies optimization opportunities and supports architectural design space exploration by enabling the tuning of key design parameters. This framework informs the design of next-generation in-SRAM computing architectures.
- We propose three key optimizations targeting data movement and layout to exploit the unique characteristics of ultra-long vector compute-in-SRAM architectures. Applied to the RAG workload, these optimizations reduce retrieval latency by up to 6.6× compared to an optimized CPU baseline, yielding up to 1.8× end-to-end speedup and matching the latency of GPU-based systems while consuming 1% of the energy. On Phoenix, the optimized APU achieves a 41.8× speedup over CPU.

2 GSI APU Architecture

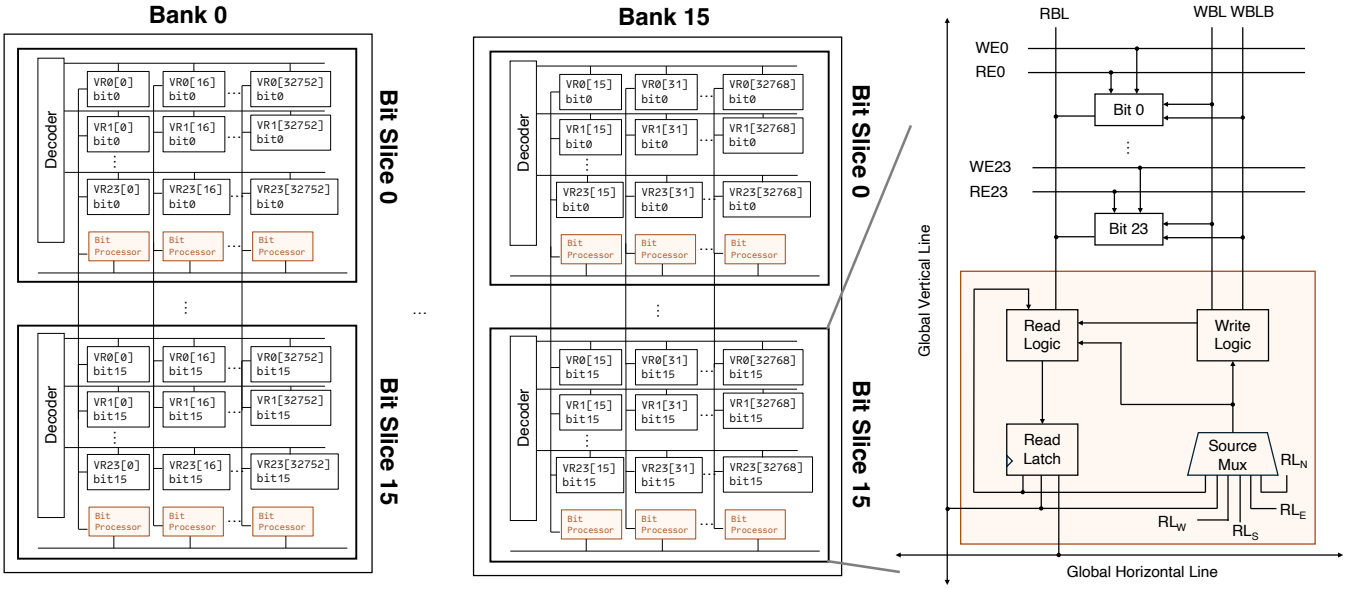
2.1 Architecture and Microarchitecture

In this section, we provide an overview of the APU's architecture and microarchitecture. As shown in Fig. 3(a), the APU platform comprises a standard x86-64 host CPU and a four-core APU chip, connected via PCIe and sharing a DDR4 DRAM. Each APU core functions as a vector engine, processing 32K-element vectors of



(a) System overview. (b) APU core logic view.

Figure 3: The GSI APU system and APU core logic view, the memory hierarchy is highlighted in blue. The APU consists of a control processor, four APU cores, and a four-level memory hierarchy. Each core has 24 vector registers (VR), and each VR has 32768 elements.



(a) APU core physical bank organization. (b) Bit processor architecture.

Figure 4: The physical bank organization of one GSI APU core and the bit processor architecture. The data is stored in a bit-slice fashion, each column of each bit-slice contains a single-bit read latch and associated read and write logic.

16-bit data, as shown in Fig. 3(b). The 32-bit ARC control processor (CP) issues vector commands to the Vector Command Unit (VCU), and the VCU decodes the vector command to microcode operations to load/store vectors to the Vector Registers (VRs) and perform arithmetic computations.

The memory hierarchy is highlighted in blue in Fig. 3, which consists of a 16 GB device DRAM, a 1 MB control processor (CP) cache (L3), a 64KB scratch pad memory (L2), and a 3 MB vector memory register file (L1). The DDR memory is shared by four APU cores, and each core has its private L2 and L1 memory. The L2 scrachpad memory serves as a DMA buffer to contain one 32K-element, 16-bit vector. The L1 memory is organized as 48 "background" registers as additional storage to the computation-enabled VRs.

Figure 4(a) shows the physical bank organization of the VRs. The 24 VRs are striped across 16 physical banks, and each bank contains 2048 16-bit elements. Within one physical bank, the data is stored in a bit-slice fashion, where each bit-slice contains one bit for all 24 VRs. Each column of each bit-slice integrates a bit processor with 24 custom 12 T SRAM cells. The bit processor microarchitecture is shown in Fig. 4(b). The bit processors are collectively equivalent to the vector arithmetic unit (VXU). The read logic can perform AND, OR, and XOR on two or more operands, including data from the read bit-line (RBL), the read latch (RL), the global vertical line,

Table 2: Microarchitectural state and operations on state.

State	Description
RL	read latch
GVL	global vertical latch
GHL	global horizontal latch
VR[i]	vector register source i
Operations	**Description**
RL = **VR**[$vrs0$]	read VR
RL = **VR**[$vrs0, vrs1$]	read and bitwise AND of two VRs
RL = L	read value from a source latch
RL = **VR**[$vrs0$] op L	operate on a VR and a latch
RL op = **VR**[$vrs0$]	operate on current RL and a VR
RL op = L	operate on current RL and a latch
RL op = **VR**[$vrs0$] op L	operate on RL, a VR, and a latch
VR[$vrs0$] = L	write to VR from source latch

the global horizontal line, and the RLs of bit processors to its north (RL_N), south (RL_S), east (RL_E), or west (RL_W). The global horizontal line connects all bit processors in the same row, while the global vertical line connects those in the same column. Each line includes a 1-bit latch: the global horizontal latch (GHL) and the global vertical latch (GVL). If multiple values are read to GHL simultaneously, an OR operation is performed before storing the result to the latch. For GVL, it performs an AND on the multiple values. The write logic updates the SRAM cells through the write bit-line (WBL) or its negation (WBLB). By default, bit processors in all bit-slices are issued the same micro operation. However, a 16-mask can be used to perform the operations on a subset of the bit slices. The VRs, RL, GHL, and GVL are the microarchitectural states. The operations on the microarchitectural states are listed in Table 2.

2.1.1 Arithmetic Operations and Data Types. Unlike bit-serial architectures that process only one bit at a time, the APU supports both bit-serial arithmetic and bit-parallel boolean operations. This flexibility is achieved through the bit-slice bank organization shown in Fig. 4(a), allowing all bits of a VR to be accessed simultaneously by the bit processors. The APU natively supports 16-bit signed and unsigned integers, 16-bit IEEE floating point, and a custom GSI floating point format with a 6-bit exponent and a 9-bit mantissa.

2.1.2 Data Movement. The APU supports both direct memory access (DMA) and programmable I/O (PIO). As shown in Fig. 3(b), each APU core is equipped with two parallel DMA engines that transfer contiguous data in 512-byte chunks, enabling high memory bandwidth for VR transfers within the memory hierarchy. For random access or single-element extraction from the VR, the ARC control processor can perform these operations using PIO. For DRAM ↔ L3 transfers, both DMA and PIO can be used, whereas for DRAM ↔ L2, only DMA is available.

DRAM ↔ L3, DRAM ↔ L2: For these types of data movement, data layout transformations can be applied. With DMA, the source and target 512-byte chunk addresses can be programmed to enable contiguous, strided, and duplicated data layout transformations. PIO enables arbitrary data layout transformations, though with lower bandwidth compared to DMA.

L2 ↔ L1, L1 ↔ VR: For these types of data movement, data layout transformations are not supported. Data is transferred at the granularity of an entire vector, meaning only full VR loads/stores (32K by 16-bit) are possible.

L3 ↔ VR: PIO enables direct data transfers between L3 and VRs via a response FIFO (RSP FIFO). It supports serial retrieval (get) from VR and parallel insertion (set) into VR. The CP can broadcast scalars or immediate values to entire VRs or masked elements, while retrieval from VR is limited to one element at a time.

Inter-VR vs. intra-VR: Due to the bit-slice organization, element-wise data movement between VRs can be performed efficiently, as all elements and bits can be processed in parallel by the bit processors. However, intra-VR data movements, such as vector shifts or bank copies, depend on the GHL or RSP FIFO and thus cannot be fully parallelized.

2.1.3 Implications on Data Layout. The differing costs of inter- and intra-VR data movement impact how data layout in the memory hierarchy affects performance in several ways: (1) For device DRAM and L3, data layout influences the bandwidth of data movement. When data is contiguous or has a regular stride, DMA offers higher bandwidth than PIO. (2) Data layout within the VR also affects data movement time. If computation results are contiguous within the VR, DMA can efficiently transfer them back to L1, L2, and then device DRAM. However, if they are scattered, PIO must be used to move them sequentially. (3) Data layout in the VR impacts computational efficiency. For instance, a reduction operation can be mapped to either inter-VR or intra-VR operations, depending on the data layout in the VR. As discussed in Section 3.2, intra-VR data movement is more costly than inter-VR movement, making intra-VR reductions more expensive due to data movement overhead.

2.2 Programming Model

The APU uses a host-accelerator programming model, where an x86 host manages kernel execution, shared memory, and kernel invocation on the APU device. Fig. 5 shows this model using a simple vector addition example to demonstrate host-device interaction.

2.2.1 Host Program. The host program, written in C, manages kernel invocation, shared DRAM (L4) memory allocation and deallocation, and data transfers between the host and device memory. Fig. 5(a) shows a snippet of the host-side code. Initialization of the calling context and input data is omitted for simplicity. The host and device communicate through a program command structure, detailed in lines L1–L9. Memory management, including device memory allocation, data movement, and kernel invocation, is handled by the GSI GDL library, a memory management library from GSI.

2.2.2 Device Program. The device program, also in C, runs on the APU control processor and uses general-purpose control flow statements. The system macro `GAL_TASK_ENTRY_POINT` defines the entry point of the device program, extracts the data structure from the command, and calls the `vec_add` function. The device program manages data transfers from device memory to L1 memory and performs vector computations using Vector Registers (VRs). Vector processing uses the GSI Vector Math Library (GVML), which provides functions for vector operations, including arithmetic, logical,

```c
// define APU program data and command
struct program_data {
    uint64_t mem_hndl_vec1, mem_hndl_vec2, mem_hndl_out;
} __attribute__((packed));

struct program_cmd {
    char* program_name;
    struct program_data data;
} __attribute__((packed));

const uint64_t vec1_size = sizeof(uint16_t) * LENGTH;
const uint64_t vec2_size = sizeof(uint16_t) * LENGTH;
const uint64_t out_size = sizeof(uint16_t) * LENGTH;
const uint64_t total_io_size = vec1_size + vec2_size + out_size;

// Allocate device DRAM memory
gdl_mem_handle_t L4_buf = gdl_mem_alloc_aligned(total_io_size);
// Add device DRAM ptrs  to the program command
struct program_cmd cmd = {
    .data.mem_hndl_vec1 = L4_buf,
    .data.mem_hndl_vec2 = L4_buf + vec1_size,
    .data.mem_hndl_out = L4_buf + vec1_size + vec2_size
};
// Copy data from host to device DRAM
gdl_mem_cpy_to_dev(&cmd.data.mem_hndl_vec1, vec1_host, vec1_size);
gdl_mem_cpy_to_dev(&cmd.data.mem_hndl_vec2, vec2_host, vec2_size);
// Copy program cmd to device DRAM
gdl_mem_handle_t L4_cmd = gdl_mem_alloc_aligned(sizeof(cmd));
gdl_mem_cpy_to_dev(L4_cmd, cmd, sizeof(cmd));

// Invoke APU code
gdl_run_task_timeout(
    ctx_id,
    GDL_TASK(vec_add_task),
    L4_cmd);

// Copy output from device DRAM to host
gdl_mem_cpy_from_dev(out, cmd.data.mem_hndl_out, out_size);
```

(a) APU host code.

```c
static int vec_add(struct program_data *data) {
    // convert mem handles to device DRAM pointers
    void *vec1_L4ptr =
        gal_mem_handle_to_apu_ptr(data->mem_hndl_vec1);
    uint16_t *vec2_L4ptr =
        (uint16_t *)gal_mem_handle_to_apu_ptr(
        data->mem_hndl_vec2);
    uint16_t *out_L4ptr =
        (uint16_t *)gal_mem_handle_to_apu_ptr(
        data->mem_hndl_out);
    // move data from device DRAM to L1
    direct_dma_l4_to_l1_32k(GVML_VM_0, vec1_L4ptr);
    direct_dma_l4_to_l1_32k(GVML_VM_1, vec2_L4ptr);
    // load to VR, perform addition, store to L1
    gvml_load_32(vec1_vr, GVML_VM_0);
    gvml_load_32(vec2_vr, GVML_VM_1);
    gvml_add_u16(result_vr, vec1_vr, vec2_vr);
    gvml_store_16(GVML_VM_3, result_vr);
    // move data from L1 to device DRAM
    direct_dma_l1_to_l4_32k(out_L4ptr, GVML_VM_3);
    return 0;
}

GAL_TASK_ENTRY_POINT(vec_add_task, in) {
    struct program_cmd *cmd = (struct program_cmd *)in;
    gvml_init_once();
    return vec_add(&cmd->data);
}
```

(b) APU device code.

Figure 5: Simple vector addition code demonstrating APU programming model.

bitwise, trigonometric, and min/max operations. Once computations are complete, the device program transfers data back to device memory.

The GVML library is implemented using APU microcode instructions. APU microcode instructions directly operate on the microarchitectural states listed in Table. 2. An APU programmer can implement a different vector abstraction with microcode instructions. For example, Golden et al. [19] implemented a RISC-V vector abstraction using APU microcode. In this work, we use the abstractions provided by GVML to focus on optimizing performance through these data movement and computation operations.

Table 3: Notations

Notation	Description	Notation	Description
d	Data size in bytes	BW	Memory bandwidth
r	VR group size	s	Subgroup size
σ	Lookup table size	C	Constant

3 Analytical Framework

We propose a flexible analytical framework to model performance characteristics of compute-in-SRAM platforms. This framework parameterizes critical architectural factors, including computation latency, data movement bandwidth, and communication patterns with potentially non-uniform costs. Such generalization enables applicability across various compute-in-SRAM architectures, aiding in performance analysis and optimization strategies beyond specific device implementations.

3.1 Applicability and Assumptions

This analytical framework targets compute-in-SRAM system models as illustrated in Fig. 1. The model assumes a PCIe-based accelerator with a multi-level memory hierarchy and a vector processor abstraction, where data movement costs are non-uniform across memory levels and within vector registers. While the framework is validated using the GSI APU, it is not limited to this device. It can be extended to other compute-in-SRAM platforms that follow the same system model by deriving the necessary parameters through profiling.

Table 3 summarizes notations used throughout the framework. Tables 4 and 5 provide generic models of latency for data movement and computation operations, respectively. Framework validation against measured latencies on a real device is discussed in Section 5.2.2.

3.2 Data Movement

Effective data movement is crucial for compute-in-SRAM systems, particularly in data-intensive applications. Below, we discuss key data movement paradigms typically supported by these architectures.

3.2.1 DMA Transfers. Direct Memory Access (DMA) operations facilitate efficient bulk data transfers within compute-in-SRAM platforms. DMA latency generally scales linearly with transfer size, captured by the model $T_{\text{DMA}} = d/BW + T_{\text{init}}$, where d is data size, BW is bandwidth, and T_{init} is a fixed initialization overhead. While DMA provides high throughput for continuous data movement, off-chip memory bandwidth constraints can limit performance for very large data sizes.

3.2.2 Programmable I/O (PIO). PIO enables fine-grained, irregular data access patterns. The latency of PIO transfers typically scales

Table 4: Data movement analytical framework

Operation	Description	Execution Time (cycles) Analytical	Meas.
dma_l4_l3	L4→L3 DMA	$d/\text{BW} + T_{\text{init}}$	$0.19d + 41164$
dma_l4_l2	L4→L2 DMA	$d/\text{BW} + T_{\text{init}}$	$0.63d + 548$
dma_l2_l1	L2→L1 DMA, 16-bit × 32K	$T_{\text{l2}\to\text{l1}}$	386
dma_l4_l1	L4→L1 DMA, 16-bit × 32K	$T_{\text{l4}\to\text{l1}}$	22272
dma_l1_l4	L1→L4 DMA, 16-bit × 32K	$T_{\text{l1}\to\text{l4}}$	22186
pio_ld	PIO load, L4→VR	$n \cdot T_{\text{pio\_ld}}$	$57n$
pio_st	PIO store, VR→L4	$n \cdot T_{\text{pio\_st}}$	$61n$
lookup	Lookup L3 w/ index VR	$C \cdot \sigma + T_{\text{init}}$	$7.15\sigma + 629$
load, store	VR↔L1 load store	$T_{\text{ld}}, T_{\text{st}}$	29
cpy	VR↔VR element-wise copy	T_{cpy}	29
cpy_subgrp	Copy VR subgroup to group	$T_{\text{cpy\_sgp}}$	82
cpy_imm	Broadcast an immediate to VR	$T_{\text{cpy\_imm}}$	13
shift_e(k)	Shift VR entries to head/tail by k	$C \cdot k$	$373k$
shift_e(4k)	Intra-bank shift VR entries by $4 \cdot k$	$C + k$	$8 + k$

* In the analytical framework, we refer to the device DRAM as L4 memory.

with the number of individual load or store operations, modeled as $T_{\text{PIO}} = n \cdot T_{\text{access}}$, where n is the operation count. Though flexible, PIO incurs higher overhead compared to DMA, making it suited primarily for non-contiguous or sparse data transfers.

3.2.3 Indexed Lookup and Element-wise Operations. Indexed lookups handle irregular, scatter-like data transfers from higher memory levels to local vector registers (VR). The lookup latency grows proportionally with table size (σ), formulated as $T_{\text{lookup}} = C \cdot \sigma + T_{\text{init}}$, highlighting the necessity for careful indexing and data layout optimization. Element-wise copy operations, such as scalar broadcasting and VR-to-VR transfers, execute efficiently due to parallel hardware mechanisms, typically exhibiting constant-time latencies. Such operations are essential for data initialization and broadcasting in parallel workloads.

3.2.4 Vector Register (VR) Shifts. Intra-vector register shifts rearrange data locally within VRs without accessing external memory, incurring latency proportional to the shift magnitude, modeled by $T_{\text{shift\_e}} = C \cdot k$. Minimizing intra-VR shifts through optimized data layouts can significantly improve overall performance.

3.3 Computation

On compute-in-SRAM platforms, vectorized arithmetic, logical, and comparison operations typically execute in constant time due to their inherent parallel execution. Therefore, we summarize their notation and provide representative latency measurements obtained from the GSI APU in Table 5.

Reduction operations aggregate elements within vector registers, such as summation or finding extrema. Such operations often employ subgroup-based hierarchical reduction strategies to exploit parallelism. However, due to hardware constraints, inter-subgroup reductions may have non-linear costs and can be significantly higher than intra-subgroup operations. A generic cost model for subgroup-based reductions can be expressed as:

$$
\begin{aligned}
T_{\text{sg\_add}}(r,s) &= p_3(\log_2 s)^3 + p_2(\log_2 s)^2 + p_1 \log_2 s + p_0, \\
p_3 &= \alpha_3 \cdot \log_2 r + \beta_3, \quad p_2 = \alpha_2 \cdot \log_2 r + \beta_2, \\
p_1 &= \alpha_1 \cdot \log_2 r + \beta_1, \quad p_0 = \alpha_0 \cdot \log_2 r + \beta_0.
\end{aligned} \tag{1}
$$

Table 5: Computation analytical framework

Operation	Description	Execution Time (cycles) Analytical	Meas.
and_16	16-bit bit-wise and	T_{and}	12
or_16	16-bit bit-wise or	T_{or}	8
not_16	16-bit bit-wise not	T_{not}	10
xor_16	16-bit bit-wise xor	T_{xor}	12
ashift	int16 arithmetic shift	T_{ash}	15
add_u16	uint16 element-wise addition	T_{uadd}	12
add_s16	int16 element-wise addition	T_{sadd}	13
sub_u16	uint16 element-wise subtraction	T_{usub}	15
sub_s16	int16 element-wise subtraction	T_{ssub}	16
popcnt_16	16-bit population count	T_{popcnt}	23
mul_u16	uint16 element-wise multiplication	T_{umul}	115
mul_s16	int16 element-wise multiplication	T_{smul}	201
mul_f16	float16 element-wise multiplication	T_{fmul}	77
div_u16	uint16 element-wise division	T_{udiv}	664
div_s16	int16 element-wise division	T_{sdiv}	739
eq_16	16-bit element-wise equal	T_{eq}	13
gt_u16	uint16 element-wise greater than	T_{ugt}	13
lt_u16	uint16 element-wise less than	T_{ult}	13
lt_gf16	gsi float16 element-wise less than	T_{flt}	45
ge_u16	uint16 greater than or equal	T_{uge}	13
le_u16	uint16 less than or equal	T_{ule}	13
recip_u16	uint16 element-wise reciprocal	T_{recip}	735
exp_f16	float16 exponential	T_{exp}	40295
sin_fx	fixed-point sine	T_{sin}	761
cos_fx	fixed-point cosine	T_{cos}	761
count_m	count marked entries	$T_{\text{cnt\_m}}$	239
add_subgrp_s16	int16 add sub groups in each group	Eq. 1	–

```python
framework = LatencyEstimator()  # Initialize analytical framework
with framework.ctx():
    total_data_size = 1024 * 1024 * 256 * 3
    tile_data_size = 8 * 1024 * 48  # Size of one tile across 48 VMRs
    tile_num = int(total_data_size / tile_data_size)
    for tile in range(tile_num):
        for vmr in range(48):
            for t in range(2):
                fast_dma_l4_to_l2(32 * 512)  # L4 to L2 DMA
            direct_dma_l2_to_l1_32k()  # L2 to L1 DMA

        for vmr in range(48):
            gvml_load_16()
            for subgrp in range(8):
                gvml_cpy_subgrp_16_grp(8192, 1024)
                gvml_create_grp_index_u16()
                gvml_cpy_imm_16()
                for hist_grp in range(8):
                    gvml_cpy_16_msk()  # Masked copy
                    gvml_sr_imm_16()  # Shift right by immediate
                    gvml_eq_16()
                    gvml_cpy_from_mrk_16_msk()
                    ...

    for res_vr in range(8):
        gvml_store_16()
        direct_dma_l1_to_l4_32k()
latency = framework.report_latency()  # Estimate total latency
print(f"Latency: {latency} us")
```

Figure 6: An example of modeling application latency with the analytical framework. We developed a Python function library with an interface similar to the GSI-provided C++ API. This example models the latency of the `Histogram` application from the Phoenix benchmark suite [41].

The cubic term emerges due to the multi-level shifting, alignment, and accumulation operations inherent in hierarchical reductions, whose complexity grows non-linearly as subgroup size increases. Using logarithms ($\log_2 s$ and $\log_2 r$) in the model is natural since these operations typically organize data aggregation in stages that

halve the subgroup size at each step, indicating a logarithmic relationship. The polynomial coefficients p_0, p_1, p_2, p_3 themselves depend logarithmically on the group size (r), parameterized by experimentally determined constants α_i, β_i. This generalized formulation allows modeling of complex, non-linear hardware behavior common in hierarchical reduction operations.

3.4 Framework Implementation

We have developed a Python function library that mirrors the interface of the GSI-provided C++ API, enabling programmers to model arbitrary APU programs. The analytical framework interprets these programs and reports the total latency. Fig. 6 shows a code snippet modeling the `Histogram` application from the Phoenix benchmark suite [41].

3.5 Framework Implications

Our analytical framework highlights general performance trends across compute-in-SRAM architectures. Specifically, element-wise computations exhibit low latency and efficiently exploit parallel hardware. Conversely, large-scale reduction operations and certain intra-vector data movements can become significant bottlenecks. DMA transfers outperform PIO for bulk movements but lack flexibility for sparse data access. Thus, achieving high efficiency on compute-in-SRAM platforms requires careful optimization of data movement strategies, data layouts, and computational structures aligned with underlying hardware characteristics.

4 Optimizing Realistic Workloads on Compute-in-SRAM

Compute-in-SRAM provides substantial advantages in data parallelism and energy efficiency. However, it also introduces specific challenges and opportunities for optimization. Here, we use binary matrix multiplication as a motivating example to illustrate three key optimizations for compute-in-SRAM devices.

4.1 Motivating Example

Binary matrix multiplication is a crucial kernel for efficient machine learning, supporting workloads such as binary neural networks [50, 51] and binarized transformers [31, 46]. Compute-in-SRAM platform is a natural fit for this kernel due to its efficiency and speed in logical operations and integer addition. However, it is not easy to achieve high performance without careful consideration of data layout and data movement.

In the motivating example shown in Fig. 7, the input matrices $\mathbf{A}$ (M, K) and $\mathbf{B}$ (K, N) are bit-packed into `uint16` types along the K-axis. The binary matrix multiplication produces an output matrix $\mathbf{C}$ (M, N) in `int16` type. The algorithm is depicted in Fig. 7(a). To implement this inner-product algorithm on an ultra-long vector processor, the baseline approach unrolls loop j, leading to the data layout illustrated in Fig. 7(c). We refer to this loop mapping scheme as spatial reduction vector mapping, as the reduction sum occurs spatially within the VR. Let the VR length be $l = 32768$. We consider the device DRAM as off-chip memory, assuming matrix B fits in L1, the baseline operational intensity (OI) is:

$$OI = \frac{M \cdot N \cdot K \cdot \alpha}{(MK \cdot \lfloor l/K \rfloor + KN + MN) \cdot \texttt{sf(u16)}}, \tag{2}$$

where $\lfloor l/K \rfloor$ is the duplication factor of matrix A due to loop j unrolling, α is the number of logical and arithmetic operations on each scalar, and `sf()` denotes `size_of()`. Matrix A rows are duplicated in DRAM→L2 and moved to L1, with a run-time cost of:

$$T_A = \left(\frac{K \cdot \texttt{sf(u16)}}{\text{BW}} + T_{\text{init}} \right) \cdot \left\lfloor \frac{l}{K} \right\rfloor \cdot M + M \cdot T_{\text{l2}\to\text{l1}}. \tag{3}$$

We assume matrix B is stored in a column-major layout in the device DRAM, and it fits in L1, then the run time cost of moving matrix B is given by:

$$T_B = \frac{N}{\lfloor l/K \rfloor} \cdot T_{\text{l4}\to\text{l1}}. \tag{4}$$

For non-contiguous results in VR C, PIO transfers each element to L4, with a cost of:

$$T_C = M \cdot N \cdot T_{\text{pio\_st}} \tag{5}$$

The compute run-time cost is:

$$T_{\text{MAC}} = \frac{N}{\lfloor l/K \rfloor} \cdot (T_{\text{xor}} + T_{\text{popcnt}} + T_{\text{ash}} + T_{\text{ssub}} + T_{\text{sg\_add}}(K, 1)) \tag{6}$$

and the total run-time cost is the sum of the data movement costs and the compute cost.

4.2 Communication-Aware Reduction Mapping

As outlined in the analytical framework: (1) intra-VR operations are more costly than inter-VR operations, and (2) using DMA to transfer the same amount of data is significantly cheaper than using PIO. Guided by these observations, we implement binary matrix multiplication as scalar-vector product (SVP) [13]. As shown in Fig. 8, the reduction axis is mapped to the more efficient element-wise operations at each k loop iteration. We refer to this loop mapping scheme as temporal reduction vector mapping. Additionally, the output data layout becomes contiguous, enabling fast DMA. Therefore, the compute run-time cost and matrix $\mathbf{C}$ movement cost reduce to:

$$T_{\text{MAC}} = (T_{\text{xor}} + T_{\text{popcnt}} + T_{\text{ash}} + T_{\text{ssub}} + T_{\text{sadd}}) \cdot \frac{M}{\lfloor l/N \rfloor} \cdot K, \tag{7}$$

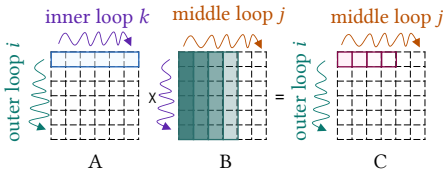
$$T_C = \frac{M}{\lfloor l/N \rfloor} \cdot T_{\text{l4}\to\text{l1}}. \tag{8}$$

Since all bit processors operate in parallel, higher VR occupancy translates to improved computational efficiency. To achieve this, we unroll loop i to fully utilize the VR, as shown in Fig. 9(a). Loop i is specifically chosen for unrolling to maintain the temporal mapping of loop k. Consequently, this approach results in two levels of data duplication in the VR layout, as shown in Fig. 9(b): the scalars from A are duplicated due to the spatial unrolling of loop j, and rows from B are duplicated due to the spatial unrolling of loop i. This data layout enables opportunities for data reuse and memory access coalescing. We implement the scalar duplication of matrix A as a lookup operation from L3. Therefore, the OI for the scalar-vector product becomes

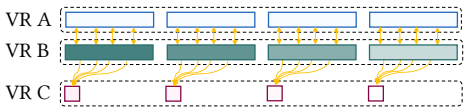
```
A = uint16 [M, K]
B = uint16 [K, N]
C = int16  [M, N]

for i in range(M):
  for j in range(N):
    for k in range(K):
      v0 = A[i, k] ^ B[k, j]
      v1 = popcount(v0)
      C[i, j] += 16 - v1 << 1
```

(a) Binary matrix multiply.

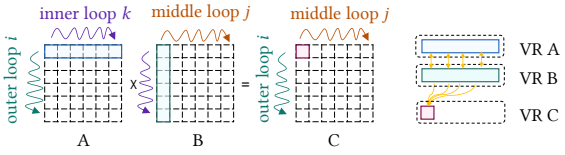


(b) Inner-product data access.

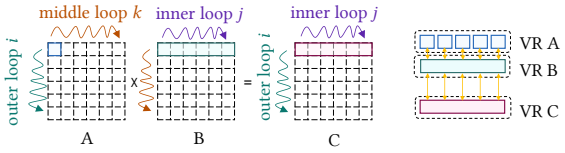


(c) Data layout on vector registers.

Figure 7: Motivating example: binary matrix multiply implemented as an inner-product algorithm on APU.

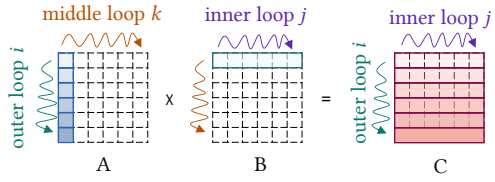


(a) Inner-product maps reduction to intra-VR operations (spatial)

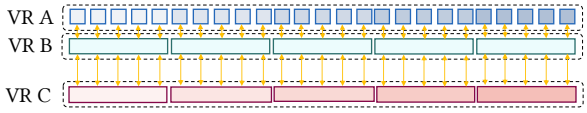


(b) Scalar-vector product maps reduction to inter-VR operations (temporal)

Figure 8: Reduction axis spatial vs. temporal mapping.



(a) Spatially unroll loop i.



(b) VR data layout for SVP with spatially unrolled loop i.

Figure 9: Spatially unrolling i-axis fully utilizes the VR, enables inter-VR reduction, and achieves a contiguous layout of results.

$$OI = \frac{M \cdot N \cdot K \cdot \alpha}{(MK + NK \cdot \lfloor l/N \rfloor + MN) \cdot \texttt{sf(u16)}} \tag{9}$$

Assuming matrix A is stored in row-major order, the run-time cost of moving it involves transferring from L4 to L3, followed by duplication via lookup:

$$T_A = M \cdot K/\text{BW} + T_{\text{init}} + T_{\text{lookup}}(N \cdot K) \cdot \frac{M}{\lfloor l/N \rfloor} \cdot K \tag{10}$$

For B, loop i spatial unrolling incurs duplication of factor $\lfloor l/N \rfloor$, the run-time cost of moving matrix B becomes

$$T_B = \left(\frac{N \cdot \texttt{sf(u16)}}{\text{BW}} + T_{\text{init}}\right) \cdot \left\lfloor \frac{l}{N} \right\rfloor \cdot K + K \cdot T_{\text{l2}\to\text{l1}} \tag{11}$$

4.3 DMA Coalescing

Once we optimize the data layout, a new bottleneck of data duplication emerges. As seen in Fig. 9, one form of data duplication is that of duplicating a chunk of data across an entire VR. In Fig. 10(a), we see that DMA transactions can be used for data duplication. However, this approach is bandwidth-inefficient since accessing off-chip memory incurs high latency, and multiple DMA transactions add initiation overhead. Because the same chunk of data from B is accessed in each iteration of loop k, we can coalesce these DMA accesses to avoid redundant data movement. Specifically, we combine DMA transactions on multiple rows of B into a single transaction, minimizing initiation overhead.

To implement this, we introduce a reuse VR to store the initial DMA result. Using the subgroup copy capability, each row of B is arranged in a subgroup and copied to fill the VR at each iteration of loop k. Notably, subgroup copy can also target a portion of the VR, providing flexibility when duplicating only part of the data. This optimization results in a lower run-time cost of moving matrix B:

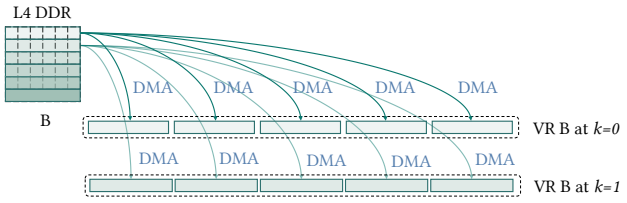
$$T_B = \left\lceil \frac{K \cdot N}{l} \right\rceil \cdot T_{\text{l4}\to\text{l1}} + K \cdot T_{\text{cpy\_sgp}} \tag{12}$$

Since DMA coalescing also removes duplicate data movement from L4, the OI is also improved:

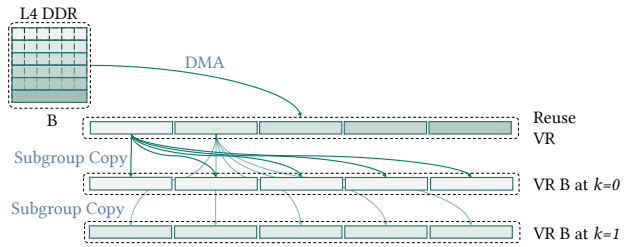
$$OI = \frac{M \cdot N \cdot K \cdot \alpha}{(MK + NK + MN) \cdot \texttt{sf(u16)}}. \tag{13}$$

4.4 Broadcast-Friendly Data Layout

After removing the redundant DMA operations, the bottleneck shifts to the lookup operation used to broadcast scalars in **A**. As shown in Table 4, the lookup latency increases with the size of the lookup table, prompting us to reduce its size. Fig. 11 illustrates the lookup operation, where three scalars are broadcast each time, highlighted by the blue-filled boxes. In the row-major layout shown in Fig. 11(a), the broadcast window initially covers indices 0, 6, and 12, and then moves to indices 1, 7, and 13 in the next iteration. Since the lookup table must be a contiguous chunk of memory, the lookup table size is at least 18 to broadcast the first three rows. To reduce



(a) Without DMA coalescing.



(b) With DMA coalescing.

Figure 10: Coalescing DMA – leverage subgroup copy to remove redundant data access and increase data reuse.

the lookup table size, we change the data layout to a broadcast-friendly format, shown in Fig. 11(b). The broadcast window initially covers indices 0, 1, 2, and then moves on to 3, 4, 5. Therefore, the lookup table sizes can be reduced to 3. We express this data layout as dimension sizes and strides, where decomposed sizes and strides are expressed as tuples, as shown in Fig. 11. This format is proposed by Graphene [23]. For the motivating example, this optimization reduces the lookup table size for broadcasting matrix $\mathbf{A}$ from $K \cdot N$ to N, thereby reducing the cost of data movement to:

$$T_A = M \cdot K/\mathrm{BW} + T_{\mathrm{init}} + T_{\mathrm{lookup}}(N) \cdot \frac{M}{\lfloor l/N \rfloor} \cdot K. \qquad (14)$$

In summary, we demonstrate how these three key optimizations for data layout and movement reduce both input/output transfer costs and computation costs for compute-in-SRAM devices.

5 Evaluation

Using the GSI APU as a commercial example, this section validates the analytical framework and evaluates the real-world performance of compute-in-SRAM with the proposed optimizations. First, a latency breakdown of binary matrix multiplication highlights the individual contributions of each optimization. Next, a benchmark study validates the analytical framework and identifies workload characteristics well-suited for in-SRAM computing. Finally, an end-to-end retrieval-augmented generation study on large corpora compares the performance and energy efficiency of compute-in-SRAM against CPU and GPU platforms.

We use the GSI Leda-E APU (500 MHz clock frequency), an Intel Xeon Gold 6230R CPU (2.1 GHz, 1.6 MB L1 cache, 52 MB L2 cache, 71.5 MB L3 cache), and an NVIDIA A6000 GPU for comparison. Latency measurements on the GSI APU are obtained using control processor cycle counts. Energy profiling is performed using a Texas Instruments UCD9090 voltage monitor and Renesas ISL8273M power modules on board, which provide point-of-load regulation and current telemetry.

5.1 Binary Matrix Multiplication

We use a 1024×1024 binary matrix multiplication kernel as a microbenchmark to analyze and demonstrate the impact of the proposed optimizations.

Fig. 12 illustrates the latency breakdown from the baseline implementation to the optimized versions. Key operations include: LD LHS / RHS, loading matrices from off-chip memory to L1 via DMA, PIO, or lookup; VR Ops, on-chip operations like subgroup copies and computations; and ST, storing results back to off-chip memory.

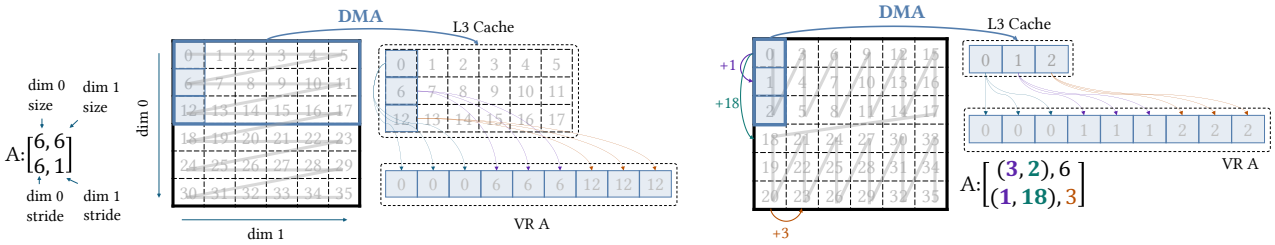
We use an inner-product algorithm as the baseline implementation (described in Section 3.2), which is bottlenecked by result data movement due to costly PIO stores for non-contiguous outputs. Applying communication-aware reduction mapping (opt1) reduces this overhead by enabling efficient DMA transfers, though it increases RHS matrix loading time due to data duplication. Adding DMA coalescing (opt2) further improves LHS loading by replacing PIO with faster DMA, at the cost of additional vector register operations for subgroup copies. Introducing a broadcast-friendly data layout (opt3) also accelerates LHS loading, but the overall bottleneck remains in the result write back. When all three optimizations are combined, results become contiguous and can be transferred using DMA. We also apply DMA coalescing for the RHS matrix using k-axis data packing and adopt a broadcast-friendly format $\begin{bmatrix} (32,\ 32) & 64 \\ (1,\ 2048) & 32 \end{bmatrix}$ for the LHS matrix broadcasting, which yields an end-to-end latency of 12.0 ms, an 18.9× improvement over the baseline latency of 226.3 ms.

This result demonstrates that while individual optimizations may yield modest speedups, they enable opportunities for further improvements. For example, communication-aware reduction mapping enables DMA coalescing to RHS matrix loading and facilitates the use of a broadcast-friendly layout for the LHS matrix. Ultimately, combining all optimizations leads to substantial performance gains.

5.2 Phoenix Benchmarks

We evaluate the optimized GSI APU performance using the Phoenix Benchmark suite [41], comparing it against two baselines: single-threaded and multi-threaded CPU implementations. We select Phoenix for its data-intensive benchmarks and its prior use in compute-in-SRAM studies such as CAPE [11]. The official Phoenix benchmark provides optimized CPU implementations, either single-threaded or multi-threaded. We use the official repository[2], with the multi-threaded version configured for up to 16 threads using the MapReduce programming model. Table 6 summarizes the eight applications in the suite, including their CPU instruction count (measured with Valgrind) and APU μCode instruction count (reported by the Vector Command Unit). Input sizes range from 10 MB to 1.5 GB. We measure the total APU kernel latency, covering data movement from device memory to L1 and back. We achieve optimized performance on these benchmarks by applying all proposed data movement and data layout optimizations.

[2]https://github.com/kozyraki/phoenix



(a) Row-major data layout.

(b) A broadcast-friendly data layout

Figure 11: Broadcast-friendly data layout example – (a) a row-major data layout requires a lookup table of size 18. (b) A broadcast-friendly data layout only requires a lookup table size of 3.

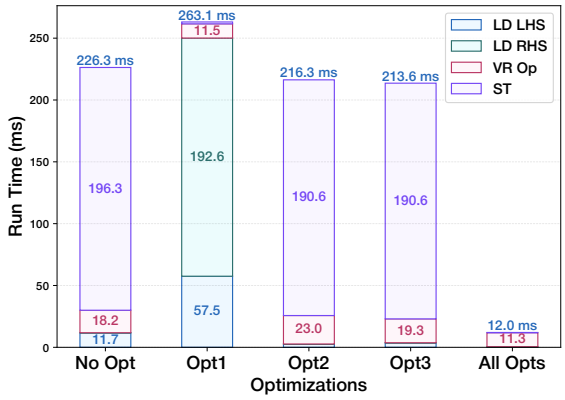


Figure 12: Binary matrix multiplication runtime breakdown with different optimizations. Opt1: communication-aware reduction mapping. Opt2: DMA Coalescing. Opt3: broadcast-friendly data layout.

Table 6: Statistics of the phoenix benchmark suite.

Application	Input Size	#Inst. on CPU	#Inst. of APU μCode
Histogram	1.5GB	4.8 billion	110.7 million
Linear Regression	512MB	3.8 billion	1.6 million
Matrix Multiply	1,024×1,024	22.6 billion	69.7 million
Kmeans	128k	0.4 billion	0.04 million
Reverse Index	100MB	4.8 billion	11.0 million
String Match	512MB	101.8 billion	0.09 million
Word Count	10MB	0.7 billion	0.17 million

Figure 13 compares the latency of the APU implementations against CPU baselines. Relative to the single-threaded CPU, the APU implementation with all optimizations applied achieves an average speedup of 41.8× (mean), 14.4× (geometric mean), and a peak speedup of 128.3×. Compared to the multi-threaded CPU execution, the APU achieves an average speedup of 12.5× (mean), 2.6× (geometric mean), and a maximum speedup of 68.1×.

5.2.1 Results Analysis. The APU implementations shown in Fig. 13 include a baseline with no optimizations, as well as versions applying only communication-aware reduction mapping (Opt1), only DMA coalescing (Opt2), only broadcast-friendly data layout (Opt3), and all three optimizations together (APU all opts). Individually, communication-aware reduction mapping provides large gains in workloads involving comparison or distance computation over large volumes of data, such as `kmeans`, `reverse index`, `string match`, and `word count`. DMA coalescing reduces data movement costs in cases where data duplication is required (e.g., `matmul`) or where input data can be packed to improve vector register utilization (e.g., `linear regression`, `string match`). Broadcast-friendly data layout is beneficial when scalar values are broadcast via lookup operations, such as in `kmeans`, although these opportunities often emerge only after other optimizations have been applied. Overall, we observe that applying all three optimizations consistently yields greater performance improvements than applying any single optimization in isolation.

Table 7: Phoenix benchmark suite latency measured vs. analytical framework.

Application	Meas. Latency (ms)	Predicted (ms)	Error
Histogram	1644.8	1650.1	+0.32%
Linear Regression	92.3	94.5	+2.3%
Matrix Multiply	421.3	402.5	-4.5%
Kmeans	1.6	1.4	-6.2%
Reverse Index	182.0	181.1	-0.49%
String Match	90.9	92.6	+1.8%
Word Count	3.2	3.1	-3.1%

The fully optimized results on the benchmark suite suggest that compute-in-SRAM platforms are best suited for a specific subset of applications. The optimized APU implementation outperforms a multi-threaded CPU on linear regression, k-means, string match, and word count: applications characterized by high data parallelism and minimal intra-VR computation. With the proposed optimizations, most arithmetic operations are efficiently mapped to inter-VR element-wise instructions, and data duplication overhead is reduced. In contrast, other applications including histogram, matrix multiply, reverse index, still involve frequent intra-VR operations and fine-grained element access due to their algorithmic nature, limiting the performance benefits from compute-in-SRAM acceleration.

5.2.2 Analytical framework validation. We validate the analytical framework using the Phoenix Benchmark suite by comparing the measured latency with the predicted latency. Table 7 summarizes the results across the eight benchmarks. On average, the analytical framework achieves 97.3% accuracy, with a maximum error of 6.2%.

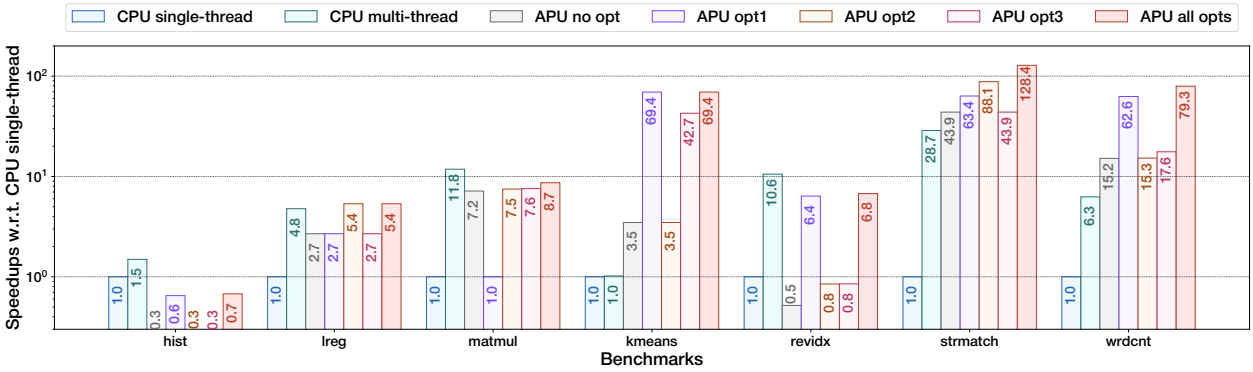


Figure 13: Latency comparison across workloads from the *Phoenix Benchmark Suite*, normalized to the single-threaded Intel Xeon Gold CPU baseline. Opt1: communication-aware reduction mapping. Opt2: DMA Coalescing. Opt3: broadcast-friendly data layout.

The primary source of error arises from the model's inability to account for memory subsystem details or cache behavior.

5.3 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) improves language model responses by retrieving relevant knowledge during generation [16, 34]. It embeds queries and documents, then performs vector similarity search [42]. Large corpora often require CPUs/GPUs to use Approximate Nearest Neighbor Search (ANNS), trading accuracy for latency and memory. This can cause significant accuracy loss (22%–53% for `Llama-8B` and `Llama-80B` [40]) compared to Exact Nearest Neighbor Search (ENNS). Compute-in-SRAM platforms, with massive parallelism, can accelerate ENNS efficiently, avoiding this compromise. We study optimized compute-in-SRAM ENNS for RAG, focusing on latency and energy benefits.

5.3.1 Experimental Setup. We implement the ENNS RAG retrieval process on the GSI APU. However, the device's limited DDR bandwidth (23.8 GB/s) would unequivocally create an off-chip memory bottleneck, hindering a fair performance comparison. To mitigate this, we model a more representative off-chip memory system by simulating HBM2e memory (16 GB, 2 ranks, 8 channels, 1.6 GHz, yielding 380–420 GB/s peak bandwidth) using Ramulator 2 [35] and DRAMPower 5.0 [12]. The compute-in-SRAM performance results presented incorporate these simulated off-chip memory timings, while all other components—including on-chip data movement, computation, and system overheads are measured directly on the GSI APU hardware.

We use `Llama3.1-8B` [20] with 16-bit number format as the generation model and sample questions from the Natural Questions (NQ) dataset [28]. The evaluation system comprises two GPUs (one dedicated to generation and the other to retrieval), a CPU, and a GSI APU. Generation runs on a single GPU, and retrieval is performed using ENNS across three corpus sizes: 10 GB, 50 GB, and 200 GB, on CPU, GPU, and a compute-in-SRAM accelerator. Each corpus is chunked into segments of 16,384 tokens. As a result, the 10 GB corpus contains 163K chunks (120 MB embedding size), the 50 GB corpus contains 819K chunks (600 MB embedding size), and the 200 GB corpus contains 3.3M chunks (2.4 GB embedding size).

For both the GPU and compute-in-SRAM accelerator, corpus embeddings are transferred to device memory once at the start of the workload. All subsequent queries are served without reloading the embeddings. Meanwhile, the corpus chunks reside in the CPU's main memory. In the results that follow, we report the time-to-interactive latency on each platform—also referred to as time-to-first-token latency—which serves as the primary metric for evaluating the interactivity of the LLM inference system. All latency results are averaged across 10 queries.

5.3.2 Software Configurations. We evaluate RAG performance on the CPU and GPU using FAISS [14], a widely adopted library for efficient similarity search and clustering of dense vectors at scale. Our experiments use FAISS v1.7.2 to run ENNS inner product search with `IndexFlat`, leveraging AVX512 intrinsics and OpenMP-based multithreading on the CPU.

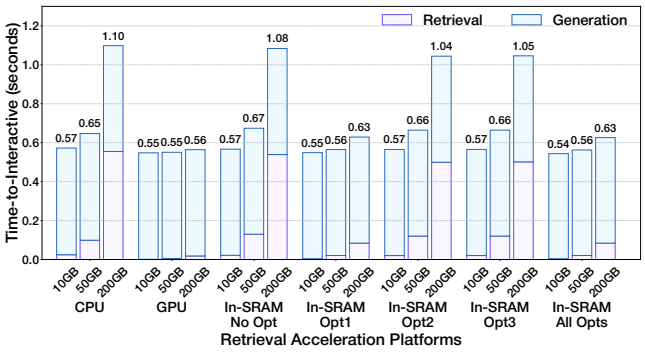


Figure 14: Inference time breakdown of CPU, GPU, vs. compute-in-SRAM with and without optimizations. The `Llama3.1-8B` generative model runs on a dedicated GPU. Opt1: communication-aware reduction mapping. Opt2: DMA Coalescing. Opt3: broadcast-friendly data layout.

5.3.3 End-to-End RAG Performance. As shown in Fig. 14, retrieval accounts for an increasing portion of end-to-end inference time as corpus size scales (CPU-based retrieval: 4.3% at 10 GB → 50.5%

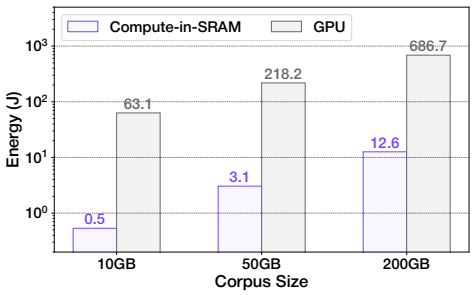


Figure 15: Top-5 retrieval process energy comparison with GPU. Results are measured on GSI Leda-E APU and NVIDIA A6000 GPU.

at 200 GB). We optimize inner-product search on the compute-in-SRAM accelerator via communication-aware reduction mapping and a broadcast-friendly query layout, mapping reductions to inter-VR instructions. Retrieval speedups over CPU are 6.3×/4.8×/6.6× at 10/50/200 GB, yielding 1.05×/1.15×/1.75× end-to-end gains. Versus an unoptimized compute-in-SRAM baseline, retrieval latency reduces up to 6.4×. The optimized system attains GPU-level end-to-end latency, underscoring the effectiveness of retrieval-side acceleration.

5.3.4 Optimization Impact and Retrieval Latency Breakdown. APU retrieval latency for RAG at 10/50/200 GB is 21.8/129.5/539.2 ms without optimization, comparable to CPU performance but slower than GPU. Communication-aware reduction mapping (opt1) addresses output data movement bottlenecks, cutting retrieval latency to 4.0/21.0/86.1 ms. DMA coalescing (opt2) and a broadcast-friendly layout (opt3) give modest standalone gains but compound with opt1; with all three, latency drops to 3.9/20.6/84.2 ms. Figure 14 shows that while opt2 and opt3 have limited standalone effect, they enhance opt1 by improving data movement and vector utilization.

As shown in Table 8, most of the optimized compute-in-SRAM retrieval speedup comes from the distance calculation stage. The key is communication-aware reduction mapping—which maps inner-product reductions to inter-VR ops to reduce intra-VR movement and improves alignment (e.g., embedding-load time drops from 8.2 ms to 6.1 ms at 200 GB). A broadcast-friendly layout further lowers query-broadcast overhead and boosts vector reuse, adding additional compute-time savings.

5.3.5 Energy Efficiency Comparison with GPU. We benchmark top-5 retrieval on our optimized compute-in-SRAM accelerator against an NVIDIA A6000 GPU, measuring GPU energy with `nvidia-smi`. As shown in Fig. 15, the APU is 54.4×–117.9× more energy-efficient than the GPU. At 200 GB, APU energy is dominated by static (71.4%), followed by compute (24.7%), DRAM (2.7%), other (1.1%), and cache (0.005%); smaller corpora show similar distributions, indicating static power dominates while compute scales modestly with workload size.

6 Related Work

Compute-in-Memory architectures hold the promise of being a highly energy-efficient approach for data-intensive applications by reducing data movement between memory and compute units. The Intelligent RAM (IRAM) [36–38] was one of the earliest efforts to integrate computational logic directly into DRAM, demonstrating the potential of coupling memory with vector processing. Building on this idea, VIRAM [17] introduced a full vector processor with embedded DRAM to accelerate bandwidth-bound workloads. DIVA [4] brought SPMD (single-program multiple-data) models to Processing-in-Memory (PNM), enabling more flexible parallelism, while FlexRAM [5, 10] extended this model within embedded DRAM systems, highlighting the importance of programmable abstractions for general-purpose compute-in-memory platforms.

Table 8: Compute-in-SRAM retrieval latency breakdown across corpus size with and without optimizations.

	Compute-in-SRAM No Opt			Compute-in-SRAM All Opts		
Corpus Size	10 GB	50 GB	200 GB	10 GB	50 GB	200 GB
Load Embedding*	0.4 ms	2.0 ms	8.2 ms	0.3 ms	1.5 ms	6.1 ms
Load Query	10 µs	11 µs	10 µs	62 µs	62 µs	65 µs
Calc Distance	21.0 ms	126.5 ms	527.9 ms	3.1 ms	18.0 ms	74.6 ms
Top-K Aggregation	69 µs	325 µs	1.30 ms	72 µs	317 µs	1.24 ms
Return Top-K	14 µs	14 µs	14 µs	15 µs	16 µs	16 µs
Total	**21.8 ms**	**129.5 ms**	**539.2 ms**	**3.9 ms**	**20.6 ms**	**84.2 ms**

* Load embedding latency reflects simulated HBM2e performance; all other values are measured on GSI APU hardware.

Compute-in-SRAM architectures has been explored to realize boolean [2, 3, 8, 26], multiply-and-accumulate (MAC) [6, 9, 24, 25, 29, 30, 32, 45, 48], and associative computing [15, 21, 39, 43] mechanisms. Jeloka et al. [26] introduced bit-line compute techniques in SRAMs, enabling bitwise logical operations between rows. Compute caches [1] applied bit-line compute to transform chip multiprocessor (CMP) caches into logical compute engines. SRAM-based technologies have also proven effective for in-situ MAC operations due to the high on/off impedance ratio of SRAM bit cells [9, 27, 32, 49]. Associative computing, which uses primitives like search and multi-write to achieve in-memory compute [15, 43], has seen renewed interest with modern technologies. CAPE [11], for example, demonstrates a CMOS-based associative engine with high programmability and low area cost.

APU Microbenchmarking. Prior work mapped RISC-V vector abstractions to the APU [19], accelerated genomics kernels [18], and implemented cryptographic primitives [33], but these handtuned microkernels highlight architectural features rather than system-level behavior. In contrast, we evaluate end-to-end Phoenix and RAG workloads, providing detailed performance characterization, and analysis of realistic compute and memory demands.

7 Conclusion

This work provides a comprehensive evaluation of compute-in-SRAM devices under realistic workloads. Our analytical framework highlights key optimizations for general-purpose in-SRAM computing. With communication-aware reduction mapping, coalesced DMA, and broadcast-friendly data layouts, we accelerated RAG retrieval stage by 4.8×–6.6× and reduced time-to-interactive latency by 1.1×–1.8× over an optimized CPU baseline. Our system matched the performance of an NVIDIA A6000 GPU while consuming 54.4×–117.9× less energy, underscoring the practicality and efficiency of compute-in-SRAM architectures.

Acknowledgments – This work was supported in part by the NSF PPoSS Award #2118709, the NSF Graduate Research Fellowship Program (GRFP) Award #2141064, and the ACE Center for Evolvable Computing, one of the seven centers in JUMP 2.0.

References

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (Austin, TX, USA). IEEE Computer Society, Los Alamitos, CA, USA, 481–492. https://doi.org/10.1109/HPCA.2017.21

[2] Amogh Agrawal, Akhilesh Jaiswal, Chankyu Lee, and Kaushik Roy. 2018. X-SRAM: Enabling in-memory Boolean computations in CMOS static random access memories. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 12 (2018), 4219–4232.

[3] Amogh Agrawal, Akhilesh Jaiswal, Deboleena Roy, Bing Han, Gopalakrishnan Srinivasan, Aayush Ankit, and Kaushik Roy. 2019. Xcel-RAM: Accelerating binary neural networks in high-throughput SRAM compute arrays. *IEEE Transactions on Circuits and Systems I: Regular Papers* 66, 8 (2019), 3064–3076.

[4] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing. In *ISCA '15: Proceedings of the 42nd Annual International Symposium on Computer Architecture* (Portland, OR, USA). Association for Computing Machinery, New York, NY, USA, 105–117. https://doi.org/10.1145/2749469.2750386

[5] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. *ACM SIGARCH Computer Architecture News* 43, 3S (2015), 336–348.

[6] Khalid Al-Hawaj, Olalekan Afuye, Shady Agwa, Alyssa Apsel, and Christopher Batten. 2020. Towards a Reconfigurable Bit-Serial/Bit-Parallel Vector Accelerator Using In-Situ Processing-in-SRAM. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)* (Seville, Spain (Virtual)). Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, 1–5. https://doi.org/10.1109/ISCAS45731.2020.9181068

[7] Khalid Al-Hawaj, Tuan Ta, Nick Cebry, Shady Agwa, Olalekan Afuye, Eric Hall, Courtney Golden, Alyssa B. Apsel, and Christopher Batten. 2023. EVE: Ephemeral Vector Engines. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Montreal, QC, Canada). IEEE Computer Society, Los Alamitos, CA, USA, 691–704. https://doi.org/10.1109/HPCA56546.2023.10071074

[8] Daniel Bankman, Lita Yang, Bert Moons, Marian Verhelst, and Boris Murmann. 2018. An Always-On 3.8μJ 86% CIFAR-10 mixed-signal binary CNN processor with all memory on chip in 28-nm CMOS. *IEEE Journal of Solid-State Circuits* 54, 1 (2018), 158–172.

[9] Avishek Biswas and Anantha P. Chandrakasan. 2018. Conv-RAM: An Energy-Efficient SRAM with Embedded Convolution Computation for Low-Power CNN-Based Machine Learning Applications. In *2018 IEEE International Solid-State Circuits Conference (ISSCC)* (San Francisco, CA, USA). Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, 488–490. https://doi.org/10.1109/ISSCC.2018.8310397

[10] Jay B. Brockman, Shyamkumar Thoziyoor, Shannon K. Kuntz, and Peter M. Kogge. 2004. A Low Cost, Multithreaded Processing-in-Memory System. In *WMPI '04: Proceedings of the 3rd Workshop on Memory Performance Issues, in conjunction with the 31st International Symposium on Computer Architecture* (Munich, Germany). Association for Computing Machinery, New York, NY, USA, 16–22. https://doi.org/10.1145/1054943.1054946

[11] Helena Caminal, Kailin Yang, Srivatsa Srinivasa, Akshay Krishna Ramanathan, Khalid Al-Hawaj, Tianshu Wu, Vijaykrishnan Narayanan, Christopher Batten, and José F. Martínez. 2021. CAPE: A Content-Addressable Processing Engine. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Seoul, South Korea (Virtual)). IEEE Computer Society, Los Alamitos, CA, USA, 557–569. https://doi.org/10.1109/HPCA51647.2021.00054

[12] Karthik Chandrasekar, Christian Weis, Yonghui Li, Sven Goossens, Matthias Jung, Omar Naji, Benny Akesson, Norbert Wehn, and Kees Goossens. 2024. *DRAMPower: Open-source DRAM Power & Energy Estimation Tool.* TU Kaiserslautern, Microelectronic Systems Design (MSD) Research Group. http://www.drampower.info

[13] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. 2020. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems* 32, 5 (2020), 1014–1029.

[14] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss Library. https://doi.org/10.48550/arXiv.2401.08281 arXiv:2401.08281 [cs.LG]

[15] Caxton C. Foster. 1976. *Content Addressable Parallel Processors.* Van Nostrand Reinhold Company, New York, NY, USA.

[16] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey. https://doi.org/10.48550/arXiv.2312.10997 arXiv:2312.10997 [cs.CL]

[17] Joseph Gebis, Sam Williams, David Patterson, and Christos Kozyrakis. 2004. VIRAM1: A Media-Oriented Vector Processor with Embedded DRAM. In *DAC '04: Proceedings of the 41st Annual Design Automation Conference – Student Design Contest* (San Diego, CA, USA). Association for Computing Machinery, New York, NY, USA, 6 pages. https://csl.stanford.edu/~christos/publications/2004.dac.iram.pdf Student Design Contest paper.

[18] Courtney Golden, Dan Ilan, Nicholas Cebry, and Christopher Batten. 2023. Accelerating Seed Location Filtering in DNA Read Mapping Using a Commercial Compute-in-SRAM Architecture. In *Proceedings of the 5th Workshop on Accelerator Architecture in Computational Biology and Bioinformatics (AACBB)* (Orlando, FL, USA). Association for Computing Machinery, New York, NY, USA, 10 pages. https://www.csl.cornell.edu/~cbatten/pdfs/golden-apu-filtering-aacbb2023.pdf Workshop held in conjunction with ISCA'50 (FCRC 2023); see also arXiv:2401.11685.

[19] Courtney Golden, Dan Ilan, Caroline Huang, Niansong Zhang, Zhiru Zhang, and Christopher Batten. 2024. Supporting a Virtual Vector Instruction Set on a Commercial Compute-in-SRAM Accelerator. *IEEE Computer Architecture Letters* 23, 1 (Jan 2024), 29–32. https://doi.org/10.1109/LCA.2023.3341389

[20] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The Llama 3 Herd of Models. https://doi.org/10.48550/arXiv.2407.21783 arXiv:2407.21783 [cs.LG]

[21] Qing Guo, Xiaochen Guo, Ravi Patel, Engin Ipek, and Eby G. Friedman. 2013. AC-DIMM: Associative Computing with STT-MRAM. In *ISCA '13: Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel). Association for Computing Machinery, New York, NY, USA, 189–200. https://doi.org/10.1145/2485922.2485939

[22] Linley Gwennap. 2020. *In-Memory Acceleration for Big Data.* Technical Report. The Linley Group. https://gsitechnology.com/wp-content/uploads/2023/01/GSIT-Gemini-WP-Final-Linley.pdf

[23] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An IR for Optimized Tensor Computations on GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23)* (Vancouver, BC, Canada). Association for Computing Machinery, New York, NY, USA, 302–313. https://doi.org/10.1145/3582016.3582018

[24] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. FloatPIM: In-Memory Acceleration of Deep Neural Network Training with High Precision. In *ISCA '19: Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, AZ, USA). Association for Computing Machinery, New York, NY, USA, 802–815. https://doi.org/10.1145/3307650.3322237

[25] M. Ishida, T. Kawakami, A. Tsuji, N. Kawamoto, M. Motoyoshi, and N. Ouchi. 1998. A Novel 6T-SRAM Cell Technology Designed with Rectangular Patterns Scalable Beyond 0.18 μm Generation and Desirable for Ultra High Speed Operation. In *1998 IEEE International Electron Devices Meeting (IEDM) Technical Digest* (San Francisco, CA, USA). Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, 201–204. https://doi.org/10.1109/IEDM.1998.746322

[26] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. 2016. A 28 nm configurable memory (TCAM/BCAM/SRAM) using push-rule 6T bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits* 51, 4 (2016), 1009–1021.

[27] Zhewei Jiang, Shihui Yin, Jae-Sun Seo, and Mingoo Seok. 2020. C3SRAM: An in-memory-computing SRAM macro based on robust capacitive coupling computing mechanism. *IEEE Journal of Solid-State Circuits* 55, 7 (2020), 1888–1897.

[28] Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. 2019. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics* 7 (2019), 453–466.

[29] Ann Franchesca Laguna, Arman Kazemi, Michael T. Niemier, and X. Sharon Hu. 2021. In-Memory Computing Based Accelerator for Transformer Networks for Long Sequences. In *Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (Grenoble, France (Virtual)). Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, 1839–1844. https://doi.org/10.23919/DATE51398.2021.9474146

[30] Ann Franchesca Laguna, Mohammed Mehdi Sharifi, Arman Kazemi, Xunzhao Yin, Michael Niemier, and X. Sharon Hu. 2022. Hardware-Software Co-Design of an In-Memory Transformer Network Accelerator. *Frontiers in Electronics* 3, Article 847069 (Apr 2022), 21 pages. https://doi.org/10.3389/felec.2022.847069

[31] Phuoc-Hoan Charles Le and Xinlin Li. 2023. BinaryViT: Pushing Binary Vision Transformers Towards Convolutional Models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (Vancouver, BC, Canada). IEEE Computer Society, Los Alamitos, CA, USA, 4664–4673. https://doi.org/10.1109/CVPRW59228.2023.00492

[32] Eunyoung Lee, Taeyoung Han, Donguk Seo, Gicheol Shin, Jaerok Kim, Seonho Kim, Soyoun Jeong, Johnny Rhe, Jaehyun Park, Jong Hwan Ko, et al. 2021. A charge-domain scalable-weight in-memory computing macro with dual-SRAM architecture for precision-scalable DNN accelerators. *IEEE Transactions on Circuits and Systems I: Regular Papers* 68, 8 (2021), 3305–3316.

[33] Kaitlyn Lee, Brian Donnelly, Tomer Sery, Dan Ilan, Bertrand Cambou, and Michael Gowanlock. 2023. Evaluating Accelerators for a High-Throughput Hash-Based Security Protocol. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops (ICPP-W '23)* (Salt Lake City, UT, USA). Association for Computing Machinery, New York, NY, USA, 40–49. https://doi.org/10.1145/

3605731.3605745

[34] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[35] Haocong Luo, Yahya Can Tu, F. Nisa Bostanci, Ataberk Olgun, A. Giray Yaglikci, and Onur Mutlu. 2024. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* 23, 1 (2024), 112–116. https://doi.org/10.1109/LCA.2023.3333759

[36] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. 1998. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA '98)* (Barcelona, Spain). IEEE Computer Society, Los Alamitos, CA, USA, 192–203. https://doi.org/10.1109/ISCA.1998.694774

[37] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. A case for intelligent RAM. *IEEE micro* 17, 2 (1997), 34–44.

[38] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberley Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. 1997. Intelligent RAM (IRAM): Chips that Remember and Compute. In *1997 IEEE International Solid-State Circuits Conference (ISSCC) Digest of Technical Papers* (San Francisco, CA, USA). Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, 224–225. https://doi.org/10.1109/ISSCC.1997.585348

[39] Jerry Potter, Johnnie Baker, Stephen Scott, Arvind Bansal, Chokchai Leangsuksun, and Chandra Asthagiri. 1994. ASC: an associative-computing paradigm. *Computer* 27, 11 (1994), 19–25.

[40] Derrick Quinn, Mohammad Nouri, Neel Patel, John Salihu, Alireza Salemi, Sukhan Lee, Hamed Zamani, and Mohammad Alian. 2025. Accelerating Retrieval-Augmented Generation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 15–32. https://doi.org/10.1145/3669940.3707264

[41] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)* (Phoenix, AZ, USA). IEEE Computer Society, Los Alamitos, CA, USA, 13–24. https://doi.org/10.1109/HPCA.2007.346181

[42] Alireza Salemi and Hamed Zamani. 2024. Evaluating Retrieval Quality in Retrieval-Augmented Generation. In *SIGIR '24: Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval* (Washington, DC, USA). Association for Computing Machinery, New York, NY, USA, 2395–2400. https://doi.org/10.1145/3626772.3657957

[43] Gordon E. Sayre. 1976. Staran: An Associative Approach to Multiprocessor Architecture. In *Computer Architecture: Workshop of the Gesellschaft für Informatik, Erlangen, May 22–23, 1975*. Springer, Berlin, Heidelberg, 199–221. https://doi.org/10.1007/978-3-642-66400-7_9

[44] GSI Technology. 2023. GSI Technology's Gemini-I® APU Showcased in "In-Memory Acceleration for Big Data". https://ir.gsitechnology.com/news-releases/news-release-details/gsi-technologys-gemini-ir-apu-showcased-memory-acceleration-big.

[45] Fengbin Tu, Zihan Wu, Yiqi Wang, Ling Liang, Liu Liu, Yufei Ding, Leibo Liu, Shaojun Wei, Yuan Xie, and Shouyi Yin. 2023. TranCIM: Full-Digital Bitline-Transpose CIM-based Sparse Transformer Accelerator With Pipeline/Parallel Reconfigurable Modes. *IEEE Journal of Solid-State Circuits* 58, 6 (Jun 2023), 1798–1809. https://doi.org/10.1109/JSSC.2022.3213542

[46] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. 2023. BitNet: Scaling 1-Bit Transformers for Large Language Models. https://doi.org/10.48550/arXiv.2310.11453 arXiv:2310.11453 [cs.CL]

[47] Yue Zha and Jing Li. 2020. Hyper-AP: Enhancing Associative Processing Through a Full-Stack Optimization. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (Valencia, Spain (Virtual)). Institute of Electrical and Electronics Engineers, Piscataway, NJ, USA, 846–859. https://doi.org/10.1109/ISCA45697.2020.00074

[48] Bo Zhang, Shihui Yin, Minkyu Kim, Jyotishman Saikia, Soonwan Kwon, Sungmeen Myung, Hyunsoo Kim, Sang Joon Kim, Jae-Sun Seo, and Mingoo Seok. 2023. PIMCA: A Programmable In-Memory Computing Accelerator for Energy-Efficient DNN Inference. *IEEE Journal of Solid-State Circuits* 58, 5 (May 2023), 1436–1449. https://doi.org/10.1109/JSSC.2022.3211290

[49] Jintao Zhang, Zhuo Wang, and Naveen Verma. 2017. In-memory computation of a machine-learning classifier in a standard 6T SRAM array. *IEEE Journal of Solid-State Circuits* 52, 4 (2017), 915–924.

[50] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. 2021. FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations. In *FPGA '21: Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA). Association for Computing Machinery, New York, NY, USA, 171–182. https://doi.org/10.1145/3431920.3439296

[51] Yichi Zhang, Zhiru Zhang, and Lukasz Lew. 2022. PokeBNN: A Binary Pursuit of Lightweight Accuracy. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (New Orleans, LA, USA). IEEE Computer Society, Los Alamitos, CA, USA, 12475–12485. https://doi.org/10.1109/CVPR52688.2022.01215

A Artifact Appendix

A.1 Abstract

This artifact contains GSI APU programs, profiling results, HBM simulation traces, and the analytical framework described in Sec. 3.4. It reproduces the key results presented in the paper through six experiments, each corresponding to a specific figure or table:

- Binary matrix multiplication (Fig. 12)
- Phoenix benchmark (Fig. 13)
- Analytical framework validation (Table 6)
- End-to-end RAG inference (Fig. 14)
- RAG energy analysis (Fig. 15)
- RAG latency breakdown (Table 8)

A.2 Artifact Check-list (Meta-information)

- **Program:** GSI LedaG Tools, Version 100.12.0.1.1000.25
- **Compilation:** GSI Device Library (GDL), GSI APU Library (GAL), GSI Vector Math Library (GVML), `gcc`, `meson`
- **Hardware:** GSI Gemini APU Leda-E PCIe board
- **Execution:** `bash`, `python3`
- **Metrics:** Measured latency, speedup against CPU/GPU baselines, analytical prediction error, energy consumption
- **Output:** Reproduces Fig. 12, Fig. 13, Fig. 14, Fig. 15, Table 6, and Table 8
- **Experiments:** Six automated experiments matching the paper's figures and tables
- **Disk space required:** 20 GB
- **Setup time:** None. Evaluators access the artifact via JupyterHub on our server
- **Runtime:** 10 minutes
- **Publicly available:** Yes, code hosted on GitHub [3]
- **License:** Apache License 2.0
- **Workflow automation:** Jupyter Notebook
- **Archived (DOI):** 10.5281/zenodo.16730562

A.3 Description

A.3.1 How to Access. While the artifact is publicly available, the experiments require access to a specialized APU accelerator. We provide access to our research server via JupyterHub at zhang-capra-xcel.ece.cornell.edu. For login credentials, the artifact evaluator should contact the authors. Access is restricted and valid only during the artifact evaluation period.

A.4 Installation

No installation is necessary. All dependencies and data are pre-installed and accessible via the provided JupyterHub server.

A.5 Experiment Workflow

The experiments are fully automated through a Jupyter Notebook. After logging into the JupyterHub, the evaluator will find the main notebook at `artifact_evaluation.ipynb`. This notebook includes detailed instructions for each experiment.

The evaluator may run individual experiments using `Run > Run Selected Cells`, or execute all experiments at once using `Run > Run All Cells`. Each experiment invokes APU kernels, profiles execution, parses outputs, and generates figures or tables matching those in the paper.

The six included experiments are:

(1) **Binary Matrix Multiplication (1-bmatmul):** Performance breakdown across optimization levels
(2) **Phoenix Benchmark Suite (2-phoenix):** Speedup comparison across CPU, GPU, and APU backends
(3) **Analytical Model Validation (3-analytical):** Comparison of model predictions and measured performance
(4) **RAG End-to-End Inference (4-rag-e2e):** Inference time analysis for retrieval-augmented generation
(5) **RAG Energy Analysis (5-rag-energy):** Energy comparison between GPU and compute-in-SRAM
(6) **RAG Latency Breakdown (6-rag-latency-breakdown):** Latency analysis across RAG components

A.6 Evaluation and Expected Results

The artifact reproduces the key figures and tables with minor variations due to hardware and runtime effects:

- Fig. 12 — Binary matrix multiplication
- Fig. 13 — Phoenix benchmark speedup
- Table 6 — Analytical model validation
- Fig. 14 — RAG end-to-end inference
- Fig. 15 — Energy analysis
- Table 8 — Latency breakdown

A.7 Notes

A demo video [4] is provided to assist the evaluator in reproducing the results. It walks through the entire process step by step.

[3] https://github.com/cornell-zhang/apu-micro25-artifact

[4] https://youtu.be/A_By1ShFXbc