PIMSAB: A Processing-In-Memory System with Spatially-Aware Communication and Bit-Serial-Aware Computation

SIYUAN MA, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, United States
KAUSTUBH MHATRE, Arizona State University, Tempe, United States
JIAN WENG, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia
BAGUS HANINDHITO, The University of Texas at Austin, Austin, United States
ZHENGRONG WANG, University of California, Los Angeles, Los Angeles, United States
TONY NOWATZKI, University of California, Los Angeles, Los Angeles, United States
LIZY JOHN, The University of Texas at Austin, Austin, United States
AMAN ARORA, Arizona State University, Tempe, United States

Bit-serial Processing-In-Memory (PIM) is an attractive paradigm for accelerator architectures, for parallel workloads such as Deep Learning (DL), because of its capability to achieve massive data parallelism at a low area overhead and provide orders-of-magnitude data movement savings by moving computational resources closer to the data. While many PIM architectures have been proposed, improvements are needed in communicating intermediate results to consumer kernels, for communication between tiles at scale, for reduction operations, and for efficiently performing bit-serial operations with constants. We present PIMSAB, a scalable architecture that provides a spatially aware communication network for efficient intra-tile and inter-tile data movement and provides efficient computation support for generally inefficient bit-serial compute patterns. Our architecture consists of a massive hierarchical array of compute-enabled SRAMs (CRAMs), which is codesigned with a compiler to achieve high utilization. The key novelties of our architecture are (1) in providing efficient support for spatially aware communication by providing local H-tree network for reductions, by adding explicit hardware for shuffling operands, and by deploying systolic broadcasting, as well as (2) by taking advantage of the divisible nature of bit-serial computations through adaptive precision and efficient handling of constant operations. These innovations are integrated into a tensor expressions-based programming framework (including a compiler for easy programmability) that enables simple programmer control of optimizations for mapping programs into massively parallel binaries for millions of PIM processing elements. When compared against a similarly provisioned modern Tensor Core GPU (NVIDIA A100), across common DL kernels and end-to-end DL networks (Resnet18 and BERT), PIMSAB outperforms the GPU by

Authors' Contact Information: Siyuan Ma, Department of Electrical and Computer Engineering, The University of Texas at Austin, Austin, Texas, United States; e-mail: siyuan.ma@utexas.edu; Kaustubh Mhatre, Arizona State University, Tempe, Arizona, United States; e-mail: kmhatre@asu.edu; Jian Weng, King Abdullah University of Science and Technology, Thuwal, Saudi Arabia; e-mail: jian.weng@kaust.edu.sa; Bagus Hanindhito, The University of Texas at Austin, Austin, Texas, United States; e-mail: hanindhito@bagus.my.id; Zhengrong Wang, University of California, Los Angeles, Los Angeles, California, United States; e-mail: seanzw@ucla.edu; Tony Nowatzki, University of California, Los Angeles, Los Angeles, California, United States; e-mail: tjn@cs.ucla.edu; Lizy John, The University of Texas at Austin, Austin, Texas, United States; e-mail: ljohn@ece.utexas.edu; Aman Arora, Arizona State University, Tempe, Arizona, United States; e-mail: aman.kbm@asu.edu.

ACM 1544-3973/2024/11-ART91
https://doi.org/10.1145/3690824

4.80×, and reduces energy by 3.76×. We compare PIMSAB with similarly provisioned state-of-the-art SRAM PIM (Duality Cache) and DRAM PIM (SIMDRAM), and observe a speedup of 3.7× and 3.88×, respectively.

CCS Concepts: • **Hardware → Memory and dense storage**; • **Computer systems organization → Other architectures;**

Additional Key Words and Phrases: Processing-in-memory, bit-serial computing, accelerators, compilers, machine learning

ACM Reference Format:
Siyuan Ma, Kaustubh Mhatre, Jian Weng, Bagus Hanindhito, Zhengrong Wang, Tony Nowatzki, Lizy John, and Aman Arora. 2024. PIMSAB: A Processing-In-Memory System with Spatially-Aware Communication and Bit-Serial-Aware Computation. *ACM Trans. Arch. Code Optim.* 21, 4, Article 91 (November 2024), 27 pages. https://doi.org/10.1145/3690824

1 Introduction

Bit-serial **Processing-In-Memory (PIM)** is a promising accelerator paradigm [11, 14, 21, 22, 29] with both high compute density and abundant on-chip memory capacity, especially considering the recent surge in demands on computing power and memory bandwidth in multiple application domains, including but not limited to deep learning, image processing, and signal processing. The essential principle of this paradigm is to integrate a single-bit **processing element (PE)** at the output of the sense amplifier under each bitline of a memory array so that massive data parallelism can be exploited over a transposed data layout.

This technology provides compute density that is competitive with the state-of-the-art GPUs. The theoretical throughput of a PIM system based on prior technologies [5, 11] is in the range of 310-340 GOPS/mm^2 for int8 precision, for the same area and DRAM bandwidth as that of an NVIDIA A100 GPU. The GPU has a much lower vector throughput of 24 GOPS/mm^2, but has a higher throughput of 755 GOPS/mm^2 for Tensor Cores. However, Tensor Cores can only achieve high utilization for specific kernels and parameters. In addition, bit-serial PIM supports arbitrary precision, which can be extremely beneficial for saving memory bandwidth and increasing compute throughput. The paradigm keeps data near compute units to avoid data movement overhead and thwart the memory wall [44]. Overall, bit-serial PIM is a promising paradigm that has competitive compute density without needing specialized units like Tensor Cores and can be a pathforward for DL workloads.

State-of-the-art PIM systems [14, 18] have showcased improved performance compared to previous generation GPUs. To make PIM systems outperform the state-of-the-art GPUs, we need to fully unlock the potential of the PIM paradigm by taking a system-level approach - co-optimizing hardware and software. Hardware should be carefully architected, given the area budget, to optimize computation and communication. Prior works [11, 18] ignore the overhead in on-chip data communication, which is significant without hardware specialization for common data access behaviors. Similarly, the software can be tuned to make better use of the underlying hardware. Prior works [14, 18, 34] do not enable the software to take advantage of the hardware's bit-serial nature to perform optimized data allocation and computation. Also, though some prior works claim to have a full-stack implementation [14], their programming interfaces are rather low-level. These low-level interfaces limit the productivity of both application development and performance tuning. Furthermore, other PIM systems are either cache-based or DRAM-based requiring new system-level execution models. Such limitations of existing PIM systems motivate us to build a PIM accelerator with easy programmability that can outperform state-of-the-art GPUs and contemporary PIM systems by incorporating multiple novel features that optimize both computation and communication.

Our goal is to build a **Processing-In-Memory (PIM)** system—ISA, microarchitecture and compiler—that can exceed the performance and energy efficiency of similarly provisioned GPUs and prior PIM systems, with a focus on DL workloads. There are two key principles that form the basis of our proposed design: (1) We optimize on-chip **communications to be spatially-aware**: H-tree interconnect topology for faster reductions at lower level of hierarchy & dynamic routing at higher level of hierarchy for scalability, explicit hardware for shuffling (multicasting and broadcasting) operands for common data patterns in DL workloads, and systolic broadcasting; (2) We **optimize bit-serial computations** that are common in PIM architectures: memory allocation and cycles required can expand/shrink dynamically based on precision requirements (adaptive precision) and bit level sparsity can be exploited using constant operations saving space and time. Operations such as reductions, constant multiplication, multicasting, broadcasting are common in workloads like DL.

Our overall system is a hierarchical and spatial PIM accelerator, abbreviated as PIMSAB. PIMSAB uses a hierarchical structure, where each tile is composed of many SRAM arrays capable of bit-serial PIM, along with an instruction controller that broadcasts commands to SRAM arrays in its tile. The ISA enables efficient expression of mixed scalar/vector program regions. The intra-tile network is simple and static for low overhead, and uses an H-tree topology [6] to facilitate high-bandwidth reduction. At the inter-tile level, tiles communicate explicitly, and routing is done over a dynamically routed network to enable flexible parallelization strategies. Further, a mesh-based topology enables scalability to arbitrary sizes. PIMSAB's programming interface is based on the TVM tensor DSL [9], which can be used to express a wide range of applications, including linear algebra, neural networks, and stencil processing. With moderate hints from the developers, the compiler can easily generate portable and high-performance code, by partitioning work across millions of PEs and balancing buffer occupancy and data parallelism.

Our evaluation shows that with sufficient co-design, PIMSAB can rival and surpass state-of-the-art GPUs as well as prior PIM systems. Specifically, we achieve 4.80× speedup over NVIDIA A100, while having 3.76× energy improvement for the same area and the same memory bandwidth. We also observe a speedup of 3.7× with similarly provisioned state-of-the-art SRAM PIM (Duality Cache), and a speedup of 3.88× with similarly provisioned state-of-the-art DRAM PIM (SIMDRAM). To sum up, the contributions are:

- A hierarchical and spatial PIM system with an ISA, a microarchitecture, a compiler and a programming interface.
  - A microarchitecture that deploys dual-ported SRAM arrays with configurable PEs for PIM, without any modifications to the internal SRAM array, as opposed to some of the prior works [11, 14].
  - An ISA that exposes PIM-specific features of the hardware that can be utilized by the compiler.
  - A compiler that can automatically tune the parallelism and on-chip buffer allocation, with moderate hints from the application developer.
  - A user-friendly programming interface using TVM Tensor DSL.
- Employing techniques for spatially-aware communication (shuffle hardware, H-tree for efficient reduction, systolic broadcasting) and bit-serial-aware computation (constant operations, adaptive precision) for high performance.
- Demonstration of GPU-outperforming performance and energy efficiency across both DL microbenchmarks and end-to-end **Deep Neural Networks (DNNs)**.
- Comparison with SOTA SRAM and DRAM PIM systems, showing improved performance for realistic benchmarks.

Table 1. Single-Port Based vs Dual-Port Based SRAM PIM

Feature	Single Port Based	Dual Port Based
Activate two wordlines at the same time on one port	Yes	No
Requires extra voltage source	Yes	No
Requires extra row decoder	Yes	No
Requires modification to sense amps	Yes	No
Compute uses dual-port behavior	No	Yes
Generic/Flexible PE	No	Yes
Cross-RAM shift	No	Yes
Examples	[1, 3, 11, 14]	[5]

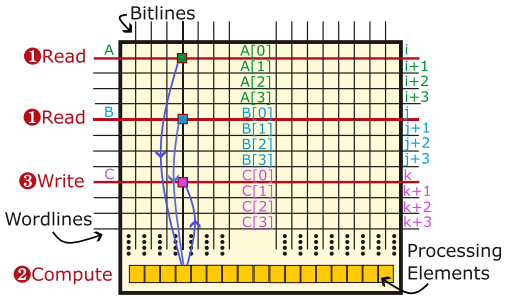


Fig. 1. Basics of bit-serial processing-in-memory.

```
# Vecadd Implementation in Tensor DSL
n = 120*256*256
a,b = tensor((n,),i32),tensor((n,),i32)
i = loop(n); a[i] = b[i] + c[i]
```

```
# Code Organization API
io,ii = split(i, 256)
ioo,ioi = split(io, 256)
parallel.bitline(ii);
parallel.array(ioi);
parallel.tile(ioo)
```

```
# Parallelism distributed
tile x in 0..120
 array y in 0..256
  bitline z in 0..256 {
   i = x*65536+y*256+z
   a[i] = b[i]+c[i] }
```

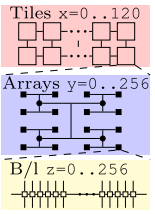


Fig. 2. Programming PIMSAB in tensor DSL.

2 Background

2.1 Bit-serial Processing-In-Memory in SRAMs

Bit-serial computing paradigm performs operations on data bit-by-bit instead of element-by-element. This makes each operation take many cycles, but massive parallelism can be achieved by utilizing simple 1-bit processing elements, enabling high throughput. Bit-serial Processing-In-Memory combines bit-serial computing with Processing-In-Memory. Analog approaches to bit-serial PIM [24, 27] require analog-to-digital and digital-to-analog converters, have high power consumption, and are, therefore, not considered in this work. In digital approaches, 1-bit processing elements (PE) are added to an SRAM block. To provide operands to the PEs, two methods are used: (1) activating multiple wordlines at the same time on one port [1, 3, 11, 14], (2) using dual ported RAMs to read two wordlines at the same time [5]. Table 1 compares various properties of the compute capable SRAM blocks that use these two methods. We use the second method in PIMSAB because this method is more robust and does not modify the memory array (e.g., modifications to sense amplifiers, requiring extra voltage source, adding an extra row decoder).

Figure 1 shows the basic principle of bit-serial In every cycle, two wordlines containing a bit of each operand are activated, the processing element performs the computation and the result is written into a wordline. Operations such as addition, multiplication, and others, can be performed by repeating this basic step over multiple cycles. We refer the reader to Neural Cache [11] for a detailed description of the algorithms for various operations. Note that floating-point and transcendental operations are also supported [14, 21]. processing in memory.

The challenges in prior PIM systems include (1) high on-chip communication overhead in moving partial results across the chip and in organizing the data in the right layout, especially in large systems with thousands of RAMs, even though the off-chip memory traffic is reduced, and (2)

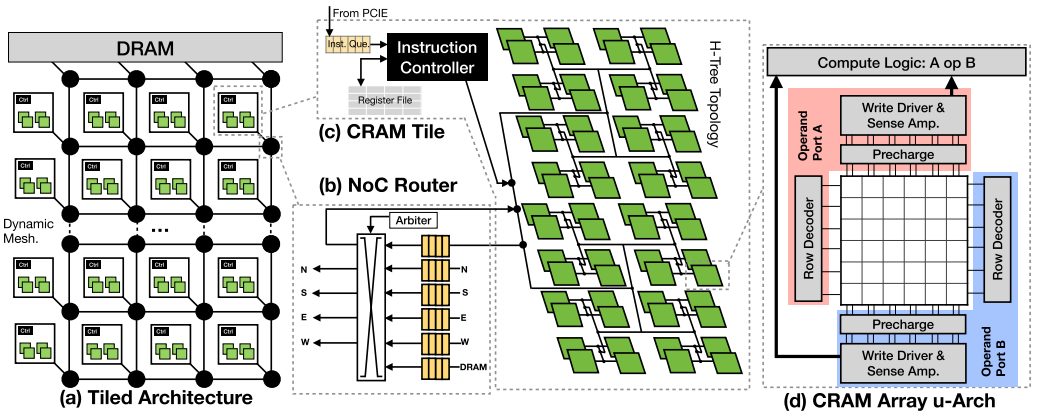


Fig. 3. PIMSAB hardware architecture.

bit-serial compute takes a large number of cycles, especially when the precision expands because compilers allocate number of bits based on traditional paradigms (e.g., int8 $*$ int8 $->$ int32) .

2.2 Tensor Domain-Specific Language (DSLs)

DSLs, like Halide [35], TVM [9], and Tensor Comprehensions [41], are developed to productively write high-performance tensor programs. The idea is to decouple the algorithm and the performance tuning controlled by loop re-organization. Consider the vector addition implementation in Figure 2: Loop variables and tensors are first declared, then a vector addition is implemented in an expression involving these declared variables. DSL allows us to tune the algorithm at an abstract level orthogonally with the specific problem tiling and mapping for a specific hardware. By tiling, ordering, and annotating the loops, the parallelism in the program can be mapped onto our hardware hierarchies.

3 Overview

3.1 Hardware Organization

Figure 3 provides an overview of the hardware organization of PIMSAB. The PIMSAB hardware deploys a large number of compute-enabled SRAMs (or **CRAMs**). Each CRAM is a dual-ported SRAM modified to add multiple single-bit PEs. We base our CRAM design on CoMeFa [5], because of its more practical design compared to Neural Cache [11]. We use their basic block to build a large scalable network of CRAMs with several enhancements for both communication and computation, allowing the PIM to operate efficiently at scale.

To communicate between the CRAMs, a **statically scheduled network** is chosen, since most communication patterns are identifiable at compile time. We choose an **H-Tree** topology for this network, because it is well suited for partial sum reduction, a common computation pattern in DL and many modern applications. Statically scheduling the entire chip would put too much burden on the compiler. So, we introduce another level of hierarchy: **tiles**. Tiles communicate using a dynamically scheduled packet-switched **NoC**. We choose a **2D mesh** topology for the NoC because this enables scalability. The NoC is used to send and receive data across tiles and to/from DRAM. Having parallelism at three levels of hierarchy—CRAM, tile, chip—enables PIMSAB to capture different types of parallelism in highly data-parallel applications.

Each CRAM needs to be fed micro-ops to perform computation. An **instruction controller** decodes the instructions and provides micro-ops to the CRAMs every cycle. However, connecting an

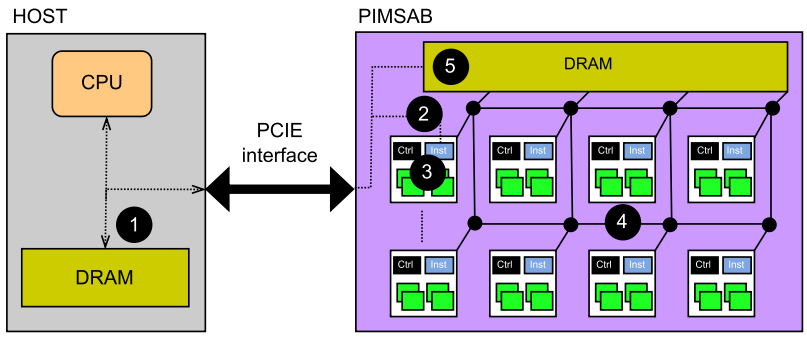


Fig. 4. System level architecture.

instruction controller to each CRAM would result in significant overhead. We reduce this overhead by having one instruction controller in a tile, making CRAMs in each tile operate in a SIMD fashion.

The PIMSAB system defines three memory locations: main memory (**DRAM**), CRAMs and **register file**. **HBM (High Bandwidth Memory)** DRAM is adopted to sustain the high bandwidth required by massive parallelism. To simplify physical design, DRAM controllers are connected to the routers at the edges of the NoC. For similar reasons, we connect all DRAM controllers to the top edge of the mesh NoC. A register file is provided in each tile to store constants or scalars. We assume a PCIe interface, both for loading instructions and for transferring data (like GPUs).

3.2 System Architecture

Figure 4 shows a system level diagram of PIMSAB interfaced with a host machine. PIMSAB is an accelerator/device connected to the host via a PCIe interface, similar to a GPU. Instructions and data are communicated over the PCIe link. We explain the system-level data flow of a vector addition example as follows: ❶ The two input vectors to be added are moved from host DRAM to PIMSAB DRAM. ❷ Instructions of a compiled vector addition program are streamed from the host to instruction queues in PIMSAB. ❸ The instructions are decoded by the instruction controllers and executed within the tiles—computations happen in CRAMs using the PEs and communication happens via on-chip networks. ❹ During instruction execution, the input vectors are loaded into CRAMs from the PIMSAB DRAM and the result vector is stored into the PIMSAB DRAM after computation. The output vector is then transferred back to the host DRAM ❺. Any intermediate synchronization between the host CPU and the device also happens via PCIe.

3.3 Enabling Scalable and Performant Processing-In-Memory

In this section, we present the innovations—for spatially-aware communication and for bit-serially-aware computation—that make PIMSAB a scalable and performant PIM system. We provide a high-level overview of these innovations here; details are provided in the upcoming sections.

Register File and Constant Operations. A frequent operation in applications such as DL is multiplying a scalar (or a constant) with an array or vector operand. With the computation paradigm of bit-serial PIM, we would have to replicate this scalar over multiple bitlines in the CRAM. A more efficient way is to keep scalars outside the CRAM and perform what we call *constant multiplication* (explained in Section 4.2). To store these scalar operands, we introduce a **register file (RF)** in each tile. Additionally, this approach can enable exploiting bit-level sparsity in the constant operand by skipping operations for zero-bits, leading to up to 2× speedup in operations like multiplication and 4× speedup in operations like dot product. This feature is exposed to the compiler through the ISA (`mul_const` instruction).

Dedicated Shuffle Hardware. When data is loaded from the DRAM, it often needs to be broadcasted or multicasted to various CRAMs in different patterns to avoid loading data multiple times or to ensure high utilization of CRAMs in a tile. In addition to just loading data from DRAM, broadcasting or multicasting is useful when data is transferred from one tile to another, or from one CRAM to another, or from one bitline in a CRAM to other bitlines. We provide explicit hardware near each CRAM to support this. Several multicast and broadcast patterns are supported, governed by the requirements of common workloads such as GEMM.

Adaptive Precision. Since PIMSAB uses bit-serial operations, any precision is supported, including floating point (for algorithms of operations using various precisions, we refer the reader to [11] and [14]). In PIMSAB ISA, the precision for each operand can be specified separately. This capability of specifying a custom precision at operand granularity enables using just the number of bits that are required and allocating only the required number of wordlines. For example, when multiplying numbers of precision 8 and 10, 18 wordlines can be allocated to store the result, instead of 32 bits as in a normal CPU. Our compiler exploits this feature to pack as many operands as possible in each bitline (enabling high reuse), even splitting portions of an operand across non-consecutive wordlines, and saving wordlines while performing accumulations by directly adding to those that already have partial results instead of allocating new wordlines.

Cross-CRAM Shift. Shifts are commonly used in operations like stencils, filters, and the like. Vectorization widths in PIM architectures can get really large (e.g., in PIMSAB, the vectorization width for maximum utilization is 256*256). Supporting only intra-CRAM shifting (i.e., shifting data from a bitline to the next within a CRAM using connections between PEs) limits the utility of the shift operation to only a CRAM. To support shifting data from a bitline to the next across the whole vectorization width, we provide CRAM-to-CRAM shift connections within a tile. This gives PIMSAB the ability to perform filters and stencils much more efficiently.

Systolic Broadcasting. Chip level communications, such as broadcast, are essential for workloads such as convolutions, where weights need to be broadcasted to multiple tiles. However, naive broadcast algorithms, like one-to-many transfers, can cause extreme network congestion and overheads. To optimize this, we support a near-neighbor systolic-like data transfer supported in hardware and exposed to the compiler through the `tile_bcast` library function. This efficiently utilizes the available NoC bandwidth and reduces congestion.

Transposing Data: An important feature of bit-serial PIM approaches is the requirement of storing data in a transposed format in the PIM-enabled memory blocks. Transposing data can be challenging and can become a bottleneck in achieving high performance. For DL inference, weights can be transposed prior to execution and stored in DRAM. However, this optimization cannot be applied for inputs or activations. We tackle this challenge by adding dedicated transpose hardware in the DRAM controllers used in PIMSAB. This hardware can be enabled or disabled as needed using instructions.

Hierarchical Interconnect. A two-tiered interconnect is used in PIMSAB. A statically scheduled H-tree interconnect topology at the lower (intra-tile) level is orchestrated internally by each tile's controller and a dynamically scheduled mesh interconnect topology at the higher (inter-tile) level is distributively manipulated by dynamic data packets.

The lower level interconnect enables faster reductions due to H-tree topology and reduces area overhead by simple switch designs. The higher level mesh interconnect reduces the compiler's burden to schedule communications and allows more flexible communication patterns. As a result, combining those two levels increases the scalability of PIMSAB.

4 Architecture

4.1 Instruction Set Architecture (ISA)

In this section, we elaborate the PIMSAB ISA, including Compute, Data Transfer and Synchronization instructions.

Compute Instructions. Compute instructions support arithmetic and logical operations (`add`, `mult`, `or`, `and`, `xor`, `max`, and `min`), operate on data in the CRAMs, and are vectorized across bitlines. We also support inter-bitline instructions, like shifting data across bitlines (`shift`). Instructions to reduce data within a CRAM (`reduce_cram`) and across the CRAMs in a tile (`reduce_tile`) are also provided. We also provide an instruction, `set_mask`, which copies the data of wordline into the mask latches in PEs, to enable predicating operations per bitline. Additionally, each instruction has a field to specify what should be used for predication—the mask latch or the carry latch (Section 4.2 describes the PE architecture including the latches). Precision of each operand can be expressed in the instruction through the `pr*` fields. Exposing precision through the ISA provides more control to the programmer over the benefits of adaptive precision. In most cases, all compute instructions are executed across all the CRAMs in tile, but we also provide a field (called `size`) to specify the number of bitlines involved in the operation across the tile.

Operations with Scalars or Constants. For multiplication operation, a special instruction called `mul_const` is provided where one operand is from the RF (scalar or constant), instead of being replicated in the CRAM. This instruction skips zeros in the constant operand in the RF, reducing the execution time.

Data Transfer Instrucfions. These instructions are used to move data between the DRAM, CRAMs, the RF. Specifically, we support bidirectional data transfer between DRAM and CRAMs (`load` and `store`), as well as DRAM and the RF (`load_rf` and `store_rf`). We support transferring data between CRAMs within a tile (`cram_tx_rx`), and transferring data between tiles (`tile_tx` and `tile_rx`). Such communication blocks the receiver until the data arrives. Broadcasting from one tile to other tiles is supported by a library function called `tile_bcast`. Two modes of broadcasting are supported—(1) one_to_many, in which one tile sends data to all receivers, and (2) systolic, in which each tile receives data from one neighbor and passes it to another neighbor.

Data Shuffling Instructions. When loading data from DRAM into CRAMs, the shuffling can be enabled by using the `load_shf` instruction. The `shf` argument specifies the shuffling pattern and the `bcast` bit specifies whether broadcast is enabled or not (see the Shuffle Logic subsection of Section 4.2 for details). Furthermore, we provide the capability to shuffle data that is already stored within a CRAM, using the `cram_local_shf` instruction.

Synchronization Instructions. These instructions coordinate data transfers and computations among tiles. Two synchronization instructions provided are `signal` and `wait`. `signal` sends a message from a source CRAM to a destination CRAM and is non-blocking. A CRAM can wait for a message (blocking) from a source CRAM using the `wait` instruction.

Transposing Data. In load and store instructions, besides the source address, destination address, size and precision, there is an additional `tr` field specifying if the data is transposed or not. This can be used when, e.g., an immediate/constant operand read from the main memory need not be transposed.

Program Example. A simple elementwise vector multiplication is shown in Listing 1. The program generates an instruction stream for all tiles in the chip (`NUM_TILES`). Two operand arrays, each with elements of precision `int8`, are loaded from the main memory. `vec_width` is specified to be the full width of a tile. Then a multiplication instruction is used to generate a result with precision `int6`. The result is then stored back in the main memory.

Table 2. Microarchitectural Parameters of PIMSAB

Parameter	Value
CRAM geometry	256×256
PEs per CRAM	256
CRAM size	8 KB
Mesh dimensions	12×10
DRAM bandwidth	12288 bits/clock
Clock frequency	1.5 GHz
Num tiles	120
Num CRAMs per tile	256
Total CRAMs	30720
RF size	32 32-bit regs
Tile-to-Tile bandwidth	1024 bits/clock
CRAM-to-CRAM bandwidth	256 bits/clock

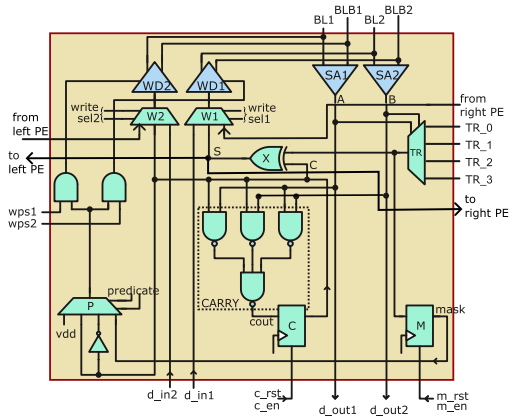


Fig. 5. Processing element used in CRAMs.

```
int vec_width = NUM_CRAMS_IN_TILE * NUM_BITLINES_IN_CRAM;
for (i = 1; i<NUM_TILES; i++) {
  load tile_addr1, dram_addr1, vec_width, i8
  load tile_addr2, dram_addr2, vec_width, i8
  mult tile_addr3, i16, tile_addr2, i8, tile_addr1, i8
  store dram_addr3, tile_addr3, vec_width, i16 }
```

Listing 1. Simple program to add two arrays

4.2 Microarchitecture

Here we discuss PIMSAB's microarchitecture. Table 2 provides a list of hardware parameters.

CRAMs. We employ dual-ported **compute-enabled RAMs** (called *CRAMs*) similar to CoMeFa [5]. A CRAM has two modes: compute and memory. In compute mode, a data word written into the memory is treated as a micro-op. Each micro-op takes 1 cycle during which two wordlines are read (one on each port), computation is performed in the PE, and the result is written into a wordline. In memory mode, the CRAM behaves like normal RAM. CRAMs are grouped into tiles; all CRAMs in a tile execute in lock-step in a SIMD fashion (except when executing CRAM-to-CRAM data transfer or inter-CRAM intra-tile reduction). CRAMs in a tile are connected using the intra-tile network. In addition, there is a single wire ring interconnect between all CRAMs in a tile to facilitate `shift` instructions.

Processing Element (PE). PIMSAB adopts the PE architecture from CoMeFa [5], as shown in Figure 5. Each PE can perform any logical operation between two operands, using the **TR** mux. The **TR** mux allows the PE to be more flexible, compared to [11]. With the addition of an XOR gate (**X**), it can perform a 1-bit full adder operation. A carry latch (**C**) is used to store the carry-out, which can be used as carry-in for the next timestep. The output of the TR mux can be stored in the mask latch (**M**). Predication based on mask bits and carry bits is supported, through the predication mux (**P**). There are as many PEs in a CRAM as many bitlines. The operation performed by the PE is governed by the micro-op received by the CRAM from the instruction controller. The micro-op bits are decoded in the CRAM's sequencing logic and generate the various control signals present in the PE. The write select muxes **W1** and **W2** select what to write back to the bitlines using the write drivers - data from left or right PEs, or the sum or carry output calculated by this PE. In each cycle, the PE receives two bits of operands from the sense amplifiers and performs the operation specified by the micro-op. The result of the operation is then written back into the CRAM through the write drivers (unless predicated off).

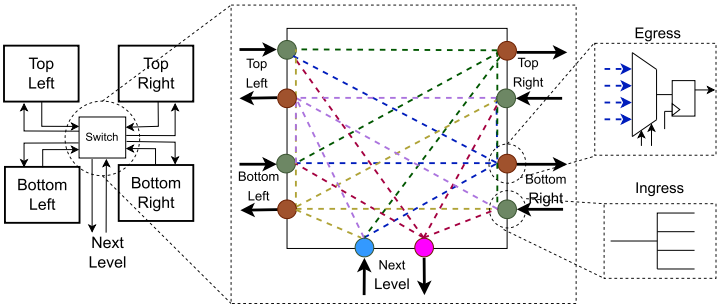


Fig. 6. Structure of a switch in the intra-tile H-Tree network.

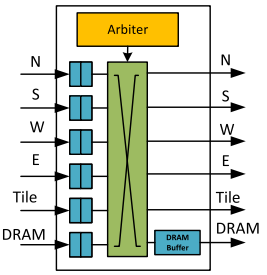


Fig. 7. Structure of a router in the inter-tile mesh network.

Instruction Controller. Instructions are received from the HOST over PCIe. Each tile has an instruction controller to decode and farm-out execution to corresponding units. For compute instructions (add, multiply, reduction, etc.), it generates micro-ops for the CRAMs every cycle. For data transfer instructions (CRAM-to-CRAM transfer, tile-to-tile transfer, DRAM transfers), it reads the CRAM and sends data into the static network's switches, and also writes data coming in from the switches into the CRAMs.

Inter-Tile Dynamic Network. The inter-tile network uses a standard wormhole-switched dynamic NoC, with X-Y routing. Each router, shown in Figure 7, has a crossbar connecting five input and output ports—local tile, north, south, west, east. Routers connected to DRAM have an extra input/output port to receive/send data from/to DRAM. The transferred data is broken into flits (flow control units). Each input port has multiple circular queues to buffer input flits into multiple virtual channels. Upon sending, header and data flits are pushed into a circular queue of the local tile port one after another. Each router performs wormhole switching on the incoming flits. Upon decoding the flit header, the router controller performs minimal routing to route incoming flits towards their destination. Upon receiving, data flits are popped from one of the input queues selected by the crossbar. Due to simple flow-control and routing strategy, small flit and queue sizes, area and power overheads of routers are minimized.

We choose a mesh topology to connect the tiles as opposed to a ring topology. A mesh topology helps in scalability and reduces the burden on the compiler. Section 7.7 shows the advantage of mesh topology compared to the ring topology.

Intra-Tile Static Network. The intra-tile network is a static circuit-switched network using an H-Tree topology. This is similar to a hierarchical FPGA [2, 39], but with a much smaller configuration overhead because of the coarser granularity (word-level instead of bit-level). Figure 6 provides the details of the microarchitecture of a switch in the intra-tile network. Each switch is a buffered crossbar with five ingress (input) and egress (output) ports. Each output port can be driven by the other four input ports—three from other directions at the same hierarchical level and the fourth from the next level of hierarchy (shown using separate colors in the figure). There is one switch at the top of the tile that interfaces with the NoC router of the inter-tile network. For a tile with 256 CRAMs, there are four levels of switches, for a total of 1 + 4 + 16 + 64 = 85 switches. The last set of 64 switches are connected to 256 CRAMs. The intra-tile network reconfigures its switches when a incoming data transfer instruction indicates new communication pattern.

Reductions can be performed on data within a single CRAM, using an algorithm similar to [11]. We refer to this as intra-CRAM reductions. This method requires iteratively shifting values across bitlines and adding them. Moving bits to adjacent bitlines takes 1 cycle, but moving bits to a bitlines

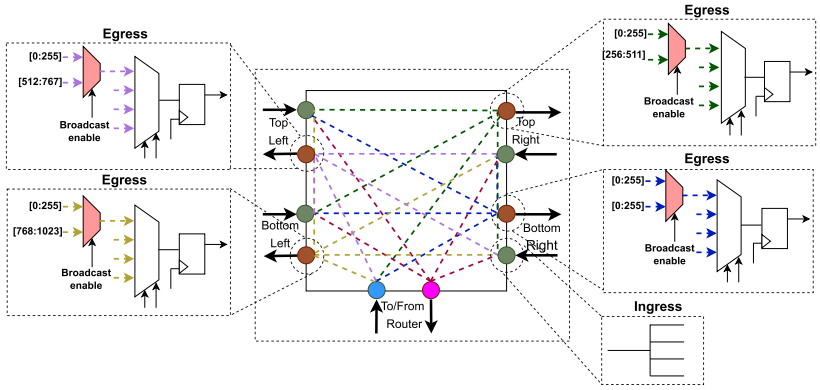


Fig. 8. Shuffle logic at input of each tile in PIMSAB.

N-bitlines away takes N cycles, because each bit has to be shifted cycle-by-cycle, based on the connectivity across PEs provided in the CRAMs. The number of cycles consumed in the reduction operation is linearly related to distance (in terms of bitlines). Furthermore, the number of bitlines utilized reduces as the reduction operation progresses, and the result is available in the first bitline of the CRAM.

The H-Tree topology in PIMSAB facilitates pairwise reduction across multiple CRAMs. We refer to this as inter-CRAM (or intra-tile) reduction. Data to be reduced are transferred across pairs of CRAMs through levels of the H-Tree and added at each level. Therefore, the reduction time is logarithmically related to the number of CRAMs that the operand occupies. As a result, inter-CRAM reduction is more efficient than intra-CRAM reduction and is prioritized by our compiler. The number of utilized CRAMs in a tile reduces as the reduction operation progresses, and the results are available in the first CRAM of each tile.

Shuffle Logic. Operations like GEMM and convolution can greatly benefit from custom data layout patterns. For example, we may need a data element to be duplicated in each bitline or repeated every four bitlines in a CRAM. These custom layout patterns can be achieved by data duplication in the CRAM thereby avoiding unnecessary traffic from/to DRAM. We refer to this duplication of data in various layouts as shuffling. We implement dedicated hardware in PIMSAB to enable efficient shuffling of data. This hardware is implemented in two parts. The first part is implemented at the input of each tile, by employing careful modifications to the structure of the top-level intra-tile switch. This hardware provides the capability to broadcast data received at the top of the tile to each CRAM in the tile. The second part is implemented at the input of each CRAM. Additional multiplexing hardware is provided to enable common data patterns observed in the DL benchmarks.

Figure 8 shows the modifications done to the top-level intra-tile switch to support shuffling. The data coming from the NoC (skyblue circle) goes to all the ports as shown previously in Figure 6, but now an additional red multiplexer is added on each port. The first input of all the red multiplexers is data bits 255:0. Thus, the lower significant 256 bits from 1,024 bits received at the NoC router are broadcasted to all the four ports and flow through the intra-tile network of switches to CRAMs. The second input of the red multiplexer is the normal path, through which different set of bits received from the NoC router are sent downstream to the CRAMs through the intra-tile network. The selection between these inputs is controlled by a broadcast enable bit. If broadcast is enabled, all ports (and hence the CRAMs in the tile) receive the same 256 bits of

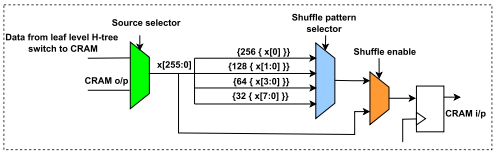


Fig. 9. Shuffle logic at input of each CRAM in PIMSAB.

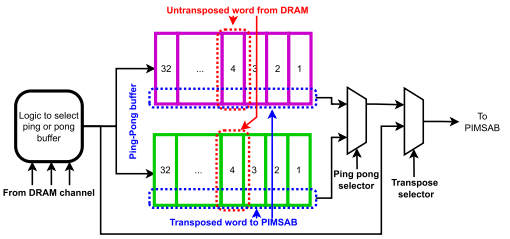


Fig. 10. Transpose unit.

data. This broadcast enable is exposed to the compiler through a `bcast` argument in the `load_shf` instruction.

Figure 9 illustrates the additional hardware designed to shuffle bits at the periphery of each CRAM. This unit enables data shuffling in four distinct patterns. The source selector multiplexer, shown in green, enables choosing between the data coming from the leaf level intra-tile switch, or the output of the CRAM itself. The former is used when data originating from outside (either in DRAM or another tile or another CRAM) is being written into the CRAM. Currently, we only support shuffling data loaded from DRAM through the `load_shf` instruction. The latter is used when data already present in a CRAM needs to be shuffled and written back to the same CRAM. This path is supported through the `cram_local_shf` instruction. The shuffle pattern selector multiplexer, shown in blue, allows selecting between four data patterns generated by shuffling the data bits of the data output by the source selector. For instance, the first pattern duplicates the 0th bit 256 times, while the fourth pattern duplicates bits 7:0 32 times. The functionality of this multiplexer is exposed to the compiler through the `shf` argument in the `load_shf` and `cram_local_shf` instructions. Eventually, a shuffle enable bit of orange multiplexer selects whether to enable shuffling or not, and the resulting data is written into the CRAM. Shuffling is disabled if `load_shf` or `cram_local_shf` instructions are not used.

DRAM Interface and Transpose Unit. All tiles in the top row of the mesh NoC are connected to DRAM controllers. The data from DRAM must be transposed before storing into CRAMs, so that bit-serial arithmetic can be performed. Results need to be untransposed when writing back. In PIMSAB, transpose units are integrated within the DRAM controllers. This unit can be disabled if not needed (through the `tr` field of the DRAM load/store instructions). Some common situations where transpose is not required include loading the RF and reading/writing spilled data during operations. Another example is for deep learning, where we enable this for input activations while we disable it for weights, as weights can be stored pre-transposed in DRAM. We use the transpose unit shown in Figure 10 similar to CoMeFa's [5]. It employs a ping-pong FIFO. Data enters from one side into the ping part in the non-transposed format. When full, transposed output is obtained by reading bit slices of the loaded elements, while the pong part is filled with new data. When the pong part is full, the roles are reversed and the process repeats. The bandwidth for each DRAM channel in PIMSAB is 1024 bits per cycle. There are 32 transpose units for each DRAM channel. 32 bits can be read from 1 transpose unit in 1 cycle. For the highest bandwidth achievable, the transpose unit adds a latency of 32 cycles. Data of different precisions can be handled and transposed using the transpose unit.

Register File and Operations with Constants or Scalars. Many applications, including DL, heavily rely on constant operations like vector-scalar multiplication. Instead of replicating the constant operand in all bitlines like Figure 11(a), PIMSAB holds the constant operand in a register file (RF) present in every tile. Figure 11(b) shows the operation of the instruction `mul_const`. After the

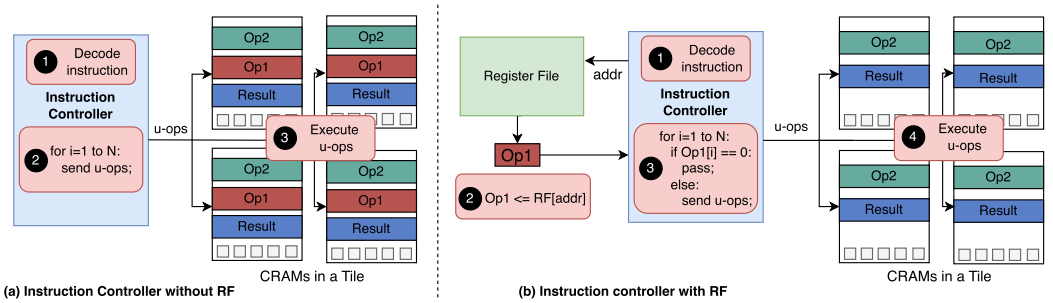


Fig. 11. Flow for constant operations.

instruction is decoded ❶, the instruction controller fetches the scalar operand from the RF ❷, and sends micro-ops to the CRAMs according to the bits of the constant that are set. When a bit of the constant is zero, corresponding computations and micro-ops can be skipped ❸. Finally, the CRAMs execute the u-ops to perform the computation ❹. Besides exploiting such bit-level sparsity, constant operations also save CRAM space and reduce data spilling to DRAM. The RF is flip-flop-based and does not have any port restrictions. Any number of registers can be read and written in each cycle. Instructions `load_rf` and `store_rf` are provided to load/store RF from/to DRAM.

5 Compiler

5.1 Overview

Programming Interface. We adopt a tensor DSL as our high-level interface, because of its portability and ease of performance tuning. As shown in Figure 12(a), a matrix multiplication is implemented in a tensor DSL, by declaring loops, and tensors, and describing the program behaviors in expressions involving these declared variables. The loop organizations are the key to the performance tuning in tensor programs, and can be easily explored by invoking several loop organization primitives (e.g., `split` and `reorder` shown in Figure 13). The parallelism is naturally encoded in the declared loops with different types, either data-parallel or reduction. These different loop types may lead to different program behaviors when mapping loops to different hardware hierarchies.

Performance Tuning. Different implementations significantly affect the on-chip buffer occupancy, memory traffic, on-chip network traffic, and parallelism distribution, and lead to different performances. Considering the excessively large space of code organizations, we decide to leave loop organization and data layout tunings to developers, so that the compiler can figure out the best parallelism distribution and buffer allocation under such organization and layout.

To explain, consider the matrix multiplication example in Figure 12(a). Its imperative version in Figure 12(a') shows that the innermost reduction loop is hard to parallelize across bitlines specialized for vector parallelism. Thus, one important transformation is to place a data-parallel dimension in the inner loop. As shown in Figures 12(b) and 12(b'), the outermost dimension of tensor `a` is tiled by 256, and reordered to the innermost for mapping to bitline PEs. Then, Figure 13 shows that users are required to call the loop organization APIs to determine a code organization for the compiler to distribute the parallelism and allocate CRAM memory buffers.

Compiler Analysis and Optimizations. After the data layout and loop organization are determined, the compiler analyzes the program. It distributes parallelism to hardware hierarchies, and performs memory buffer allocation (Section 5.2). Then it performs CRAM data optimizations (Section 5.3) that take advantage of the properties of bit-serial arithmetic to reduce on-chip memory occupancy. Since the exploration space of parallelism distribution and memory buffer allocation is

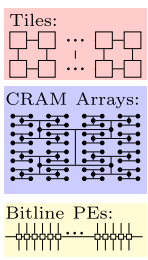
```
# (a) Vanilla Matrix Multiply
n,m,p = 12*256*64, 10*32, 1024
a = tensor((n, p), i8)
b = tensor((m, p), i8)
x, y = loop(0, n), loop(0, m)
k = red_loop(0, p)
c[x,y] =
    sum(i32(a[x,k])*i32(b[y,k]))
```

```
# (a') Imperative IR
for x in 0..n
 for y in 0..m {
  c[x,y] = i32(0)
  for k in 0..p
   c[x,y] +=
     a[x,k]*b[y,k]
}
```

```
# (b) Relayout Matrix Multiply
a = tensor((n/256, p, 256), i8)
b = tensor((m, p), i8)
xo, xi = loop(0, n/256), loop(0, 256)
y = loop(0, m)
k = red_loop(0, p)
c[xo,y,xi] = sum(
  i32(a[xo,k,xi])*i32(b[y,k]))
```

```
# (b') Imperative IR
for xo in 0..n/256
 for y in 0..m {
  c[xo,y,0..256] = i32x256(0)
  for k in 0..p
   # xi "vectorized" 0..256
   c[xo,y,0..256] +=
    a[xo,k,0..256]*b[y,k] }
```

Fig. 12. Matrix-matrix multiplication implemented in tensor expression language and array packing.




```
# Code Reorganization API called by user
xo.o, xo.i = split(xo,64)
y.o, y.i = split(y,32)
reorder(xo.o, y.o, k, xo.i, y.i)
```

```
# Tiled imperative IR
for xo.o in 0..12 # to tiles
 for y.o in 0..10
  for k in 0..1024    # to arrays
   for xo.i in 0..64 # explored by compiler
    for y.i in 0..32
     for xi in 0..256 { # b/l vectorization
      xo, y = ... # compute tiled indices
      c[xo,y,xi] += a[xo,y,xi]*b[y,k] }
```

Fig. 13. Reorganize the loops for parallelism distribution.

small, the compiler exhaustively evaluates each point, and adopts the one with the best objective. As shown in Figure 14, the parallelism distribution affects the memory buffer allocation. If the required buffer size exceeds the on-chip resources available, this exploration point is considered invalid. To make more exploration points more likely to succeed, CRAM data optimizations will squeeze the buffer size requirement.

Code Generation and Feedback Loop. After the favored parallelism distribution and buffer allocation is *successfully* determined on the given loop organization, the compiler extracts all the computational instructions to be offloaded to PIM and rewrites them in hardware intrinsics. Then, the transformed IR is ready for code generation. If all the parallelism distribution fails under the given loop organization, the compiler will throw an error to the developer, and the developer is required to find another more conservative loop organization.

5.2 Parallelism Distribution & Memory Allocation

Parallelism distribution determines how much of these loops should be tiled and parallelized across hardware hierarchies, and how much should be executed in serial. Since the parallel degree of each hierarchy is at an order of hundreds, the loop tiling space is small enough for the compiler to search exhaustively. Next, we explain how the loops are mapped to parallelism across and within tiles.

Inter-Tile Parallelism Distribution. Data communication between tiles (which happens using dynamically routed NoC) is expensive compared to the data communication between CRAMs within a tile (which happens using the static H-tree interconnect). Considering the overhead of communicating data between tiles, it is often inefficient to reduce the partial sum across different tiles. Therefore, our compiler only seeks to map data parallel loops to inter-tile parallelism. Assuming we have 120 tiles, each iteration of `xo.o` and `y.o` in Figure 13 are mapped to each tile exactly. If the iterations exceed the number of tiles, the compiler will seek to tile the loops and execute parts serially.

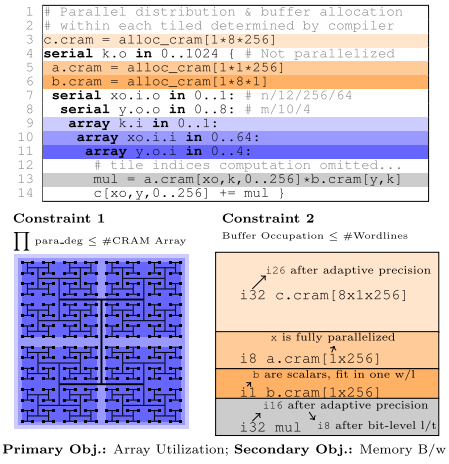


Fig. 14. Distributing the parallelism intra-tile.

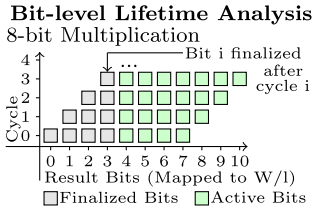


Fig. 15. Bit-level lifetime analysis.

Intra-Tile Parallelism Distribution. Figure 14 shows that, after the inter-tile parallelism is fixed, the compiler distributes the intra-tile parallelism by exploring the space of loop tiling. Each tiled outer loop (with `.o` suffix) will be executed in serial by each tile's controller, and each tiled inner loop (with `.i` suffix) will be mapped to a CRAM array. The total iterations of these `.i`-loop should not exceed the number of arrays. This can easily be enforced when tiling the loop by multiplying the tiling factor. Besides, the CRAM buffer allocation should not exceed the wordlines available in each array. When there are multiple distributions that fulfill these two constraints (parallelization degree and CRAM buffer), we use two objectives to determine the best one. The primary objective is more computing resource occupancy, and the second is less DRAM memory bandwidth. The rationale behind this objective order is that a high computing resource occupancy often requires high data bandwidth to sustain.

CRAM Buffer Allocation. CRAM buffer allocation is the key to determining the feasibility of a parallel distribution. Here, we first explain how the compiler greedily exploits data reuse, and compute occupancy, while not exceeding the CRAM capacity. Some overused CRAM capacity can be false positive and eliminated through adaptive precision. If it turns out to be a true overuse, feedback will be sent to the developer for a conservative initial loop organization. For the example shown in Figure 14, the compiler greedily allocates the memory buffer at the highest serialized loop with reuse. Then, the compiler tries to minimize the CRAM buffer occupancy by analyzing the data access pattern of each operand. In the case shown in Figure 14, the size of each buffer is proportional to the iterations of serialized loops. For example, the `c.cram` buffer size is 1×8×32, where 1 is the serial iteration of `xo.i.o`, 8 is the serial iteration of `y.o.o`, and 32 is the precision of the integers. Parallelizing `xo.i` is favored, because it saves more buffer occupancy for both `a`&`c`, considering `b.cram` are all scalars. After fully parallelizing `xo.i`, `y.o` is further parallelized to fill the remaining arrays for compute resource occupancy. Finally, `c` buffer, 1×8×32 = 256 wordlines, already occupies each entire CRAM array, with no space remaining for other operands, intermediate values, which indicates the unfeasibility. In the next section, we will explain how we squeeze CRAM allocation to optimize this false overuse.

5.3 Optimizing CRAM Data through Adaptive Precision

False overused CRAM allocation can be optimized so that potentially more aggressive parallel distribution can be feasible. For the example in Figure 14, 32 ×8 = 256 wordlines are required for

the accumulated results (`c.cram`), 8 ×1 = 8 wordlines for the operand `a`, 1 wordline for the operand `b`, and implicit 32 wordlines for the intermediate results. In total, 256 + 8 + 32 + 1 = 297 wordlines, which exceeds 256 wordlines of each array. Next, we will explain how our optimizations make them fit. These optimizations mostly take advantage of the divisible nature of bit-serial arithmetic—each bit of results is independently accessible. We call these optimizations *Adaptive Precision*.

The minimum feasible precisions are adopted to override the precision in the original program: Multiplying an a-bit and a b-bit number is at most $a+b$ bits; accumulating k a-bit numbers requires only $a+\lceil \log(k) \rceil$ bits. Specific to the example shown in Figure 12, though the results are accumulated on `i32`, only `i26` is required. The input operands are both `i8`, so the result of multiplication is within `i16`. We now have `p = 1024 i16` accumulated, in total $\log_2 1024 + 16 = 26$ bits for each accumulation. Therefore, we in total saved (32 - 26) x 8 + 16 = 64 wordlines, so now only 249 wordlines are required. In addition to saving space in CRAMs, using this technique also improves performance. Without this technique, existing data in wordlines would have to be spilled to DRAM and read back later, consuming additional cycles. Alternatively, the compiler would need to use a less aggressive parallelization strategy leading to lower performance.

The divisible nature of bit serial arithmetic makes each bit have its own lifetime. We extend the code scheduling in CHOPPER [34] to an even broader applicability. For a multiplication that is consumed immediately by an addition, instead of keeping the whole 16 bits of multiplication, we can add it to the accumulator as soon as a bit is finalized. As shown in Figure 15, after i cycles of multiplication, the ith bit is finalized, and it always maintains a half-width active window when doing a multiplication. Therefore, this saves 16/2=8 wordlines, and now we occupy only 243 wordlines.

5.4 Data Loading & Packing

Once parallelism distribution passed the constraint check after these optimizations, the compiler will then inspect the memory access pattern to generate code for data loading and packing. If several tiles are using the same data, the common memory traffic will be converted to on-chip network communication. The compiler uses data transfer instructions such as `tile_tx` and `tile_rx` for such cases. If the data needs to be broadcasted to multiple tiles, the compiler invokes library function `tile_bcast` provided for broadcasting data across tiles. This broadcast can be invoked using the systolic mode. The compiler also analyzes the operands loaded to the CRAM arrays within a tile to determine how they should be shuffled. For example, `xo` is not related to reading `b.cram[y,k]`, so loaded `b` should be broadcast to all the arrays mapped to different iterations of `xo`. For such cases, the compiler uses the `load_shf` or the `cram_local_shf` instructions with the shuffle pattern specified through the `shf` argument. The shuffling pattern can be broadcast or different types of multicast as described in Section 4.2. When constants or scalars are used for multiplications, the compiler uses the `load_rf` and `store_rf` instructions for loading data to the RF and storing data from the RF respectively, and the instruction `mul_const` is subsequently used for the computation operations.

5.5 Implementation

We integrate our compiler analysis and transformations to the TVM compilation flow. TVM provides rich code organization primitives to tune loop organizations and allocate memory buffers. We start with the initial code organization provided by user, and apply our explored parallelism and memory allocation. This is lowered to an IR with all the loops and buffers instantiated. Then CRAM data optimization is done on this level of IR. If the memory occupancy satisfies the hardware constraints, all the arithmetic operations are rewritten in bit-serial intrinsics, including arithmetic operations, memory loading, and data transfer, which are ready for code generation. If not, we invoke a feedback loop to explore a more conservative code organization for less memory occupancy.

6 Evaluation Methodology

6.1 Modeling PIMSAB

We develop a cycle-accurate simulator in C++ to model the PIMSAB hardware. The simulator executes a program written in the ISA described in Section 4.1. An input configuration file is used to specify various parameters of the microarchitecture. The values of area and energy for various blocks and instructions (as described below) are incorporated into the simulator. Various metrics like cycles and energy breakdown by component or instructions are reported by the simulator. We validated the simulator using simple handwritten kernels first and then with microbenchmarks. Thus, our model is well calibrated and realistic.

We use a Verilog behavioral model of the CRAM to obtain instruction cycle counts. For the PEs in each SRAM, we write transistor level code and evaluate area and energy using SPICE with 22 nm ASU PTM technology [40]. Area and power of RAMs is obtained using the OpenRAM memory compiler [17]. We write Verilog RTL for components such as the static network's H-tree switch, shuffle logic, instruction controller, transpose unit, and register file. We use Synopsys VCS for simulation and Synopsys Design Compiler (using FreePDK45 [31]) to obtain post-synthesis area and power. We further assume a 15% area overhead for place and route [20]. For the dynamic NoC, we use PAT-Noxim simulator [33] and extract area and energy values for the routers and links. For cycles required for packet transmission through the dynamic NoC, we model the NoC in our simulator. For the on-chip DRAM and PCIe controllers and transceivers, we obtain areas from A100 die analysis. For DRAM energy, we use a simple analytical model calibrated from memory-only microbenchmarks on the A100. We scale all values to 22 nm using scaling factors for area, power and delay from [38].

6.2 Baselines

GPUs are the most common commercially available accelerators for DL workloads; so we compare PIMSAB against NVIDIA A100 GPU. Additionally, we compare against state-of-the-art prior SRAM and DRAM based PIMs (DualityCache and SIMDRAM). To make fair comparisons, we build three different PIMSAB configurations for each of the comparisons. Table 3 shows these different architectures and compares their type, programming model, and level of automation with the ability to handle reuse. Table 4 shows the configuration details of these architectures.

NVIDIA A100 GPU. We provision PIMSAB to have the same area (825 mm^2 in 7nm, i.e., 2,950 mm^2 in 22 nm) and DRAM bandwidth (12,288 bits/cycle, i.e., 1,866 GB/s @ 1,215 MHz). We also assume the same DRAM size as A100 (40 GB), and all benchmarks fit in this memory. PIMSAB has a memory controller to interface with external HBM DRAM, similar to A100. GPU performance is measured by running on an A100 using NVIDIA's profiler NSight Compute. Each kernel is measured by averaging 500 launches to exclude the device overhead. To compare the dynamic energy of PIMSAB with A100, the static energy is normalized indirectly to A100 through having the same area footprint and DRAM bandwidth. The NVIDIA GPU uses CUDA as the programming language. NVIDIA's CUDA compiler and support in high-level frameworks such as PyTorch provides high degree of automation, enabling expressing workloads at a high level of abstraction. With caches on the GPU, the workloads can take advantage of data reuse.

Duality Cache. **Duality Cache (DC)** [14] is an in-cache SRAM PIM architecture that builds on Neural Cache [11], and uses a subset of CUDA as a programming language. It is the state-of-the-art SRAM PIM architecture with a full-stack implementation, similar to PIMSAB. This makes DC relevant to compare with PIMSAB. DC integrates PIM into on-chip CPU caches whereas PIMSAB is a dedicated accelerator chip. DC has 1.14 million processing elements and runs at a frequency of 2.6 GHz. We design a PIMSAB chip (PIMSAB-D) sized to match the compute throughput of DC

Table 3. Qualitative Attributes of PIMSAB and Baselines

Architecture	Type of chip	Programming model	Level of automation	Ability to handle reuse
PIMSAB	Accelerator	Tensor DSL	High	Yes
Duality Cache	Cache	CUDA	High	Yes
SIMDRAM	DRAM chip	Manual coding using intrinsics	Low	No
GPU Nvidia A100	Accelerator	CUDA	High	Yes

Table 4. Configuration Details PIMSAB, Baselines, and PIMSAB Variants Provisioned for Comparison

	PIMSAB	Nvidia A100	PIMSAB-D	Duality Cache	PIMSAB-S	SIMDRAM
Compute Throughput ($int8$)	352 TOPS	624 TOPS (Tensor Core), 19.5 TOPS	49.5 TOPS	49.5 TOPS	24.75 TOPS	27 GOPS
Area (mm^2)	825	826	206 (7nm)	471 (22nm)	103 (7nm)	–
Frequency (GHz)	1.5	1.4	1.5	2.6	1.5	2.4
On-Chip memory (MB)	256	87	64	35	32	16384
DRAM B/W (GB/s)	1866	1866	933	–	777	19.2 (external), 4.68 TB/s (internal)
Number of PEs	7864320	–	1966080	1146880	983040	983040

for a fair comparison. PIMSAB-D has 30 tiles (organized in a 6 ×5 mesh). The CRAM size for DC is the same as for PIMSAB (256 bitlines ×256 wordlines). DC provides compiler support, enabling high degree of automation. Since DC is a cache-based architecture, workloads can take advantage of data reuse.

SIMDRAM. SIMDRAM [18] is a DRAM PIM architecture and was shown by the authors to perform better than DC for some workloads. We use the 1 bank configuration mentioned in their article, when comparing SIMDRAM to PIMSAB. We design a PIMSAB configuration (PIMSAB-S) to match the number of processing elements in SIMDRAM. PIMSAB-S has one tile. We use three full Deep Neural Networks (LeNet, VGG-13, and VGG-16) as benchmarks for comparison since they were used in the SIMDRAM work. SIMDRAM does not have a compiler. The programming is done using intrinsics, which is tedious and error prone. Since SIMDRAM is a DRAM-based architecture, it cannot exploit the data locality inherently present in compute-intensive workloads like GEMM.

6.3 Benchmarks

Table 5 lists the benchmarks, along with input size and precision, used in our evaluations. When comparing with A100, we choose a set of microbenchmarks from high-performance libraries, including ArrayFire[45] (`fir`), and CUTLASS[28] (`gemm`, `gemv`, `conv2d`). These microbenchmarks represent the fundamental operation in popular applications like deep learning and signal processing. We run full networks like quantized Resnet18 from MxNet Model Zoo and BERT to show our support for end-to-end applications. We also support various precisions like FP32, INT8, INT16, making our system precision agnostic. Our kernels are using mixed precision—the input data, the compute operations, the storage use different precisions. PIMSAB's adaptive precision feature is more sophisticated than NVIDIA Tensor Core's mixed precision support because it can support arbitrary precisions, e.g., different stages of accumulation can use different precisions. PIMSAB's architecture can support other mixed precision modes such multiplying operands of two different precision; however, this is not used in our benchmarks currently.

To ensure an apple-to-apple comparison with DC and SIMDRAM, we use benchmarks used by those works. DC uses Rodinia benchmarks for their evaluation. So, we also use Rodinia benchmarks to compare with DC directly. However, some benchmarks from the Rodinia suite were

Table 5. Benchmarks used for Evaluation

Benchmark	Size	Precision	Comparison	DRAM Usage
vecadd	input=15728640	int8	A100	RD=30 MB,WR=15MB
fir	input=7833600, filter=32	int16, acc=int16	A100	RD=15.6MB,WR=15.6MB
gemv	m=61440, k=2048, n=1	int8, acc=int32	A100	RD=128MB,WR=262KB
gemm	m=61440, n=32, k=2048	int4, acc=int16	A100	RD=62MB,WR=3.9MB
conv2d	input=9×9×256×2, weights=3×3×256×256	int8, acc=int32	A100	RD=815KB ,WR=100KB
resnet18	input=224×224×3×1, output=1000×1	int8, acc=int32	A100	RD=6.1MB,WR=1.3MB
bert	input=384×768, output=384×768	int8, acc=int32	A100	RD=26 MB,WR=9MB
backprop	input=65536×16	fp32	DC	RD=8.3MB,WR=4.1MB
dwt2d	input=1024×1024	fp32	DC	RD=4.1MB,WR=6.2MB
gausselim	input=256×256	fp32	DC	RD=512KB,WR=262KB
hotspot	input=1024×1024	fp32	DC	RD=20MB,WR=4.1MB
hotspot3d	input=512×512	fp32	DC	RD=16.7MB,WR=8.3MB
vgg13	input=224×224×3×1, output=1000×1	binarized	SIMDRAM	RD=4.9MB,WR=65KB
vgg16	input=224×224×3×1, output=1000×1	binarized	SIMDRAM	RD=5.9MB,WR=65KB
lenet	input=32×32, output=10×1	binarized	SIMDRAM	RD=139KB,WR=16

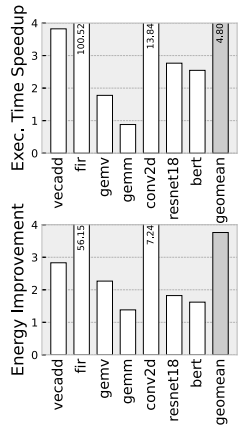


Fig. 16. Comparing PIMSAB with NVIDIA A100.

excluded from comparison because those benchmarks have irregular parallelism or indirect memory accesses, which are not supported by PIMSAB. For SIMDRAM comparison, we use three binary neural networks: VGG-13, LeNet, and VGG-16. We obtained the raw runtimes numbers for the benchmarks they used by directly reaching out to the authors and used those for comparison.

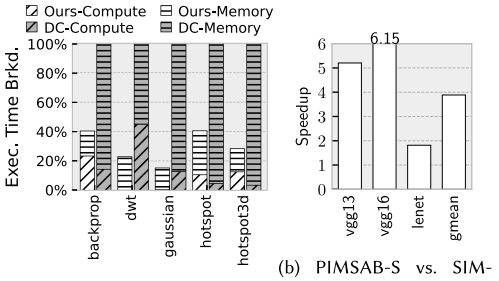
7 Results

7.1 Comparison with NVIDIA A100 GPU

Figure 16 shows the execution time and energy comparison against NVIDIA A100 GPU. On average, PIMSAB achieves 4.80× improvement in execution time, and 3.76× improvement in energy efficiency. PIMSAB outperforms A100 on vecadd because of the higher compute throughput. PIMSAB significantly surpasses A100 on `fir` because of the unaligned memory access caused by the sliding window. In PIMSAB, this program behavior can easily be handled by shifting bits across bitlines, while it prevents the GPU from fully utilizing the memory bandwidth. PIMSAB can achieve slightly less performance as A100 for `gemm`, even though A100 uses Tensor Cores for GEMM, which provide 2× peak TOPs compared with PIMSAB. In terms of energy consumption, PIMSAB is better than A100 for `gemm`. `conv2d` is sped up by 13.84× compared to A100 and `gemv` is sped up by 1.78×. The two main sources of the speedup are the high data parallelism in PIMSAB leading to reduced instruction overhead, and the larger on-chip buffer (256 MB on PIMSAB vs. 96 MB, including L2, shared memory and RF, on A100) leading to more data reuse and reduces off-chip memory traffic. These reasons also lead to significant energy savings. PIMSAB utilizes broadcast operations to avoid excess data read from DRAM speeding up the `conv2d` operations. We observe a speedup of 2.8× and 2.3×on `resnet18` and `bert`, respectively.

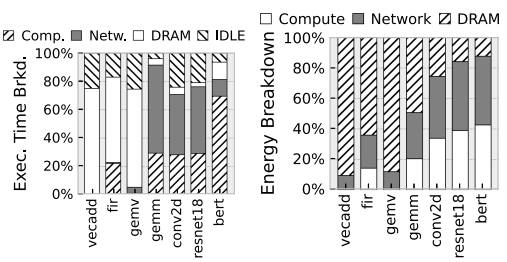
7.2 Comparison with SRAM PIM (Duality Cache)

Figure 17(a) shows PIMSAB-D outperforms Duality Cache by 3.7× on average across several Rodinia benchmarks. PIMSAB-D shows speedups over Duality Cache on `backprop`, `hotspot2d`, and `hotspot3d`, because the tensor DSL programming compiler can easily analyze the memory footprint and allocate buffers for memory reuse. In addition, Duality Cache still adopts a GPU-like warp-wise execution, which imposes high overhead to coordinate unaligned data loading. PIMSAB



(a) PIMSAB-D vs. Duality Cache (b) PIMSAB-S vs. SIMDRAM

Fig. 17. Appropriately provisioned PIMSAB compared with prior in-SRAM and in-DRAM systems.



(a) Execution time breakdown (b) Energy breakdown

Fig. 18. Categorized breakdown of each workload.

can simply shift across bitlines, even across CRAMs in a tile, so it outperforms DC on `dwt2d`. `gaussian` is bound by memory packing on DC, but our hardware is well specialized for it because of the H-tree intra-tile interconnect, which also leads to fewer computational instructions.

7.3 Comparison with DRAM PIM (SIMDRAM)

Figure 17(b) shows our comparison against SIMDRAM. PIMSAB-S outperforms SIMDRAM [18] by 3.88× on average across real world neural networks, because in-SRAM processing takes advantage of data reuse in on-chip buffers. SIMDRAM has to pay DRAM read latencies for every computation and is at a disadvantage for workloads with data reuse. PIMSAB's speedup is lower on LeNet because the LeNet model is relatively small—SRAM-DRAM transfer occupies a larger portion of execution, compared to the other networks.

7.4 Time and energy breakdown

Figure 18(a) shows the breakdown of time spent in each benchmark. Since `vecadd` has low arithmetic intensity, most of the time is spent on DRAM loads and stores, as expected. In `fir`, about 60% of the time is spent on DRAM traffic. `gemv` is also DRAM bound because of low reuse. `gemm` and `conv2d` are dominated by network traffic, because our optimization objective is to minimize the estimated DRAM traffic by converting them to network data transfer through broadcasting. `resnet18` is mainly a sequence of convolution layers followed by element-wise operations. The execution time breakdown for `resnet18` is very similar to `conv2d`. `bert` is mainly composed of several GEMM layers and a single softmax layer. As mentioned earlier, our GEMM kernels in PIMSAB are primarily limited by network traffic. However, BERT, due to its distinct shapes with high compute density, is compute-bound.

Figure 18(b) shows the breakdown of energy consumed in each benchmark. `vecadd`, `fir`, and `gemv` are dominated by DRAM energy because of the limited reuse. In microbenchmarks like `gemm` and `conv2d`, 20-40% of the energy is spent on computation. In `resnet18` and `bert`, energy is majorly spent on compute and on-chip network traffic.

7.5 Sensitivity to different hardware parameters

As shown in Figure 19, we analyze a set of seven different hardware configurations, obtained by varying three hardware parameters, with the microbenchmarks. Figure 19(a) studies the sensitivity of the number of compute resources (PEs) by tuning the size of each CRAM while retaining a constant memory capacity. Assuming each CRAM is a square (#wordlines = #bitlines), halving the number of bitlines results in 4× more compute intensity (more PEs for the same amount of memory). For `vecadd` and `gemv`, changing computational intensity has an insignificant impact

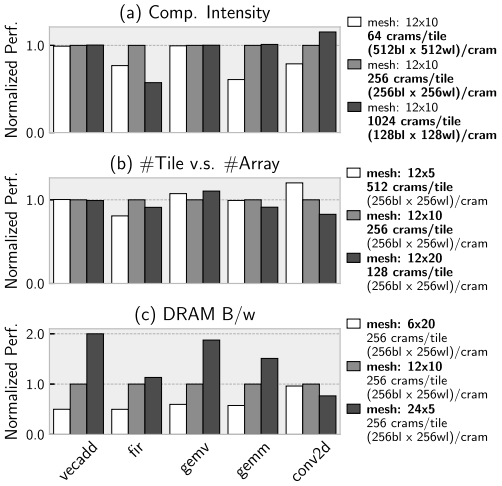


Fig. 19. Perf. sensitivity to hardware parameters.

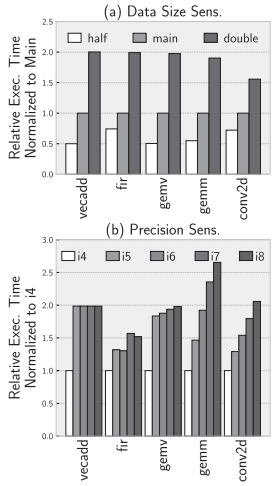


Fig. 20. Perf. sensitivity to workload parameters.

because the benchmarks are memory bound. In `fir`, we see an interesting observation. Performance improves as the CRAM size is changed from 512 ×512 to 256 × 256 on account of increased compute throughput, but the performance degrades to 57% for a CRAM size of 128 × 128. This is because the workload is too small to fully utilize the available compute throughput. The `gemm` workload exhibits ˜40% increase in performance when compute intensity increases from 512 ×512 CRAM size to 256 × 256 CRAM size. However, when the CRAM size is changed to 128 ×128, the performance only increases by 11%. This is due to the time taken by intra-tile inter-CRAM reductions starts to increase significantly; smaller CRAMs means more inter-CRAM reduction is required. The `conv2d` workload still exhibits ˜ 20% increase in performance when CRAM size changes to 128 × 128, because `conv2d` has less network traffic.

Figure 19(b) studies the tradeoff between the number of tiles and CRAMs per tile, while retaining the same number of compute resources. Increasing the number of tiles implies a larger inter-tile dynamic network (NoC), but smaller intra-tile static network, and vice-versa. The results of this study suggest that more tiles (darker grey bars in the figure) hurt performance by 4.3%, and larger tiles (white bars in the figure) provide diminishing returns (~2.6% improvement).

Figure 19(c) shows the tradeoff by changing the memory bandwidth. This is achieved by changing the mesh geometry, since only the top-row tiles have memory controllers. The massive data parallelism requires massive data to sustain. Therefore, workloads with poor reuse, like `vecadd`, and `gemv`, which are bounded by memory accesses, achieve nearly linear speedup when doubling the memory bandwidth (i.e., number of columns in the mesh is doubled). Although according to Figure 18(a), `gemm`'s execution time is not dominated by DRAM bandwidth, the performance is significantly improved when number of columns in the mesh increases. This speedup is attributed to reduced data transfer time because the mesh height is reduced to half. `conv2d` is an outlier; the performance is slightly lowered by the memory bandwidth increase. Because broadcasting dominates the data loading time, memory bandwidth is not fully utilized. Because of the wider mesh width, loaded data is broadcasted to further tiles, increasing the overall loading time.

7.6 Sensitivity to different workload parameters

Figure 20(a) shows the sensitivity of PIMSAB's performance to workload sizes, by studying two additional sizes, i.e., halving and doubling the data. The execution time of workloads with limited

data reuse (e.g., vecadd, gemv) is linearly proportional to the data size. fir shows a higher than 0.5× execution time when the data size is reduced to 0.5×. The smaller sized benchmark causes underutilization of the hardware. gemm has slightly lower than 2× execution time when the data size is increased by 2×; larger sizes lead to better hardware utilization because of data reuse. Because of compute underutilization caused by shapes, conv2d execution time does not vary linearly with change in input. Bigger input increases input loading and computing time, but does not increase weight loading time which is significant, leading to only a 1.5×increase in overall time. Figure 20(b) shows the sensitivity of PIMSAB's performance to the precision of the inputs. A unique capability of bit-serial systems is to support any arbitrary precision. We vary the input precisions from 4-bits to 8-bits. Since the DRAM representation always aligns to a power of 2, the DRAM traffic remains the same for int5 to int8. The DRAM bound benchmarks like vecadd and gemv tend to show the effect. vecadd is highly dominated by DRAM reads; thus, it shows no change in performance for 5-bits to 8-bits. gemv which is also a DRAM bound workload but has more compute and reuse as compared to vecadd shows slight increase in execution time going from 5-bits to 8-bits. Since computation and on-chip network traffic account for a significant portion of the execution time in conv2d and gemm operations, the performance of these workloads varies almost linearly with precision. Note that adaptive precision eliminates the requirement to utilize 8-bit computations for smaller precisions.

7.7 Ablation Studies

In this section, we show the results of ablation studies, where we disable microarchitectural and compiler features one-at-a-time and observe their impact of the performance of each benchmark. In these experiments, the baseline is the PIMSAB with all features enabled. Figure 21 shows the performance of each benchmark relative to this baseline, with one feature disabled. In Figure 21(a), we disable the shuffle units. We observe that the performance of conv, gemm, resnet18 and bert degrades significantly. Performance degradation comes mainly from redundant data loads which are required in the absence of shuffle units. Other benchmarks do not utilize shuffling and therefore have no performance differences. Figure 21(b) shows the impact of disabling constant operations. Up to 45% degradation of performance is seen (in the fir case). Disabling constant operations means duplicating data in the CRAM, which means additional rows will be used, which can cause data spills to DRAM. The fir benchmark spends relatively more time on constant operations than gemv, therefore it suffers more performance degradation. Figure 21(c) shows the performance degradation if we disable the Htree-based interconnect within each tile and replace it with a simplistic bus-based interconnect (similar to [19]). We see a reduction in performance in conv, gemm, resnet18 and bert. These benchmarks require reductions within tiles that are not done efficiently with the bus. Other benchmarks do not use the H-tree for reduction and their performances do not change as much. Figure 21(d) shows that disabling the cross-CRAM shifting feature affects the fir benchmark significantly, because shifts have to be done external to PIMSAB (e.g., by the transpose unit). The other benchmarks do not use the cross-CRAM shift feature. Figure 21(e) shows the impact of disabling systolic broadcasting. When systolic broadcasting is disabled, we use a one-to-all broadcast instead. In conv and resnet18, broadcasting weights or inputs takes majority of time; therefore, their performance degrades by up to 40%. In other benchmarks, broadcasting only takes a small portion of the overall time. Therefore, the performances are not significantly impacted. In Figure 21(f), we show the impact of disabling the mesh topology for the inter-tile dynamic network and replace it with a ring interconnect instead. The performances of all workloads degrade significantly since they all become interconnect-bound, because the average latency of communication is higher in a ring interconnect than a mesh interconnect. Some workloads such as conv2d and gemm have broadcast operations, which become very expensive in ring interconnect. Also, when multiple

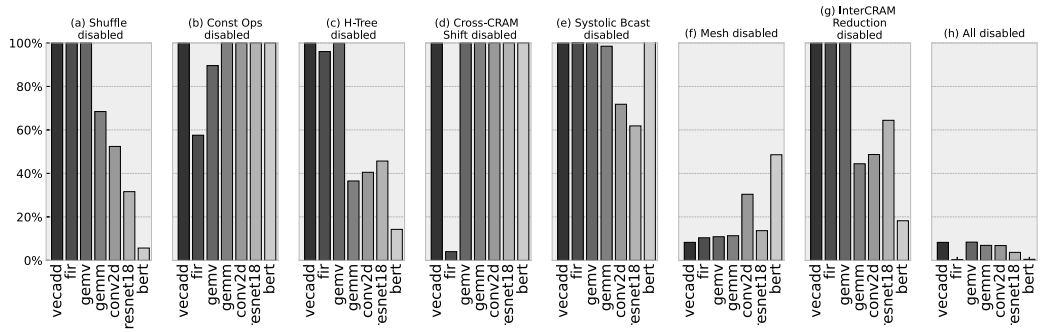


Fig. 21. Ablation studies. Y-axis is the performance compared to the baseline.

tiles load from DRAM at the same time, the ring topology causes significant resource contention and high latency. A ring interconnect is commonly used in CPU caches and is used in Duality Cache [14]. Figure 21(g) shows impact of disabling the inter-CRAM reduction and use intra-CRAM reduction instead. Intra-CRAM reduction is done by shifting data between bitlines as described in [11]. This results in more cycles compared to inter-CRAM reduction, because shifting data through N bitlines takes N cycles, but shifting data from one CRAM to another takes log(N) cycles, attributed to the H-tree. Hence, all workloads that heavily utilize inter-CRAM reduction (`conv2d`, `gemm`, `resnet18`, `bert`) show significant performance degradation. We also perform an experiment where all these features are disabled (Figure 21(h)). We see >90% degradation in all workloads.

We also study the effects of adaptive precision optimization mentioned in Section 5.3. As discussed in that section, adaptive precision helps performance by using only the number of bits required for a particular operation. This saves memory as well as cycles. The results of this study are presented in Table 6. No speedups are observed for `vecadd` and `fir` benchmarks. `vecadd` uses int8 precision and does not have any accumulations. For `fir`, the accumulation precision (int16) is the same as multiplication precision (int16), so adaptive precision is not used. `gemv` is sped up by only 0.34% because of the small number of accumulation operations in the workload. Speedups of 2.77%, 4%, and 3.5% are observed in `gemm`, `conv2d`, and `resnet18` respectively. Speedup of 15% is observed in `bert` as the matrix multiplications in `bert` involve significantly more accumulations than `gemm` and take more advantage of adaptive precision.

7.8 Chip Area Distribution

Figure 22 shows the area distribution of the PIMSAB chip. 72% of the chip area is consumed by the CRAMs, indicating a large percentage of useful compute/storage area. The dynamic and static networks take ~7.5% of the chip area, while the shuffle logic occupies ~1.5% of the area. The DRAM controller, transpose units and transceivers (XCVR) occupy 17% of the chip. Considering the additional capabilities enabled by PIMSAB, the overhead is relatively low.

8 Related Work

Instead of moving data to distant compute units, PIM brings computation closer to the data. Recent works have used **Non-Volatile Memories (NVM)** like **Resistive RAMs (ReRAM)** or **Spin-Transfer Torque Magnetic RAM (STT-MRAM)** [10, 19, 21, 22, 37]. NVM-based solutions are nascent and are yet to reach large scale production, and have endurance and technology scaling limitations.

Many DRAM-based PIM were proposed [16, 18, 29, 36], without compilers, so they are difficult to program them. CHOPPER [34] is a full-stack DRAM PIM that is programmed from a bit-sliced

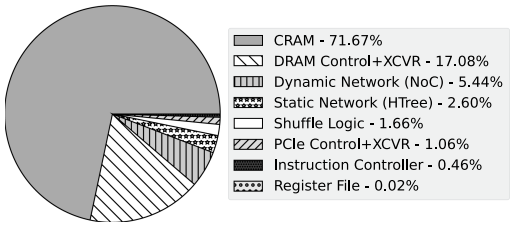


Fig. 22. Chip area distribution of PIMSAB.

Table 6. Speedup Obtained by Enabling Adaptive Precision Compiler Optimization

Benchmark	Speedup (%)
vecadd	0
fir	0
gemv	0.34
gemm	2.77
conv2d	4
resnet18	3.5
bert	15

DSL. Inspired by their code scheduling strategies, we developed our bit-level lifetime analysis technique. UPMEM [32] and Samsung HBM-PIM [12] are recent commercial DRAM PIM architectures. SRAM-based PIM has the advantage of simple integration with compute logic using the same process, and also the ability to exploit data reuse in applications. PIMSAB uses SRAM-based PIM. Some SRAM-based approaches are analog [7, 25, 26], requiring expensive DACs and ADCs. Other approaches use the property of enabling multiple wordlines in an SRAM at the same time [11, 14, 42]. This requires reducing wordline voltage to avoid data corruption, and modifying sense amplifiers. PIMSAB uses conventional dual ported RAMs instead, based on CoMeFa [5]. This costs area, but is practical and robust.

Neural Cache [11] and Duality Cache [14] are popular SRAM-based PIM architectures in which the focus is to repurpose existing caches in CPUs to perform in-situ computations. Neural cache uses an ad-hoc programming approach, and Duality Cache introduces a restricted version of the CUDA programming interface; both of these are lower-level and expose hardware aspects to programmers (e.g. SRAM-array dimensions). Our Tensor DSL abstracts hardware and is easier to program and perform explorations with.

Fujiwara et al. [15] design an SRAM-based compute-in-memory chip that provides 254 TOP-S/W throughput for 4-bit operations. They fabricate the chip, but no performance evaluation of real workloads is provided. Other works such as PUMA [4] and IMDPP [13] develop compilers to make PIM systems easier to program. Their TensorFlow or C++ based graph-level programming interfaces are harder to perform performance tuning with than our Tensor DSL. Also, they use ReRAM instead of SRAM-based PIM.

Recently, Processing-In-Memory has been proposed for FPGAs as well. CCB [43] uses the same technology as Neural Cache [11] to enable block RAMs on an FPGA to perform computation, while CoMeFa [5] uses the dual-ported nature of block RAMs. Comparing with PIMSAB's Tensor DSL programming interface, these works still require users to design finite state machines to send instructions to the RAM blocks, which is error-prone and time-consuming.

The bit-serial approach has a long history, going back to their use for neural networks in the 1980s [8, 30]. Stripes [23] is a more recent such DNN accelerator. PIMSAB combines a bit-serial approach with PIM.

9 Conclusion

We present PIMSAB, a system for in-memory acceleration of massively parallel workloads like Deep Learning. Our system employs novel mechanisms for spatially aware communication and bit-serial-aware computation. While other PIM accelerators have been proposed for DL, our work makes significant strides in making PIM-based accelerators feasible for real-world DL problems. With the scalable hierarchical architecture combined with the H-tree and mesh interconnects at

different levels, along with the shuffle network, adaptive precision and constant operation support, we make significant improvement in the capability of PIM-based accelerators.

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)*. 481–492. https://doi.org/10.1109/HPCA.2017.21

[2] A. A. Aggarwal and D. M. Lewis. 1994. Routing architectures for hierarchical field programmable gate arrays. In *1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 475–478. https://doi.org/10.1109/ICCD.1994.331954

[3] 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)*. 1–5.

[4] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W. Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. 2019. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. ACM, New York, , 715–731. https://doi.org/10.1145/3297858.3304049

[5] Aman Arora, Tanmay Anand, Aatman Borda, Rishabh Sehgal, Bagus Hanindhito, Jaydeep Kulkarni, and Lizy K. John. 2022. CoMeFa: Compute-in-memory blocks for FPGAs. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–9. https://doi.org/10.1109/FCCM53951.2022.9786179

[6] H. B. Bakoglu. 1990. Circuits, interconnections, and packaging for VLSI. Addison-Wesley Pub. Co.

[7] Avishek Biswas and Anantha P. Chandrakasan. 2019. CONV-SRAM: An energy-efficient sram with in-memory dot-product computation for low-power convolutional neural networks. *IEEE Journal of Solid-State Circuits* 54, 1 (2019), 217–230. https://doi.org/10.1109/JSSC.2018.2880918

[8] Zoe Butler, Alan Murray, and Anthony Smith. 1989. *VLSI Bit-Serial Neural Networks*. Springer US, Boston, MA, 201–208.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th OSDI*.

[10] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 27–39. https://doi.org/10.1109/ISCA.2016.13

[11] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaaauw, and Reetuparna Das. 2018. Neural cache: Bit-serial in-cache acceleration of deep neural networks. In *45th Annual International Symposium on Computer Architecture (ISCA '18)*. (Los Angeles, California). IEEE Press, 383–396. https://doi.org/10.1109/ISCA.2018.00040

[12] Young-Cheon Kwon, Suk Han Lee, Jaehoon Lee, Sang-Hyuk Kwon, Je Min Ryu, Jong-Pil Son, O. Seongil, Hak-Soo Yu, Haesuk Lee, Soo Young Kim, Youngmin Cho, Jin Guk Kim, Jongyoon Choi, Hyun-Sung Shin, Jin Kim, BengSeng Phuah, HyoungMin Kim, Myeong Jun Song, Ahn Choi, Daeho Kim, SooYoung Kim, Eun-Bong Kim, David Wang, Shinhaeng Kang, Yuhwan Ro, Seungwoo Seo, JoonHo Song, Jaeyoun Youn, Kyomin Sohn, and Nam Sung Kim. 2021. 25.4 A 20nm 6GB function-in-memory DRAM, based on HBM2 with a 1.2TFLOPS programmable computing unit using bank-level parallelism, for machine learning applications. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*. 350–352. DOI: https://doi.org/10.1109/ISSCC42613.2021.9365862

[13] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-memory data parallel processor. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, , 1–14. https://doi.org/10.1145/3173162.3173171

[14] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2019. Duality cache for data parallel acceleration. In *46th International Symposium on Computer Architecture (ISCA '19)* (Phoenix, Arizona). ACM, New York, , 397–410.

[15] Hidehiro Fujiwara, Haruki Mori, Wei-Chang Zhao, Mei-Chen Chuang, Rawan Naous, Chao-Kai Chuang, Takeshi Hashizume, Dar Sun, Chia-Fu Lee, Kerem Akarvardar, Saman Adham, Tan-Li Chou, Mahmut Ersin Sinangil, Yih Wang, Yu-Der Chih, Yen-Huei Chen, Hung-Jen Liao, and Tsung-Yung Jonathan Chang. 2022. A 5-nm 254-TOPS/W 221-TOPS/mm2 fully-digital computing-in-memory macro supporting wide-range dynamic-voltage-frequency scaling and simultaneous MAC and write operations. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 1–3. https://doi.org/10.1109/ISSCC42614.2022.9731754

[16] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. 2019. ComputeDRAM: In-memory compute using off-the-shelf DRAMs. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture* . ACM, New York, , 100–113. https://doi.org/10.1145/3352460.3358260

[17] M. R. Guthaus, J. E. Stine, S. Ataei, Brian Chen, Bin Wu, and M. Sarwar. 2016. OpenRAM: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. https://doi.org/10.1145/2966986.2980098

[18] Nastaran Hajinazar, Geraldo F. Oliveira, Sven Gregorio, João Dinis Ferreira, Nika Mansouri Ghiasi, Minesh Patel, Mohammed Alser, Saugata Ghose, Juan Gómez-Luna, and Onur Mutlu. 2021. SIMDRAM: a framework for bit-serial SIMD processing using DRAM. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*. ACM, New York, , 329–345.

[19] Bagus Hanindhito, Ruihao Li, Dimitrios Gourounas, Arash Fathi, Karan Govil, Dimitar Trenev, Andreas Gerstlauer, and Lizy John. 2021. Wave-PIM: Accelerating wave simulation using processing-in-memory. In *50th International Conference on Parallel Processing (ICPP 2021)* (Lemont, IL). ACM, New York, , Article 8, 11 pages. https://doi.org/10.1145/3472456.3472512

[20] Chun Hok Ho, Chi Wai Yu, Philip H. W. Leong, Wayne Luk, and Steven J. E. Wilton. 2007. Domain-specific hybrid FPGA: Architecture and floating point applications. In *2007 International Conference on Field Programmable Logic and Applications*. 196–201. DOI: https://doi.org/10.1109/FPL.2007.4380647

[21] Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. 2019. FloatPIM: In-memory acceleration of deep neural network training with high precision. In *46th International Symposium on Computer Architecture*. 802–815. https://doi.org/10.1145/3307650.3322237

[22] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. 2018. Computing in memory with spin-transfer torque magnetic RAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 3 (2018), 470–483. https://doi.org/10.1109/TVLSI.2017.2776954

[23] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

[24] Mingu Kang, Sujan K. Gonugondla, Ameya Patil, and Naresh R. Shanbhag. 2018. A multi-functional in-memory inference processor using a standard 6T SRAM array. *IEEE Journal of Solid-State Circuits* 53, 2 (Feb. 2018), 642–655. https://doi.org/10.1109/JSSC.2017.2782087

[25] Mingu Kang, Sujan K. Gonugondla, and Naresh R. Shanbhag. 2020. Deep in-memory architectures in SRAM: An analog approach to approximate computing. *Proc. IEEE* 108, 12 (2020), 2251–2275. https://doi.org/10.1109/JPROC.2020.3034117

[26] Mingu Kang, Min-Sun Keel, Naresh R. Shanbhag, Sean Eilert, and Ken Curewitz. 2014. An energy-efficient VLSI architecture for pattern recognition via deep embedding of computation in SRAM. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 8326–8330. https://doi.org/10.1109/ICASSP.2014.6855225

[27] Mingu Kang, Min-Sun Keel, Naresh R. Shanbhag, Sean Eilert, and Ken Curewitz. 2014. An energy-efficient VLSI architecture for pattern recognition via deep embedding of computation in SRAM. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 8326–8330. DOI: https://doi.org/10.1109/ICASSP.2014.6855225

[28] Andrew Kerr, Haicheng Wu, Manish Gupta, Dustyn Blasig, Pradeep Ramini, Duane Merrill, Aniket Shivam, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Matt Nicely. 2022. *CUTLASS*. https://github.com/NVIDIA/cutlass

[29] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-based reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 288–301.

[30] Alan Murray, Anthony Smith, and Zoe Butler. 1987. Bit-serial neural networks. In *Neural Information Processing Systems*, D. Anderson (Ed.), Vol. 0. American Institute of Physics.

[31] NCSU. 2018. *FreePDK45*. https://www.eda.ncsu.edu/wiki/FreePDK45:Contents

[32] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, and Alexandra Fedorova. 2021. A case study of processing-in-memory in off-the-shelf systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 117–130. https://www.usenix.org/conference/atc21/presentation/nider

[33] Amin Norollah, Danesh Derafshi, Hakem Beitollahi, and Ahmad Patooghy. 2018. PAT-noxim: A precise power & thermal cycle-accurate NoC simulator. In *2018 31st IEEE International System-on-Chip Conference (SOCC)*. 163–168. https://doi.org/10.1109/SOCC.2018.8618491

[34] Xiangjun Peng, Yaohua Wang, and Ming-Chang Yang. 2023. CHOPPER: A compiler infrastructure for programmable bit-serial SIMD processing using memory in DRAM. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1275–1288.

[35] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (Jul. 2012), 12 pages.

[36] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-memory accelerator for bulk bitwise operations using commodity DRAM technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 273–287.

[37] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar. 2016. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 14–26.

[38] A. Stillmaker and B. Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration, the VLSI Journal* 58 (2017), 74–81. http://vcl.ece.ucdavis.edu/pubs/2017.02.VLSIintegration.TechScale/

[39] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek, and André DeHon. 1999. HSRA: High-speed, hierarchical synchronous reconfigurable array. In *1999 ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays (FPGA '99)* (Monterey, CA) . ACM, New York, , 125–134. https://doi.org/10.1145/296399.296442

[40] Arizona State University. 2012. *Predictive Technology Model*. http://ptm.asu.edu/

[41] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[42] Jingcheng Wang, Xiaowei Wang, Charles Eckert, Arun Subramaniyan, Reetuparna Das, David Blaauw, and Dennis Sylvester. 2020. A 28-nm compute SRAM with bit-serial logic/arithmetic operations for programmable in-memory vector computing. *IEEE Journal of Solid-State Circuits* 55, 1 (2020), 76–86. https://doi.org/10.1109/JSSC.2019.2939682

[43] Xiaowei Wang, Vidushi Goyal, Jiecao Yu, Valeria Bertacco, Andrew Boutros, Eriko Nurvitadhi, Charles Augustine, Ravi Iyer, and Reetuparna Das. 2021. Compute-capable block RAMs for efficient deep learning acceleration on FPGAs. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 88–96. https://doi.org/10.1109/FCCM51124.2021.00018

[44] Wm. A. Wulf and Sally A. McKee. 1995. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News* 23, 1 (1995).

[45] Pavan Yalamanchili, Umar Arshad, Zakiuddin Mohammed, Pradeep Garigipati, Peter Entschev, Brian Kloppenborg, James Malcolm, and John Melonakos. 2015. ArrayFire - A high performance software library for parallel computing with an easy-to-use API. https://github.com/arrayfire/arrayfire

Received 28 December 2023; revised 22 June 2024; accepted 13 August 2024