AME-PIM: Can Memory be Your Next Tensor Accelerator?

Emanuele Venieri
University of Bologna
Bologna, Italy
emanuele.venieri2@unibo.it

Simone Manoni
University of Bologna
Bologna, Italy
s.manoni@unibo.it

Alberto Florian
University of Bologna
Bologna, Italy
alberto.florian@studio.unibo.it

Jaehyun Park
Samsung Electronics
Hwaseong, Republic of Korea
jh0416.park@samsung.com

Kyomin Sohn
Samsung Electronics
Hwaseong, Republic of Korea
kyomin.sohn@samsung.com

Andrea Bartolini
University of Bologna
Bologna, Italy
a.bartolini@unibo.it

Abstract

High Bandwidth Memory with Processing-in-Memory (HBM-PIM) offers an opportunity to reduce data movement by executing computation directly inside memory, but current commercial platforms expose limited instruction sets and require specialized software stacks. In this work, we investigate whether HBM-PIM can serve as a backend for ISA-level matrix acceleration, using the RISC-V Attached Matrix Extension (AME) as a semantic reference. We propose a PEP-based execution model that maps AME element-wise and matrix instructions to HBM-PIM micro-kernels and data instructions in memory operations. Differently from SoA HBM-PIM, we introduce a reduction-free outer-product dataflow that enables accumulation entirely within memory despite the lack of native reduction support. Our approach supports end-to-end execution of element-wise operations, GEMV, and GEMM in PIM mode, minimizing host involvement and off-chip transfers. An experimental evaluation on Samsung Aquabolt-XL shows that AME matrix tile multiplication achieves up to 14.9 GFLOP/s (59.4 FLOP/cycle) on a single HBM pseudo-channel.

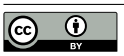
CCS Concepts

• **Computer systems organization** → **Parallel architectures**; **Single instruction, multiple data**; **Other architectures**; • **Hardware** → **Emerging technologies**.

Keywords

Processing-in-Memory, AME, RISC-V, HBM-PIM

ACM Reference Format:
Emanuele Venieri, Simone Manoni, Alberto Florian, Jaehyun Park, Kyomin Sohn, and Andrea Bartolini. 2026. AME-PIM: Can Memory be Your Next Tensor Accelerator?. In *Proceedings of the 23rd ACM International Conference on Computing Frontiers (CF '26), May 19–21, 2026, Catania, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3801487.3806067



CF '26, Catania, Italy

ACM ISBN 979-8-4007-2568-5/26/05
https://doi.org/10.1145/3801487.3806067

1 Introduction

The performance of modern processors is increasingly limited not by the availability of arithmetic units, but by the ability to supply them with data [14]. As workloads in machine learning, scientific computing, and data analytics continue to scale, large working sets and low reuse often push execution beyond the effective capacity of on-chip caches. In these regimes, computation is dominated by off-chip memory traffic, and system efficiency is determined by memory bandwidth and the energy cost of moving data across the memory hierarchy. Taken together, these effects constitute the *von Neumann bottleneck*, where data must continuously flow from memory into the CPU pipeline. This limitation is further amplified in the era of generative AI and foundation models, whose working sets frequently exceed on-chip memory capacity by several orders of magnitude.

High Bandwidth Memory (HBM) [7] was introduced to alleviate this bottleneck by providing substantially higher bandwidth than conventional DDR-based DRAM [11, 14]. By stacking DRAM dies and integrating them within the same package as the host using wide interfaces, HBM exposes much higher I/O parallelism while reducing energy per bit transferred [11]. This makes HBM an attractive solution for memory, and it has become a common choice in GPUs and high-performance accelerators. However, even with HBM, emerging workloads can remain memory-limited [8], and further scaling memory bandwidth through wider interfaces or faster signalling faces practical constraints in packaging, power delivery, thermals, and signal integrity [3, 11, 16].

Processing-in-Memory (PIM) architectures offer a complementary path forward: rather than only increasing the rate at which data can be moved to compute units, PIM reduces the need for data movement by performing computation close to where the data resides [14]. In HBM-PIM, lightweight compute logic in the memory stack can execute simple arithmetic kernels directly within the memory device, leveraging internal bank-level parallelism and minimizing host involvement. Commercial HBM-PIM platforms such as Samsung Aquabolt-XL [12] demonstrate the practicality of this approach, enabling in-memory execution through JEDEC-compliant host commands without custom HBM memory controllers. Moreover, the use of HBM-PIM proved to be effective in both reducing execution time (2.1× faster) and power consumption (71% less) than conventional GPU with HBM memory [13].

Despite these benefits, effectively using today's HBM-PIM remains difficult due to architectural and software constraints. In particular, the set of natively supported operations is limited, and

current commercial HBM-PIM Instruction-Set Architectures (ISAs) do not provide native reduction support [4, 12]. This is largely because the arithmetic blocks in HBM-PIM are implemented in a memory-oriented logic technology with limited logic density, which constrains the amount and complexity of compute that can be integrated on the HBM logic layer. For matrix and vector workloads, this becomes a critical obstacle: while in-memory Multiply-and-Accumulate (MAC) units can efficiently generate partial products, many common dataflows require reductions across those partial results. Consequently, prior studies typically perform reductions on the host CPU [9, 14] or on an attached accelerator such as an FPGA [4], increasing data movement and partially undermining the motivation for PIM.

Beyond architectural limits, the interface exposed by commercial HBM-PIM platforms also makes integration challenging. Rather than being invoked through a general architectural mechanism, PIM execution is accessed through device-specific primitives and hand-written microkernels that must be explicitly orchestrated by runtime software [12]. As a result, applications must manage PIM execution as a specialized workflow that involves mode transitions, constrained memory placement, and carefully scheduled command sequences, instead of relying on standard compiler and OS abstractions. This accelerator-style programming model complicates portability across host processors and toolchains, limits compiler-driven optimization, and substantially increases the engineering effort required to deploy PIM kernels in real systems.

In contrast, modern CPUs increasingly expose matrix acceleration through ISA-level extensions, where tiled GEMM/GEMV-style computation is expressed using well-defined architectural semantics. This includes dedicated register files, with the purpose of containing matrix tiles and direct support to algebraic matrix–matrix operations that uses said tile registers as source operands. Examples include Intel Advanced Matrix Extensions (AMX) [6] and Arm's Scalable Matrix Extension (SME) [1]. These extensions show how matrix workloads can be integrated into the ISA of a processor, enabling compilers and software stacks to target specialized execution engines without relying on device-specific APIs.

Motivated by this perspective, can HBM-PIM be used as a backend for CPU ISA extensions for matrix operations, with the goal of enabling more general and portable exploitation of PIM through well-defined architectural semantics? As a semantic reference point, we use the T-Head proposal for the RISC-V Attached Matrix Extension (AME) [17], which provides an ISA-level abstraction for tiled matrix execution that could be naturally exposed to compilers and software stacks. Rather than treating HBM-PIM as a specialized memory device accessed through custom APIs, we study how its compute capabilities can be mapped onto matrix-extension-style operations in a way that preserves clean semantics while respecting real platform constraints.

We propose an outer-product–based mapping that remaps matrix execution to exploit in-memory MAC units for accumulation directly within HBM, avoiding explicit reduction operations that are unsupported in current HBM-PIM platforms. Previous works [4, 9] relied on external reduction engines to overcome this limitation, while our approach enables element-wise operations as well as GEMV and GEMM to execute entirely inside HBM-PIM, reducing host interaction and minimizing off-chip transfers. Our results show that carefully designed dataflows can significantly improve the practicality of HBM-PIM for ISA-level acceleration, and provide insights for future PIM architectures and programming models.

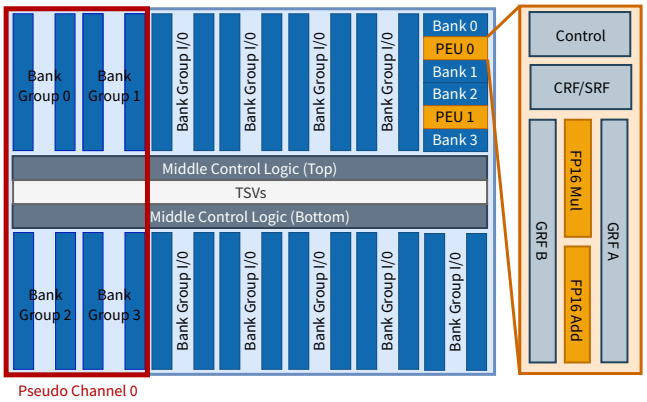


Figure 1: Samsung Aquabolt-XL PIM DRAM architecture

Overall, this paper makes the following contributions:

- We study how commercial HBM-PIM devices can serve as a practical backend for matrix ISA extensions subsets, using AME semantics as a reference abstraction.
- We introduce an outer-product–based execution strategy that performs accumulation within HBM via in-memory MAC units, avoiding host-side reduction phases.
- We experimentally evaluate the proposed mapping on Samsung Aquabolt-XL. Our results show that AME GEMM, GEMV, and element-wise instructions can execute fully in PIM mode, reaching 14.9 GFLOP/s (59.4 FLOP/cycle) on a single pseudo-channel for the mfmacc instruction on 128×4096 tiles.
- We identify key ISA and microarchitectural limitations — missing reduction support and restricted operand routing —and show how they constrain AME execution efficiency, informing the design of future HBM-PIM architectures.

2 Background

This section summarizes the organization and programming interface of Samsung's Aquabolt-XL HBM-PIM platform, which serves as the commercial baseline for our work. We focus on the architectural features most relevant to matrix and vector execution, including the placement and structure of the in-memory compute units, the operating modes used to expose bank-level parallelism, and the ISA through which PIM microkernels are expressed and executed.

2.1 HBM-PIM Hardware Architecture

Aquabolt-XL augments an HBM2-based stacked memory system with in-memory compute capability while preserving conventional DRAM operation. Rather than modifying DRAM subarrays, PIM functionality is implemented by inserting a lightweight *PIM unit* at the I/O boundary between the DRAM cells and the peripheral circuitry. Each PIM unit is shared by a pair of DRAM banks, referred to as the even and odd bank, and can access one of the two banks at a time. This design choice reduces area and power overhead while maintaining direct access to bank-local data.

Each PIM unit consists of three components: a 16-wide SIMD floating-point unit, a set of local register files, and a controller. The SIMD datapath comprises sixteen FP16 lanes, each equipped with a multiplier and an adder, organized as a five-stage pipeline. The register files include a Command Register File (CRF) with 32 entries of 32 bits each, a General Register File (GRF) with sixteen 256-bit registers split between the even (GRF_A) and odd banks (GRF_B), and a Scalar Register File (SRF) composed of two banks of eight registers for scalar multiplication (SRF_M) and addition (SRF_A). The controller fetches and decodes PIM instructions from the CRF and orchestrates operand selection, SIMD execution, and write-back to the GRFs. The HBM stack consists of four DRAM dies and four PIM-enabled DRAM dies. Each PIM-enabled DRAM die is divided into four pseudo-channels, each connected to sixteen DRAM banks. With one PIM unit shared between each pair of banks, each pseudo-channel integrates eight PIM units. Figure 1 shows the internal architecture of an Aquabolt-XL HBM-PIM die. All PIM units operate at the same frequency as the DRAM core, between 250 MHz and 300 MHz, and collectively provide substantial on-chip compute bandwidth for vector and matrix operations.

From the host perspective, Aquabolt-XL is accessed using standard HBM transactions. In PIM mode, DRAM column read and write commands are repurposed to trigger PIM unit execution with deterministic latency. This command-driven execution model preserves compatibility with JEDEC-compliant memory controllers while enabling tightly synchronized, high-throughput execution across multiple banks inside the HBM stack.

2.2 Operating Modes

For each pseudo-channel Aquabolt-XL HBM-PIM supports three operating modes:(i) *Single-Bank (SB) Mode* corresponds to standard HBM behavior, where an Activate (ACT) command selects a specific bank and subsequent column commands to that bank access its open row. This mode is used for conventional DRAM memory accesses. (ii) *All-Bank (AB) Mode* overrides bank address fields so that a single column command targets the same row and column across all banks in a pseudo-channel simultaneously. This exposes bank-level parallelism in lock-step fashion and is also used to broadcast microkernel instructions into the CRF of every PIM unit during the programming phase. (iii) *All-Bank PIM (AB-PIM) Mode* builds on all-bank (AB) mode by coupling each DRAM column command with in-memory execution. Each such command advances an internal PIM program counter, causing all PIM units within a pseudo-channel to step through a preloaded microkernel in lock-step, while also acting as a pointer to the bank data used as a source operand. (iv) *Mode transitions* are performed via predefined DRAM command patterns to a reserved configuration memory region, preserving compatibility with unmodified JEDEC-compliant controllers.

2.3 PIM Instruction Set Architecture

Aquabolt-XL exposes a compact instruction set designed to support memory-bound vector and matrix kernels directly within the HBM stack. PIM instructions are executed by the PIM units placed at the bank I/O boundary and are stored in a per-pseudo-channel CRF. Each instruction is encoded in a fixed 32-bit format and is fetched and executed in response to DRAM column commands when operating in AB-PIM mode. The PIM instruction set is intentionally minimal and consists of three main classes: arithmetic operations, data movement operations, and control-flow instructions.

2.3.1 Arithmetic Instructions. Inside each PIM unit, arithmetic instructions operate on the 16-lane FP16 datapath and are optimized for vector-vector and vector-matrix computation. The supported operations include `ADD`, `MUL`, `MAD`, and `MAC`, which together enable fused multiply-add style computation commonly used in linear algebra and machine learning workloads. In particular, the `MAC` instruction accumulates the product of two operands into a destination register, making it the primary building block for GEMV- and GEMM-like kernels. There is no availability of cross-lane reduction-like instructions.

Operands for arithmetic instructions can be sourced from multiple locations, including the GRFs, SRFs, or directly from the active DRAM row buffer associated with a bank. Results are written back to the GRF or to bank-local storage, allowing accumulation to remain entirely within the memory device. To further increase efficiency, Aquabolt-XL supports an address-aligned mode (AAM) in which operand selection is implicitly derived from the command's DRAM row and column addresses, enabling multiple arithmetic operations to be issued with a small number of instructions.

2.3.2 Data Movement Instructions. Data movement instructions are used to transfer data between the DRAM row buffer and the PIM-local registers. The instruction set provides `MOV` and `FILL` operations, which load data from banks into registers or write register contents back to memory. These instructions support multiple operand routing configurations, allowing flexible movement between bank storage, GRFs and SRFs. Broadcasts of data from a single bank to all GRFs of a pseudo-channel is also possible.

Certain data movement operations can optionally apply simple activation functions such as ReLU as part of the transfer, further reducing the need for additional compute stages. However, all data movement remains explicitly programmed and tightly coupled to the underlying DRAM access pattern. In this work we do not leverage this capability as the AME specification does not have dedicated instructions for activation functions on matrix tiles. Future works can combine the AME-PIM with Neural Networks compilers/Design Specific Architecture to leverage it.

2.3.3 Control-Flow Instructions. To support repetitive kernel execution, the PIM ISA includes a small set of control-flow instructions, namely `NOP`, `JUMP`, and `EXIT`. These instructions allow microkernels stored in the CRF to implement loops and termination without host intervention. In particular, `JUMP` instructions are optimized for zero-cycle execution by relying on predecoded iteration counts, enabling efficient looping despite the absence of a full-fledged control unit. This approach has the drawbacks not allowing `JUMP` nesting and limiting the loop counter to a maximum of 255 iterations.

2.3.4 Implications for Matrix Execution. Overall, the PIM instruction set is sufficient to express the core arithmetic patterns of matrix and vector workloads, especially those dominated by multiply-accumulate operations. At the same time, it lacks support for more complex operations such as comparisons, reductions, and data-type widening, which constrains the class of computations that can be

executed entirely within memory. These characteristics directly influence how higher-level matrix abstractions can be mapped onto HBM-PIM and motivate the execution strategies we explored in this work.

2.4 AME extension by T-Head

To provide an ISA-level abstraction for tiled matrix computation, we rely on the AME proposal published by T-Head [17]. It defines a complete programmer-visible model for an attached matrix extension, including architectural state, configuration registers, tile-level and element-wise instruction semantics. We leverage this proposal as our reference specification since, despite not being ratified yet, it has already been used as a foundation in prior works [2, 15].

2.4.1 Architectural state and configuration. AME introduces dedicated architectural resources for matrix execution in addition to the RISC-V scalar register file. The extension defines two groups of matrix registers: a set of *tile registers* (`tr0–tr3`) used to hold input tiles, and a set of *accumulation registers* (`acc0–acc3`) used to store partial sums and output tiles.

To support flexible tiling, AME includes configuration state that determines the logical tile dimensions used by subsequent instructions. In particular, the proposal defines configuration Control and Status Registers (CSRs) controlling the active matrix shape through the parameters M, K, and N (e.g., `mtilem`, `mtilek`, `mtilen`), allowing software to adapt tile sizes at runtime for boundary conditions or problem-specific blocking.

2.4.2 Instruction set overview. The T-Head AME proposal organizes its instruction set into five groups. *Configuration instructions* set the active tile dimensions M, K, and N via `msettile{m,k,n}`-style mechanisms and provide lifecycle management via `mrelease`. *Matrix multiplication instructions* define the core tiled compute primitive: a multiply-and-accumulate that consumes two source tiles from `tr` registers and accumulates results into an `acc` register. It supports multiple operand/accumulator datatype combinations, including floating-point and integer dot-product variants, as well as hybrid-precision/widening forms depending on the supported implementation subset. *Load/store instructions* move matrix tiles between memory and matrix registers. These instructions explicitly encode the element size and tile layout, and include variants to support access patterns such as whole-register transfers and transposed forms, which are commonly required to implement blocked GEMM/GEMV kernels efficiently. *Misc instructions* cover auxiliary functionality such as data movement between matrix registers, scalar–matrix moves, broadcast operations, packing, and slide operations over rows/columns. *element-wise instructions* operate per element of a tile register and include arithmetic (add, subtract, multiply)comparison-based (min, max), shifts, and data type conversions.

Overall, AME provides a tile-based ISA abstraction in which matrix computation is expressed in terms of explicit architectural registers, configurable tile dimensions, and structured tile-level memory and arithmetic operations.

3 Methodology

This subsection describes how in our work the AME programming model is realized on the Aquabolt-XL HBM-PIM platform. It is structured as follows. Section 3.1 analyzes which AME computational primitives can be directly mapped onto the native PIM instruction set and identifies the architectural constraints that limit full ISA coverage. Section 3.2 presents our implementation strategy in which AME tile and accumulation registers are realized as memory-resident abstractions within HBM-PIM and their execution is driven by PIM execution primitives. Section 3.3 discusses the realization of AME control and status registers and the exposed programming interface, completing the description of a PIM-based AME implementation.

3.1 AME-PIM ISA comparison

We first evaluate whether AME arithmetic and matrix instructions can be executed natively within HBM-PIM, without relying on external computational hardware.

Table 1 summarizes the mapping between individual AME instructions and the corresponding PIM instructions used for their implementation. In most cases, an association is feasible. However, exceptions arise for operations involving widening, as the PIM datapath supports only the FP16 format, and for minimum and maximum operations, due to the absence of conditional or comparison instructions in the PIM execution model.

AME instruction	PIM instructions
mfadd.h.mm	ADD
mfadd.h.mv.i	ADD
mfsub.h.mm	MUL, ADD
mfsub.h.mv.i	MUL, ADD
mfmul.h.mm	MUL
mfmul.h.mv.i	MUL
mfmax.h.mm	Not supported
mfmax.h.mv.i	Not supported
mfmin.h.mm	Not supported
mfmin.h.mv.i	Not supported
mfmacc.h	MAC
mfmacc (with widening)	Not supported

Table 1: Summary of the association of AME arithmetic element-wise and matrix multiplication instructions with PIM instructions.

3.2 AME-PIM Implementation

In this paper we propose small computational kernels, called PIM Execution Primitive (PEP), to implement AME instructions using the PIM ISA. The PEPs encapsulate sequences of native PIM instructions that are executed entirely within the HBM-PIM. The defined PEPs are used iteratively and in combination with tile register memory mapping we adopted to implement AME instructions.

The PEPs employ a dataflow similar to the AME execution model, transferring data from Even Banks to Odd Banks through internal PIM operations involving the GRFs. This organization aligns with AME requirements for tile and accumulation registers. In this work we propose not to physically instantiate these registers but to abstract them as memory-resident data structures accessible by the PIM units as described in the previous paragraph. Even Banks store data corresponding to AME tile registers, while accumulation registers are mapped to Odd Banks.

```
def add (mul)-pep:
    loop: for i = 0 to 7
        fill GRF_A[i], EVEN_BANK[bt_0 + 32*i];
    for i = 0 to 7
        add (mul) GRF_B[i], EVEN_BANK[bt_1 + 32*i], GRF_A[i];
    for i = 0 to 7
        mov ODD_BANK[ba_0 + 32*i], GRF_B[i];
    jump 255 times, loop;
exit;
```

(a) ADD(MUL)-PEP listing.

```
def sub-pep:
    for i = 0 to 7
        fill SRF_M[i], EVEN_BANK[minus_one_vector];
    loop: for i = 0 to 7
        fill GRF_A[i], EVEN_BANK[bt_0 + 32*i];
    for i = 0 to 7
        mul GRF_B[i], EVEN_BANK[bt_1 + 32*i], SRF_M[i];
    for i = 0 to 7
        add GRF_B[i], GRF_A[i], GRF_B[i];
    for i = 0 to 7
        mov ODD_BANK[ba_0 + 32*i], GRF_B[i];
    jump 255 times, loop;
exit;
```

(b) SUB-PEP listing.

```
def mac-pep:
    loop: fill GRF_B[0], ODD_BANK[ba_0];
    for i = 0 to 7
        fill SRF_A[i], EVEN_BANK[bt_1 + 2*i];
        add GRF_A[i], EVEN_BANK[zero_vector], SRF_A[i];
    for i = 0 to 7
        mac GRF_B[0], EVEN_BANK[bt_0 + 32*i], GRF_A[i];
    mov ODD_BANK[ba_0], GRF_B[0];
    jump 255 times, loop;
exit;
```

(c) MAC-PEP listing.

Listing 1: Pseudocode listings of the four proposed PEPs. ADD-PEP and MUL-PEP are presented as a single listing for brevity, since their implementations differ only in the core PIM arithmetic instruction.

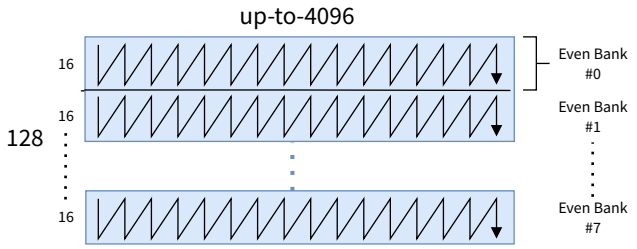


Figure 3: Tile register memory mapping in the Even Banks of a pseudo-channel.

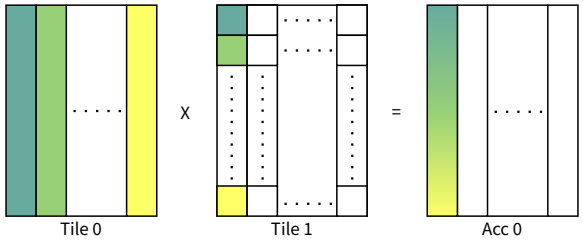


Figure 4: Reduction-free matrix multiplication via MAC-PEP: outer-product decomposition through in-memory vector–scalar MACs.

3.2.1 Tile Registers Memory Layout. The organization of matrix tiles in memory is dictated by the execution model of HBM-PIM. Each PIM unit operates on 16 elements in parallel, and PEPs are executed concurrently across the eight PIM units within a pseudo-channel. Moreover, the absence of cross-lane reduction requires each SIMD lane to process elements from the same row of a tile. To satisfy these constraints, we map tile rows across the Even Banks of a pseudo-channel and store data in column-major order within each bank. This layout ensures that contiguous memory accesses correspond to elements belonging to a single row, aligning with the SIMD execution requirements. As a consequence, the number of rows processed in parallel is fixed by the number of Even Banks, resulting in 128 rows per tile. This fixed row size is a fundamental constraint that drives our implementation of AME matrix tiles.

The maximum number of columns of a Tile register is then derived from the number of Tile registers required by AME specification, in this case equal to four, and the available storage of the Even Banks. From HBM-PIM specification each pseudo channel has available 1 Gb of storage. Considering that this is halved in Even and Odd banks, then the total amount of space available allows the presence of 4 Tile registers of up to 32768 columns. However, this exceeds the maximum dimension supported by the AME specification, which for FP16 data is limited to 4,096 elements per row. Accordingly, in our implementation we restrict tile dimensions to 128×4096. In Figure 3 it is represented this matrix tile mapping inside a pseudo-channel.

3.2.2 AME instructions mapping onto PIM. On HBM-PIM, each PEP is a binary code of PIM instructions loaded in the CRF of each PIM unit during the setup phase. In the listings showed in Listing 1, operations are expressed in pseudo-code, with each operand representing a block of 16 FP16 values processed in a SIMD fashion. Execution proceeds in AB-PIM Mode as described in Section 2.2, addressing operands from a base address plus an offset (in Listings indicated as `bt_#` or `ba_#`, for the base of tiles and accumulators respectively). In general, FP16 instructions relies on AAM, allowing eight iterations of the same instruction on consecutive eight blocks of FP16 values. In the listings this is reported as `for` loop with a counter that increments from 0 to 7. This behaviour is enabled or disabled in the instructions themselves via a dedicated bit-field.

3.2.3 mfadd/mfmul instructions. The `ADD-PEP` and `MUL-PEP` (Listing 1a) implement the AME addition and multiplication instructions on two 128×2048 tiles, iterating 256 times on 128×8 windows and writing results to the Odd Banks. Smaller column dimension are supported by reducing the `JUMP` instruction *loop counter*, at the cost of increased relative setup overhead; smaller row dimensions result in unused SIMD lanes, as the parallel execution width is fixed by the PIM architecture.

3.2.4 mfsub instruction. Subtraction is not natively supported by the PIM ISA, and it is emulated by multiplying the subtrahend by `-1` using the `MUL` instruction with the `SRF_Ms` scalar broadcast register, then adding the result to the minuend. This requires reserving a small portion of the Even Bank to store a vector of `-1` values for the `SRF_Ms` initialization. The resulting `SUB-PEP` (Listing 1b), operates identically to the other PEPs with a moderate throughput reduction due to the additional instructions required.

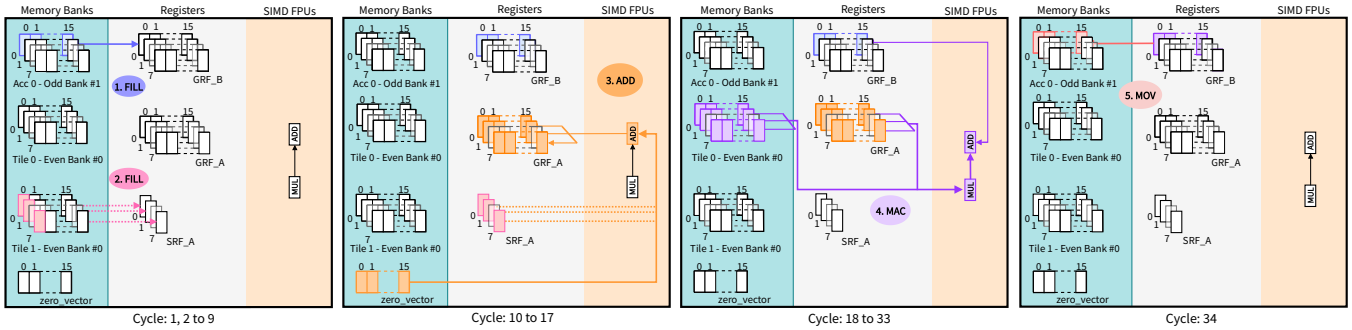


Figure 5: Cycle-by-cyle execution of the *mfmacc* AME instruction through the MAC-PEP, illustrating the dataflow across memory banks, PIM registers (GRF, SRF), and SIMD FPUs over the five micro-operations.

AME implementation constant	Value
ELEN	16 bits
TLEN	2^{23} bits
TRLEN	2^{16} bits
ROWNUM	128
ARLEN	2^{16} bits
ALEN	2^{23} bits

Table 2: Summary of the implementation-defined constant parameters values

3.2.5 mfmacc instruction. The proposed `MAC-PEP` (Listing 1c), in conjunction with the memory mapping of matrix tiles described in Paragraph 3.2.1, enables the complete execution of matrix multiplication within HBM-PIM without relying on external resources such as reduction engines, as required by previously proposed solutions [4]. The computation follows an outer-product formulation implemented through a sequence of matrix–vector operations. At each step, a column of the first source tile is multiplied by a scalar element from the second source tile, and the resulting vector is accumulated into the corresponding column of the destination accumulator tile. We leverage here a double broadcasting of the second source matrix operand, from a memory location to all PIM units `SRF_As`, then from each of them to GRF_As using an `ADD` with a vector of zeros. This requires two instructions in the current PIM ISA. This is due to the impossibility of using `SRF_Ms` as input operand for PIM `MAC` instruction in Address Aligned Mode (AAM). As a result, matrix multiplication can be performed entirely within the native data movement and execution model of the HBM-PIM.

The sequence of operations is depicted in Figure 5, which shows the different execution phases as a timeline of events across the HBM-PIM micro-architecture relative to `PIM Unit #0`, with each phase replicated separately for clarity.

The `MAC-PEP` operates independently of the tile shape, supporting both $128 \times 2048 \times 1$ matrix–vector and $128 \times 8 \times 256$ matrix–matrix multiplications within a single execution, with the tile size bounded by the `JUMP` instruction's maximum iteration count of 256. The impact of tile shape on computational efficiency is discussed in Section 4.

3.2.6 Data movement instructions. As previously described, in this work we propose to map the AME ISA directly onto the HBM-PIM computational primitives with AME tiles and accumulation registers abstracted as memory regions in the HBM-PIM DRAM dies. In addition to computational instructions, AME defines dedicated *load* and *store* operations, as well as instructions for matrix register management. We implemented these in our proposed approach by means of a hardware table that tracks the memory addresses associated with each AME tile and accumulation register. Specifically, for each `tr0-tr3` and `acc0-acc3`, the table records the corresponding memory base addresses. This indirection enables the translation of operand fields in issued AME instructions into the appropriate memory transactions required by the PIM units to operate on the correct data.

Instructions that write, reorganize, or transform data — such as transposed load, store, move, broadcast, pack, and slide — are implemented as memory operations that produce new memory locations. Upon completion, the hardware table is updated to associate the affected AME registers with the newly generated locations, while the requested data transformation is carried out within memory.

It has to be noted that data movement instructions in AME are very important, because they enable packing and unpacking routines, which pre-process data to leverage cache locality. In our proposal these instructions can be resolved without a real data movement, but by just updating the pointers in the hardware table.

3.3 Programming Model

The AME-PIM implementation fully supports the CSRs, which expose configuration, synchronization, and execution state to software in a way that is consistent with the original proposal. Table 2 summarizes the maximum implementation-defined constant AME parameters based on the proposed implementation. Tile operands are memory-resident data structures within HBM-PIM, with a maximum supported dimension of 128×4096. The *msettilem|k|n[i]* configuration instructions update both the CSR state and the PEP loop counters, ensuring that tile dimension changes are correctly reflected in all subsequent PIM kernel executions.

4 Results

This section presents the experimental results, providing an initial assessment of the performance of an AME implementation based on HBM-PIM.

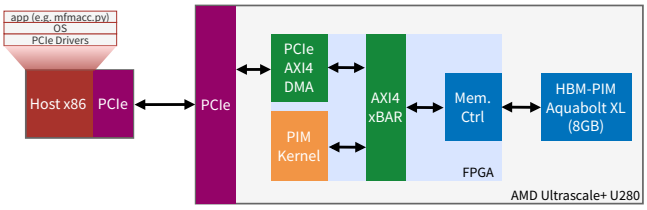


Figure 6: Evaluation platform.

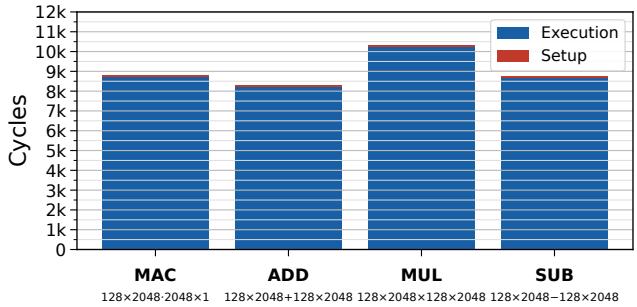
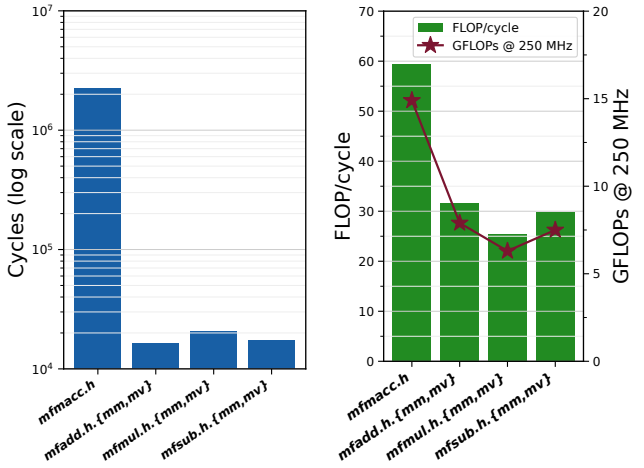


Figure 7: Cycle counts measured for PIM Execution Primitives. Each primitive is annotated with operand dimensions.



(a) Cycles of AME instructions **(b) FLOP/Cycle and GFLOPS AME instructions**

Figure 8: Relative and absolute performance of AME instructions mapped via PEPs onto HBM-PIM

4.1 Experimental Setup

The evaluation was conducted on a Xilinx Alveo U280 FPGA equipped with Samsung Aquabolt-XL HBM-PIM, connected to an x86 host processor via PCIe. The FPGA fabric hosts a simple logic design comprising a memory controller, a PCIe DMA IP, an AXI4 crossbar, and a dedicated encrypted PIM management block that issues PIM commands, referred to as `PIM_kernel`, as illustrated in Figure 6. AME instructions are emulated through host-side applications that issue commands to the HBM-PIM via the PCIe driver, triggering execution on the FPGA. This setup is adopted purely for evaluation purposes, as we do not envision AME-PIM as a PCIe accelerator. The `PIM_kernel` block coordinates execution within a single pseudo-channel but is designed around a fixed access pattern for matrix–vector multiplications, which limits the set of programs that can be directly evaluated [4].

In the following evaluation we characterized both the execution cycles as well as the setup cycles used for programming the PEP kernel and configuring the HBM-PIM. Cycle counts were collected using FPGA-mapped hardware counters triggered by the specific memory transactions that enable PIM functionality. As performance counters are not directly available within the HBM-PIM, the reported measurements reflect performance observed from the bus side. As reported by Lee et al. [14] the FPGA bus frequency of 250 MHz matches the PIM units one.

4.2 PEP and AME Instructions Characterization

In this subsection, we evaluate the cycles associated with the execution of individual PEPs and performance of AME instructions implemented via those. These measurements are reported in Figure 7 and exhibit limited variability across different operations. This behaviour is primarily attributable to the constraints of the available HBM-PIM experimental platform, in which PIM trigger commands are generated by PIM_kernel implemented on the FPGA, resulting in a largely uniform execution latency across PEP types.

Overall we can notice that on the maximum tile size the setup cost is lower than the 1% of the run-time cost which means it can be neglected. Applying the previously reported results to AME instructions requires accounting for the number of times each PEP must be invoked to complete a given instruction when operating on maximum-dimension tiles. Specifically, the *mfmacc* instruction requires 256 MAC-PEP invocations, *mfadd*, *mfmul* and *mfsub* require a double invocation. The resulting cycle counts are shown in Figure 8a, along with relative performance in `FLOP/cycle` and absolute performance in `GFLOP/s` considering the FPGA frequency of 250MHz (Figure 8b).

The maximum theoretical throughput of a pseudo-channel is `128 FLOP/cycle`. In practice, this performance is not attainable due to the setup phase overhead (for small tile sizes) and the additional operations required to overcome intrinsic limitations of the PIM ISA, such as the lack of cross-lane reduction and native subtraction support.

As reported in Figure 8, the measured `FLOP/cycle` for the *mfmacc* instruction reflects the ratio between PIM instructions devoted to actual computation within the `MAC-PEP` and those required for data movement and arrangement. Within the central execution loop, this ratio is approximately 1-to-1, implying that at most half of the theoretical peak performance can be sustained, excluding the initial setup and final data movement overheads. This results in the observed throughput of 59.4 `FLOP/cycle`. As a comparison, MPC-Wrapper[4] shows a performance of 58.1 `FLOP/cycle` per single pseudo-channel on a GEMV on a 1024×128 matrix. The same principle applies to all other AME instructions implemented.

To overcome this limitation, we observe for *mfmacc* the need for a single-cycle memory-to-all-bank lane broadcast instruction, which would allow a single operand tile to be distributed across all compute lanes without incurring per-bank load instructions,

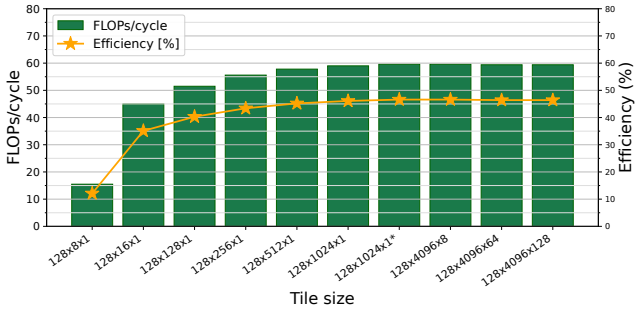


Figure 9: Scaling of FLOPs per cycle for the *mfmacc* instruction with increasing tile size. (* same performance is obtained by 128×8×256 configuration).

effectively reducing the data-movement overhead. While such support would require additional interconnect and control complexity, similar broadcast mechanisms are commonly employed in SIMD and GPU architectures, suggesting its feasibility. For element-wise operations, this limitation can be mitigated by fusing multiple instructions (e.g., combining operand loading and arithmetic) or by allowing two bank-resident operands, thereby amortizing memory accesses across multiple computations. These approaches reduce, but do not fully eliminate, the imbalance, as they remain constrained by memory bandwidth and instruction encoding complexity.

Computational efficiency scales strongly with tile size, as shown in Figure 9. For smaller tiles, the `MAC-PEP` requires fewer `JUMP` iterations than the maximum supported by the PIM ISA, causing setup overheads to dominate. As tile dimensions grow toward $128 \times 2048 \times 1$, efficiency saturates at 59.4 `FLOP/cycle`, bounded solely by the PIM ISA limitations by the 1-to-1 ratio of compute-to-data-movement instructions within the MAC-PEP loop. The same trend applies to all element-wise PEPs. This scaling behaviour confirms that the proposed HBM-PIM matrix acceleration is best suited to large, tiled dataflows, making it a natural fit for an AME-based implementation.

5 Related Works

Prior FPGA-based HBM-PIM accelerators rely on custom designs that retain reduction logic outside memory. The RNN-T accelerator [9] offloads GEMV to in-memory units but follows a conventional inner-product dataflow, requiring host-side reduction of partial products after write-back to DRAM. MPC-Wrapper [4] scales performance across 16 pseudo-channels on Aquabolt-XL but similarly depends on external FPGA logic to aggregate intermediate results. Both approaches partially undermine the motivation for PIM by introducing significant intermediate data movement. SparsePIM [10] and IMPRINT [5] extend the HBM-PIM ISA to support irregular workloads, but are evaluated through modelling rather than physical implementation, leaving their feasibility under memory-compatible logic density constraints unclear. On the software side, the programming model shipped with Aquabolt-XL [12] exposes PIM execution as raw sequences of DRAM commands and mode transitions, tightly coupling applications to device-specific details and limiting portability. Our work differs from all of the above by enabling in-memory accumulation for GEMM, GEMV, and element-wise operations on a single pseudo-channel through a reduction-free outer-product dataflow, while targeting a portable ISA-level abstraction.

Table 3: Comparison of AME-PIM with related works.

Work	Pseudo-channels (pCH)	In-memory acc.	GEMM	GEMV	Eltwise Ops	GEMM/GEMV FLOP/cycle per pCH
RNN-T [9]	1	✘	✘	✔	✘	n.a.
MPC-Wrapper [4]	16	✘	✔	✔	✘	58.1
This work	**1**	✔	✔	✔	✔	59.4

6 Conclusions and Future Work

With the objective of addressing the *von Neumann bottleneck*, this work analysed the capabilities of HBM-PIM with the objective of employing it as a hardware substrate for the implementation of an ISA-level matrix extension. Using the RISC-V AME T-HEAD proposal as a target, we investigated whether the majority of matrix computations could be executed directly within HBM-PIM, without relying on external computational engines and dedicated software stacks. For most operations, this approach proved feasible, ranging from element-wise arithmetic to matrix tile multiplication. We proposed mappings that rely on specific dataflows and architectural workarounds to compensate for missing features in the PIM execution model, such as native reduction and subtraction support. However, certain operations, including element-wise maximum and minimum, could not be implemented under these constraints.

We conducted a preliminary performance evaluation, indicating promising performance and scaling for matrix tile multiplication, while also highlighting the significant computational overhead required to overcome PIM ISA restrictions. These results emphasize both the potential and the current limitations of using HBM-PIM as a backend for an ISA-level matrix extension.

Future work will focus on a more comprehensive performance evaluation of the proposed programming model, including the cost of data movement instructions and the coordinated use of all available HBM-PIM pseudo-channels. The ultimate goal is to achieve a fully functional AME implementation that fully exploits the computational capabilities of the HBM-PIM platform.

Acknowledgments

This activity has been supported by the DARE (g.a. 101143421) project and the ICSC Spoke future HPC. The authors gratefully acknowledge Samsung for the donation of modified Xilinx Alveo U280 platforms with HBM-PIM support used in this work.

References

[1] Arm Ltd. 2024. Arm Scalable Matrix Extension (SME) Introduction. https://developer.arm.com/community/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-scalable-matrix-extension-introduction. Accessed: 2026-01-26.

[2] Danilo Cammarata, Matteo Perotti, Marco Bertuletti, Angelo Garofalo, Pasquale Davide Schiavone, David Atienza, and Luca Benini. 2025. Quadrilatero: A RISC-V programmable matrix coprocessor for low-power edge applications.

In *Proceedings of the 22nd ACM International Conference on Computing Frontiers: Workshops and Special Sessions (CF '25 Companion)*. Association for Computing Machinery, New York, NY, USA, 66–69. doi:10.1145/3706594.3726978

[3] Kyungjun Cho, Hyunsuk Lee, and Joungho Kim. 2016. Signal and power integrity design of 2.5D HBM (High bandwidth memory module) on SI interposer. In *2016 Pan Pacific Microelectronics Symposium (Pan Pacific)*. 1–5. doi:10.1109/PanPacific.2016.7428425

[4] Jinwoo Choi, Yeonan Ha, Hanna Cha, Seil Lee, Sungchul Lee, Jounghoo Lee, Shinhaeng Kang, Bongjun Kim, Hanwoong Jung, Hanjun Kim, and Youngsok Kim. 2024. MPC-Wrapper: Fully Harnessing the Potential of Samsung Aquabolt-XL HBM2-PIM on FPGAs. In *2024 IEEE 32nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 162–172. doi:10.1109/FCCM60383.2024.00027

[5] Ersin Cukurtas, Kavish Ranawella, Kevin Skadron, and Mircea Stan. 2026. IMPRINT: In-Memory Processing with Indirect Addressing Techniques for GPU-hosted HBM-PIM. In *Proceedings of the International Symposium on Memory Systems (MemSys '25)*. Association for Computing Machinery, New York, NY, USA, 165–176. doi:10.1145/3767110.3767121

[6] Intel Corporation. 2023. Intel® Architecture Instruction Set Extensions and Future Features Programming Reference. https://cdrdv2-public.intel.com/836496/architecture-instruction-set-extensions-programming-reference.pdf. Includes Intel AMX architectural overview and instructions. Accessed: 2026-01-26.

[7] JEDEC Solid State Technology Association. 2013. High Bandwidth Memory (HBM) DRAM. JESD235. www.jedec.org Standard document, available from www.jedec.org.

[8] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the NVIDIA volta GPU architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

[9] Shinhaeng Kang, Sukhan Lee, Byeongho Kim, Hweesoo Kim, Kyomin Sohn, Nam Sung Kim, and Eojin Lee. 2022. An FPGA-based RNN-T Inference Accelerator with PIM-HBM. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 146–152. doi:10.1145/3490422.3502355

[10] Taewoon Kang, Geonwoo Choi, Taeweon Suh, and Gunjae Koo. 2025. SparsePIM: An Efficient HBM-Based PIM Architecture for Sparse Matrix-Vector Multiplications. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 495–512. doi:10.1145/3721145.3735111

[11] Joonyoung Kim and Younsu Kim. 2014. HBM: Memory solution for bandwidth-hungry processors. In *2014 IEEE Hot Chips 26 Symposium (HCS)*. 1–24. doi:10.1109/HOTCHIPS.2014.7478812

[12] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jangseok Choi, Jeonghyeon Cho, Kyomin Sohn, Youngsoo Sohn, Kwangil Park, and Nam Sung Kim. 2021. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. 1–26. doi:10.1109/HCS52781.2021.9567191

[13] 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)*, Vol. 64. 350–352. doi:10.1109/ISSCC42613.2021.9365862

[14] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, O Seongil, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology : Industrial Product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 43–56. doi:10.1109/ISCA52012.2021.00013

[15] RISC-V International. 2024. Enhancing the Future of AI/ML with Attached Matrix Extension. https://riscv.org/blog/enhancing-the-future-of-ai-ml-with-attached-matrix-extension/. Accessed: 2026-01-22.

[16] Kyomin Sohn, Won-Joo Yun, Reum Oh, Chi-Sung Oh, Seong-Young Seo, Min-Sang Park, Dong-Hak Shin, Won-Chang Jung, Sang-Hoon Shin, Je-Min Ryu, Hye-Seung Yu, Jae-Hun Jung, Hyunui Lee, Seok-Yong Kang, Young-Soo Sohn, Jung-Hwan Choi, Yong-Cheol Bae, Seong-Jin Jang, and Gyoyoung Jin. 2017. A 1.2 V 20 nm 307 GB/s HBM DRAM With At-Speed Wafer-Level IO Test Scheme and Adaptive Refresh Considering Temperature Distribution. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 250–260. doi:10.1109/JSSC.2016.2602221

[17] XuanTie RISC-V Team. 2024. RISC-V Matrix Specification Proposal. https://github.com/XUANTIE-RV/riscv-matrix-extension-spec. Version v0.6.0, 2024-12-11: Draft.