2026 IEEE International Symposium on Performance Analysis of Systems and Software

MINISA: Minimal Instruction Set Architecture for Next-gen Reconfigurable Inference Accelerator

Jianming Tong[‡], Devansh Jain[†], Yujie Li[‡], Charith Mendis[†], Tushar Krishna[‡]
[†]*University of Illinois*, Urbana-Champaign, USA
devansh9@illinois.edu, charithm@illinois.edu
[‡]*Georgia Institute of Technology*, Atlanta, Georgia, USA
jianming.tong@gatech.edu, yli3821@gatech.edu, tushar@ece.gatech.edu

***Abstract*— Modern reconfigurable AI accelerators rely on rich mapping and data-layout flexibility to sustain high utilization across matrix multiplication, convolution, and emerging applications beyond AI. However, exposing this flexibility through fine-grained micro-control results in prohibitive control overhead of fetching configuration bits from off-chip memory. This paper presents MINISA, a minimal instruction set that programs a reconfigurable accelerator at the granularity of Virtual Neurons (VNs), the coarsest control granularity that retains flexibility of hardware and the finest granularity that avoids unnecessary control costs. First, we introduce FEATHER+, a modest refinement of FEATHER, that eliminates redundant on-chip replication needed for runtime dataflow/layout co-switching and supports dynamic cases where input and weight data are unavailable before execution for offline layout manipulation. MINISA then abstracts control of FEATHER+ into three layout-setting instructions for input, weight, and output VNs and a single mapping instruction for setting dataflow. This reduces the control and instruction footprint while preserving the legal mapping and layout space supported by the FEATHER+. Our results show that MINISA reduces geometric mean off-chip instruction traffic by factors ranging from $35\times$ to $(4 \times 10^5)\times$ under various sizes under 50 GEMM workloads spanning AI (GPT-oss), FHE, and ZKP. This eliminates instruction-fetch stalls that consume 96.9% of micro-instruction cycles, yielding up to $31.6\times$ end-to-end speedup for 16×256 FEATHER+. Our code: https://github.com/maeri-project/FEATHER/tree/main/minisa.**

I. Introduction

The new AI revolution drives crazily increasing installs of data centers for AI. With AI workloads become diverse, AI accelerators increasingly become more reconfigurable for efficiently support workloads of diverse shapes. Their performance stems from two sources: (i) high-throughput low-precision matrix/vector execution and (ii) programmable data movement that enables diverse dataflows and layouts. This programmability is now crucial not only for CNNs and Transformers, but also for workloads that can be compiled into matrix and vector operators, including homomorphic encryption (HE [22]) and zero-knowledge proofs (ZKP [21]). Yet this same flexibility creates a challenge: **control overhead**.

FEATHER [23] represents the SotA accelerator enabling (dataflow, layout) coswitching for each individual workload. However, its programming interfaces expose **fine-grained** or **micro-coded** control over buffer layouts, on-chip routing, and data-to-PE mappings. While expressive, such a paradigm inflates instruction traces, increases off-chip instruction fetch traffic, and consumes limited valuable on-chip memory for storing instructions. The resulting control footprint can directly limit the size of compute tiles that fit on the chip, reducing arithmetic intensity and hurting end-to-end throughput. As accelerators scale in array dimensions and workload diversity, this control overhead becomes a first-order bottleneck (§III-D).

This paper argues that the right abstraction boundary for minimizing control overhead is the **Virtual Neuron (VN)**. A VN corresponds to the smallest hardware dot-product atom. A **coarser-grained** abstraction than current VN would ignore inter-PE mapping flexibility, while a **finer-grained** abstraction would introduce unnecessary control costs without gaining flexibility. Therefore, programming at the level of VN is exactly the level to minimize control overhead without losing hardware built-in flexibility.

Building on this insight, we present a compact VN-abstracted instruction set architecture, MINISA, that captures the flexibility in mapping and layout of FEATHER [23], one SotA reconfigurable accelerator, using only eight instructions. Layout-setting instructions parameterize the nested-loop order of VNs in on-chip buffers. The mapping instruction specifies dataflow and triggers execution of a compute tile under the current layout configurations. For a single layer, the program trace consists of three layout instructions followed by a sequence of `ExecuteMapping` and `ExecuteStreaming` invocations. For consecutive layers, the output of the first layer becomes the input of the next, allowing `SetIVNLayout` of the next layer to be used to specify the output layout of next one. This allows skipping `SetOVNLayout` of the current one.

While MINISA compresses FEATHER's control footprint, the baseline FEATHER microarchitecture still creates two practical frictions for dynamic workloads: it can induce data duplication in on-chip buffers, and its buffer/interconnect design relies on the assumption that one operand (often weights) needs to be pre-known and pre-reordered into preferred layout before program execution. This assumption limits applicability when both operands may arrive or change at runtime, such as LLM inference. To remove these barriers without redesigning the accelerator, we augment FEATHER [23] into FEATHER+ with minimal architectural refinements on (1) interconnect and (2) buffer organization to add distribution flexibility for (1) eliminating data duplication in on-chip buffers, and (2) supporting scenarios when weights are not pre-known for

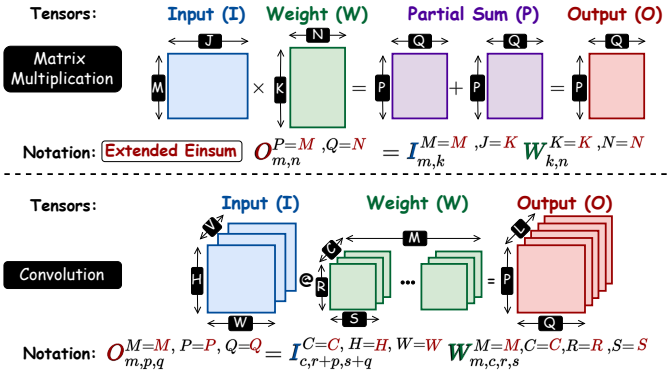

Fig. 1: Workload Illustration. Convolution is converted to MatMul via im2col. We use color coding: blue/green/purple/red for Input(I)/Weight(W)/Partial-sum(psum, P)/Outputs(O).

compile-time layout pre-reorder. Our contributions are:

- We identify control overhead as a key limiter of tile size and instruction efficiency in reconfigurable AI accelerators, instruction fetching stall share rises from 0% (FEATHER+ with smaller than 8×8 PE Array) to 96.9% (16×256 FEATHER+).
- We propose FEATHER+, a modest enhancement that (1) supports dynamic input data, (2) removes data duplication in on-chip buffers, and (3) enables inputs to be loaded inside NEST and streaming weights.
- We introduce MINISA, a minimal VN-level ISA for FEATHER that preserves hardware-supported mapping and layout flexibility with eight instructions.
- We demonstrate that MINISA enables up-to $31.6\times$ speedups by avoiding instruction-fetch stalls. We show MINISA reduces instruction-to-data ratio from $\sim100\times$ (micro-instruction programming) to negligible (MINISA) for 16×256 FEATHER+. MINISA maintains $< 0.1\%$ instruction-cycle fraction.

II. Background

A. Workload Specification

Modern AI inference workloads—from LLM attention blocks to text-to-image pipelines—are dominated by matrix multiplications and convolutions whose *operand values and shapes* can vary across layers and requests. To describe this diversity in a uniform way, we adopt *extended einsum* notation [5]. Taking matrix multiplication as an example (Fig. 1).

$$O^{P=M,Q=N}_{m,n} = I^{M=M,J=K}_{m,k} \cdot W^{K=K,N=N}_{k,n}$$

Each tensor is annotated with:

- **Superscript**: an ordered list of *rank names* that define the logical shape. Ranks are capitalized, such as $I \in \mathbb{R}^{M\times J}$, $W \in \mathbb{R}^{K\times N}$, $P \in \mathbb{R}^{P\times Q}$, $O \in \mathbb{R}^{P\times Q}$.
- **Subscript**: an ordered list of quasi-affine *index expressions* over rank variables and constants (e.g., $[m, k]$). Rank variables use the lowercase form of the corresponding rank name.

Partial sums represent intermediate accumulation states that arise when the rank variable of reduction rank (e.g., k) is distributed across non-consecutive loop levels in mapping, requiring aggregation of partial results to produce final outputs.

B. Mapping and Layout

A given extended einsum (Fig. 1) can be executed with many legal schedules. We refer to a fine-grained schedule of compute and memory accesses as a **mapping** which specifies the partitioning, ordering and parallelism of ranks [11], [20].

Independently, each tensor can be organized differently in on-chip memories. We use **layout** to denote the fine-grained organization of data in the on-chip buffers [20], [23].

C. Reconfigurable Accelerator

No single dataflow is globally optimal across the wide shape space of modern operators [17], [20]. This motivates **reconfigurable accelerators** that can adapt mappings and layouts at runtime to sustain high utilization across workloads [3], [12]–[15], [18], [23], [24]. FEATHER [23] is a recent practical design that enables low-cost *co-switching* of dataflow and layout, making reconfigurable execution deployable.

However, FEATHER has two limitations.

- **Pre-known weights offline reordered into ideal layout**. This is because in FEATHER, there is only a single reorder-in-reduction NoC designed to support output layout reordering. For weights, the rigid point-to-point connections from buffers to computation array require weights to be preknown and offline reordered into desired layout. It works for general convolution but is inefficient for recent LLMs when both inputs and weights can arrive or change at runtime. Further, the offline reorganized weights might contain *redundant duplicated data in on-chip buffer*.
- **Heavy Control Overhead**. Programming individual switches require dominating configurations overhead at scale.

To eliminate both, we introduce **FEATHER+** and its programmer view in §III, and Virtual Neuron abstraction and VN-based ISA, MINISA, for FEATHER+ in §IV.

III. FEATHER+ and Programming View

This section introduces FEATHER+ and programming view.

A. FEATHER+ Overview

At its compute, FEATHER+ (Fig. 2) comprises:

- **NEST**: a column-wise independent $AH\times AW$ PE array. Each PE holds $2 \times AH$ local registers (double buffered to hide loading latency) and performs AH-element dot product.
- **BIRRD**: a multi-stage reordering-in-reduction network to change its target bank during the data reduction.

In its memory, FEATHER+ has:

- **Streaming Buffer** stores **"streaming tensor"**, which is pipelined from top to bottom within each NEST column and reused by all PEs in that column.
- **Stationary Buffer** stores **"stationary tensor"** in local PE registers (green). These register values participate in a single dot product with the streaming tensor as it passes through PE.
- **Output Buffer** (OB) stores partial sums that require future update, i.e. temporal reduction. OB is the only multi-bank buffer with individual address generation per bank for supporting flexible output layout reordering.

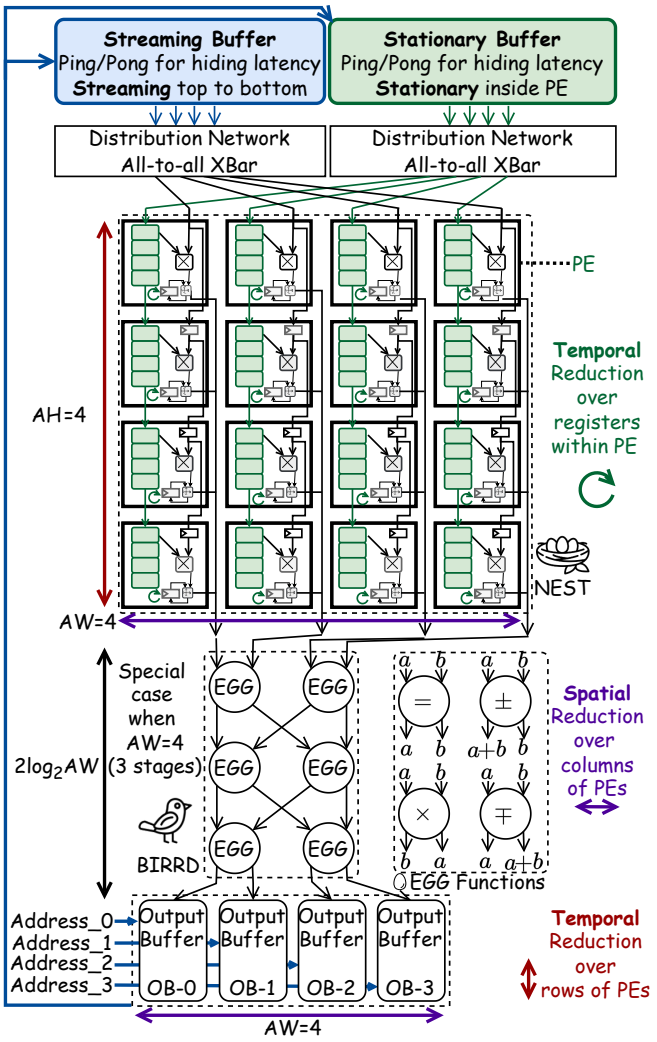

Fig. 2: Programmer view of FEATHER+, the reconfigurable accelerator capable of co-switching both mapping and layout.

B. Architectural Refinement of FEATHER into FEATHER+

FEATHER+ (Fig. 2) has three architectural refinement:

- **All-to-all Distribution from Buffers to NEST**. We replace the per-column point-to-point connection from the streaming and stationary buffers to NEST with two separate all-to-all crossbars. This allows any resident data to be flexibly multicasted to arbitrary PE columns, eliminating the need to explicitly materialize duplicated copies of the same data in on-chip buffers. Such crossbar-style distribution is common in contemporary AI ASICs [8]–[10], being resource-reasonable.
- **Simplified Streaming Buffer Banking**. With all-to-all distribution, FEATHER+ no longer requires a multi-bank streaming-buffer interface to access different rows per bank. We regress to single bank with simplified address generation.
- **Links from Output Buffer to Stationary Buffer**. This allows current layer's outputs, used as the next layer's inputs, to either stay stationary in local registers of PEs or stream through PEs for dot product with the next layer's weights.

Together, these changes (1) remove the requirement that "stationary tensor" must be known ahead of execution to be placed in a preferred layout in stationary buffer, (2) free duplicated data in on-chip buffers, and (3) support both inputs and weights to be used as "stationary tensor". FEATHER+ serves as the default hardware backend for this work.

C. Programming View of FEATHER+

FEATHER+ is a *constrained dataflow accelerator* [19]. A programmer needs to map workloads into "*three-level reduction*" with "*two dataflow choices*" under "*two constraints*".

1) Flexible Mappings:

a) Three-level Temporal and Spatial Reduction:

- **Temporal reduction within a PE** performed by an AH-element dot product between local registers and streamed data over time, which generates a single psum.
- **Spatial reduction** over **a row of PEs via BIRRD**.
- **Temporal reduction** over **a column of PEs in OB**.

Such flexible reduction allows any selected PEs to be grouped together to compute one dot product.

b) Two Flexible Mixed Dataflow:

- **Input-Output Stationary (IO-S)**: store inputs in each PE, stream weights, and accumulate psums locally within PE.
- **Weight-Output Stationary (WO-S)**: store weights in each PE, stream inputs, and accumulate psums locally within PE.

For MatMul, pick IO-S when $M > N$, otherwise WO-S.

2) Mapping Constraints:

a) Constraint 1: Intra-PE Dot Product: All AH data in a PE's local registers must participate in the same dot product, ensuring each PE's stationary tensor is reduced over time.

b) Constraint 2: Intra-Column Data Reuse: The streaming tensor is reused across all PEs within the same column to compute dot products with their respective stationary tensors. However, all columns in NEST operate entirely independently.

D. Control Overhead of FEATHER+

While FEATHER+ supports board mapping/layout choices, *realizing* this flexibility via switch- and cycle-level micro-control can incur substantial overheads. Programs must specify control for BIRRD and buffer address generation for each cycle. This has two direct consequences. First, control state consumes on-chip storage that could otherwise be used for activations and weights, limiting the largest feasible tile size and reducing arithmetic intensity. Second, fetching long instruction traces increases off-chip instruction traffic, which is especially harmful for memory-bound layers. As accelerator arrays scale, this control pressure becomes a first-order limiter of end-to-end throughput. When adopting micro-control to configure each individual switches for FEATHER+, significant memory overhead of instructions fetching would dominate at larger scale, as shown in Tab. I. Such control overhead motivates MINISA – a low-cost abstraction of hardware flexibility.

TABLE I: Explicit stall of fetching instructions for matrix multiplication of $\sum_k I^{M=65536,J=40}_{m,k} \cdot W^{K=40,N=88}_{k,n}$

FEATHER	4×4	8×8	4×64	16×16	8×128	16×256
stall	0	0	75.3%	65.2%	90.4%	96.9%

IV. MINISA

This section introduces MINISA, a minimal instruction set that configures FEATHER+ at the Virtual Neuron (VN) granularity. We detail the VN abstraction, its mapping constraints under VN, the corresponding ISAs, and the execution model.

A. Key Idea: VN (AH-element Dot Product) as Abstraction

MINISA is inspired by a simple architectural observation:

FEATHER+ always computes at the granularity of an AH-element dot product in its PE, and supports flexible reduction over the the dot-product results from any PEs.

We therefore define a **Virtual Neuron**[1] as the smallest software operand fragment that matches this hardware atom, i.e. AH-element dot product. This leads to a key insight:

Programming at VN granularity is the coarsest control that still preserves inter-PE mapping flexibility and the finest control that avoids unnecessary switch-level overhead.

Concretely, MINISA re-expresses FEATHER+'s legal mapping and layout space as the granularity of VN. This factorization retains FEATHER+'s built-in reconfigurability while dramatically compressing instruction footprint, enabling larger on-chip tiles and higher end-to-end performance.

B. MINISA Abstraction – Virtual Neuron (VN)

1) Definition: A *Virtual Neuron* is the minimal hardware dot-product atom, which is also the largest atomic unit in software (compilation) without losing flexibility in hardware. For an FEATHER+ with $AH \times AW$ NEST, each PE performs an AH-element dot product, hence the VN size should be smaller than or equal to AH. We use VN size as AH in the following of this paper for clear idea demonstration.

2) Operand-specific VNs: Different operands have their own VNs, including inputs (I_{VN}), weights (W_{VN}), partial sums (P_{VN}) and outputs (O_{VN}). Each VN is obtained by grouping consecutive AH elements along the reduction dimension, i.e. J of inputs, K of weights and Q of partial sum and outputs (which becomes J of input for the layer following it). For example, weight VNs are indexed by $W_{VN}(r,c)$, where $r \in [0, \lceil K//AH \rceil - 1]$ and $c \in [0, N-1]$ denote the row and column indices. In this way, I_{VN} and W_{VN} exactly denote a group of elements from inputs and weights to be **consumed by local dot product in one PE**, respectively. Each PE's dot product yields a single element in either P_{VN} or O_{VN}, depending on whether further reduction is required.

3) FEATHER+ Mapping Constraints under VNs: The VN abstraction hides Constraint 1, leaving Constraint 2 (Intra-Column I_{VN} Reuse) as the only restriction. I_{VN} pipelines top-to-bottom within each NEST column, requiring all PEs in a column to compute dot products between their W_{VN} and this shared I_{VN} stream for correct local reduction.

C. MINISA ISA Overview

1) ISA: MINISA defines four ISAs (Tab. II) to reconfigure mapping and data layout in on-chip buffer at VN granularity, and supporting ISAs for data loading/writing and activation.

TABLE II: Overview of MINISA instructions.

ISA name	ISA meaning
`SetIVNLayout`	Configure data layout for I_{VN} in on-chip buffer.
`SetWVNLayout`	Configure data layout for W_{VN} in on-chip buffer.
`SetOVNLayout`	Configure the *output* buffer layout for output VNs and clear (initialize) the on-chip output buffer for accumulation.
`ExecuteMapping`	Specify the VN-level mapping parameters and trigger execution of a single compute tile on the PE array under the current VN layouts.
`ExecuteStreaming`	Specify how VNs are streamed in each column and swap Dataflow between IO-S and WO-S.
`Load`	Load data from off-chip memory into streaming / stationary buffer.
`Write`	write data from streaming / stationary buffer to off-chip memory.
`Activation`	Perform activation function on data in streaming / stationary buffer.

- `SetIVNLayout` / `SetWVNLayout`: configure the on-chip streaming and stationary buffer layouts for Virtual Neurons of Input (I_{VN}) and Weight (W_{VN}).
- `SetOVNLayout`: Configures the layout for Output Virtual Neurons (O_{VN}) in the output buffer, which becomes the layout of input for next successive operation.
- `ExecuteMapping`: Configures mapping of stationary tensor (W_{VN} or I_{VN}) to PEs in one compute tile.

2) ISA Bitwidth Analysis: ISA parameter bitwidths are designed to support the maximum ratio between on-chip buffer capacities and architectural dimensions. Specifically, this applies to the ratio of stationary or streaming buffer depths ($D = D_{sta} = D_{str}$) to the NEST width or height (AW, AH). Workload parameters can be configured to any value within these bitwidth limits; if the actual element index exceeds the workload shape, it is automatically zero-padded.

ExecuteMapping	*OpCode*	G_r	G_c	r_0	c_0	s_r	s_c
Bitwidth	3	$\lceil \log_2(AW) \rceil$		$\lceil \log_2(D/AH \cdot AW) \rceil$		$\lceil \log_2(D/AH) \rceil$	
value ranges	111	$[1, AW]$		$0 \le r_0 c_0 \le \lceil D/AH \cdot AW \rceil$		$[0, D/AH]$	
E.g. non-zero value when in range for W_{VN}:				$r_0 < K, c_0 < N, s_r \le K/2, s_c \le N/2$			

ExecuteStreaming	*OpCode*	*Dataflow*(df)	m_0	s_m	VN_{SIZE}	T
Bitwidth	3	1	$\lceil \log_2(D/AH) \rceil$-1		$\log_2(AH)$	$\lceil \log_2(D/AH) \rceil$
value ranges	011	0/1 for IO-S/WO-S	$1 \le x \le \lceil D/AH \rceil$-1		$1 \le x \le AH$	$1 \le x \le \lceil D/AH \rceil$

Fig. 3: `Execute*` Field definition for $AH \times AW$ NEST.

D. ExecuteMapping: Flexible Mapping for NEST

1) Definition: For an $AH \times AW$ PE array, PE (a_h, a_w) denotes the PE at row $a_h \in [0, AH)$ and column $a_w \in [0, AW)$. Each PE could holds a single W_{VN}. Therefore, our objective is to specify which $W_{VN}(r,c)$ is stored in each PE. Using WO-S dataflow as example, MINISA realizes all legal mappings using six parameters $\theta_{EM} = (r_0, c_0, G_r, G_c, s_r, s_c)$.

$$r = r_0 + \left\lfloor \frac{a_w}{G_r} \right\rfloor \qquad c = c_0 + s_r\, a_h + s_c \left(a_w \bmod G_c\right) \tag{1}$$

where, (r_0, G_r) control placement of W_{VN}s within each column, preserving the architectural constraint that all PEs in one PE column share the same W_{VN} *"row index (r)"*.

- r_0 denotes the starting row index of the W_{VN} in NEST.
- G_r specifies how many consecutive PE columns share the same W_{VN} row index (r) before incrementing to next row index, and hence is bounded by the number of columns (AW).

(c_0, s_r, G_c, s_c) control W_{VN}s across PE columns.

- c_0 selects the starting W_{VN} column index (c).
- G_c defines replication period for c of W_{VN} across columns.
- s_c determines the spacing in column index (c) of W_{VN} among distinct PE-column patterns within one period.
- s_r is the temporal stride across PE rows, determining how W_{VN} column indices grow within a PE column.

[1] we borrow this term from MAERI [13] which was a flexible dataflow accelerator that defined VN to be a flexible sized dot product engine. But here we use it as the abstraction of a fix-size dot product performed by a PE.

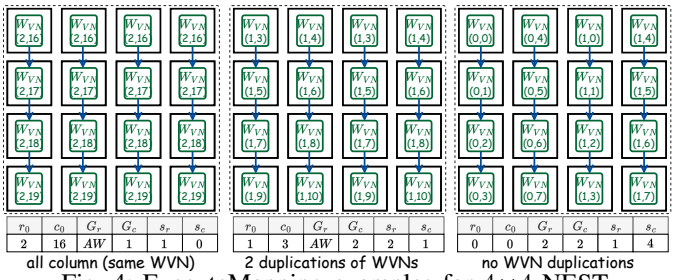

Fig. 4: ExecuteMapping examples for 4×4 NEST.

TABLE III: Unified 3-bit permutation encoding across VN layouts. Each column applies the same `order_id` value to the operand-specific three-rank set remaining after the VN constraint ($K_{L0}=AH$, $J_{L0}=AH$, $Q_{L0}=AH$).

order	W_{VN} order	I_{VN} order	O_{VN} order
000	$k_{L1}\to n_{L0}\to n_{L1}$	$j_{L1}\to m_{L0}\to m_{L1}$	$p_{L1}\to p_{L0}\to q_{L1}$
001	$k_{L1}\to n_{L1}\to n_{L0}$	$j_{L1}\to m_{L1}\to m_{L0}$	$p_{L1}\to q_{L1}\to p_{L0}$
010	$n_{L0}\to k_{L1}\to n_{L1}$	$m_{L0}\to j_{L1}\to m_{L1}$	$p_{L0}\to p_{L1}\to q_{L1}$
011	$n_{L0}\to n_{L1}\to k_{L1}$	$m_{L0}\to m_{L1}\to j_{L1}$	$p_{L0}\to q_{L1}\to p_{L1}$
100	$n_{L1}\to k_{L1}\to n_{L0}$	$m_{L1}\to j_{L1}\to m_{L0}$	$q_{L1}\to p_{L1}\to p_{L0}$
101	$n_{L1}\to n_{L0}\to k_{L1}$	$m_{L1}\to m_{L0}\to j_{L1}$	$q_{L1}\to p_{L0}\to p_{L1}$
110/111	reserved	reserved	reserved

SetWVNLayout	*OpCode*	*Order*	N_{L0}	N_{L1}	K_{L1}
Bitwidth	3	3	$\lceil\log_2(AW)\rceil$	$\lceil\log_2(D/AH)\rceil$	$\lceil\log_2(D/AH)\rceil$
value ranges	000	$[0,5]$	$N_{L1}N_{L0}K_{L1}\le[D/AH]$		

SetIVNLayout	*OpCode*	*Order*	M_{L0}	M_{L1}	J_{L1}
Bitwidth	3	3	$\lceil\log_2(AW)\rceil$	$\lceil\log_2(D/AH)\rceil$	$\lceil\log_2(D/AH)\rceil$
value ranges	001	$[0,5]$	$M_{L1}M_{L0}J_{L1}\le[D/AH]$		

SetOVNLayout	*OpCode*	*Order*	P_{L0}	P_{L1}	Q_{L1}
Bitwidth	3	3	$\lceil\log_2(AW)\rceil$	$\lceil\log_2(D/AH)\rceil$	$\lceil\log_2(D/AH)\rceil$
value ranges	010	$[0,5]$	$P_{L1}P_{L0}Q_{L1}\le[D/AH]$		

Load/Store	*OpCode*	*HBM Address*	*Target*
Bitwidth	3	$\log_2(HBM\ Cap)$	1
value ranges	101/ 100		0 / 1: Stationary/Streaming Buffer

Fig. 5: MINISA specifications for layout, load/store, and dataflow. D refers to the depth of stationary / streaming buffer.

Each buffer could holds at most $D/AH{\cdot}AW$ VNs, such that $r_0{\cdot}c_0$ peaks at $D/AH{\cdot}AW$. Stride peaks at half. Any $W_{VN}(r,c)$ falling outside the tensor bounds is implicitly zero-padded.

2) Case Studies: Because columns are independent, Fig. 4 shows three ExecuteMapping choices: (1) replicate the same WVNs across all columns, enabling all IVN chunks to be processed in parallel across every column. (2) partition the columns into two replicated groups, allowing the IVN stream to be divided into two independent substreams that are processed in parallel. and (3) assign each column a different set of WVNs, so each column processes an independent IVN stream.

E. ExecuteStreaming

1) Definition: Under WO-S, streamed I_{VN}s enter from the top of each PE column and are reused across consecutive cycles as they propagate downward. Therefore, we shall specify which $I_{VN}(m,j)$ is injected into each PE column over time.

To minimize ISA overhead, `ExecuteStreaming` reuses parameters θ_{EM} defined by the paired `ExecuteMapping`, and introduces give additional parameters $\theta_{ES}=(m_0,s_m,T,VN_{size},df)$.

- m_0 is the starting streamed-row index.
- s_m is the temporal stride of streamed I_{VN}.
- T is the number of I_{VN} injected into each PE column.
- df changes the dataflow choice (`IO-S` or `WO-S`).
- VN_{size} defines the size of Virtual Neuron.

All fields encode "value-1" omitting zero to reduce bitwidth.

Given the paired `ExecuteMapping`, the $I_{VN}(m,\ j)$ injected into PE column a_w at temporal step $t\in[0,T)$ is

$$j=r_0+\left\lfloor\frac{a_w}{G_r}\right\rfloor,\quad m=m_0+s_m\,t+\left\lfloor\frac{a_w \bmod G_r}{G_c}\right\rfloor.$$

2) Case Study: As an example, consider an $AH\times4$ PE array with the paired `ExecuteMapping` parameters chosen as $(r_0,G_r,G_c)=(0,2,1)$, and `ExecuteStreaming` set to $(m_0,s_m,T,df)=(0,3,3,1)$. PE columns 0 and 1 belong to reduction group $j=0$, while PE columns 2 and 3 belong to reduction group $j=1$. Within each two-column group, the two columns split and process overall I_{VN} in parallel. Therefore, over three injection cycles, the I_{VN}s entering the top of the four PE columns are:

2	$I_{VN}(6,0)\downarrow$	$I_{VN}(7,0)\downarrow$	$I_{VN}(6,1)\downarrow$	$I_{VN}(7,1)\downarrow$
1	$I_{VN}(3,0)\downarrow$	$I_{VN}(4,0)\downarrow$	$I_{VN}(3,1)\downarrow$	$I_{VN}(4,1)\downarrow$
0	$I_{VN}(0,0)\downarrow$	$I_{VN}(1,0)\downarrow$	$I_{VN}(0,1)\downarrow$	$I_{VN}(1,1)\downarrow$
cycle	PE col 0	PE col 1	PE col 2	PE col 3

In the array, each column corresponds to one PE column in hardware, while each row denotes the I_{VN} injected at the top of that PE column in one cycle. After injection, each I_{VN} propagates downward through the corresponding PE column over consecutive cycles, so the same streamed tensor (I_{VN} under WO-S) is temporally reused by all PEs in that column.

F. Set*VNLayout: Flexible Layout for On-chip Buffers

1) Key Idea: 4-level Partitioned Tensor in AW-width Buffer: The three layout instructions (`SetIVNLayout`, `SetWVNLayout`, and `SetOVNLayout`) define how a logical 2-rank tensor is placed into a physical $D\times AW$ on-chip buffer. Each instruction targets different operand types (I_{VN}/W_{VN}/P_{VN} and O_{VN}). Under WO-S, I_{VN} is mapped to the streaming buffer and W_{VN} to the stationary buffer. Under IO-S, these roles are swapped.

At a high level, layout specification has three steps: (1) choose rank partitioning factors, (2) choose an ordering of the post-partitioned ranks, and (3) fold the resulting VN sequence into the $D\times AW$ buffer.

2) Partitioning Factors: Using the weight matrix (K,N) as an example, we first express layout at element granularity. Each rank is split into two levels:

$$K=K_{L1}K_{L0},\qquad N=N_{L1}N_{L0},$$
$$k=k_{L1}K_{L0}+k_{L0},\qquad n=n_{L1}N_{L0}+n_{L0}.$$

This yields four loop ranks $\{k_{L0},k_{L1},n_{L0},n_{L1}\}$, which can represent both contiguous and strided layouts.

VNs always group consecutive elements along the reduction rank (i.e., $K/J/Q$ for W_{VN}/I_{VN}/O_{VN} in Fig. 6), therefore MINISA fixs the innermost reduction-level factor at the size

of VN, e.g., enforces K_{L0} as VN size for W_{VN}. VN size can be any value up to AH. This paper uses AH for illustration.

Further, all elements of a single VN are accessed serially across multiple cycles (rather than in a single-cycle concurrent access), all elements of that VN are placed in contiguous buffer rows at a fixed buffer-column index.

3) Ordering: The VN abstracts away the level-0 partitioning factor of the reduction rank, the remaining freedom is only the order of the three other remaining ranks, e.g., $\{K_{L1},\ N_{L0},\ N_{L1}\}$ for W_{VN}. This reduced loop space is still complete for VN-granularity layouts. Under such rank partitioning, we identify weight VNs by $W_{VN}(r, c)$, where

$$r = k_{L1}, \qquad c = n_{L1} \cdot N_{L0} + n_{L0}.$$

a) Addressing Generation: We first flatten all VNs according to the chosen rank order to obtain a 1D index L, and then map L to buffer coordinates by row-major placement over the $D \times AW$ buffer. For W_{VN} with ranks $\{K_{L1}, N_{L0}, N_{L1}\}$,

$$\mathbf{R} = [K_{L1},\ N_{L0},\ N_{L1}], \qquad \mathbf{RV} = [k_{L1},\ n_{L0},\ n_{L1}],$$

where $\mathbf{R}$ denotes the ordered ranks and $\mathbf{RV}$ their associated rank variables. Let $P = (p_0, p_1, p_2)$ be a permutation of $0, 1, 2$ defining the loop order from outermost to innermost. The resulting flattened VN index is

$$L = RV_{p_0} \cdot R_{p_1} R_{p_2} + RV_{p_1} \cdot R_{p_2} + RV_{p_2}$$

and the physical buffer address is

$$addr_{row} = \left\lfloor \frac{L}{AW} \right\rfloor, \qquad addr_{col} = L \bmod AW.$$

4) Bitwidth Analysis:

a) Order: Arbitrary ordering of three partitioned ranks offers $3! = 6$ legal permutation choices, requiring $\lceil \log_2 6 \rceil = 3$ bits. Therefore, *order* is encoded with 3 bits. The operand-specific mapping of each code is listed in Tab. III.

b) Partitioning Factors: Partition factors determine size of tensors which must fit in overall buffer capacity. For W_{VN}, the number of VNs that could be stored by a $D \times AW$ buffer is bounded by $K_{L1} \cdot N_{L1} \cdot N_{L0} \le \lfloor \frac{D}{AH} \rfloor \cdot AW$. This restricts the overall bitwidth. We also cap level-0 non-reduction factors (e.g., $N_{L0} \le AW$), since larger values are performance-equivalent to existing value within AW.

5) Case study: Fig. 6 uses loop order $n_{L0} \to k_{L1} \to n_{L1}$ (with $N_{L0} = 4$, $K_{L1} = 2$, $N_{L1} = 2$). For $n_{L0} = 0$, the first buffer row places $W_{VN}(0, 0), W_{VN}(0, 4), W_{VN}(1, 0), W_{VN}(1, 4)$ across the $AW = 4$ columns. The same pattern repeats for $n_{L0} = 1, 2, 3$. This case study makes explicit that `SetWVNLayout` encodes the full legal layout spaces with a compact VN-level descriptor (partition factors + order).

G. Execution Model

1) Roles of MINISA ISAs: MINISA cleanly separates *configuration*, *data movement*, and *compute triggering*.

• **Configuration-only ISAs:** `Swap`, `SetIVNLayout`, and `SetWVNLayout` only program internal control/state registers. They do not move data and do not launch computation.

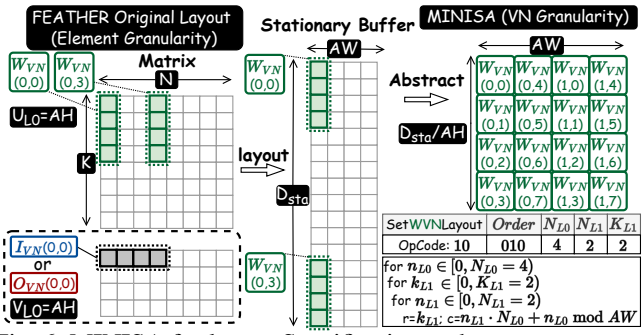

Fig. 6: MINISA for layout Specification and `SetWVNLayout` illustration ($K = 8, N = 8, AH = AW = 4 \Rightarrow K_{L0} = AH$). Takeaway: MINISA organizes layout at VN granularity.

• **Memory-movement ISAs:** `Load` and `Write` trigger data transfers between off-chip memory and streaming/stationary buffers. `SetOVNLayout` additionally manages output-buffer lifecycle at tile boundaries: it initializes the output tile before accumulation and commits the finished tile from output buffer to the next operand buffer according to dataflow (IO-S: streaming, WO-S: stationary).

• **Compute-trigger ISA:** FEATHER+ only triggers on-chip activity when reciving `E.Streaming` and `E.Mapping`. Each issue loads stationary tensors into NEST from on-chip buffers, and computes on one tile under current layouts.

• **Sub-tiled execution:** Layout configurations are reused across multiple pairs of (`ExecuteMapping`, `ExecuteStreaming`) issues, enabling a sequence of sub-tiles to contribute on the same outputs.

2) Execution Model: The canonical trace for one layer is:

$$\texttt{Set*VNLayout} \to \{\texttt{E.Mapping/E.Streaming}\}^{\times T}$$

where T is the number of compute tiles needed to consume the currently loaded I_{VN}/W_{VN} for that layer. The layout instructions first define operand placement and initialize the output tile, and the following `ExecuteMapping` sequence processes all tiles with unique mapping.

For consecutive layers, the output of layer i becomes the input of layer $i{+}1$. Therefore, `SetIVNLayout` is issued once for the first layer, and the `SetOVNLayout` of layer i is reused as the `SetIVNLayout` of layer $i{+}1$, which could be optionally skipped. Each new layer still issues its own `SetWVNLayout`, `SetOVNLayout`, and a batch of `ExecuteMapping` instructions.

3) Walk-through Case Study: Fig. 7 illustrates this execution semantics for matrix multiplication. Assume `SetIVNLayout`, `SetWVNLayout` and `Load` have already established layouts and loaded required VNs. The on-chip tensors are then consumed by two successive `ExecuteMapping/Streaming` instructions (❸) on the same $AH \times AW$ NEST. Both tiles share the same `SetOVNLayout` and accumulate P_{VN}s into the same final O_{VN}. Because layout of O_{VN} remain invariant across the two mappings, P_{VN}s from both tiles are guaranteed to accumulate into consistent locations, completing the full O_{VN} without extra output re-layout or intermediate data movement.

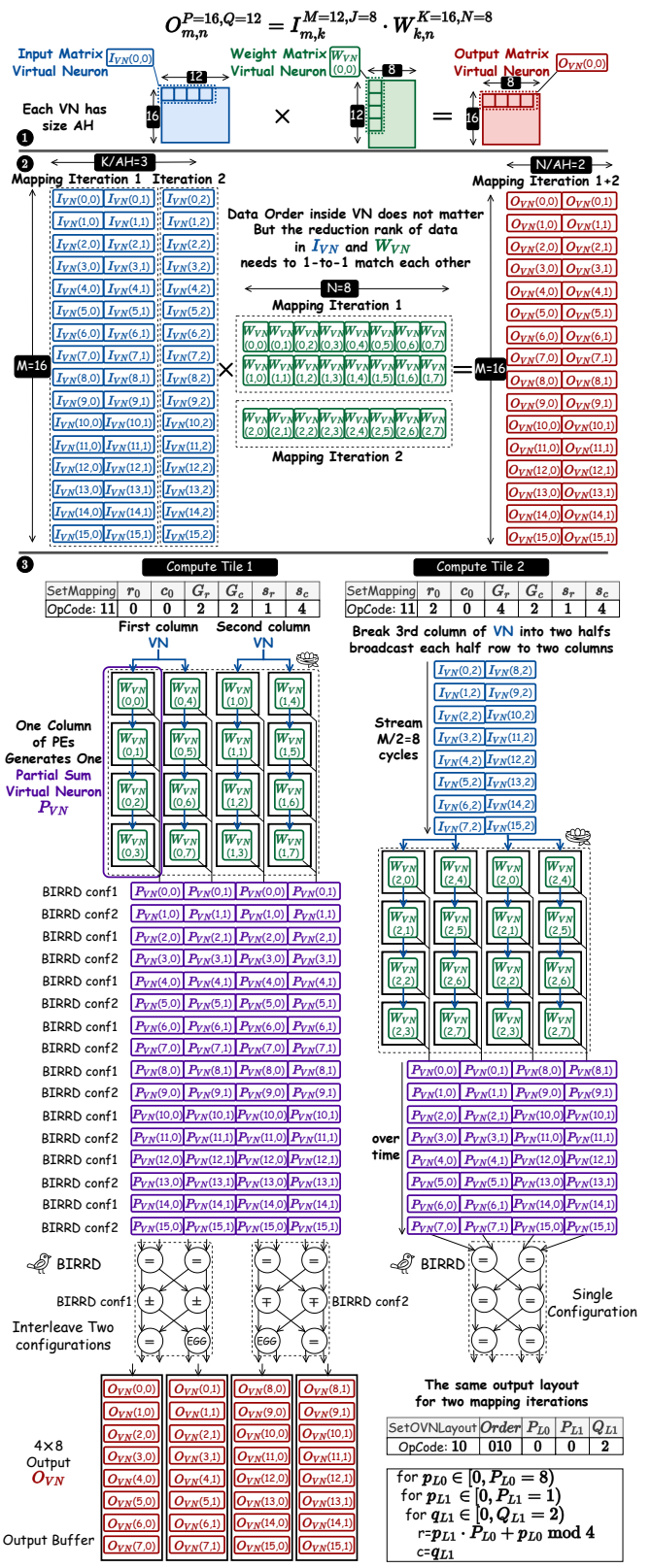

Fig. 7: Mapping a matrix multiplication to FEATHER+ (AH×AW=4×4 NEST) as 2 compute tiles under MINISA.

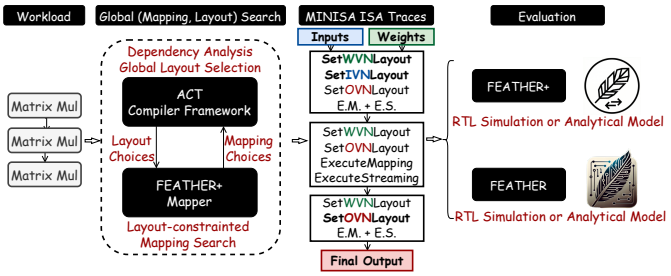

Fig. 8: Compilation flow for FEATHER+.

V. Compilation Pipeline

A. Overview

For a workload with a chain of matrix multiplication, we provide an analytical (mapping, layout) co-search framework, FEATHER+ Mapper. Chosen (dataflow, layout) is deterministically translated into a trace of MINISA ISAs, then lowered into FEATHER+ "micro-instructions".

For a workload represented as a chain (or DAG) of layers including matrix multiplications and activations (e.g., softmax), we integrate FEATHER+ Mapper as a layout-constrained mapping search framework into ACT Ecosystem [6], [7], as shown in Fig. 8. Specifically, ACT first performs graph-level analyses and identifies layout-flexible regions—subgraphs where tensor layouts may be changed while maintaining correctness, subject to any required layout constraints at region boundaries. For each layer in such a region, ACT invokes FEATHER+ mapper to perform layout-constrained mapping search. The selected per-layer mapping choices are sent back to ACT to finalize the global (mapping, layout) choice with the lowest latency. The final choices are lowered into MINISA ISA traces, which are executed in FEATHER / FEATHER+ RTL simulation or analytical simulation model.

B. Overall Flow of the FEATHER+ Mapper

The FEATHER+ mapper performs *mapping-first, layout-second* search. It first enumerates candidate mappings, and then derives buffer layouts that realize each mapping without bank conflicts. This ordering is motivated by a key observation: although the full joint space of mapping and layout is large, the mapping space itself can be compactly parameterized by only three knobs: (1) compute-tile size, (2) VN-group formation, and (3) column duplication. Once a mapping candidate is fixed, the remaining layout search is restricted to layout *orders* and level-0 *partition factors*, which is much smaller and can be checked efficiently for feasibility. Among all feasible {mapping, layout} pairs, the mapper returns the one with minimum estimated latency. Fig. 7 and Fig. 9 illustrate this flow using a matrix multiplication example lowered into two compute tiles. From the mapper's perspective, IO-S is equivalent to a transposed WO-S configuration.

1) Step 1 - Lower the Workload into Virtual Neurons: The input workload, such as convolution or matrix multiplication, is first rewritten into a set of *Virtual Neurons* (VNs), e.g., ❷ in Fig. 7. Each tensor is partitioned along its reduction dimension into fixed-length VNs, where each VN has length at most AH

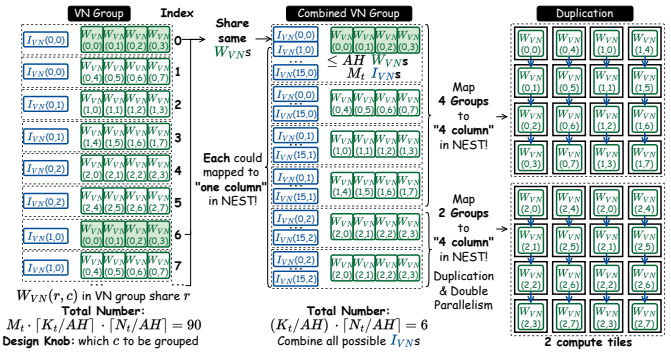

Fig. 9: Mapper illustration of "VN Group", "Combination" and "Duplication" to lower ❷ in Fig. 7 to two compute tiles.

so that it fits the per-PE local registers. VN is the canonical software abstraction manipulated by the mapper.

Under this representation, each tensor becomes a 2D VN array indexed by one non-reduction rank and one reduction-tile rank, i.e., 2D logical arrays of I_{VN}, W_{VN}, and O_{VN}. Execution then reduces to matching one I_{VN} and one W_{VN} that share the same reduction-rank index onto the same PE for a dot product. For instance, $I_{VN}(*, 0)$ can only pair with $W_{VN}(0, *)$, where $*$ denotes any non-reduction index. In other word, workloads become a logical GEMM ($O_{m,n}^{P=M,Q=N} = I_{m,k}^{M=M,J=K} \cdot W_{k,n}^{K=K,N=N}$) with rank shape (M, K, N).

2) Step 2 - Tile the Workload: Tiling serves two purposes: it bounds tensors within the on-chip buffer capacity and exposes control to tuning the tile sizes for overlapping off-chip transfers with computation. For GEMM, a tile therefore represents a subproblem of shape $M_t \times K_t \times N_t$ to be executed by one or more NEST invocations.

3) Step 3 - Form VN Groups: Each NEST column is an independent mapping unit consisting of AH PEs, with each PE storing one W_{VN} under WO-S or I_{VN} under IO-S. The mapper therefore groups VNs into *VN groups*, where each VN group contains one streaming I_{VN} and up to AH W_{VN}s that can consume that same I_{VN} when mapped onto one column. A VN group is thus the smallest schedulable unit for a single column. For a GEMM tile, the number of VN groups is $M_t \cdot \left\lceil \frac{K_t}{AH} \right\rceil \cdot \left\lceil \frac{N_t}{AH} \right\rceil$. Within a VN group, all W_{VN}s share the same reduction-rank index, but the mapper may choose different non-reduction indices to co-locate in the same group. This choice defines the second mapping knob.

4) Step 4 - Combine VN Groups Across Streaming Inputs: VN groups are then merged to capture reuse of stationary W_{VN}s across multiple streamed I_{VN}s. A *combined VN group* contains all I_{VN}s that reuse the same set of up to AH W_{VN}s when executed on one NEST column. Equivalently, it specifies the full workload assigned to one column before moving to a different stationary-W_{VN} configuration.

For a GEMM tile, the number of combined VN groups is $\left\lceil \frac{K_t}{AH} \right\rceil \cdot \left\lceil \frac{N_t}{AH} \right\rceil$. All I_{VN}s and W_{VN}s in the same combined VN group share the same reduction-tile index. This step makes explicit the reuse opportunity that would otherwise be hidden across many individual VN groups.

5) Step 5 - Select Column Duplication: Since NEST has AW columns, the mapper must decide how many combined VN groups are executed concurrently and whether some of them should be duplicated across columns. At one extreme, the mapper selects AW distinct combined VN groups and assigns one to each column. At the other extreme, it may select fewer than AW groups and replicate their W_{VN}s across multiple columns, allowing different subsets of I_{VN}s from the same group to be processed in parallel. This replication factor is the third mapping knob.

For example, if a tile produces six combined VN groups and $AW = 4$, the mapper may schedule four distinct groups in the first invocation, then schedule the remaining two groups with duplication factor two in the second invocation. This choice improves column utilization when the number of available groups does not evenly match the physical column count, and it highlights why FEATHER+ benefits from reconfigurable mappings even within a single logical tile.

6) Step 6 - Search for Feasible Layouts: The three mapping knobs above generate a pool of candidate mappings. For each mapping candidate, the mapper searches for tensor layouts that realize the candidate without bank or port conflicts. The layout search is restricted to layout orders and level-0 partition factors, and each candidate is checked against the following feasibility conditions.

a) Buffer-capacity legality: The I_{VN}, W_{VN}, and O_{VN} must fit within/com streaming, stationary, and output buffers.

b) Streaming/stationary-buffer legality: A combination of `ExecuteMapping`, `SetIVNLayout`, `SetWVNLayout` must not create bank conflicts when reading I_{VN}s or W_{VN}s.

c) Output-buffer legality: The chosen `SetOVNLayout` must not introduce output-buffer port conflicts under.

Any candidate that violates above conditions is discarded.

7) Step 7 - Generate the MINISA ISA Trace: Among all feasible {mapping, layout} pairs, the mapper selects the one with minimum estimated end-to-end latency. The selected solution is then deterministically lowered into a MINISA ISA trace consisting of `ExecuteMapping`, `SetWVNLayout`, `SetIVNLayout`, and `SetOVNLayout` instructions. The resulting trace is finally evaluated by the FEATHER+ analytical performance model to estimate latency and utilization.

For multi-layer workloads, the mapper additionally enforces inter-layer layout compatibility: the output layout of layer i must match the input layout expected by layer $i + 1$. It then searches over all surviving cross-layer combinations and selects the choice with minimum overall latency.

VI. Evaluation

A. Evaluation Setup

Methodology. We use a two-level simulation framework. A cycle-accurate RTL model of FEATHER+ (supporting both MINISA and micro-instructions) validates functional correctness and timing. The RTL-verified FEATHER+ mapper with an analytical model extends the study to larger array scales.

Configuration. We use FEATHER+ with (AH, AW) sweeping $\in \{(4, 4/16/64), (8, 8/32/128), (16, 16/64/256)\}$.

TABLE IV: Workload specifications of matrix multiplication.

Workload	Matrix Multiplication of Shape $O^{P=M,Q=N}_{m,n} = I^{M=M,J=K}_{m,k} \cdot W^{K=K,N=N}_{k,n}$
FHE: BConv	$(65536\times K)\cdot(K\times N)$, $K\in[28,60]$, $N\in[72,160]$ (41 shapes)
FHE: NTT	$J=K=N\in\{1024,2048,4096\}$, $M\in\{64,128,256\}$, $M\le K/16$
ZKP: NTT	$J=K=N\in\{8192,16384,32768\}$, $M\in\{K/32,K/16\}$
GPT-oss	$M=2048$, $(J=K,N)\in\{(64,2048),(2880,4096/5120/201088),(4096,2880)\}$

TABLE V: Experimental setup for FEATHER+.

$(AH,\ AW)$	On-chip capacity (MB)	MINISA ISA Bitwidth		
	(StrB/ StaB, OB, Instr.)	`Set*VNLayout`	`E.Mapping`	`E.Streaming`
(4, 4/16/64)	(1.6, 0.8, 0.5)	42/40/38 bits	81/83/85 bits	57/51/45 bits
(8, 8/32/128)	(6.4, 3.2, 1.0)	43/41/39 bits	86/88/90 bits	58/52/46 bits
(16, 16/64/256)	(25.6, 12.8, 2.0)	44/42/40 bits	91/93/95 bits	59/53/47 bits

On-chip data SRAM scales with AH and is partitioned into streaming (40%), stationary (40%), and output (20%) buffers. A dedicated instruction buffer (0.5 MB, 1 MB, 2 MB) is served by a fixed off-chip instruction interface of 9 B/cycle. Off-chip data bandwidths are modeled as AW B/cycle for inputs/weights and $4AW$ B/cycle for outputs.

Workloads. We evaluate 50 GEMM kernels from three domains: LLM inference (GPT-OSS 20B), Homomorphic Encryption (FHE) bootstrapping [2], and Number-Theoretic Transform (NTT) kernels from both FHE [22] and Zero-Knowledge Proof (ZKP) [21]. Shapes are listed in Tab. IV.

Baselines. For each workload and array size, we compare the following two configurations under identical mappings.

- **Micro-instruction (Baseline):** Explicit, fine-grained control where every switch and PE is configured per cycle.
- **MINISA (Ours):** VN-based MINISA instructions.

Metrics. (1) *Latency*: GPU/TPU latencies are measured with Nsight/JAX-profiler on realistic traces; FEATHER+ results come from cycle-accurate RTL. We report best-latency from: CUDA kernels with tiled/strided/continguous layouts on GPU; and best sharding of (M, N) over eight tensor-cores(TPU). (2) *Utilization*: average compute utilization of the entire end-to-end execution of matrix multiplications.

B. Control Overhead Evaluation

We first quantifies and compares total instruction bytes and fetching latency for MINISA against the baseline in Fig. 12.

1) Instruction Storage Reduction: Across all 50 workloads, micro-instructions could take up-to 100× storage overhead than data. Because the micro-instruction stream must explicitly configure switch and addresses fabric. MINISA reduces such costs by a geometric mean of $2\times10^4\times$ at $(AH, AW) = (16, 256)$. The reduction grows roughly with workload sizes as more instructions are needed for bigger workloads.

2) Instruction Fetch Latency Reduction: We evaluate how instruction compression translates into end-to-end speedup (Fig. 10) for FEATHER+ with different scales.

- **Small scale (≤ 64 PEs):** At $(AH, AW) = (4, 4), (4, 16), (8, 8)$, micro-instructions consumption is small enough to be hidden by compute, making speedup as 1.
- **Large scale (>64 PEs):** As arrays grow, BIRRD instructions and buffer addresses grow by $O(AW\log_2(AW))$ and $O(D\times AW)$. Stall cycles from instruction fetch rise from **75.3%** at $(4, 64)$ to **96.9%** at $(16, 256)$ in baseline.

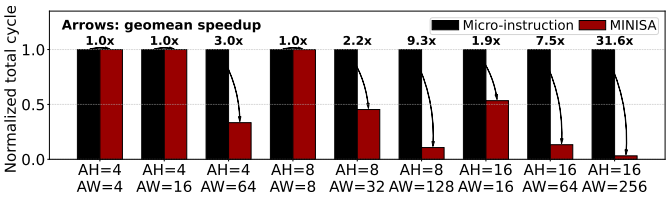

Fig. 10: End-to-end speedup and stall analysis. MINISA achieves up to 31.6× speedup as array size scales to 16×256. **Takeaway:** The performance gain stems from eliminating instruction-fetch stalls. The micro-instruction baseline becomes instruction-bound at large scales, which MINISA reduces to zero instruction-fetch stalls across all configurations.

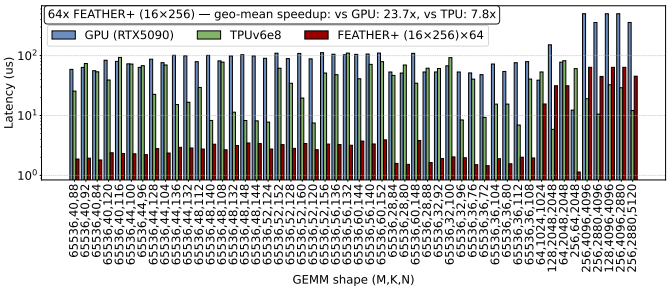

Fig. 11: Latency comparison among GPU (RTX 5090), TPUv6e-8 (256×256×8), and FEATHER+ ((16×256)×64).

Because MINISA removes fetch stalls across all tested sizes, geometric-mean speedup increases with scale: 1.9× at 16×16, 7.5× at 16×64, and up to 31.6× at 16×256. These results proves the efficacy of VN abstraction in concealing rich reconfigurations in hardware with succinct ISAs.

C. Performance Evaluation

1) Comparison against industry baselines (TPU&GPU): We compare FEATHER+ vs. RTX5090 and TPUv6e. All devices are scaled to roughly the same power budget of 575W: eight TPUv6e tensor cores and 64 instances of FEATHER+ 16×256 connected as a 8×8 mesh, as shown in Fig. 11.

FEATHER+ achieves 23.7× (vs. RTX5090) and 7.8× (vs. TPUv6e) geo-mean speedup. The key reason is granularity mismatch in less-flexible GPUs/TPUs (e.g., for INT8, TPUv6e process GEMM at minimal granularity of 8×256×256, while RTX5090 does it at 16×32×8). When GEMM shape does not divide these fixed granularity, compute will be underutilized. FEATHER+ supports matrix multiplication at the granularity of $T \times AH \times AH$ per PE column to align diverse shape of GEMM in runtime, where $T \in [1, M]$ is total number of I_{VN}s to be streamed per row.

2) Latency Breakdown: We further dissect architectural efficiency using cycle-level breakdowns of representative workloads in Fig. 13. We separate execution time into four overlapping components: *Compute*, input/weight streaming (*Load In/W*), output movement from the output buffer to the Streaming/Stationary Buffer (*Out→Stream*), and off-chip output transfer (*Store Out*).

Robustness to Irregular Shapes. The compute-utilization curve (red line in Fig. 11) remains consistently high across

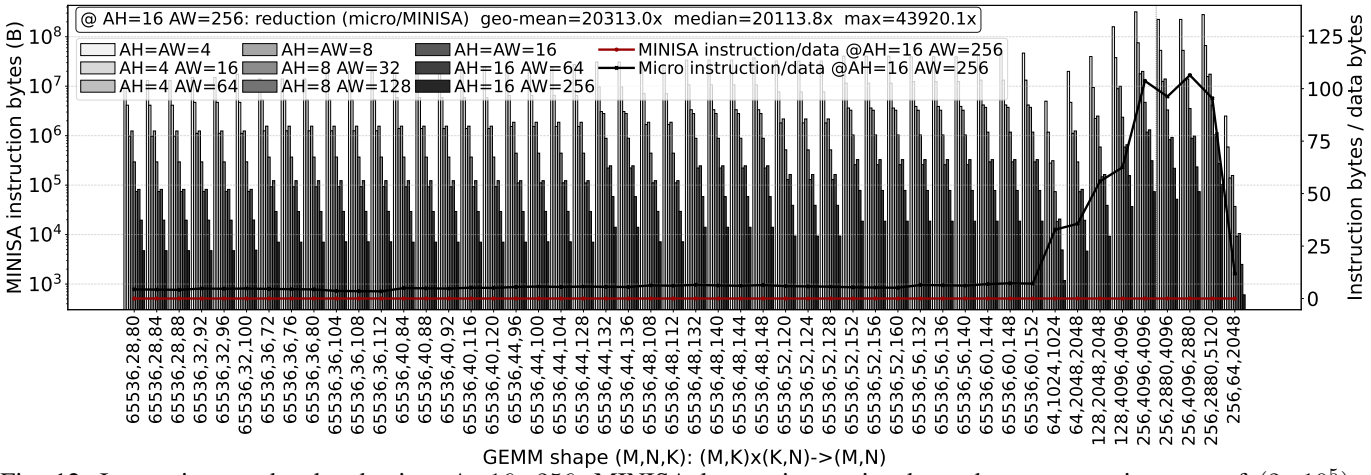

Fig. 12: Instruction-overhead reduction. At 16×256, MINISA lowers instruction bytes by a geometric mean of $(2\times10^5)\times$ versus micro-instructions (bar chart). Black and red lines show instruction-to-data ratio for baseline and MINISA, respectively. **Takeaway:** MINISA shrinks instruction volume by at most $4.4\times10^5\times$, removing instruction loading from the critical path.

a wide range of tensor shapes, indicating that FEATHER+ is largely robust to shape irregularity.

• *Irregular workloads:* For FHE and ZKP kernels with shape not dividing the compute granularity (e.g., K=10 or N=21), FEATHER+ still sustains $> 60\%$ average compute utilization. This is in sharp contrast to rigid systolic arrays with 3% compute utilization, where mismatched dimensions typically introduce padding overhead and leave a majority of PEs idle.

• *Regular workloads:* For GEMMs whose dimensions align exactly with TPUv6e execution granularity (e.g., $K, N \in \{1024, 2048\}$), all architectures can approach peak utilization. In this regime, FEATHER+ is about 30% slower due to reconfiguration overhead rather than underutilization. Even so, this overhead remains lower than that of RTX5090, demonstrating that MINISA effectively amortizes reconfiguration through a compact control path and low instruction overhead.

Takeaway: Reconfigurability allows FEATHER+ to preserve high compute utilization even for highly irregular workloads, where rigid architectures suffer from padding-induced inefficiency. MINISA further keeps the cost of this flexibility low by minimizing reconfiguration overhead.

D. Scalability Ablation Study

We analyze how performance and resources of FEATHER+ scale with array width (AW) and array height (AH).

1) Scaling AW: More Independent Parallelism: AW is the number of NEST columns. Because columns operate independently, increasing AW adds column-level parallelism without changing per-column execution granularity.

With AH=16, scaling AW from 64 to 256 delivers nearly linear scaling: FEATHER+ achieves about $4\times$ average speedup with almost unchanged compute utilization across benchmarks (Fig. 13). Thus, wider arrays translate directly into throughput rather than additional underutilization.

The hardware cost of scaling AW is also favorable. NEST and on-chip buffers grow linearly, i.e., $O(AW)$. Interconnect overhead grows faster but remains subquadratic: BIRRD scales as $O(AW \log AW)$, while the distribution network is bounded by $O(AW^2)$. Overall, scaling AW is an efficient way to increase throughput, with cost dominated by interconnect.

2) Scaling AH: Higher Parallelism with Temporal Reuse Restriction: Increasing AH grows overall MACs and parallelism but also increases workload granularity for fully utilizing one PE column to $1\times AH\times AH$ matrix multiplication.

Specifically, each PE performs an AH-element local dot product. AH determines the maximum VN size supported by each PE, and thus sets FEATHER+'s compute granularity. All PE in a column reuses the same streaming tensor, therefore scaling AH adding more temporal reuse restriction.

Full utilization of an $AH\times AW$ array requires VN_{size}=AH. When $VN_{\text{size}} < AH$, FEATHER+ activates only $VN_{\text{size}}\times AW$ PEs and uses only VN_{size} local registers per PE, skipping the remaining rows to reduce stationary tensor loading latency and pipeline latency.

This tradeoff appears in performance. With AW=64, scaling AH from 4 to 16 yields $2.6\times$–$4\times$ speedup, depending on workload size. The gain comes from larger dot products and higher intra-column parallelism, but a larger AH also raises the minimum workload granularity needed for full utilization. Hence, scaling AH improves peak throughput but makes utilization more sensitive to VN size.

Resource-wise, local storage grows quadratically, i.e., $O(AH^2)$; number of multipliers scale linearly, i.e., $O(AH)$.

Takeaway. Increasing AW scales independent parallelism. Increasing AH raises parallelism with temporal reuse requirement and increases *compute granularity*.

E. Resource Overhead Evaluation

Finally, we quantify the hardware cost of FEATHER+ architectural refinements (including all-to-all distribution) needed to support dynamic workloads. Tab. VI summarizes area results.

Compared to FEATHER [23], FEATHER+ only adds up-to 7% resources overhead because the area overhead of introduced all-to-all distribution network are amortized over distributed register and compute resources.

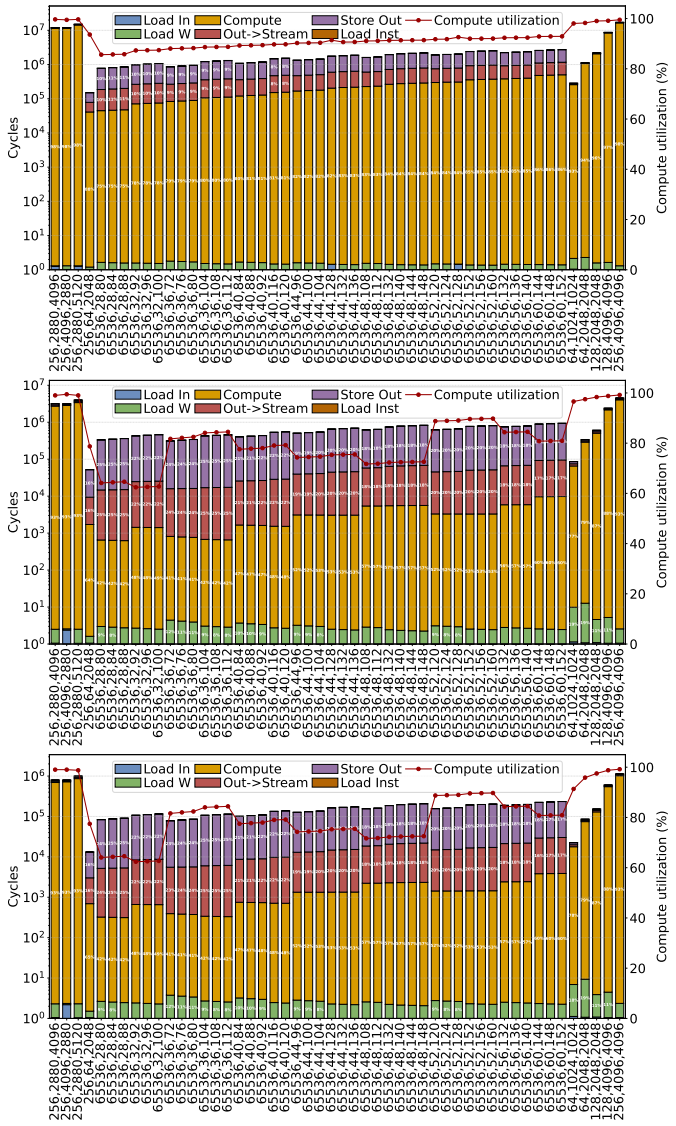

Fig. 13: Latency breakdown and compute utilization of representative workloads. **Takeaway**: FEATHER+ (4×64, upper chart), (16×64, middle chart) and (16×256, lower chart) with MINISA could successfully utilize available PEs for all irregular workload shapes, such as K=10 and $K = 2^n, n \in \mathbb{Z}^+$.

VII. Related Work

Instruction Set for AI Accelerators. Commercial AI accelerators typically employ monolithic instruction sets driving fixed functional units to maximize efficiency for standard deep learning operators. Architectures like Google's TPU [8], Meta's MTIA [4], and Amazon's Trainium [1] are designed for coarser-grained regular matrix multiplication. These rigid gran lack the fine-grained distinct programmability required to support the irregular dataflows and complex algebraic structures found in emerging workloads like Homomorphic Encryption (HE) and Zero-Knowledge Proofs (ZKP).

Reconfigurable Dataflow Accelerators. To address the diversity of tensor shapes and operators, reconfigurable architectures such as MAERI [13], SIGMA [18], and Flexagon [16] introduce flexible interconnects to support diverse dataflows [17]. Further efforts like PolyGraph [3], DSA-Gen [24], and Over-Gen [15] explore overlay generation to map varied kernels. However, these designs often decouple computation from memory layout, incurring significant data marshalling overheads. FEATHER [23] addresses this by enabling low-cost co-switching of dataflow and data layout. *Crucially, however, prior reconfigurable baselines—including FEATHER—rely on fine-grained micro-configuration.* As array sizes scale, this results in excessive control overhead and instruction-fetch stalls that bottleneck end-to-end performance. MINISA specifically targets this scalability crisis by raising the abstraction level to Virtual Neurons, eliminating redundant control traffic while preserving architectural flexibility.

AI ASICs for Cryptography (HE & ZKP). The high computational cost of cryptographic primitives has driven a bifurcation in hardware support. Prior works have demonstrated the superiority of leveraging AI ASICs for HE [22] and ZKP primitives [21] by converting high-precision modular arithmetic down to low-precision dense matrix multiplication with diverse shapes. MINISA and FEATHER+ improves the compute utilization for small matrix multiplications which are not aligned with the dimensions of hardware computation.

TABLE VI: Post-PnR area (μm^2) and power (mW) comparison FEATHER (F) vs. FEATHER+ (F+) in TSMC 28nm.

Setup	Area (μm^2)			Power (mW)		
	F	F+	Increase ↑	F	F+	Increase ↑
4×4	70598	71573	1.38%	44.59	45.34	1.67%
8×8	174370	176573	1.26%	108.97	110.49	1.39%
16×16	476174	482044	1.23%	293.47	297.72	1.45%
4×64	1259903	1352697	7.37%	854.77	915.14	7.06%
8×128	3198595	3441146	7.58%	2240.27	2350.88	4.94%

We fix the depth of all on-chip buffers to 64 and implement them as registers for PnR. In a practical deployment, larger buffers should instead use SRAM macros.

VIII. Conclusion

MINISA is the first ISA that enables runtime co-switching of dataflow and data layout with negligible instruction overhead, allowing one accelerator to efficiently support workloads with widely varying and irregular shapes. Its key insight is the Virtual Neuron (VN) abstraction, which lifts reconfiguration from individual elements to the granularity of the hardware's atomic dot-product unit. This abstraction exactly captures the reconfigurability of hardware without adding extra redundant costs, and unifies mapping and layout control in a compiler-friendly form. Coupled with the FEATHER+ architecture, MINISA makes reconfiguration tractable in practice and enables efficient matrix multiplication across diverse shapes from various domains such as FHE, ZKP and AI.

IX. Acknowledge

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. We thank Niansong Zhang, Zhiru Zhang, and reviewers for insightful feedbacks.

REFERENCES

[1] AWS. (2025) Aws nki instruction set architecture. [Online]. Available: https://awsdocs-neuron.readthedocs-hosted.com/en/latest/nki/api/nki.isa.html

[2] A. A. Badawi, J. Bates, F. Bergamaschi, D. B. Cousins, S. Erabelli, N. Genise, S. Halevi, H. Hunt, A. Kim, Y. Lee, Z. Liu, D. Micciancio, I. Quah, Y. Polyakov, S. R.V., K. Rohloff, J. Saylor, D. Suponitsky, M. Triplett, V. Vaikuntanathan, and V. Zucca, "Openfhe: Open-source fully homomorphic encryption library," Cryptology ePrint Archive, Paper 2022/915, 2022, https://eprint.iacr.org/2022/915. [Online]. Available: https://eprint.iacr.org/2022/915

[3] V. Dadu, S. Liu, and T. Nowatzki, "Polygraph: exposing the value of flexibility for graph processing accelerators," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ser. ISCA '21. IEEE Press, 2021, p. 595–608. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00053

[4] A. Firoozshahian, J. Coburn, R. Levenstein, R. Nattoji, A. Kamath, O. Wu, G. Grewal, H. Aepala, B. Jakka, B. Dreyer, A. Hutchin, U. Diril, K. Nair, E. K. Aredestani, M. Schatz, Y. Hao, R. Komuravelli, K. Ho, S. Abu Asal, J. Shajrawi, K. Quinn, N. Sreedhara, P. Kansal, W. Wei, D. Jayaraman, L. Cheng, P. Chopda, E. Wang, A. Bikumandla, A. Karthik Sengottuvel, K. Thottempudi, A. Narasimha, B. Dodds, C. Gao, J. Zhang, M. Al-Sanabani, A. Zehtabioskuie, J. Fix, H. Yu, R. Li, K. Gondkar, J. Montgomery, M. Tsai, S. Dwarakapuram, S. Desai, N. Avidan, P. Ramani, K. Narayanan, A. Mathews, S. Gopal, M. Naumov, V. Rao, K. Noru, H. Reddy, P. Venkatapuram, and A. Bjorlin, "Mtia: First generation silicon targeting meta's recommendation systems," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589348

[5] M. Gilbert, Y. N. Wu, A. Parashar, V. Sze, and J. S. Emer, "Looptree: Enabling exploration of fused-layer dataflow accelerators," in *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2023, pp. 316–318.

[6] D. Jain, M. Frigo, J. Arora, A. Pardeshi, Z. Wang, K. Patel, and C. Mendis, *TAIDL: Tensor Accelerator ISA Definition Language with Auto-generation of Scalable Test Oracles*. New York, NY, USA: Association for Computing Machinery, 2025, p. 1316–1333. [Online]. Available: https://doi.org/10.1145/3725843.3756075

[7] D. Jain, A. Pardeshi, M. Frigo, K. Patel, K. Khulbe, J. Arora, and C. Mendis, "Act: Automatically generating compiler backends from tensor accelerator isa descriptions," Oct. 2025. [Online]. Available: https://arxiv.org/abs/2510.09932

[8] N. P. Jouppi, D. Hyun Yoon, M. Ashcraft, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, "Ten lessons from three generations shaped google's tpuv4i : Industrial product," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1–14.

[9] N. P. Jouppi, G. Kurian, S. Li, P. Ma, R. Nagarajan, L. Nai, N. Patil, S. Subramanian, A. Swing, B. Towles, C. Young, X. Zhou, Z. Zhou, and D. Patterson, "Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," 2023.

[10] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. A. Patterson, "A domain-specific supercomputer for training deep neural networks," *Commun. ACM*, vol. 63, no. 7, pp. 67–78, 2020. [Online]. Available: https://doi.org/10.1145/3360307

[11] T. Krishna, H. Kwon, A. Parashar, M. Pellauer, and A. Samajdar, "Data orchestration in deep learning accelerators," 2020.

[12] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

[13] H. Kwon, A. Samajdar, and T. Krishna, "MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects," in *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[14] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design: Taxonomy, challenges, and applications," *ACM Comput. Surv.*, vol. 52, no. 6, oct 2019. [Online]. Available: https://doi.org/10.1145/3357375

[15] S. Liu, J. Weng, D. Kupsh, A. Sohrabizadeh, Z. Wang, L. Guo, J. Liu, M. Zhulin, R. Mani, L. Zhang, J. Cong, and T. Nowatzki, "Overgen: Improving fpga usability through domain-specific overlay generation," in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '22. IEEE Press, 2023, p. 35–56. [Online]. Available: https://doi.org/10.1109/MICRO56248.2022.00018

[16] F. Muñoz-Martínez, R. Garg, J. L. Abellán, M. Pellauer, M. E. Acacio, and T. Krishna, "Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing," 2023. [Online]. Available: https://arxiv.org/abs/2301.10852

[17] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[18] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[19] J. Seo, J. Tong, T. Krishna, and H. Kwon, "Exploring constrained dataflow accelerators for real-time multi-task multi-model ml workloads," in *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2025, pp. 1–11.

[20] J. Strzeszynski, J. Tong, K. Lee, N. Xiong, A. Parashar, J. S. Emer, T. Krishna, and M. Yan, "Squareloop: Explore optimal authentication block strategy for ml," in *Proceedings of the 14th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 37–45. [Online]. Available: https://doi.org/10.1145/3768725.3768732

[21] J. Tong, J. Dang, S. Langowski, T. Huang, A. Ali, J. Kun, S. Devadas, and T. Krishna, "Morph: Enabling ai asics for zero knowledge proof," in *Proceedings of the 63nd Annual ACM/IEEE Design Automation Conference*, ser. DAC '26. IEEE Press, 2026.

[22] J. Tong, T. Huang, J. Dang, L. de Castro, A. Itagi, A. Golder, A. Ali, J. Jiang, J. Kun, Arvind, G. E. Suh, and T. Krishna, "Leveraging asic ai chips for homomorphic encryption," in *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, ser. HPCA'26, Australia, 2026.

[23] J. Tong, A. Itagi, P. Chatarasi, and T. Krishna, "Feather: A reconfigurable accelerator with data reordering support for low-cost on-chip dataflow switching," in *Proceedings of the 51th Annual International Symposium on Computer Architecture*, ser. ISCA '24. Argentina: Association for Computing Machinery, 2024.

[24] J. Weng, S. Liu, V. Dadu, Z. Wang, P. Shah, and T. Nowatzki, "Dsagen: Synthesizing programmable spatial accelerators," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 268–281.

APPENDIX

A. Abstract

This artifact contains the MINISA toolchain for the FEATHER+ reconfigurable accelerator. It includes: (1) the FEATHER+ mapper that searches the optimal mapping, layout choice for processing GEMM workloads under $AH \times AW$-FEATHAER+, (2) a cycle-accurate analytical performance model with a 5-engine asynchronous execution simulator, (3) a GUI to illustrate how FEATHER+ works with cycle-by-cycle animation (4) instruction compression analysis comparing MINISA against explicit micro-configuration, and (5) latency comparison against GPU (NVIDIA RTX 5090) and TPU (Google TPU v6e-8) baselines. The artifact reproduces all evaluation figures: instruction reduction ratios, speedup over micro-instruction, latency breakdown with compute utilization, and FEATHER+ vs. GPU/TPU latency comparison.

TABLE VII: FEATHER+ Mapper Search Knobs for a Symbolic GEMM $O^{P=M,Q=N}_{m,n} = I^{M=M,J=K}_{m,k} \cdot W^{K=K,N=N}_{k,n}$.

Category	Knob	Symbol / Choice Set	Meaning
Dataflow choice $\mathrm{WO}-\mathrm{S}, \mathrm{IO}-\mathrm{S}$	$\mathrm{WO}-\mathrm{S}$	$(M_s, K_s, N_s) = (M, K, N)$	Search is performed on the original GEMM.
	$\mathrm{IO}-\mathrm{S}$	$(M_s, K_s, N_s) = (N, K, M)$	Swaps the outer dimensions before search.
Tiling choice (Decide Partition factors)	Output-row Rank	$M_t \in \{AH, 2AH, 4AH, \ldots, M_s\}$	Number of output rows processed per tile.
	Reduction Rank	$K_t \in \{AH, 2AH, 4AH, \ldots, K_s\}$	Reduction depth processed per tile.
	Output-column Rank	$N_t \in \{1, 2, 4, \ldots, N_s\}$	Number of output columns processed per tile.
Layout choice	weights order	$o_{Wvn} \in \{0, 1, 2, 3, 4, 5\}$	Tab. III permutation used for W_{VN} layout.
	inputs order	$o_{Ivn} \in \{0, 1, 2, 3, 4, 5\}$	Tab. III permutation used for I_{VN} layout.
	outputs order	$o_{Ovn} \in \{0, 1, 2, 3, 4, 5\}$	Tab. III permutation used for O_{VN} layout.
	Duplication factor	$d \in \{1, 2, \ldots, \lfloor AW/n_{\mathrm{col}} \rfloor\}$	Replicates each column type to many columns.
Combined VN group CG(K_g, N_t)	$W_{VN}(k_g, n_t)$ inter-column n_t stride	$\in$ {block, strided}	Block: $s_r = 1$, $s_c = AH$. Strided: $s_r = n_{\mathrm{col}}$, $s_c = 1$. Changes concurrent WVN bank access pattern. Only differs when $n_{\mathrm{col}} > 1$.
	$I_{VN}(m_t, k_g)$ inter-column m_t stride	$\in$ {interleaved, consecutive}	Interleaved: $s_m = n_{\mathrm{rep}}$ (strided rows). Consecutive: $s_m = 1$ (contiguous blocks). Changes concurrent IVN bank access pattern. Only differs when $n_{\mathrm{rep}} > 1$.

B. Artifact check-list (meta-information)

- **Algorithm:** MINISA ISA compilation for matrix multiplication on reconfigurable accelerators; Virtual Neuron (VN) abstraction with brute-force tiling search over choies in Tab. VII.
- **Program:** Python scripts: `evaluate.py` (mapping-layout cosearch), `analyze.py` (analysis + GPU/TPU comparison), `figure_drawer/*.py` (plot generation)
- **Compilation pipeline:** Workload → VN groups → combined VN groups → duplication → MINISA trace → latency.
- **Dataset:** 50 GEMM workloads from three domains: Fully Homomorphic Encryption (FHE), Zero-Knowledge Proofs (ZKP), and ChatGPT-class open-source LLM inference; pre-collected GPU/TPU baseline latency measurements
- **Run-time environment:** Linux (Ubuntu 22.04); Python 3.10+; Conda environment with numpy, pandas, matplotlib, pyyaml
- **Hardware:** Any x86-64 or ARM64 machine with ≥16 GB RAM. No GPU/TPU required (baseline data is pre-collected). We also provide GPU and TPU scripts.
- **Run-time state:** Deterministic analytical model (no random).
- **Execution:** Supporting multiple-threading (`--jobs <n>`)
- **Metrics:** Compute utilization (%), cycle breakdown (load-in, load-weight, compute, out-to-stream, store-out, instruction fetch), instruction bytes (MINISA vs. micro-instruction), instruction reduction ratio, latency (μs)
- **Output:** 18 PDF figures, 6 CSV data files (benchmark summary, instruction comparison, GPU/TPU comparison, utilization/reduction/memory summaries)
- **Experiments:** 50 workloads × 9 array configurations = 450 evaluation points (Stage 1); GPU/TPU comparison across all workloads (Stage 2); 21 publication-quality figures (Stage 3)
- **How much disk space required (approximately)?:** ∼500 MB
- **How much time is needed to prepare workflow (approximately)?:** 5–10 minutes (conda environment setup)
- **How much time is needed to complete experiments (approximately)?:** 3–10 hours for full evaluation (Stage 1); <2 minutes for analysis and plots (Stages 2–3)
- **Publicly?:** https://github.com/maeri-project/FEATHER
- **Archived?:** https://doi.org/10.5281/zenodo.18921947

C. Description

The MINISA framework runs on any device supporting Python. Please refer to the README.md in the open-sourced Github repository or archived artifacts DOI to install the MINISA framework, and run (1) mapping-layout cosearching, (2) layout-constrainted mapping search, (3) comparison against GPU / TPU. We provide 50 GEMM workloads as benchmark suites, as discussed in §VI-A. We also provide pre-collected results on GPUs and TPUs.

D. Evaluation and expected results

(1) the FEATHER+ mapper that searches the optimal mapping, layout choice for processing GEMM/convolution workloads under $AH \times AW$-FEATHAER+, (2) a cycle-accurate analytical performance model with a 5-engine asynchronous execution simulator, (3) a GUI to illustrate how FEATHER+ works with cycle-by-cycle animation (4) instruction compression analysis comparing MINISA against explicit micro-configuration, and (5) latency comparison against GPU (NVIDIA RTX 5090) and TPU (Google TPU v6e-8) baselines.

(1) (mapping, layout) co-search for all 50 workloads under 9 configs: `python -m minisa evaluate`

(2) FEATHER+ GUI: `python -m minisa gui`

(3) vs. TPU/GPU, `python -m minisa analyze`

(4) Instruction overhead comparison (MINISA vs. micro-instruction baseline): `python -m minisa compare`

(5) (mapping, layout) co-search for GEMM/conv. under $AH \times AW$-FEATHAER+: `python -m minisa search`

(6) layout-constrainted mapping search (GEMM/conv.) under $AH \times AW$-FEATHAER+: `python -m minisa search --layout-constrained`

(7) plotting figures, `python -m minisa plot`

E. Experiment customization

- **Array configurations:** Change `--ah` and `--aw` to evaluate different FEATHER+ sizes. Use `--aw same` for square arrays (AW=AH).
- **Workloads:** Supply a different CSV with different workloads. `--csv` with columns `category`, `name`, `M`, `K`, `N`.

F. Mapper Design Space

We summarize the mapper design space in Tab. VII and apply heuristics to prune unpromising candidates early, accelerating the search. For 50 workloads with $AH{=}AW{=}16$, joint mapping and layout search completes in 17 minutes on an M5 Pro powered MacBook Pro using 16 parallel jobs.