A Comparative Analysis of Loosely and Tightly Coupled Accelerator Architectures for Machine Learning

Amin Firoozshahian, *Rain AI, San Francisco, CA, 94110, USA*

Joel Coburn, Ajit Punj, Aravind Sukumaran Rajam, Colby Boyer, Rakesh Nattoji, and Mahima Bathla, *Meta Platforms Inc., Menlo Park, CA, 94025, USA*

Bob Dreyer, *Accel Dynamics, Palo Alto, CA, 94301, USA*

Sujith Srinivasan, Harshitha Pilla, Michael Rotzin, Surendra Rajupalem, K. Rajesh Jagannath, Krishna Noru, and Harikrishna Reddy, *Meta Platforms Inc., Menlo Park, CA, 94025, USA*

Chris Yang, Charlie Hong-Men Su, and Charlie Cheng, *Andes Technology, Hsinchu City, 300042, Taiwan*

With the rapid innovation of artificial intelligence (AI)/machine learning (ML) workloads, accelerators have become an important part of the computing resources in data centers. Although characteristics such as performance, power, and efficiency are critical, developer velocity, which dictates how quickly a new workload can be deployed on an accelerator, is equally important. The accelerator architecture and the properties that it exposes to the higher-level programming model play a key role in determining the programming model and economic viability of the accelerators. We compare two classes of AI/ML accelerator architectures: an efficient, loosely coupled accelerator already implemented in silicon, and a tightly coupled accelerator with a traditional and user-friendly programming model. Our comparison shows that the performance of the design depends on the number of hardware resources and not on how they are integrated, concluding that it is possible to architect accelerators in a much simpler and more compiler-friendly manner while still maintaining the benefits of offload computing.

Hardware acceleration is important for artificial intelligence (AI)/machine learning (ML) workloads as they demand continued improvements in performance and efficiency to meet the escalating demands of workloads and applications. These accelerators, the most prevalent being GPUs, provide high peak floating-point operations per second and memory bandwidth while supporting a large set of general matrix multiplication and other mathematical operators to accommodate a variety of neural network architectures.

Digital Object Identifier 10.1109/MM.2025.3603837
Date of publication 2 September 2025; date of current version 31 October 2025.

Model trends in domains such as generative AI [e.g., large language models (LLMs) and latent diffusion models] and ranking and recommendations (e.g., deep and hierarchical ensemble network and hierarchical sequential transduction unit) are driving innovation in accelerator design. These models require flexibility from accelerators to handle new operators, memory access patterns, data types, and higher compute capacity. Model designers iterate by inventing and tweaking important operators such as normalization layers and activation functions. It becomes an ongoing challenge to efficiently map these operators to hardware. Therefore, the efficiency and suitability of accelerators should not only be measured merely by their performance, power, and area; their flexibility and programmability should be considered equally important, if not more so. The speed with which AI/ML

workloads can be deployed on a particular accelerator determines its practicality and usefulness.

The accelerator architecture and the programming model affect the software stack, developer ecosystem, and efficient mapping of operators to hardware, and therefore directly impact performance and developer efficiency. The choice of accelerator architecture directly influences these factors, both for manual developers and for building compiler support for the accelerator. In this article, we compare two classes of these machines based on the lessons learned from constructing two generations of production AI/ML accelerators.

MOTIVATION

GPUs have been the preferred accelerators for AI workloads, but new architectures that claim better performance or efficiency have emerged. Google's TPU,[1,2,3,4] Amazon's Inferentia,[5] and Meta's Training and Inference Accelerator (MTIA)[6,7,8] are a few such examples. We recognize two distinct architectural schemes in these accelerators. In one scheme, special-purpose hardware blocks perform the bulk of the computation, controlled by processor cores. These blocks can be systolic arrays, matrix multipliers, single-instruction, multiple data (SIMD) engines, or other blocks that perform a specific compute task or data movement efficiently. The programmable cores orchestrate the operations and data movement between the special-purpose hardware blocks. In this text, we refer to these accelerators as loosely coupled, or offload architectures. Meta's MTIA and Google's TPU are examples of such architectures.

The second category of accelerator architectures relies on programmable cores as the main computing engine, while augmenting them with AI/ML specific functional units for acceleration. The program on the processors directly performs the computation to produce results. We refer to this type of accelerator architecture as *processor-based*, *tightly coupled*, or *integrated* architecture. Nvidia's tensor cores or AMD's matrix core engines are examples of augmenting programmable cores with tightly coupled engines.

There are major differences between these architectures, as studied in the literature. Tightly coupled architectures are more constrained by the instruction set architecture (ISA) definition and microarchitectural implementation, whereas loosely coupled architectures have more freedom to construct hardware resources and control them through processors. Therefore, these architectures are not limited to what a given ISA provides and hence can implement custom functions independently. Complex ML operations such as matrix multiplication and nonlinear functions typically take many more cycles than standard ISA instructions. In offload architectures, accelerator blocks are loosely coupled with the processor's pipeline and hence can asynchronously perform these operations without causing stalls or delaying the rest of the instructions. In contrast, in tightly coupled accelerators, long latency instructions might create dependencies or fill up resources, negatively impacting the execution of subsequent instructions. Hence, it is critical to create enough parallelism within the processor to allow running such long latency instructions in the background.

Loosely coupled architectures can operate directly on memory, unlike load/store reduced-instruction-set computing or computer (RISC) architectures that require data be loaded into registers. The AI/ML workloads that perform "streaming" operations tend to have low arithmetic intensity on large tensors without any temporal locality. Performing arithmetic operations directly on memory therefore saves power by avoiding unnecessary register read/write and removing fetch, decode, and issue overheads.

In loosely coupled architectures, compute blocks can be constructed with the sufficient memory bandwidth needed to feed all their operands. In contrast, the operand bandwidth of processor-based architectures is limited by the bandwidth to the memory hierarchy. Processors provide enough register bandwidth to feed the arithmetic logic units (ALUs) but typically limit memory accesses to one or two per cycle. Hence, a tightly coupled accelerator needs to have other means to access memory at high bandwidth.

Finally, in processor-based architectures, instruction fetch, decode, issue, and other management operations are known sources of inefficiency.[9] But offload engines in loosely coupled architectures perform a large amount of work every time that they are activated, amortizing operation initiation and tracking overheads. Even with the SIMD paradigm and register grouping in modern vector architectures, offload engines can still better amortize overheads because they are not subject to ISA limitations such as vector length.

However, loosely coupled architectures are more challenging to program because they define their own semantics for execution, memory access, and synchronization. The programmer is responsible for the orchestration and communication between offload blocks via the control code that follows the invented semantics, which is often complicated and

nonintuitive. Further, what is being computed tends to be a black box that cannot be targeted by the compiler for traditional optimizations. On the other hand, tightly coupled architectures, which resemble more traditional CPU/GPU-like systems, can leverage modern compiler optimizations for code generation, scheduling, and even auto-vectorization.

Inspired by building two generations of our MTIA architecture and experiencing challenges with its programming model, this work tries to determine whether one can construct a tightly coupled accelerator that provides an intuitive programming model while offering the same efficiency and performance benefits as loosely coupled ones. To compare the two architectures, we systematically decompose a loosely coupled accelerator and incrementally augment a tightly coupled design. We find that RISC architectures need to adopt more complex, complex instruction set computer (CISC)-like features to close the performance gap with offload architectures.

TWO ARCHITECTURES

RISC-V architecture is a common building block in many recent AI/ML accelerators.[6,10,11] In our work, we use a modern, extensible RISC-V processor core from Andes Technology as the basis for the two architectures. Andes processors use a customization framework called *ACE* (*Andes Custom Extensions*).[12] Although several vendors permit different levels of customization to their processor offerings, ACE provides a comprehensive environment that allows the user to add a set of custom features to a baseline processor. The framework seamlessly ties the user-defined custom hardware to the main processor pipeline. In addition to custom instructions, the framework allows the definition of a rich set of features such as custom registers, custom ports, and custom exceptions. It also provisions the addition of superscalar pipelines for executing the custom instructions in parallel, and support for custom RISC-V vector instructions, as well as hardware loops.

The ACE framework is used for implementing both accelerator architectures in this work. The loosely coupled architecture is MTIA,[6,7,8] where a set of fixed-function blocks are coupled with two processor cores to form the processing elements (PEs). In the tightly coupled architecture, the functionality of these blocks is integrated into the processor core itself. The remainder of the architecture (memory subsystem and interconnect) is assumed to be the same for the two accelerators.

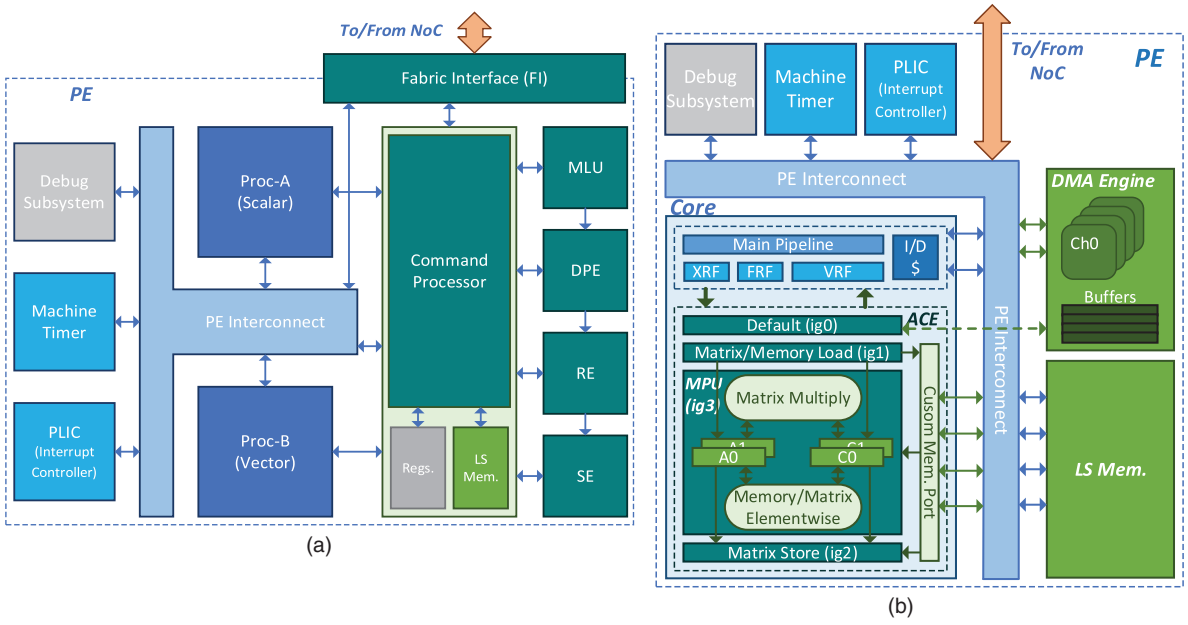

FIGURE 1. PE architecture in both accelerators. (a) Loosely coupled[6] and (b) tightly coupled. MLU: memory layout unit; DPE: dot product engine; mem.: memory; FI: fabric interface; NoC: network on chip; PLIC: platform level interrupt controller; LS: local scratchpad; Regs.: registers; XRF: auxiliary register file/general purpose register file; FRF: floating-point register file; VRF: vector register file.

TABLE 1. Custom instructions defined in both architectures.

Loosely coupled architecture			Tightly coupled architecture		
Instruction	**Unit**	**Description**	**Instruction Family**	**Pipe**	**Description**
DMA_IN	FI	Copy data from external memory to the CB	Matrix–vector move	ig0	Move values between the vector and matrix registers, both row- and columnwise
DMA_OUT	FI	Copy data from the CB to external memory	Matrix–scalar move	ig0	Move values between scalar and matrix registers
MATMUL	DPE	Perform matrix multiply on data from two CBs and accumulate the result in a CREG	Matrix load	ig1	Load values from memory to a matrix register (unit-stride, strided, and transposed)
LOAD_CREG	RE	Load values from the CB to CREG (accumulator)	Matrix store	ig2	Store values from a matrix register to memory (unit-stride, strided, and transposed)
REDUCE	RE	Send partial results from one PE to another	Matrix–matrix multiply	ig3	Multiply a matrix register with a second matrix stored in memory
FX	SE	Perform nonlinear operation on data from the CB	Matrix–vector multiply	ig3	Multiply a matrix register with a vector register
ELEMENTWISE	SE	Perform various elementwise operations on data from two CBs	Matrix–matrix elementwise	ig3	Perform arithmetic operation on elements of matrix registers
TRANSPOSE	MLU	Transpose data from one CB to another	Matrix–vector elementwise	ig3	Perform arithmetic operation on elements of a row of a matrix register and a vector register
ECOPY	MLU	Copy data from one CB to another	Memory elementwise	ig1	Perform arithmetic operations directly on vectors stored in memory
			Nonlinearity (reg)	Vec.	Perform nonlinear operations on data stored in vector registers
			Nonlinearity (mem)	ig1	Perform nonlinear operations on vectors stored in memory
			DMA communication	ig0	Transmit commands and status between core and DMA engine

CB: circular buffer; CREG: C-registers (accumulators).

Loosely Coupled Accelerator

In the loosely coupled architecture's PE [Figure 1(a)], processors are extended with custom interfaces to issue commands to the fixed-function blocks. These interfaces are driven using custom instructions, as described in Firoozshahian et al.[6] Table 1 lists custom instructions in this architecture and how they are mapped to the fixed-function blocks, while Table 2 lists the hardware resources in the fixed-function blocks in the PE. Custom instructions are retired after delivery to the blocks outside of the processor, allowing the core to execute subsequent instructions while fixed-function units execute the issued commands. Most of the fixed-function blocks in the PE operate directly on local memory using circular buffers (CBs),[6] except for the reduction engine (RE) block, which uses accumulator registers. Processor cores can also access CBs using custom instructions and interfaces.

Tightly Coupled Accelerator

The tightly coupled architecture [Figure 1(b)] starts from a standard RISC-V processor with vector extension and incrementally integrates all the functionality of the fixed-function blocks of the loosely coupled architecture into the processor core. It adds custom instructions to perform the same operations that are defined in the loosely coupled architecture, such as matrix multiplication, nonlinear function calculation, and so on, however, it defines these custom instructions such that they operate directly on memory (by

TABLE 2. Hardware resources in both PEs.

Loosely Coupled Architecture		Tightly Coupled Architecture	
Block	**Description**	**Block**	**Description**
Processor A	NX27V+, single issue, five stage, RV64IMCF	Processor	NX45V, dual issue, eight stage, RV64IMACFDV,
Processor B	NX27V+, single issue, five stage, RV64IMCFV		VLEN=512, DELN=512
	VLEN=512, DLEN=512		4 × 64 B LS memory ports (read and write)
Caches	32 KB I$, 32 KB D$, 32 B cache line	Caches	32 KB I$, 32 KB D$, 64 B cache line
LS memory	384 KB, dual port	DMA engine	Four channels, 32 × 64 B line buffers
DPE	Multiplier: Array of 32 (rows) × 32 (columns) of FP16 multipliers	LS memory	512 KB, dual port
	Memory ports: 6 × 32 B LS memory ports (read)	MPU	Multiplier: Array of 32 (rows) × 32(columns) FP16 multipliers
	Cache A: Two 32 × 32 B cache blocks for operand A		Elementwise: 16 lanes of FP32/32 lanes of FP16
	Cache B: 136-line cache of 32 B for operand B		A-Registers (A0/A1): Two 32 × 64 B
RE	Accumulators: Four 32 (rows) × 64 (columns) INT32/FP32 (CREGs)		C-Registers (C0/C1): Two 32 × 256 B (64 columns of INT32/FP32) accumulators
	Memory ports: 4 × 32 B LS memory ports (read and write)		
SE	ALUs: 32 lanes of FP16 elementwise or nonlinear operation		
	Memory ports: 4 × 32 B LS memory ports (read and write)		
MLU	Storage: 64 × 64 B internal storage for data copy and reshape		
	Memory ports: 4 × 32 B LS memory ports (read and write)		

DPE: dot product engine; RE: reduction engine; MLU: memory layout unit.

accepting memory addresses as source or destination operands). The addresses are passed to the execution units via general-purpose registers.

When necessary, the architecture also defines custom registers to specify parameters such as active regions of matrices, matrix element width, or custom vector length (for instructions that operate on memory). To achieve the aforementioned goals, the PE is updated with the following changes:

The functionality of the memory layout unit (MLU), dot product engine (DPE), and RE blocks are combined into a single matrix processing unit (MPU) integrated in the core and are invoked by the following set of custom matrix instructions:

- The accumulator registers from the RE block are carried over as part of the MPU. However, instead of four accumulators, this PE only uses two accumulators to save area and achieve similar performance. These accumulators are called *C-registers*.
- The caches in the DPE block are carried over into the MPU block but are exposed as architectural registers; the MPU defines two general-propose matrix registers for all matrix instructions, referred to as *A-registers*.
- The functionality of the SE block is split between the main vector pipeline and the MPU block. The vector pipe is augmented with custom vector instructions that perform operations such as nonlinear functions on vector registers, while the MPU block performs vector operations directly on memory using memory operands. Hardware resources such as lookup tables for performing nonlinear operations are shared between the MPU block and the custom vector instructions.
- This architecture adds four 64 B custom advanced extensible interface (AXI) memory ports to access memory operands. The read and write channels of the AXI interfaces operate independently and via same or separate custom instructions.

- This PE replaces one of the cores in the loosely coupled PE (Proc-A) with a multichannel direct memory access (DMI) engine to offload the data movement between local and main memories, with the remaining core upgraded to dual issue.
- This architecture also extends the processor with custom interfaces for tight integration with the DMA engine and low-latency job descriptor submission as well as status inspection (dotted arrow in [Figure 1(b)].

To maximize parallelism and performance, custom instructions in this architecture are mapped to four different instruction groups each with a dedicated custom pipeline. The mapping of the custom instructions to instruction groups is based on the resources used by the custom instructions to prevent structural hazards. The ACE framework generates the control logic necessary to properly enforce interlocks and prevent data hazards. The framework also takes care of arbitration, in case of multiple pipelines accessing the same hardware resources (e.g. memory ports).

The default pipeline (ig0) is used primarily for executing any register–register move instruction. The load pipeline (ig1) executes instructions that use memory ports 0 and 1 for reading data from local memory. The store pipeline (ig2) executes instructions that use memory ports for writing data back to the local memory. The matrix pipeline (ig3) uses memory ports 2 and 3 for reading local memory and performing a matrix multiply operation while using one of the matrix registers as the first operand for a matrix multiply operation. Table 1 lists custom instructions and their corresponding pipelines, and Table 2 lists the hardware resources added to the processor in this architecture.

PROGRAMMABILITY AND CODE GENERATION

The ACE framework generates the collaterals necessary to introduce custom resources (registers, instructions, and so on) to the compiler; therefore, no special runtime is required for leveraging these resources. The main difference between the two architectures, however, lays in the semantics of custom instruction execution. In the loosely coupled PE, custom instructions are used to issue commands to the external functional units. They retire immediately after the command is delivered and before the actual computation is started. Hence, the core has no knowledge of the status of computation as it is carried out in parallel and asynchronous to the core. For correct sequencing and interlocking of these instructions when dependencies exist, either core needs to poll the status of external fixed-function units to ensure proper timing, or additional hardware must be put in place (outside of the processor) to take care of such dependencies.

On the other hand, in the tightly coupled PE, custom instructions are executed in the core's custom pipelines, which are synchronous with the core's main pipeline. Therefore, the status and progress of all such instructions is tracked and observed by the core's scoreboard. This means that the processor inherently takes care of all proper dependency checking and interlocking between dependent instructions, relieving the software from this complicated task and providing a standard execution model.

When it comes to code generation, both architectures can be supported with the low level virtual machine (LLVM)-based compiler tool chain. The LLVM provides RISC-V support and low-level optimizations such as register allocation, inlining, and code generation. However, the loosely coupled architecture cannot rely on the LLVM to operate the fixed-function blocks because the fixed-function blocks are black boxes with unknown side effects and execution timing, hence, the compiler cannot reason about them. By contrast, the tightly coupled architecture integrates the fixed-function units into the back end as processor's functional units, making them a part of the exposed compute ISA. Therefore, the tightly coupled architecture can leverage LLVM support for managing registers, and tracking and respecting dependencies, optimizations, and transformations for RISC-V scalar and vector codes that interleave with custom instructions. The LLVM back end can be easily extended to support longer vector lengths, extra registers, and memory operands as these are familiar mechanisms that exist in today's architectures and compilers.

Note that higher-level languages such as Triton can be used with a front-end compiler, which generates code for the LLVM's back end. This can hide and automate the generation of the necessary boilerplate code to use the custom resources of each architecture. Each architecture can readily integrate into a PyTorch-based environment, as shown in previous work,[6,8] where PyTorch ATen operations and custom operators can be implemented manually in C++ or Triton.

EVALUATION METHODOLOGY

Evaluation Environment

We use two separate evaluation environments for the two architectures: the loosely coupled architecture's measurements are based on running kernels on the MTIAv2 silicon.[8] The tightly coupled architecture's

measurements are based on a cycle-accurate PE simulation environment built around the processor model supplied by the vendor. It accurately models all scalar and vector instructions and is extended to correctly model all custom resources, including the associated latency and throughput. The PE environment is based on SystemC and uses TLM2.0 sockets for connectivity to the local and main memory as well as peripherals such as the DMA engine.

At the high level, the programming model of both architectures relies on bringing in data from main memory into local scratchpad (LS) and processing it by the processor and fixed-function units. The interconnect and memory subsystem external to the PE is hence assumed to be the same for both architectures. In our study, which focuses on the performance and programmability of the PE, we assume that the data are already transferred and stored in the local memory and are being processed by the compute resources within the PE.

Benchmarks

For evaluation, we use a set of microbenchmarks for operators that commonly occur in deep learning recommendation models.[6] These microbenchmarks utilize all the custom functionality added to the PE, including matrix multiplication, nonlinear operations, elementwise operations, data copying, and reshaping. The input sizes are chosen to maximize the usage of available local memory. Table 3 lists these operators along with their data types and input sizes.

Measurements

Our evaluation uses the following three metrics:

1) *Cycles*: The number of cycles that are needed to run the main computation loop of the benchmark, as measured by the processors.
2) *Instructions*: The number of dynamically executed (retired) instructions in the main loop of the benchmark, as measured by the processors. This indicates the overheads that are associated with executing the compute task, in terms of instruction fetch, issue, tracking, and retirement.
3) *Throughput*: Calculated as the total size of the input tensors plus output tensors, divided by the number of execution cycles. This is the measure of the benchmark's eventual performance.

We perform two studies in our evaluation: the first uses the tightly coupled architecture's evaluation environment and is focused on understanding the benefits of adding custom features. We start by measuring the performance of the scalar version of a microbenchmark, then move to vectorizing it, using standard vector instructions. We measure the effect of using larger vectors by increasing the vector register grouping length multiplier (LMUL). Then, we use more complex custom instructions (vector, matrix, or both), utilizing different custom operations, memory operands, or other enhanced features.

Our second study provides an end-to-end comparison of the performance of the microbenchmarks executed on the two PEs. In the loosely coupled architecture, the benchmark is coded to use the fixed-function unit(s) that are most suitable for its computation. We compare this result with the results of the tightly coupled architecture where the microbenchmark uses the equivalent or similar custom instructions.

RESULTS

Quantifying Customization Benefits

Our first study looks at the improvements in each microbenchmark as it leverages enhanced features of the customized hardware. Figure 2 shows performance in terms of execution cycles (bars and left *y*-axis) and the benchmark's overall throughput (lines and right *y*-axis), while Figure 3 shows the total number of retired instructions. Table 4 compares the achieved speedup over a standard vectorized kernel.

Elementwise Add

We implement five different variations of this kernel, increasing the vector length if possible. These include scalar, vectorized, vectorized and software pipelined,

TABLE 3. Microbenchmarks used for studies.

Operator	In	Out	Dimension
Elementwise (add)	FP32	FP32	32 K 1-D tensor
Exp (e^x)	FP16	FP32	32 K 1-D tensor
Sigmoid	FP16	FP32	32 K 1-D tensor
Transpose	FP16	FP16	256 × 128 2-D tensor
Concatenation	FP16	FP16	A: 128 × 192 B: 128 × 256
FC	INT8	FP32	A: 128 × 256 B: 256 × 128
Dequantize	INT8	FP16	512 × 128 2-D tensor
EB	FP16	FP32	56 × 512 embedding table with 32 bags
Layer Norm	FP32	FP32	128 × 256 2-D tensor

FC: fully connected; EB: embedding bag.

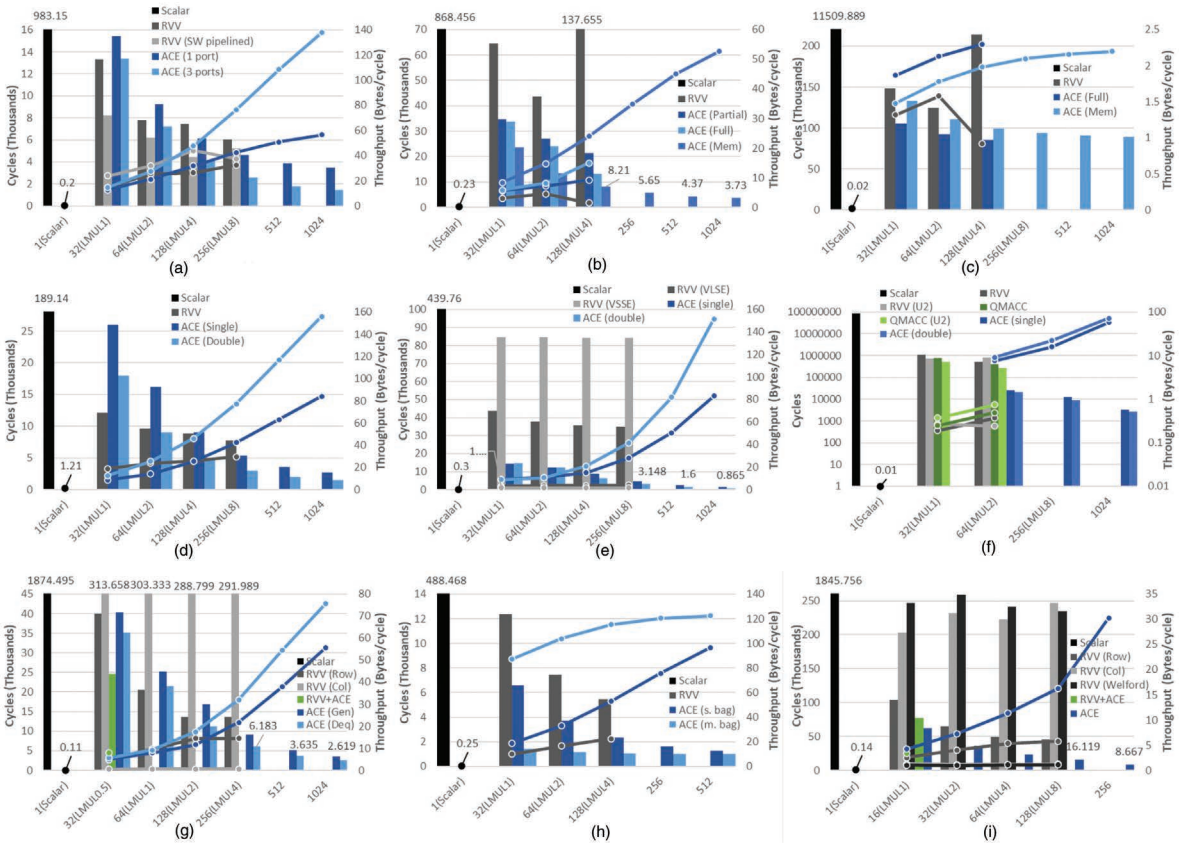

FIGURE 2. Execution time (cycles) and throughput of different microbenchmarks. (a) Elementwise add. (b) Nonlinearity: e^x. (c) Nonlinearity: sigmoid. (d) Concatenation. (e) Transpose. (f) Fully connected *(note: log scale)*. (g) Dequantize. (h) Embedding bag. (i) Layer norm. Col: column-wise; Gen: generic instructions; Deq: dequantize instruction; s. bag: single bag; m. bag: multiple bags.

custom using a single-memory port (ACE1), and custom using three memory ports (ACE3). The latter versions use custom instructions that implement a hardware loop and use memory operands for accessing elements directly in local memory. The vector length can go up to 1024, larger than the standard vector length.

The scalar version uses a quarter of the 64-bit data path (FP16 elements) and is hence very inefficient. Vectorization and longer vector lengths reduce the overheads and increase performance until the vector length and size of the vector register file becomes a limiting factor (fills/spills are inserted after a certain vector length). These limitations do not apply to the ACE1 version; this version further reduces the number of instructions as memory access and compute are specified by a single instruction using memory operands. The ACE3 version provides the most benefit as it feeds the ALUs with adequate memory bandwidth—one memory port per operand. At 256 elements, ACE1 performs 1.13× and ACE3 performs 2.03× better than the vectorized software-pipeline version.

Nonlinearity (e^x)

This kernel is a widening operation that approximates the e^x function for FP16 elements. In addition to scalar and vector implementations (which are based on the standard math library), there are three custom versions. ACE-partial uses custom instructions to look up coefficients in a table and performs cubic interpolation using vector operations. ACE-full combines coefficient lookup and interpolation into a single custom instruction. It uses more hardware resources and has longer latency. ACE-mem is the same as ACE-full but directly operates on memory by using memory operands. We observe the same trends as the previous kernel, however, ACE-partial hits the register pressure sooner as there are a lot of intermediate results passed between instructions.

Nonlinearity (Sigmoid)

This kernel calculates sigmoid using e^x as the building block. The division operation makes the kernel's latency much longer. The ACE-full version computes

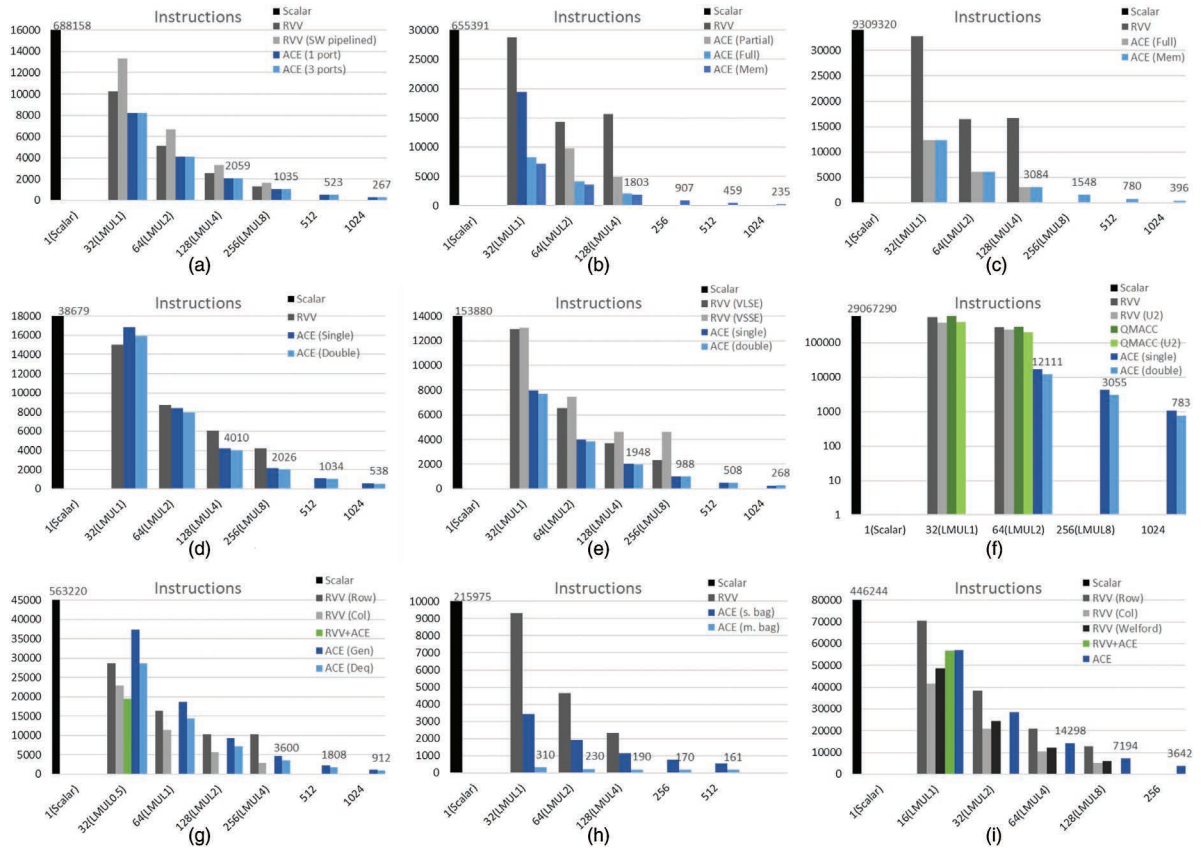

FIGURE 3. Retired instructions for each microbenchmark. (a) Elementwise add. (b) Nonlinearity: e^x. (c) Nonlinearity: sigmoid. (d) Concatenation. (e) Transpose. (f) Fully connected (*note: log scale*). (g) Dequantize. (h) Embedding bag. (i) Layer norm.

e^x in the registers and uses vector instructions for the rest of the computation. The ACE-mem version uses custom negate, subtract, and divide instructions that operate directly on long vectors in memory, where negate and subtract are pipelined but division is not. Performing operations in the registers is 1.26× better due to reuse of intermediate values, but the same vector length and vector register file-size limitations apply. Because in this case compute latency is longer than memory access, the ACE-mem version shows diminishing returns as vector length increases; memory ports go idle waiting for computations to finish. Accessing intermediate results in memory (which is not efficient) is another reason for degraded performance in the ACE-mem version.

Concatenation

This kernel stitches two tables together to create a wider one. The scalar and vector kernels use scalar (double-word) and vector load/store instructions that access the elements using the processor's bus interface. The custom versions utilize matrix registers and custom memory ports (matrix load/store instructions). The ACE-single version uses a single matrix register and cannot overlap memory read and write, while the ACE-double version uses two matrix registers to double-buffer and overlap accesses. For scaling, we increase the number of active rows in the matrix registers from one to 32. As vector length increases, the 64 B bus interface limits the performance of the vectorized kernel. On the other hand, the ACE-single version cannot fully utilize the available memory bandwidth at shorter vector lengths but performs better as the vector length increases. The double-buffered version performs 1.8× better than the single-buffer one when the vector length is increased to 128 B per instruction, allowing it to fully utilize all the available memory ports.

Transpose

This kernel transposes a tensor of FP16 elements. There are two vectorized and two custom versions: the vectorized versions use strided vector load (VSLE) or strided vector store (VSSE). The custom versions use strided matrix load and strided-transposed matrix

TABLE 4. Speedup over vectorized kernel (LMUL=1). The number of elements per instruction that matches the loosely coupled architecture are in bold.

Kernel	Elms. per instr.	32	64	128	256	512	1024
Elem. Add	*RVV*	1	1.72	1.79	2.22	—	—
	RVV (ppl)	1.62	2.16	2.99	2.55	—	—
	ACE (1 port)	0.87	1.45	2.17	2.88	3.46	3.84
	ACE (3 port)	1	1.85	3.24	5.18	7.37	**9.36**
e^x (exp)	*RVV*	1	1.48	0.47	—	—	—
	ACE (partial)	1.85	2.38	3.04	—	—	—
	ACE (full)	1.91	2.68	4.91	—	—	—
	ACE (mem)	2.74	4.48	7.86	11.43	14.77	**17.31**
Sigmoid	*RVV*	1	1.19	0.69	—	—	—
	ACE (full)	1.41	1.61	1.74	—	—	—
	ACE (mem)	1.12	1.34	1.5	1.59	1.64	1.66
Concat.	*RVV*	1	1.26	1.37	1.57	—	—
	ACE (single)	0.47	0.75	1.35	2.24	3.34	4.44
	ACE (double)	0.67	1.34	2.43	4.08	6.17	**8.25**
Trans.	*RVV (VLSE)*	1	1.15	1.22	1.25	—	—
	RVV (VSSE)	0.52	0.52	0.52	0.52	—	—
	ACE (single)	3.06	3.58	4.99	9.39	16.80	27.64
	ACE (double)	2.97	3.52	6.97	13.84	27.23	**50.36**
FC	*RVV*	1	2	—	—	—	—
	RVV (U2)	1.45	1.31	—	—	—	—
	QMACC	1.33	2.65	—	—	—	—
	QMACC (U2)	2	3.97	—	—	—	—
	ACE (single)	—	41.18	—	85.53	—	312.99
	ACE (double)	—	49.45	—	118.93	—	**382.89**
Dequant.	*RVV (Row)*	1	1.95	2.91	2.92	—	—
	RVV (Col)	0.13	0.13	0.14	0.14	—	—
	RVV+ACE	1.62	—	—	—	—	—
	ACE (Gen)	0.99	1.59	2.36	4.38	7.66	11.22
	ACE (Deq)	1.14	1.85	3.53	6.46	10.99	**15.26**
EB	*RVV*	1	1.66	2.27	—	—	—
	ACE (single)	**1.88**	3.31	5.31	7.62	9.73	—
	ACE (multi)	8.76	10.45	11.6	12.11	12.31	—
Layer norm	RVV (Row)	1	1.61	2.12	2.3	—	—
	RVV (Col)	0.51	0.45	0.47	0.42	—	—
	RVV (Welford)	0.42	0.4	0.43	0.44	—	—
	RVV+ACE	1.36	—	—	—	—	—
	ACE	1.67	2.88	4.5	6.46	12.02	—

Elem.: element-wise; Instr.: instruction.

store with an increased number of active rows in the matrix register. There is a single- and a double-buffered version as well. The vectorized kernels are very inefficient, despite doing much better than the scalar version (up to 12.6×) because strided vector load/stores generate one memory access per element and

take longer to complete. The VSSE version performs much worse, and its performance does not change with increasing vector length because the implementation uses one tracking entry for each element's memory access. In contrast, the strided vector load consumes only one tracking entry per entire instruction, resulting in marginal improvement as the vector length increases. Using matrix operations improves performance because memory accesses are aggregated. The single-buffered version performs 3–7× better than the strided vector loads at the same number of elements per instruction. The double-buffered version performs much better than the single-buffered one, improving as the number of elements per instruction increases (1.82× better at 1024 elements). The performance is lower with fewer elements because the transposed matrix store cannot utilize the full 64 B width of the custom memory port.

Fully Connected

This kernel multiplies two INT8 matrices, accumulating the INT32 result in the accumulator register (which is loaded with bias), and stores the result back. There are two vectorized versions of the kernel, one using a double-widening multiply and a double-widening add, and one using a proprietary quad-widening multiply accumulate (qmacc). For both versions, we implement a variant that unrolls the inner loop and calculates two rows per iteration (U2). As the operation is quad-widening, the maximum number of elements is limited to 128 (LMUL2). The custom versions use the MPU and the matrix registers to compute the output. Like prior kernels, there is also a double-buffered version that utilizes both available accumulators. Unrolling the standard version of the kernel hurts performance at longer vector lengths due to the increased register pressure and injected register fill/spill instructions. Using the qmacc instruction improves performance by 33% while reducing register pressure and yielding additional performance with longer vector length. However, adding custom matrix multipliers improves performance by an order of magnitude (more than 20× at 64 elements per instruction), and using two accumulators provides an additional 20%-40% improvement, demonstrating the full advantages of customizations.

Dequantize

This kernel converts elements of a 2-D tensor from INT8 to FP16. The row-wise implementation processes one row at a time and hence, increasing the vector length beyond 128 elements (number of columns) does not provide any further benefit. The columnwise implementation processes multiple rows per iteration using strided vector loads. This degrades performance, and increasing the vector length is ineffective as the latency is dominated by the strided vector load. The ACE+RVV version loads a block of the table into a matrix register, moves the columns to the vector registers, and processes them the same way as the standard vectorized kernel. It moves the results back to the matrix register for bulk writing into memory. This shows more than a 12× speedup over using strided vector loads. The custom versions of the kernel not only use matrix load/store instructions but also perform the computation on the matrix register using matrix compute instructions. The first ACE Gen version uses general matrix vector compute instructions (subtract, convert, and multiply), while the ACE Deq version uses a compound instruction to perform full dequantization as a single operation. Both of these versions show a significant improvement over the standard vector version (11 and 15×, respectively).

THE ACE-MULTI VERSION USES A CUSTOM INSTRUCTION THAT PERFORMS THE SAME OPERATION BUT RECEIVES MULTIPLE BAGS AS INPUT VIA A VECTOR REGISTER.

Embedding Bag

This kernel sums rows of a 512-wide embedding table, producing one output row per bag. The summation (vfadd.wv) is a widening operation that accumulates results as FP32 elements. The ACE-single version uses a custom instruction that receives addresses of the input and output vectors as memory operands as well as the number of rows to sum up for a single bag. It reads and sums up the rows (up to the given vector length), then writes the results back to memory. If the width of the table is wider than the number of elements in the vector, the operation is performed on vertical shards of the table. The performance improves linearly as the number of elements per vector increases. The ACE-multi version uses a custom instruction that performs the same operation but receives multiple bags as input via a vector register. The instruction produces one output per bag. This version performs much better even at smaller vector lengths as it processes up to 16 bags per invocation. Doing more work per instruction eliminates idle gaps in the ALUs between back-to-back instructions that cannot be pipelined.

Layer Norm

This kernel performs layer normalization on rows of a 2-D table. There are three vectorized versions of the kernel: the first vectorizes the operation along the rows, calculating mean and variance, and then performs the normalization by applying weight and bias. The second version performs the vectorization along the columns, calculating a vector of mean and variances, and then performing normalization on a group of rows. The third version is the same as the second but uses Welford's online algorithm instead. Like other kernels, using strided vector loads and stores in the columnwise vectorized version incurs latencies due to multiple memory accesses for vector elements. Welford's version uses long-latency floating-point vector division in the first loop, and hence, its performance is not any better, despite taking one fewer pass over the data. Like the dequantized kernel, the ACE+RVV version resolves the inefficiency of the strided vector loads and stores by utilizing matrix load/stores. This improves performance over the columnwise vectorized version by 2.65×. The ACE version uses matrix and matrix-vector elementwise operations for calculating the mean and standard deviation and performing normalization, hence considerably improving performance.

Comparison of the Two Architectures

In our second study, we compare the performance and number of retired instructions of a subset of these microbenchmarks between the tightly coupled and loosely coupled PEs (Figure 4). We exclude sigmoid and layer norm from this comparison as these kernels are not completely mapped to fixed-function blocks in the loosely coupled architecture and require the data to be handled by both fixed function and the vector processor (Proc-B) to perform vector division operations. In some cases, the tightly coupled architecture can support longer vectors, but we pick the number of elements per instruction that matches the loosely coupled architecture (bolded in Table 4). There are a few caveats to this comparison:

- The lack of pipelining in custom instructions that operate on memory is a microarchitectural limitation in the tightly coupled architecture's current implementation, which impacts the elementwise add and embedding bag kernels. The dashed blue lines demonstrate that if these instructions could be pipelined, the tightly coupled architecture would perform the same or slightly better than the loosely coupled architecture.
- A microarchitectural constraint in the loosely coupled architecture limits the memory bandwidth of the concatenation kernel to 64 B per cycle, as opposed to 128 B per cycle in the tightly coupled one. The dashed gray line shows where the throughput would be if this limitation were removed.
- The dequantization kernel in the tightly coupled architecture performs the operation by loading elements into matrix registers, performing the computation, and then writing them back, while in the loosely coupled one, the dequantization operation is performed directly on memory, hence, the performance is slightly better.

To ensure that the difference in the performance is not due to using two different processor cores,

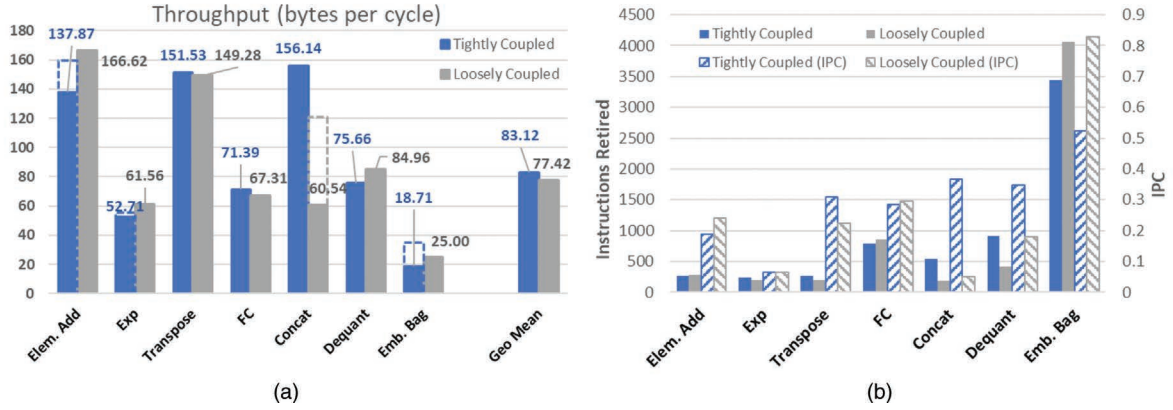

FIGURE 4. Comparison of the two architectures. (a) Throughput. (b) Retired instructions and IPC. elem.: elementwise; FC: fully connected; concat: concatenation; dequant: dequantization; emb.: embedding ; Exp: exponential (e^x).

Figure 4(b) compares the number of retired instructions as well as the instructions per clock (IPC). It should be noted that although the tightly coupled architecture uses a dual-issue core, custom instructions use only one of the issue slots, and hence, from the custom instruction-issue perspective, the processor cores in the two PEs are the same. As observed, there is no correlation between higher throughput and higher IPC; there are cases where the IPC is different, but both architectures achieve the same level of throughput. Also, the dual-issue core in the tightly coupled architecture does not always have a higher IPC; there are cases where the single-issue core in the loosely coupled architecture shows a better instruction throughput (e.g., elementwise add and embedding bag kernels).

The aforementioned comparison confirms the intuition that, to the first degree, the performance of the architecture depends on the number of available hardware resources and not on the way in which the resources are coupled with the processors. We see that tight coupling provides secondary benefits by reducing complexity and hence providing better access and management of latencies. However, as discussed, the implications on the programming model and code generation are significant. The tight integration of resources with the processor resembles a more traditional architecture that can be targeted much more easily with modern compilers, while loose coupling provides an asynchronous architecture where the software must focus on the orchestration, synchronization, and management of off-load engines, which is certainly a harder task to accomplish.

SUMMARY AND CONCLUSION

In today's evolving AI/ML workloads, the velocity of deploying a given model on an accelerator hardware is as important as the efficiency and performance benefits that the accelerator provides. The architecture of the accelerator affects its programming model and software stack, and therefore its programmability and ease of use.

We studied two different classes of accelerator architectures in this work, highlighting their merits and differences. We reviewed the potential architectural benefits that a loosely coupled architecture provides in terms of efficiency and autonomy, at the cost of difficulty and complexity of programming. We constructed a tightly coupled accelerator that provides a familiar CPU-style programming model and yields the same benefits of the loosely coupled architecture. Starting with a baseline RISC vector processor, we showed the following through our evaluation:

- The architectural restriction on vector register file size limits vector operations from their full potential performance.
- Streaming data directly from memory and bypassing the vector register file brings throughput closer to memory speeds.
- Using a RISC abstraction means we use register names which leads to register spilling when loops get large. But this also alleviates reading and writing intermediate results from and to memory.
- Default gather/scatter implementations on standard vector cores usually perform poorly as they break down memory accesses into scalar requests.
- Matrix instructions can provide significant performance gains, particularly for data movement in addition to arithmetic operations, compared to vector instructions.

These findings indicate that in the spectrum of RISC versus CISC architectures, integrating simple CISC-like features such as memory operands and complex operations in the instruction set architecture is beneficial to AI/ML workloads.

We also compared our tightly coupled accelerator against the silicon implementation of a loosely coupled one, i.e. MTIA, demonstrating that the performance of the architecture depends primarily on the number of hardware resources, not on how these resources are coupled with programmable processors. On the other hand, we argue that the way in which these resources are integrated has a pronounced impact on the eventual system's programming model and usability, and hence, with performance benefits being equal, accelerators should adopt an architecture that better lends itself to automatic compilation and code generation.

REFERENCES

1. N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Int. Symp. Comput. Archit. (ISCA)*, Toronto, ON, Canada, 2017, pp. 1–12, doi: 10.1145/3079856.3080246.
2. N. P. Jouppi et al., "A domain-specific supercomputer for training deep neural networks," *Commun. ACM*, vol. 63, no. 7, pp. 67–78, 2020, doi: 10.1145/3360307.
3. T. Norrie et al., "The design process for Google's training chips: TPUv2 and TPUv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, Mar./Apr. 2021, doi: 10.1109/MM.2021.3058217.

4. N. Jouppi et al., "TPU v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings," in *Proc. 50th Int. Symp. Comput. Archit. (ISCA)*, Orlando, FL, USA, 2023, pp. 1–14.
5. H. Zheng et al., "Optimizing memory-access patterns for deep learning accelerators," 2020, *arXiv:2002.12798*.
6. A. Firoozshahian et al., "MTIA: First generation silicon targeting meta's recommendation systems," in *Proc. 50th Int. Symp. Comput. Archit. (ISCA)*, Orlando, FL, USA, 2023, pp. 1–13, doi: 10.1145/3579371.3589348.
7. A. Firoozshahian, O. Wu, J. Coburn, and R. Levenstein, "MTIA v1: Meta's first-generation AI inference accelerator," *Meta*, May 18, 2023. [Online]. Available: https://ai.meta.com/blog/meta-training-inference-accelerator-AI-MTIA/
8. E. Tal, N. Viljoen, J. Coburn, and R. Levenstein, "Our next-generation Meta Training and Inference Accelerator," *Meta*, Apr. 10, 2024. [Online]. Available: https://ai.meta.com/blog/next-generation-meta-training-inference-accelerator-AI-MTIA/
9. R. Hameed et al., "Understanding sources of inefficiency in general-purpose chips," in *Proc. 37th Int. Symp. Comput. Archit. (ISCA)*, Saint-Malo, France, 2010, pp. 37–47.
10. D. Ditzel, RISC-V Summit, San Jose, CA, USA, *Real World Results Using Thousands of RISC-V Cores for AI and Beyond*. (Dec. 29, 2022). Accessed: Jul. 13, 2024. [Online Video]. Available: https://www.youtube.com/watch?v=5foT3huJ_Gg
11. R. Beachler, "Boqueria – Next generation at-memory inference acceleration device with 1000+ RISC-V cores," Hot Chips, Stanford, CA, USA, 2022. [Online]. Available: https://hc34.hotchips.org/assets/program/conference/day2/Machine%20Learning/HotChips_Boqueria_Presentation_v15.pdf
12. "Andes Custom Extension™." Andes Technology. Accessed: Jul. 13, 2024. [Online]. Available: https://www.andestech.com/en/products-solutions/andes-custom-extension/

AMIN FIROOZSHAHIAN is the lead architect at Rain AI, San Francisco, CA, 94110, USA. His research interests include artificial intelligence/machine learning accelerators, memory subsystems, and multiprocessors. Firoozshahian received his Ph.D. degree in electrical engineering from Stanford University. He is a Member of IEEE and the Association for Computing Machinery. He is the co-corresponding author of this article. Contact him at amin@rain.ai.

JOEL COBURN is a software engineer at Meta Platforms Inc., Menlo Park, CA, 94025, USA. His research interests include machine learning (ML) accelerators, hardware–software co-design of ML models, and emerging memory and compute technologies. Coburn received his Ph.D. degree in computer engineering from the University of California, San Diego. He is a member of the Association for Computing Machinery. Contact him at jcoburn@meta.com.

AJIT PUNJ is an application-specified integrated circuit architecture engineer at Meta Platforms Inc., Menlo Park, CA, 94025, USA. His research interests include machine learning hardware accelerators, and reconfigurable processors. Punj received his M.S. degree in electrical engineering from Stanford University. Contact him at apunj@meta.com.

ARAVIND SUKUMARAN RAJAM is a research scientist in the Hardware–Software Co-Design group at Meta Platforms Inc., Menlo Park, CA, 94025, USA. His research interests include high-performance computing and hardware–software accelerator design and their applications to machine learning and scientific computing. Rajam received his Ph.D. degree in computer science from the University of Strasbourg. Contact him at aravindsr@meta.com.

COLBY BOYER is an application-specified integrated circuit architecture engineer at Meta Platforms Inc., Menlo Park, CA, 94025, USA. His research interests include hardware–software co-design of machine learning operators, artificial intelligence accelerators, and wireless communication systems. Boyer received his Ph.D. degree in electrical engineering from the University of Washington. Contact him at csboyer@meta.com.

RAKESH NATTOJI is an application-specified integrated circuit architecture engineer at Meta Platforms Inc., Menlo Park, CA, 94025, USA. His research interests include efficient mapping of machine learning operators to industry-standard and computer architectures. Nattoji received his M.S. degree in electrical engineering from the University of California, San Diego. Contact him at rakeshnattojirajaram@gmail.com.

MAHIMA BATHLA is an application-specified integrated circuit architecture engineer at Meta Platforms Inc., Menlo Park, CA, 94025, USA. Her research interests include system-on-chip architecture and machine learning accelerators. Bathla received her M.S. degree in electrical and computer engineering from Purdue University. Contact her at mbathla@meta.com.

BOB DREYER is semiretired at Accel Dynamics in Palo Alto, CA, 94301, USA. His research interests include systems development, CPU architecture, and stream processing accelerators. Dreyer received his Master of Business Administration degree from Harvard University. He is a Member of IEEE. He is the co-corresponding author of this article. Contact him at rdreyer@alumni.princeton.edu.

SUJITH SRINIVASAN is an application-specified integrated circuit (ASIC) architecture engineer at Meta Platforms Inc., Menlo Park, CA, 94025, USA. His research interests include machine learning accelerators. Srinivasan received his M.S. degree in electrical engineering from The Ohio State University. Contact him at sujiths@meta.com.

HARSHITHA PILLA is an application-specified integrated circuit architecture engineer at Meta Platforms Inc., Menlo Park, CA, 94025, USA. Her research interests include machine learning accelerators. Pilla received her master's degree in electrical and computer engineering from North Carolina State University. Contact her at hpilla@meta.com

MICHAEL ROTZIN is an application-specified integrated circuit design engineer at Meta Platforms Inc., Menlo Park, CA, 94025, USA. His research interests include networking, machine learning accelerators, and microprocessors. Rotzin received his B.S. degree in electrical engineering from Virginia Polytechnic Institute and State University. Contact him at mrotzin@meta.com

SURENDRA RAJUPALEM is an application-specified integrated circuit design engineer at Meta Platforms Inc., Menlo Park, CA, 94025, USA. His research interests include high-performance cores, subsystem interconnects, and system-on-chip design and implementation. Rajupalem received his M.S. degree in computer engineering from the University of Louisiana at Lafayette. Contact him at srajupal@gmail.com.

K. RAJESH JAGANNATH is an application-specified integrated circuit design and processor verification engineer at Meta Platforms Inc., Menlo Park, CA, 94025, USA. His research interests include processors, inference and training accelerators, and attached coprocessors. Jagannath received his M.S. degree in computer engineering from the University of Texas at El Paso. He is a Member of IEEE. Contact him at k.rajesh.j@gmail.com.

KRISHNA NORU is a director at Meta Platforms Inc., Menlo Park, CA, 94025, USA. His research interests include building machine learning and video accelerators. Noru received his master's degree in electrical and computer engineering from Utah State University. Contact him at knoru@meta.com.

HARIKRISHNA REDDY is a technical lead and director at Meta Platforms Inc., Menlo Park, CA, 94025, USA. He works on custom accelerators that are used in Meta's infrastructure. His research interests include AI accelerators, video processing, silicon process, and packaging technologies. Reddy received his master's degree in electrical engineering from Texas A&M University. Contact him at harikr@yahoo.com.

CHRIS YANG is the director at Andes Technology, Hsin-Chu, 300042, Taiwan. His research interests include computer architecture, compute acceleration, and artificial intelligence. Yang received his Ph.D. degree in computer science from National Chung Cheng University. Contact him at chris@andestech.com.

CHARLIE HONG-MEN SU is the cofounder, president, and CTO at Andes Technology, Hsin-Chu, 300042, Taiwan. His research interests include processor and system-on-chip architecture, compute acceleration, and hardware–software interaction. Su received his Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign. Contact him at charlie@andestech.com.

CHARLIE CHENG is the board advisor at Andes Technology, Hsin-Chu, 300042, Taiwan. His research interests include computer architecture and next-generation memories. Cheng received his B.S. degree in mechanical and aerospace engineering from Cornell University. Contact him at charlie.cheng@andtestech.com.