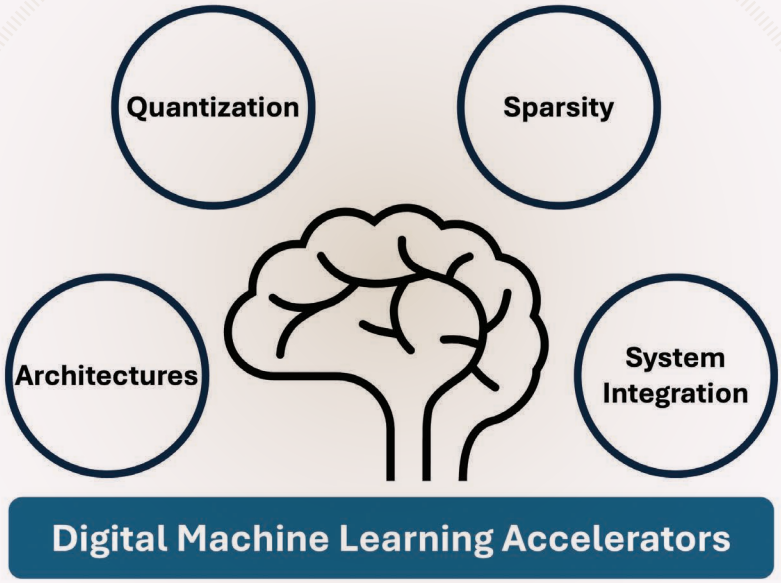
Wei Tang, Sung-Gun Cho, Jie-Fang Zhang, and Zhengya Zhang

Design and Optimization of Efficient Digital Machine Learning Accelerators

An overview of architecture choices, efficient quantization, sparsity exploration, and system integration techniques



OPENER ART WAS CREATED USING MICROSOFT POWER POINT AND ITS "BRAIN" ICON.

Digital machine learning (ML) accelerators are popular and widely used. We provide an overview of the SIMD and systolic array architectures that form the foundation of many accelerator designs. The demand for higher compute density, energy efficiency, and scalability has been increasing. To address these needs, new ML accelerator designs have adopted a range of techniques, including advanced architectural design, more efficient quantization, exploiting data-level sparsity, and leveraging new integration technologies. For each of these techniques, we review the common approaches, identify the design tradeoffs, and discuss their implications.

Digital Object Identifier 10.1109/MSSC.2025.3549361
Date of current version: 20 June 2025

Introduction

ML has found widespread use across various applications and has become a dominant computational workload. While general-purpose platforms like

GPUs and CPUs remain primary for ML, dedicated accelerators are emerging due to their higher compute density and better energy efficiency.

Dedicated accelerators can be designed effectively and efficiently because the core of ML algorithms is built on a small well-defined set of computational kernels. These kernels, such as matrix multiplication and convolution, are computationally intensive, along with activation functions, pooling, normalization, and element-wise operations. GPUs can parallelize these kernels, but they are designed for a broad range of tasks and incur a large control and memory management overhead, making them less efficient and more costly. In contrast, ML accelerators are optimized for these key computational kernels, potentially achieving higher performance per unit of silicon area and significantly improved energy efficiency.

Over the past decade, considerable research and development has been invested in ML accelerator designs. Digital accelerators offer unique advantages like higher fidelity (compared to analog and in-memory accelerators[1]), better scalability, seamless integration with CPUs, and ease of design and manufacturability across different process technologies. However, they may not achieve the top-notch energy efficiency of in-memory and analog accelerators. Most commercially available ML accelerators, such as Google TPU [1], [2], Amazon Inferentia, Groq TSP [3], [4], and Graphcore IPU, are digital.

This article provides an overview of the foundational architectures of digital ML accelerators and explores potential enhancements in this area.

[1]Recent in-memory compute has shifted toward digital in-memory computing using binary memory cells and digital circuitry to enhance robustness against PVT variation and improve accuracy compared to its analog counterpart. Digital in-memory accelerators are beginning to emerge commercially from companies like TSMC, d-Matrix, and Axelera AI.

Dedicated accelerators can be designed effectively and efficiently because the core of ML algorithms is built on a small well-defined set of computational kernels.

Foundational Architectures

A wide variety of ML accelerators have been demonstrated. Despite diverse designs, they often rely on two core architectures for matrix multiplication and convolution: SIMD [5], [6], [7] and systolic arrays [8], [9], [10]. We introduce each architecture, examining how it maps and executes computations.

SIMD Architecture

The SIMD architecture is extensively used in CPUs and GPUs [11], [12]. It consists of an array of processing elements (PEs), each capable of executing operations like MAC, as shown in Figure 1. A single instruction initiates an operation across the entire PE array, with each PE performing the same operation. For example, with one multiplication instruction, each PE fetches a pair of data, computes their product, and writes the result back to memory. A SIMD array can be used to compute a vector–vector dot product, where multiplications are followed by a summation.

The flexibility of the SIMD architecture allows it to support various ML computational kernels, including vector–matrix multiplication (VMM), matrix–matrix multiplication (MMM), and convolution. For VMM, the input vector and weight matrix are stored

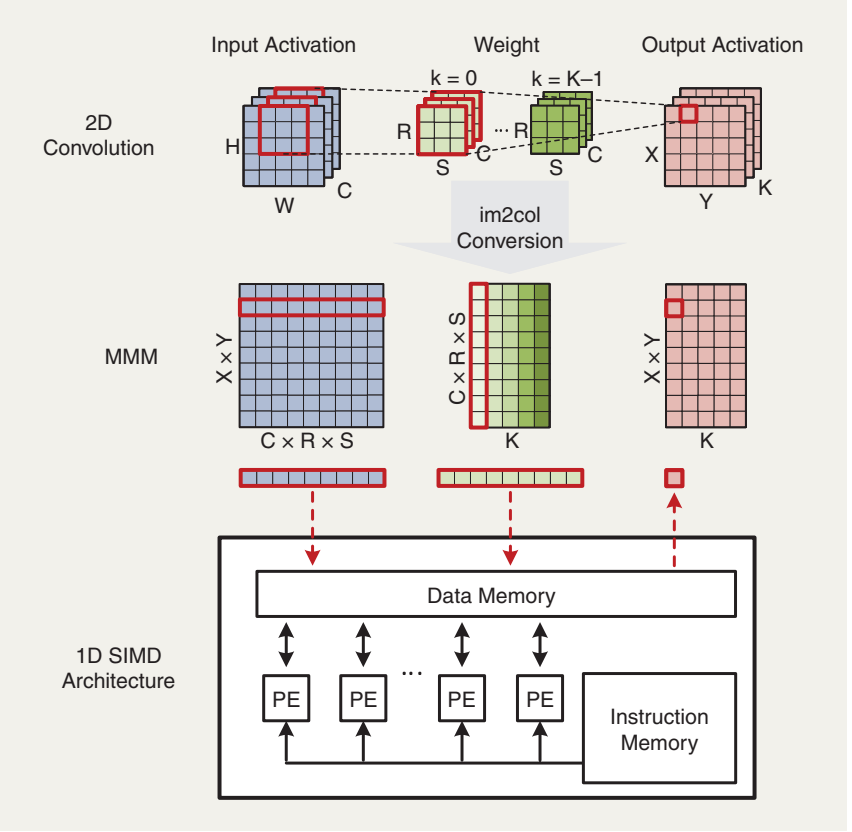


FIGURE 1: A 1D SIMD array architecture and 2D convolution operations on the array. MMM: matrix–matrix multiplication.

A SIMD array offers greater flexibility for various operations, potentially leading to higher hardware utilization but at the cost of higher control overhead.

in memory within the SIMD array. Each processing step fetches the input vector and one weight vector to perform dot product computation until the entire weight matrix is processed. Local memory within each PE can cache the input vector and reuse it, reducing memory access and enhancing efficiency. MMM extends VMM by cycling through both matrices and accessing vector pairs to compute dot products. The matrices can also be cached locally to allow reuse.

For 2D convolution, weights and activations are first organized appropriately via the im2col conversion, as described in Figure 1. The $K \times R \times S \times C$ weights are flattened into a 2D $RSC \times K$ matrix. The $H \times W \times C$ activation is divided into a series of overlapping $R \times S \times C$ segments, each used for convolution in a sliding window fashion. These segments are stacked into a 2D $XY \times RSC$ matrix, where X and Y represent the number of sliding window steps in the horizontal and vertical directions, respectively. An MMM operation by the SIMD array then produces a 2D $XY \times K$ convolution output.

The flexibility of a SIMD architecture supports various computations with simple data parallelism. Tiling a 1D SIMD array into a 2D array allows shared input and weight loading from external memory among 1D tiles, reducing bandwidth requirements.

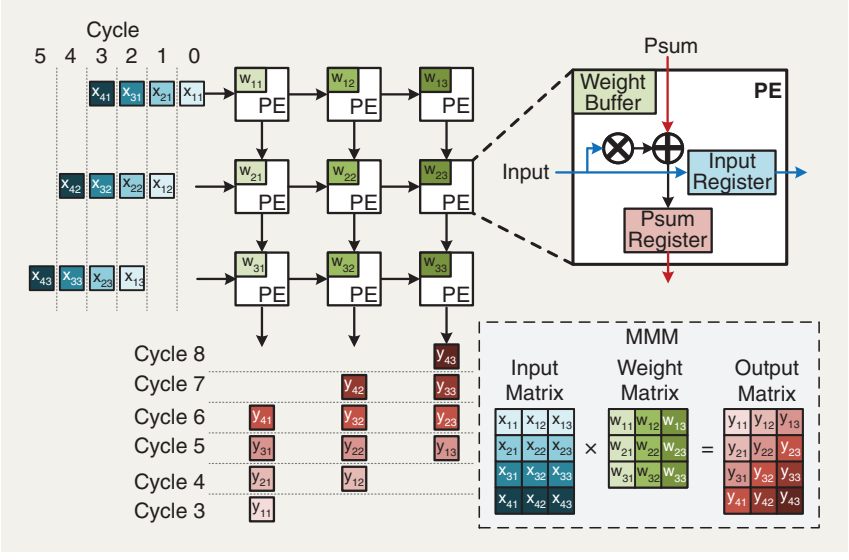


FIGURE 2: A 3 × 3 systolic array, a PE design, and the MMM operation on the systolic array.

TABLE 1: A COMPARISON OF SIMD AND SYSTOLIC ARRAY ARCHITECTURES.

	SIMD ARRAY	SYSTOLIC ARRAY
Architecture	1D/2D PE array with shared instructions	2D PE array with neighboring connectivity
Operations	VMM, MMM	MMM
Data movement	More memory access	Mostly local data movement
Compute density	Lower	Higher
Flexibility	Higher	Lower
Hardware utilization	Higher	Lower

Systolic Array Architecture

A systolic array consists of a 2D grid of PEs interconnected with their neighbors, each capable of operations like MAC and equipped with boundary registers. In each clock cycle, all PEs perform an operation, passing intermediate outputs to adjacent PEs in one direction, such as top to bottom or left to right.

To compute a VMM between an input vector and a weight matrix, the weight matrix is loaded into the array, with each weight element stored in a PE, as in Figure 2. The input vector is sequentially introduced from one side, such as the left, with one element entering per cycle. In the first cycle, x_{11} is multiplied by w_{11} to compute the partial sum $x_{11}\ w_{11}$, which is then propagated downward. In the second cycle, x_{12} enters the array, where $x_{12}\ w_{21}$, is computed and added to the partial sum from the first cycle. Meanwhile, x_{12} shifts one step to the right in the first row to calculate $x_{12}\ w_{21}$. This process continues, with input elements moving right and partial sums moving downward, creating a wavelike data flow through the array. Once the wave passes through, the VMM operation is complete. MMM can be performed similarly by launching vectors in waves during each cycle, allowing the systolic array to execute one VMM per clock cycle. 2D convolution can also be converted to MMM and mapped onto a systolic array using this method.

Table 1 compares the SIMD and systolic array architectures. In general, a SIMD array offers greater flexibility for various operations,

potentially leading to higher hardware utilization but at the cost of higher control overhead. On the other hand, a systolic array relies on data movement and reuse between neighboring PEs, resulting in higher efficiency and reduced memory bandwidth usage compared to a SIMD architecture.

Enhanced Systolic Array

While a conventional systolic array (CSA) can process inputs in a pipelined structure, scaling it up for higher performance introduces two issues. First, the pipeline length increases with the array size, resulting in longer latencies. For example, an $N \times N$ array would require N cycles to compute a column summation through N PEs. Second, the diverse data dimensions of ML workloads often prevent full utilization of all PEs in a large array.

Low-Latency and High-Utilization Systolic Array

A recent work [10] addressed these issues by arranging and connecting PEs in overlays of two structures—H tree and array—forming the PE tree array (PETRA). As a systolic array architecture, PETRA includes PEs that shift input data and weights between each other. However, PETRA differs from the CSA in its data path for summation outputs. PEs in PETRA calculate products and push them to an array-wide adder tree, structured as an H tree, as illustrated in Figure 3(a). This physical implementation-friendly structure offers two advantages: 1) logarithmic scaling of the summation latency and 2) low-overhead multiworkload mapping to subtrees.

The H tree in PETRA can sum products from an $n \times n$ PE array in $\log_2 n^2$ cycles when pipelined. For $n = 16$, the summation latency across 256 PEs is eight cycles, 32 times faster than the summation of a CSA column with $N = 256$ PEs. The H tree includes $n^2 - 1$ adders, one fewer than the CSA.

In addition to reduced latency, PETRA leverages the binary tree structure to sectionize data mapped to PEs. Since a binary tree can be divided into subtrees, PETRA can map multiple independent workloads simultaneously without a complicated network between PEs, enhancing PE utilization. Above a certain level in the H tree, a configurable adder tree (CAT) produces sums for various input combinations, while subtrees below remain fixed adder trees. PEs can be partitioned by subtrees; for example, eight subtrees can serve as separate partitions for a 16 × 16 PETRA, as in Figure 3(a). An independent

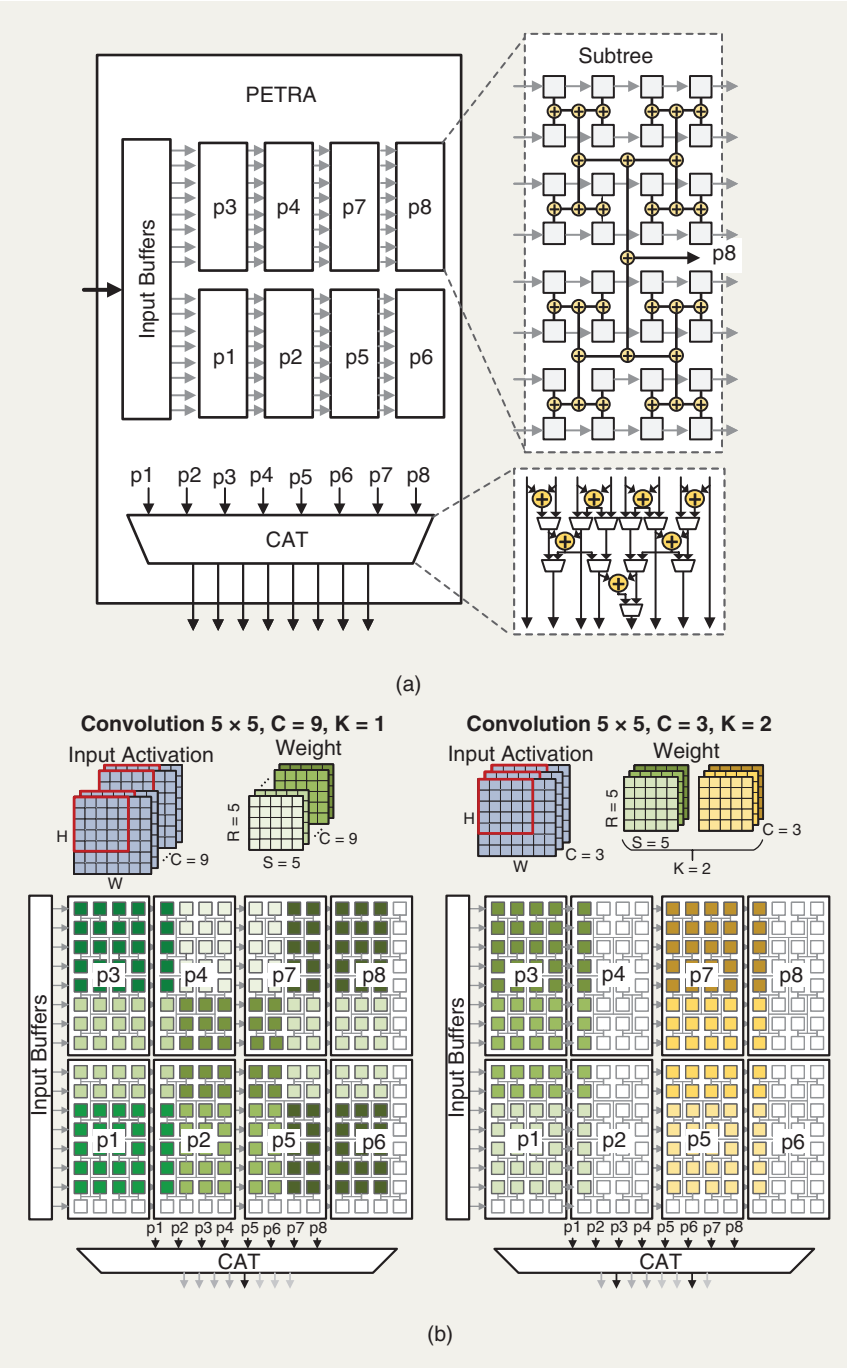


FIGURE 3: (a) PETRA with a configurable adder tree (CAT). (b) The flexible mapping of various convolution workloads to PETRA.

A systolic array relies on data movement and reuse between neighboring PEs, resulting in higher efficiency and reduced memory bandwidth usage.

workload can be mapped to a subtree or a set of subtrees. The CAT allows for either individual subtree outputs or summations of subtree outputs based on the workload partition. This support for workload partitioning allows PETRA to accommodate multiple smaller filters more efficiently, improving hardware utilization.

In addition to VMM and MMM, PETRA is also optimized for convolution. Traditional 2D convolution operations with a CSA employ im2col conversion followed by MMM, which results in significant data duplication in the input matrix, due to the overlapping sliding segments in convolution. Enhanced systolic array architectures, like Eyeriss [8] and PETRA, avoid this duplication and reuse the input by locally shifting the data. Eyeriss shifts the data between PEs both horizontally and vertically, requiring a dense inter-PE connection. In contrast, PETRA allows only horizontal shifts between PEs and performs the vertical shifts inside the input buffer, lowering inter-PE connection overhead.

Design Example

We compare the CSA and PETRA architectures for an exemplary (16 × 256) × (256 × 16) MMM workload. A CSA spatially unrolls the MMM operation using a 256 × 16 PE array. Each PE includes one MAC, one input buffer, one weight buffer, and one output buffer. The CSA takes 256 cycles to produce the first output and 256 + 16 + 16 = 288 cycles to produce all the outputs, with a throughput of 16 outputs/cycle.

With the same number of PEs, the PETRA architecture employs 16 16 × 16 PETRAs. Each PE includes one multiplier, four input buffers, 16 weight buffers, and one output buffer [10]. The larger weight buffer in a PETRA PE allows for temporal multiplexing to handle 16-times-larger workloads. The PETRA architecture takes $16 + \log_2(256) = 24$ cycles to produce the first output and $16 + \log_2(256) + 16 = 40$ cycles to produce all outputs. It achieves the same throughput as the CSA, with significantly lower latency.

Compared to the CSA, PETRA trades higher buffer usage for more efficient convolution operations, enabling input reuse without duplication. Overall, PETRA offers greater flexibility to accommodate a wide range of workload sizes and types, and it allows the mapping of multiple independent workloads to improve utilization. Two examples of convolution workloads are given in Figure 3(b). One 5 × 5 filter with channel size $C = 9$ can be mapped to an entire 16 × 16 PETRA, utilizing all eight subtrees, with the CAT producing one output channel. In the other example, two 5 × 5 filters ($K = 2$) with channel size $C = 3$ can be mapped concurrently to two sets of subtrees in a 16 × 16 PETRA, with each set combining four subtrees to support one filter. The CAT is configured to produce two output channels in this case.

A PETRA prototype consisting of four 16 × 16 PETRAs was fabricated [10]. Sample ML model mapping results are reported in Table 2. As illustrated in Figure 3(a), each PETRA can be partitioned into eight subtrees. Thus, the mapping granularity is a subtree of 32 PEs. If the compute kernels, such as filters, are of comparable or larger size, high utilization is expected. However, if the compute kernels are smaller, the utilization is likely to decrease. This is evidenced by the results: for VGG-16 and Tiny-YOLO, the utilization exceeds 80%, whereas for LeNet and AlexNet, the utilization is in the 60% range, due to the presence of layers with shallow channel depth (notably, the first and second layers) and the small number of layers in the overall model, which does not effectively average out the lower utilization of the early layers. The granularity can be adjusted in the design to maintain a high average utilization.

Since the base PETRA is composed of PEs for MAC operations, the base architecture needs to be augmented with specialized function units to support other operations, such as softmax, pooling, and normalization. Another option is to enhance some of the PEs to handle these special functions, but it would inevitably complicate the PE design itself. Although PETRA's design maximizes data reuse, high-bandwidth I/O and memory are still necessary to support a scaled-up version of PETRA, ensuring high utilization when processing large models like VGG and transformers.

TABLE 2: A SAMPLE ML MODEL MAPPING ON PETRA [10].

MODEL	THROUGHPUT	INPUT DIMENSION	UTILIZATION
LeNet	83,900 frames/s	32 × 32 (frame)	65%
AlexNet	103 frames/s	227 × 227 (frame)	61%
VGG-16	34.9 frames/s	227 × 227 (frame)	87%
Tiny-YOLO	68.5 frames/s	416 × 416 (frame)	81%
Transformer	278.5 sequences/s	768 (embedding), 64 (key/value)	100%

The throughput is measured at a 701-MHz clock frequency. Softmax and pooling are not included. The transformer workload is a 12-head attention with 512 sequences.

Quantization Optimization

Quantization is a crucial step in determining the number format and precision of an ML model's parameters and activations. Shorter word lengths, lower precision, and fixed-point (integer) quantization reduce computational complexity, data storage, and data movement but may sacrifice computation quality. Several quantization approaches exist: post-training quantization converts a high-precision trained model (e.g., 32-b floating point) to a lower-precision fixed-point format (e.g., 8-b integer), which can degrade accuracy unless fine-tuned or retrained. Quantization-aware training incorporates the effects of quantization during the training process, potentially yielding better postquantization accuracy.

Applying the same quantization for all parameters and activations in a model is not ideal, as their contributions to accuracy vary. Most ML hardware already supports different precision levels for weights and activations, such as 4-b weights and 8-b activations. However, optimal precision varies across models. To address this, recent ML hardware designs [13], [14], [15], [16] have incorporated mixed-precision quantization, offering greater flexibility and potential for higher speed and efficiency. In mixed-precision quantization, different layers or operations of an ML model can be quantized using various formats and precisions or even dynamically changing at runtime. The hardware needs to support a range of choices, such as 4, 8, and 16 b—integer or floating point—ideally reusing the same unit. For example, a 16-b unit can be split to perform multiple 8- or 4-b operations.

Nonuniform and alternative quantization methods [17], [18], [19], [20] have been proposed, such as variable-length quantization, which offers higher precision for lower values and supports a higher dynamic range, providing a better tradeoff between word length and accuracy. While these methods reduce memory size and data movement costs, they often require more complex arithmetic circuitry, lookup tables, or encoders and decoders.

Sparsity Optimization

Sparsity in activations can result from natural sparsity in input data, such as zero-valued information, static scenes in video, a rectified linear unit zeroing out negative values, or finite-precision quantization eliminating small values. Sparsity can also be introduced by pruning weights. Pruning reduces the size of ML models before deployment, minimizing data storage and movement and improving speed. Typically performed after training, pruning methods include removing weights below a threshold, insignificant filters or channels, and attention heads in transformers. Fine-tuning after pruning helps recover accuracy, with possible retraining if needed. Iterative pruning and training cycles may be conducted for optimal results. After pruning, dense model data structures become sparse, as shown in Figure 4(a), and are represented using formats like coordinate list [21], [22], [23], compressed sparse row/column [9], [24], and run length coding [20], which store only the nonzero elements and their indices, reducing memory size and bandwidth, as detailed in Figure 4(b).

Pruning can be unstructured or structured. Unstructured pruning, or fine-grained pruning, removes individual weights based on magnitude or significance, without considering the model's structure. This can lead to higher sparsity but creates irregular data structures, i.e., vectors and matrices with random zero locations, making efficient hardware mapping challenging. Structured pruning, or coarse-grained pruning, removes entire blocks, such as input channels, filters, or layers, retaining regular vector and matrix data structures, as in Figure 4(a). This facilitates efficient use of standard hardware but imposes restrictions on pruning and may have a greater impact on model accuracy.

An accelerator can be designed to process sparse (compressed) data.

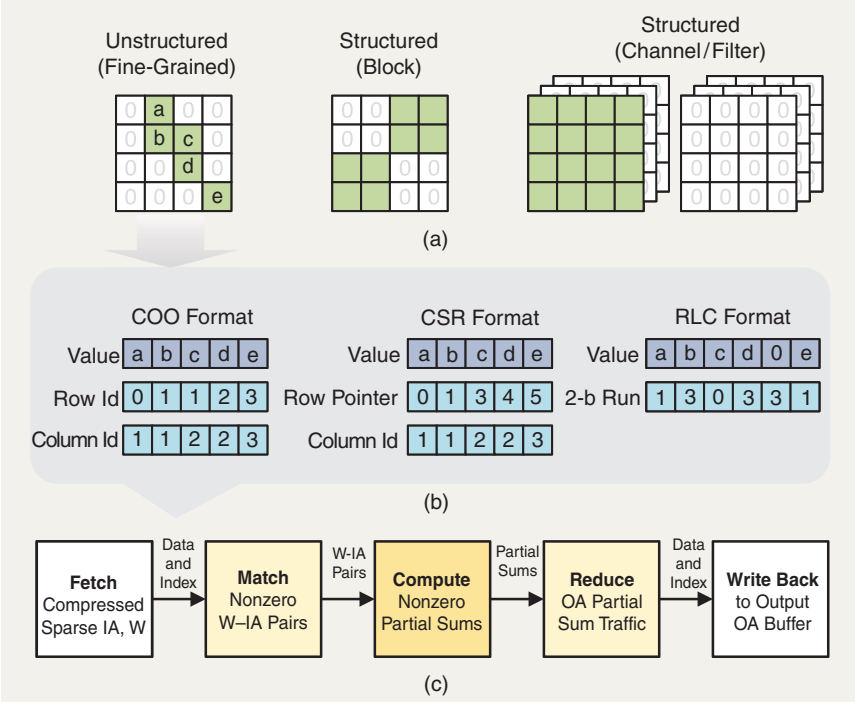


FIGURE 4: (a) Unstructured and structured pruning. (b) Compression formats: coordinate list (COO), compressed sparse row (CSR), and run length coding (RLC). (c) The sparse data processing flow. IA: input activation; W: weight; OA: output activation.

Besides the core computing unit, such as a SIMD unit or a systolic array, it requires a front-end indexing unit to match nonzero activations and weights for computing MAC and a write-back unit to determine addresses and prepare write back, as in Figure 4(c). These units must account for varying levels and structures of sparsity, which can differ significantly between models and layers. Consequently, hardware optimized for one sparsity pattern may not handle others effectively.

Recent accelerator designs [9], [20], [21], [22], [23], [24], [25] have become more flexible, supporting various sparsity levels and structures to meet different models' needs. One method uses multiple cores, each optimized for a specific sparsity level or structure, with workloads directed to the most suitable core. A more advanced method [26] imposes structure on fine-grained pruning by dividing weight matrices into uniformly sized groups, which are pruned independently. The prune rate for each group can be set to a limited number of choices, offering flexibility within a structure and ensuring efficient hardware utilization.

System Integration

ML models are rapidly growing in both size and variety, posing a major challenge in evolving hardware at the same pace. However, as models evolve, the core compute kernels remain consistent. Differences among models mainly involve the composition and sizes of these kernels. This drives the design of ML hardware using modular blocks, enabling reuse in constructing new systems.

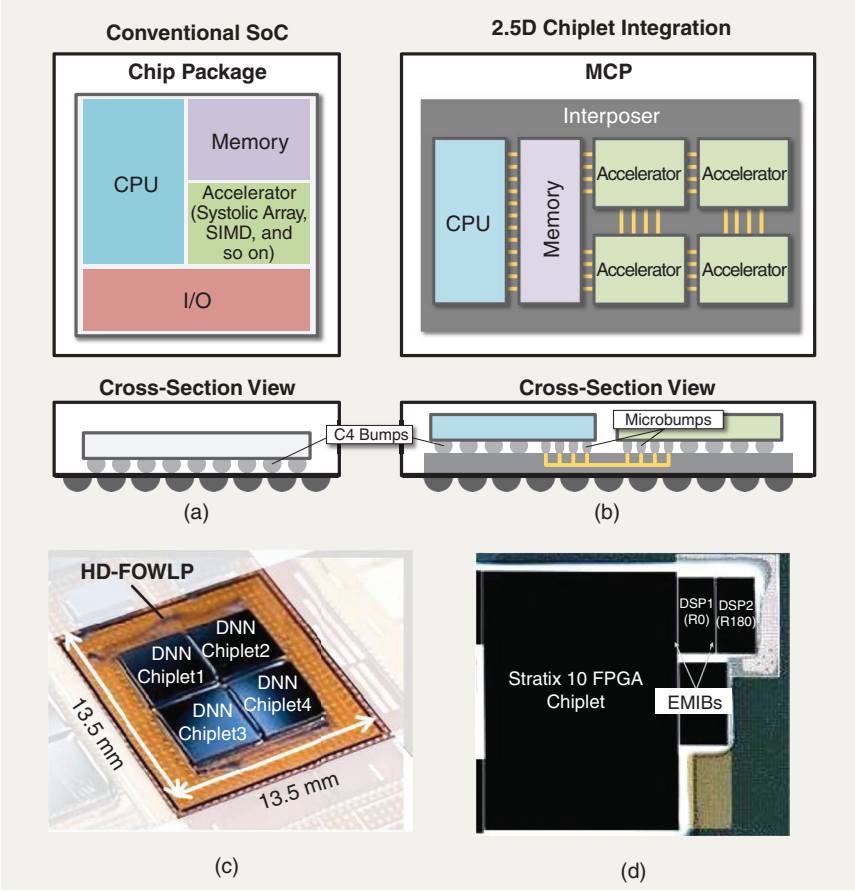


FIGURE 5: System integration: (a) conventional SoC and (b) chiplet integration. Examples of chiplet integration: (c) homogeneous [27] and (d) heterogeneous [28]. HD-FOWLP: high-density fan-out wafer-level packaging; DNN: deep neural network; EMIB: embedded multidie interconnect bridge.

Modular system construction can take the form of SoCs, typically consisting of CPU cores, a SIMD unit or systolic array for convolutions, VMM, MMM, a high-performance memory system, and fast I/Os, as displayed in Figure 5(a). The SIMD unit or systolic array handles intensive computations, while the CPU manages control, scheduling, and special functions, offering versatility for evolving workloads.

One drawback of monolithic SoCs is that every model change requires chip redesign and new fabrication. A promising alternative is chiplet-based integration, as described in Figure 5(b), where systems are constructed from modular chiplets, each specializing in a particular function. This approach allows systems to grow by incrementally adding more chiplets and different types of chiplets. To realize chiplet-based integration, it is essential to leverage advanced packaging to place chiplets close together, route dense wiring between them, and provide a high-bandwidth energy-efficient die-to-die interface for seamless communication. Advanced packaging is becoming more accessible, and die-to-die interface standards are emerging, paving the way for a wider adoption of this technique.

The chiplet approach enables rapid system scaling to meet new model needs. For example, Netflex [27], as presented in Figure 5(c), is constructed from four identical neural network chiplets using high-density fan-out wafer-level packaging [29]. Similarly, Nvidia's deep neural network (DNN) MCM [30], [31] integrates up to 36 DNN chiplets in an organic package. The chiplet approach also supports more versatile ML systems. For instance, Arvon [28], as shown in Figure 5(d), integrates an Intel FPGA chiplet with two systolic array chiplets using embedded multidie interconnect bridges [32], [33]. Convolutions and MMMs are handled by the systolic array chiplets, while the FPGA chiplet provides control and support functions. This combination

of the FPGA's flexibility and systolic array's performance addresses the needs of fast-evolving ML models.

A well-designed monolithic chip would theoretically be the best solution for achieving the highest performance and energy efficiency, as a chiplet approach incurs integration overheads, such as die-to-die I/Os. However, in practice, a well-designed monolithic chip is not always feasible due to the rapid evolution of ML models and their increasing size, resulting in unrealistic design turnaround time, design effort, and cost. Additionally, a larger die suffers from lower yield, and the monolithic die size is ultimately limited by the maximum available reticle size. Therefore, the argument in favor of a chiplet approach lies in its faster turnaround and more scalable design effort to accommodate new model changes, offering competitive, though potentially not the absolute best possible, performance and energy efficiency compared to a theoretical monolithic die.

Regarding the scaling of chiplets, a previous study examined the performance versus energy advantage of scaling up the number of chiplets for running DNN workloads [31]; another study studied the cost advantages of chipletized versus monolithic CPUs as the core numbers scale from 16 to 64 [34]. These studies demonstrate the promise of chiplets in terms of performance, energy, and cost in scaled-up designs.

Conclusion

We presented an overview of two core architectures for digital ML accelerators: SIMD and systolic arrays. SIMD architectures are flexible and provide higher utilization, while systolic arrays offer higher compute density and more efficient data movement. Systolic arrays can be further optimized to reduce computational latency for critical applications and improve utilization for various workloads, as demonstrated by the PETRA design example. Optimizing parameter quantization is crucial in ML computation, and hardware should be designed to accommodate optimal quantizations while remaining flexible for diverse workloads. Additionally, built-in hardware support for data sparsity is increasingly common, enhancing performance and efficiency with minimal overhead. Finally, ML accelerators are typically integrated into SoCs, and it is becoming practical to use 2.5D integration techniques to scale up such systems.

Acknowledgment

We acknowledge the funding of this work by the ACE Center for Evolvable Computing and the Center for Ubiquitous Connectivity, sponsored by the Semiconductor Research Corporation and DARPA, under the JUMP 2.0 Program.

References

[1] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2017, pp. 1–12.

[2] N. P. Jouppi et al., "Ten lessons from three generations shaped Google's TPUv4i: Industrial product," in *Proc. ACM/IEEE 48th Int. Symp. Comput. Archit. (ISCA)*, 2021, pp. 1–14, doi: 10.1109/ISCA52012.2021.00010.

[3] D. Abts et al., "Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2020, pp. 145–158, doi: 10.1109/ISCA45697.2020.00023.

[4] D. Abts et al., "A software-defined tensor streaming multiprocessor for large-scale machine learning," in *Proc. 49th Annu. Int. Symp. Comput. Archit. (ISCA)*, New York, NY, USA: ACM, 2022, pp. 567–580, doi: 10.1145/3470496.3527405.

[5] D. Rossi et al., "Vega: A ten-core SoC for IoT endnodes with DNN acceleration and cognitive wake-up from MRAM-based state-retentive sleep mode," *IEEE J. Solid-State Circuits*, vol. 57, no. 1, pp. 127–139, Jan. 2022, doi: 10.1109/JSSC.2021.3114881.

[6] G. K. Chen, P. C. Knag, C. Tokunaga, and R. K. Krishnamurthy, "An eight-core RISC-V processor with compute near last level cache in intel 4 CMOS," *IEEE J. Solid-State Circuits*, vol. 58, no. 4, pp. 1117–1128, Apr. 2023, doi: 10.1109/JSSC.2022.3228765.

[7] S. K. Lee, P. N. Whatmough, M. Donato, G. G. Ko, D. Brooks, and G.-Y. Wei, "SMIV: A 16-nm 25-mm^2 SoC for IoT with arm cortex-a53, eFPGA, and coherent accelerators," *IEEE J. Solid-State Circuits*, vol. 57, no. 2, pp. 639–650, Feb. 2022, doi: 10.1109/JSSC.2021.3115466.

[8] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017, doi: 10.1109/JSSC.2016.2616357.

[9] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 9, no. 2, pp. 292–308, Jun. 2019, doi: 10.1109/JETCAS.2019.2910232.

[10] S.-G. Cho, W. Tang, C. Liu, and Z. Zhang, "PETRA: A 22nm 6.97 TFLOPS/W AIB-enabled configurable matrix and convolution accelerator integrated with an Intel Stratix 10 FPGA," in *Proc. Symp. VLSI Circuits (VLSI)*, 2021, pp. 1–2, doi: 10.23919/VLSICircuits52068.2021.9492517.

[11] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, Intel, Santa Clara, CA, USA, Apr. 2022, pp. 5–16–5–19.

[12] R. Krashinsky, O. Giroux, S. Jones, N. Stam, and S. Ramaswamy, "NVIDIA ampere architecture in-depth," *NVIDIA Developer*, May 14, 2020. [Online]. Available: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

[13] C.-H. Lin et al., "A quad-core AI processing unit for generative AI in 4nm 5G smartphone SoC," in *Proc. IEEE Symp. VLSI Technol. Circuits*, 2024, pp. 1–2, doi: 10.1109/VLSITechnologyandCir46783.2024.10631508.

[14] V. Jain, S. Giraldo, J. D. Roose, B. Boons, L. Mei, and M. Verhelst, "TinyVers: A 0.8-17 TOPS/W, 1.7 mW-20 mW, tiny versatile system-on-chip with state-retentive eMRAM for machine learning inference at the extreme edge," in *Proc. IEEE Symp. VLSI Technol. Circuits*, 2022, pp. 20–21, doi: 10.1109/VLSITechnologyandCir46769.2022.9830409.

[15] B. Keller et al., "A 17–95.6 TOPS/W deep learning inference accelerator with per-vector scaled 4-bit quantization for transformers in 5nm," in *Proc. IEEE Symp. VLSI Technol. Circuits*, 2022, pp. 16–17, doi: 10.1109/VLSITechnologyandCir46769.2022.9830277.

[16] T. Tambe et al., "22.9 A 12nm 18.1TFLOPs/W sparse transformer processor with entropy-based early exit, mixed-precision predication and fine-grained power management," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2023, pp. 342–344, doi: 10.1109/ISSCC42615.2023.10067817.

[17] J. Wang, J. Lin, and Z. Wang, "Efficient hardware architectures for deep convolutional neural network," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 6, pp. 1941–1953, Jun. 2018, doi: 10.1109/TCSI.2017.2767204.

[18] Z. Liu, K.-T. Cheng, D. Huang, E. Xing, and Z. Shen, "Nonuniform-to-uniform quantization: Towards accurate quantization via generalized straight-through estimation," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Los Alamitos, CA, USA, 2022, pp. 4932–4942, doi: 10.1109/CVPR52688.2022.00489.

[19] K. Prabhu et al., "MINOTAUR: An edge transformer inference and training accelerator with 12 MBytes on-chip resistive RAM and fine-grained spatio-temporal power gating," in *Proc. IEEE Symp. VLSI Technol. Circuits*, 2024, pp. 1–2, doi: 10.1109/VLSITechnologyandCir46783.2024.10631515.

[20] S. Moon, H.-G. Mun, H. Son, and J.-Y. Sim, "Multipurpose deep-learning accelerator for arbitrary quantization with reduction of storage, logic, and latency waste," *IEEE J. Solid-State Circuits*, vol. 59, no. 1, pp. 143–156, Jan. 2024, doi: 10.1109/JSSC.2023.3312615.

[21] Z. Yuan et al., "STICKER: An energy-efficient multi-sparsity compatible

accelerator for convolutional neural networks in 65-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 55, no. 2, pp. 465–477, Feb. 2020, doi: 10.1109/JSSC.2019.2946771.

[22] Y. Qin et al., "A 52.01 TFLOPS/W diffusion model processor with inter-time-step convolution-attention-redundancy elimination and bipolar floating-point multiplication," in *Proc. IEEE Symp. VLSI Technol. Circuits*, 2024, pp. 1–2, doi: 10.1109/VLSITechnologyandCir46783.2024.10631322.

[23] X. Feng et al., "A 28-nm energy-efficient sparse neural network processor for point cloud applications using block-wise online neighbor searching," *IEEE J. Solid-State Circuits*, vol. 59, no. 9, pp. 3070–3081, Sep. 2024, doi: 10.1109/JSSC.2024.3386878.

[24] A. Parashar et al., "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2017, pp. 27–40.

[25] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, "SNAP: An efficient sparse neural acceleration processor for unstructured sparse deep neural network inference," *IEEE J. Solid-State Circuits*, vol. 56, no. 2, pp. 636–647, Feb. 2021, doi: 10.1109/JSSC.2020.3043870.

[26] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," 2016, *arXiv:1608.08710*.

[27] T. Chou et al., "NetFlex: A 22nm multi-chiplet perception accelerator in high-density fan-out wafer-level packaging," in *Proc. IEEE Symp. VLSI Technol. Circuits*, 2022, pp. 208–209, doi: 10.1109/VLSITechnologyandCir46769.2022.9830249.

[28] W. Tang et al., "Arvon: A heterogeneous system-in-package integrating FPGA and DSP chiplets for versatile workload acceleration," *IEEE J. Solid-State Circuits*, vol. 59, no. 4, pp. 1235–1245, Apr. 2024, doi: 10.1109/JSSC.2023.3343457.

[29] M. D. Rotaru, W. Tang, D. Rahul, and Z. Zhang, "Design and development of high density fan-out wafer level package (HD-FOWLP) for deep neural network (DNN) chiplet accelerators using advanced interface bus (AIB)," in *Proc. IEEE 71st Electron. Compon. Technol. Conf. (ECTC)*, 2021, pp. 1258–1263, doi: 10.1109/ECTC32696.2021.00204.

[30] B. Zimmer et al., "A 0.32–128 TOPS, scalable multi-chip-module-based deep neural network inference accelerator with ground-referenced signaling in 16 nm," *IEEE J. Solid-State Circuits*, vol. 55, no. 4, pp. 920–932, Apr. 2020, doi: 10.1109/JSSC.2019.2960488.

[31] Y. S. Shao et al., "Simba: Scaling deep-learning inference with multi-chip-module-based architecture," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 14–27, doi: 10.1145/3352460.3358302.

[32] R. Mahajan et al., "Embedded multi-die interconnect bridge (EMIB) -- A high density, high bandwidth packaging interconnect," in *Proc. IEEE 66th Electron. Compon. Technol. Conf. (ECTC)*, 2016, pp. 557–565, doi: 10.1109/ECTC.2016.201.

[33] C. Liu, J. Botimer, and Z. Zhang, "A 256Gb/s/mm-shoreline AIB-compatible 16nm FinFET CMOS chiplet for 2.5D integration with Stratix 10 FPGA on EMIB and tiling on silicon interposer," in *Proc. IEEE Custom Integr. Circuits Conf. (CICC)*, 2021, pp. 1–2, doi: 10.1109/CICC51472.2021.9431555.

[34] S. Naffziger, K. Lepak, M. Paraschou, and M. Subramony, "2.2 AMD chiplet architecture for high-performance server and desktop products," in *Proc. IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2020, pp. 44–45, doi: 10.1109/ISSCC19947.2020.9063103.

About the Authors

Wei Tang (weitang@umich.edu) received his B.S. degree from National Chiao-Tung University, Hsinchu, Taiwan, in 2011 and his M.S. and Ph.D. degrees in electrical engineering from the University of Michigan in 2019. He previously worked as a visiting Ph.D. at Lund University, Lund, Sweden, and as a graduate research intern at Intel Labs. He is currently an assistant research scientist with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA. His research interests include high-speed, energy-efficient, and flexible VLSI designs for communications, machine learning, and robotics. He is a Member of IEEE.

Sung-Gun Cho (sunggun@umich.edu) received his B.S. and M.S. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology, Daejeon, South Korea, in 2010 and 2012, respectively, and his Ph.D. degree in electrical and computer engineering from the University of Michigan, Ann Arbor, MI, USA, in 2020. From 2012 to 2015, he was with SK Hynix, South Korea, where he worked on SoC design and implementation of error control coding. In 2020, he joined the Intel Programmable Solutions Group CTO Office as an SoC design engineer to develop chiplets for various applications. He is currently with Google, Mountain View, CA 94043 USA. His research interests include energy-efficient high-performance design for signal processing, error control coding, and machine learning acceleration.

Jie-Fang Zhang (jfzhang@umich.edu) received his B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 2015 and his M.S. degree in computer science and engineering and Ph.D. degree in electrical and computer engineering from the University of Michigan, Ann Arbor, MI, USA, in 2018 and 2022, respectively. He joined Nvidia, Santa Clara, CA 95050 USA, in 2022 as a deep learning architect, focusing on GPU performance analysis, modeling, and optimization for deep learning models. His research interests include energy-efficient hardware architecture and accelerator design for machine learning, computer vision, and robotics applications. He is a Senior Member of IEEE.

Zhengya Zhang (zhengya@umich.edu) received his B.A.Sc. degree in computer engineering from the University of Waterloo, Waterloo, ON, Canada, in 2003 and his M.S. and Ph.D. degrees in electrical engineering from the University of California, Berkeley (UC Berkeley), Berkeley, CA, USA, in 2005 and 2009, respectively. He has been a faculty member with the University of Michigan, Ann Arbor, MI 48109 USA, since 2009, where he is currently a professor with the Department of Electrical Engineering and Computer Science. His research interests include low-power and high-performance VLSI circuits and systems for computing, communications, and signal processing. He was a recipient of the University of Michigan College of Engineering Neil Van Eenam Memorial Award in 2019, the Intel Early Career Faculty Award in 2013, the National Science Foundation CAREER Award in 2011, and the David J. Sakrison Memorial Prize from UC Berkeley in 2009. He has served on the technical program committee of the IEEE Custom Integrated Circuits Conference and the IEEE VLSI Symposium on Technology and Circuits. Additionally, he served as an associate editor for *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *IEEE Transactions on Circuits and Systems I: Regular Papers*, and *IEEE Transactions on Circuits and Systems II: Express Briefs*. He was also an IEEE Solid-State Circuits Society Distinguished Lecturer. He is a Senior Member of IEEE.

SSC