IMAGE LICENSED BY INGRAM PUBLISHING

Machine Learning Hardware Design for Efficiency, Flexibility, and Scalability

Jie-Fang Zhang, *Member, IEEE*, and Zhengya Zhang, *Senior Member, IEEE*

Abstract

The widespread use of deep neural networks (DNNs) and DNN-based machine learning (ML) methods justifies DNN computation as a workload class itself. Beginning with a brief review of DNN workloads and computation, we provide an overview of single instruction multiple data (SIMD) and systolic array architectures. These two basic architectures support the kernel operations for DNN computation, and they form the core of many flexible DNN accelerators. To enable a higher performance and efficiency, sparse DNN hardware can be designed to gain from data sparsity. We present common approaches from compressed storage to processing sparse data to reduce memory and bandwidth usage and improve energy efficiency and performance. To accommodate the fast evolution of new models of larger size and higher complexity, modular chiplet integration can be a promising path to meet the growing needs. We show recent work on homogeneous tiling and heterogeneous integration to scale up and scale out hardware to support larger models of more complex functions.

Index Terms—ML hardware, DNN accelerator, sparse DNN architecture, DNN chiplet, heterogeneous integration.

Digital Object Identifier 10.1109/MCAS.2023.3302390

Date of current version: 11 October 2023

I. Introduction

Deep neural network (DNN)-based machine learning (ML) methods have become the dominant way to solve problems in the fields of computer vision (CV), natural language processing (NLP), autonomous driving, and robotics [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11]. The effectiveness of the DNN-based methods leads to the proliferation of DNN models, from AlexNet [12] in 2012 for object detection and image classification to GPT-3 [7] in 2020 for natural language processing. In the quest towards higher accuracy and expanded capabilities, newer models often grow in size and require more memory and computation complexity. Fig. 1 presents the accuracy of modern network models along with their model size and complexity in terms of number of parameters and operation counts. The evolution of these models are shown in Fig. 2.

The widespread use of DNNs has made DNN computation a workload class of itself. General-purpose graphics processing units (GPUs) and central processing units (CPUs) equipped with large compute parallelism and memory bandwidth are popular hardware platforms for accelerating DNN workloads in servers and clouds, but GPUs and CPUs are not the most suitable for edge use cases due to their high cost and energy consumption. To fill the void, designing domain-specific accelerators

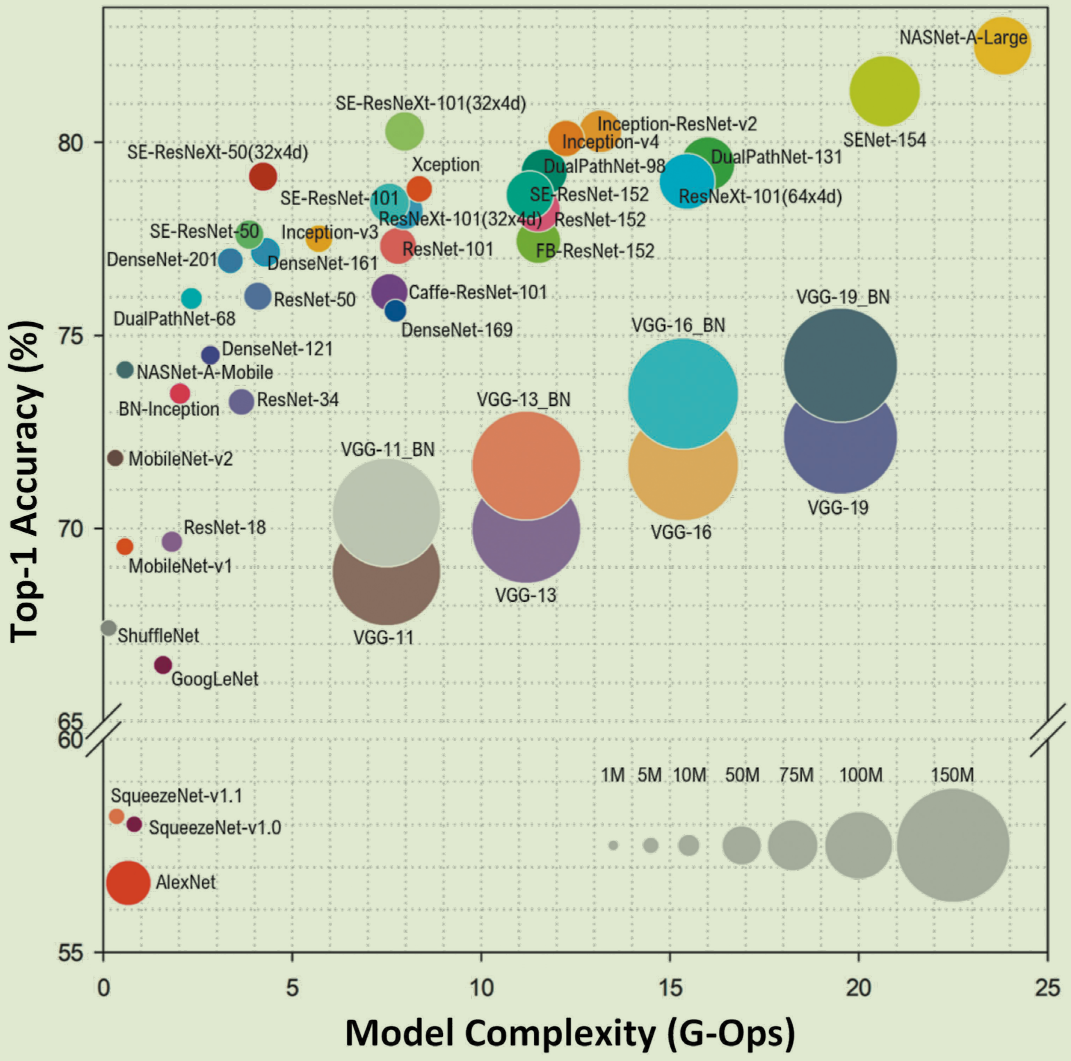

Figure 1. Top-1 accuracy, size, and complexity of modern DNN models. Adapted from [9] ©2018 IEEE.

Jie-Fang Zhang was with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA. He is now with Nividia Corporation, Santa Clara, CA 95051 USA (e-mail: jfzhang@umich.edu).
Zhengya Zhang is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: zhengya@umich.edu).

for DNN workloads is of importance to answer new application needs. A prime example of a domain-specific accelerator for DNN is Google's TPU [13].

Designing DNN-based ML accelerators has been a rapid-growing field. In general, we identify three major challenges that need to be addressed in designing these ML accelerators.

- *Flexibility:* The design needs to be flexible to support a variety of computation types and models in the DNN workload class, not only for the current generation of DNN models, but also for future generations as the models are evolving more quickly than hardware upgrades.
- *Efficiency:* The design needs to optimize both the processing and the memory access to provide a competitive advantage over GPUs and CPUs and answer new application needs.
- *Scalability:* The design needs to provide a way to support larger models with higher memory and computation requirements, and new variations of the current models to remain relevant.

In this review article, we discuss three important directions to address the computation challenges in supporting modern ML models and workloads. First, we describe the common processing architectures and the data reuse opportunities for ML computation. Then, we present the benefit of exploiting data-level sparsity to improve computation efficiency. Lastly, we provide an overview of scaling-up and scaling-out approaches to answer the scalability challenge.

This article is organized as follows. In Section II, we present the primary types of computation used in ML and DNN workloads. We then describe two common processing architectures for DNN computation and common stationary dataflows to exploit data reuse in Section III. To gain better performance and efficiency, an effective approach is by exploiting data sparsity, which is explained in Section IV along with examples of sparse compression formats and sparse architectures. To scale up designs and scale out its functionalities, a chiplet-based approach can be effectively employed. We review examples of homogeneous tiling and heterogeneous integration of chiplets in Section V to demonstrate promising recent results. Finally, we conclude this article in Section VI.

II. Background

In general, we can broadly categorize DNN models into four types based on its network structure and computation: 1) multi-layer perceptron (MLP), 2) convolutional neural network (CNN), 3) recurrent neural network (RNN), and 4) transformer. Here, we present the high-level structures of each model and explain its core computation.

A. Multi-Layer Perceptron (MLP)

An MLP consists of multiple feedforward fully-connected (FC) layers cascaded one after another. The computation of an FC layer can be formulated into a vector-matrix multiplication (VMM) between the input vector $x \in \mathbb{R}^C$ and the weight matrix $\boldsymbol{W} \in \mathbb{R}^{K \times C}$ to obtain the output vector $\boldsymbol{y} \in \mathbb{R}^K$, as described in Fig. 3(a).

B. Convolutional Neural Networks (CNNs)

CNNs are mostly specialized for 2D image processing in vision applications, e.g., image classification, object detection, and semantic segmentation [2], [3], [4], [12], [14], [15], [16], [17], [18]. A CNN uses convolution (CONV) layers for spatial feature extraction and FC layers for feature classification. The input and output are often referred as input activation (IA) and output activation (OA). A CONV layer has a weight (W) of size $R \times S \times C \times K$, which can be understood as K 3D kernels of $R \times S \times C$. A CONV layer processing takes an IA of size $H \times W \times C$ and performs 2D convolutions between the IA and the K 3D kernels to obtain an OA of size $H \times W \times K$, as shown in Fig. 3(b). The model hyperparameters C and K are the input and output channel sizes, respectively. The

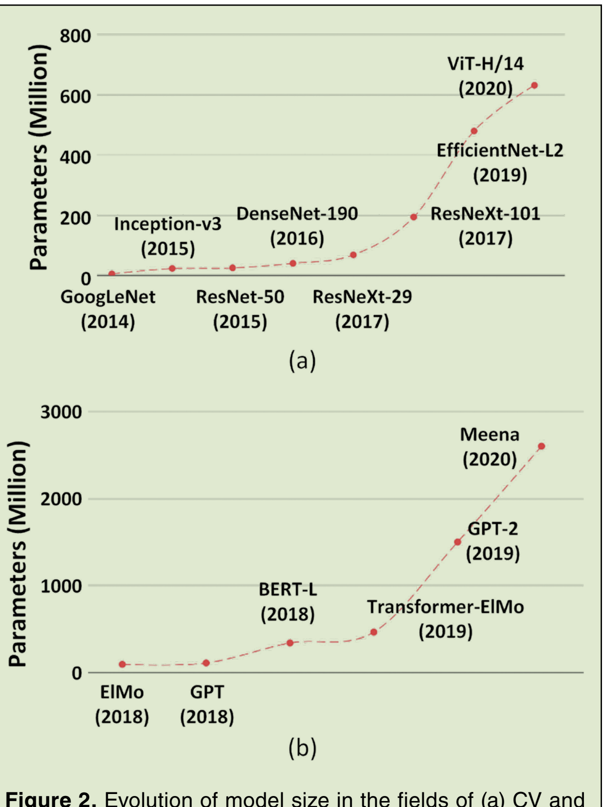

Figure 2. Evolution of model size in the fields of (a) CV and (b) NLP. Adapted from [11].

output channel size is also known as the (weight) kernel number.

The 2D convolutions can be viewed as drawing an $R \times S \times C$ cube inside the IA's $H \times W \times C$ volume, and sliding it across the IA's volume to obtain cubes of IA values. For each cube, the dot-product between the cube with each of the K $R \times S \times C$ kernels is performed to obtain K OA values. Therefore, each 2D convolution can be formulated as VMM with the $R \times S \times C$ IA cube reshaped to a vector, and the K $R \times S \times C$ kernels reshaped as a matrix of K vectors.

C. Recurrent Neural Networks (RNNs)

An RNN and its more popular variants, gated recurrent unit (GRU) and long short term memory (LSTM), are used for sequence processing in speech recognition, keyword detection, and natural language processing. An RNN uses recurrent connections to process the input sequence of the current timestep t and the output sequence from the previous timestep $t-1$. An LSTM uses input, output, forget gates, and a cell, i.e., $\boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \boldsymbol{c}$, to keep track of features that are relevant in long term and improves accuracy over traditional recurrent units. The computation of an LSTM can also be formulated into a VMM (Fig. 3(a)), where the input vectors are the input sequence $\boldsymbol{x}_t$ and the hidden sequence $\boldsymbol{h}_{t-1}$, the matrix is the concatenation of $\boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \boldsymbol{c}$ matrices with respect to the input or hidden sequences, and the output vector is the hidden sequence $\boldsymbol{h}_t$.

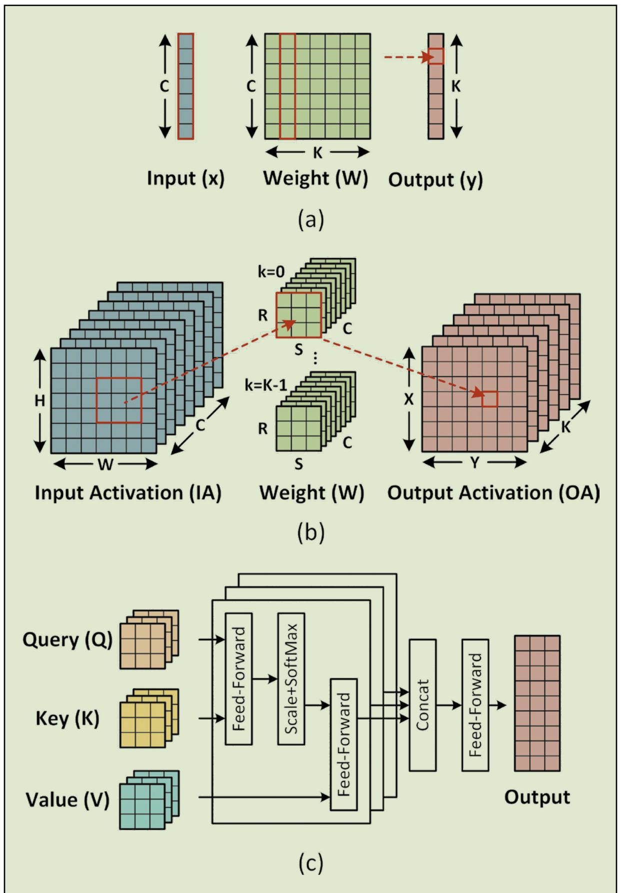

Figure 3. Core computations of DNNs: (a) vector-matrix multiplication in MLP and RNN, (b) 2D convolution in CNN, and (c) multi-head attention in transformers.

D. Transformers

Recently, transformer architectures that use self-attention and multi-head attention mechanisms are getting increasingly better performance compared to traditional LSTM in sequence and language applications [5], [6], [7] and CNN in vision applications [4]. The multi-head attention computation is described in Fig. 3(c). First, feed-forward operation, or matrix multiplication, is applied on the input sequence to obtain the key ($\boldsymbol{K}$), query ($\boldsymbol{Q}$), and value ($\boldsymbol{V}$) matrices with its weights, $\boldsymbol{W_K}, \boldsymbol{W_Q}$, and $\boldsymbol{W_V}$, respectively. The $\boldsymbol{K}, \boldsymbol{Q}, \boldsymbol{V}$ matrices are split into smaller matrices for multi-head attention. Each attention block of the multi-head attention performs the self-attention on its $\boldsymbol{K}, \boldsymbol{Q}, \boldsymbol{V}$ matrices. The outputs from each attention block are concatenated, then another feed-forward operation is applied to obtain the final output sequence. The whole computation can also be mapped into a series of matrix-matrix multiplication (MMM).

III. Classic DNN Processing Architectures and Dataflows

Single instruction multiple data (SIMD) and systolic array are the basic architectures for computing VMM and MMM. These two architectures and their variants form the core of most of the DNN accelerators. In the following, we review the two basic architectures and the common dataflows for performing the computation.

A. SIMD Architecture

In general, a single instruction multiple data (SIMD) architecture consists of an array of parallel processing elements (PEs) or functional units (FUs) and performs vector operations across an array of data. Only one instruction is decoded and issued to trigger the computation on multiple data across the array of PEs. Fig. 4(a) illustrates the SIMD architecture for vector processing. A SIMD array can be used to compute the dot-product between two data vectors. Each PE receives a pair of data from the memory or register file for multiplication, then the result from each PE is written back to the memory for the next summation instruction. Alternatively, the results may be directly summed using an adder tree.

The SIMD architecture is flexible and can be programmed to support diverse computation. In particular, we illustrate how the VMM, MMM, and CONV operations can be mapped onto a SIMD array for DNN processing.

VMM and MMM Operations: Fig. 4(b) shows the MMM operation between an input matrix and a weight matrix on a SIMD array. For an MMM operation, the vectors of the input and weight matrices are loaded into the SIMD array's memory to prepare for processing. During processing, corresponding input data and weight data pairs are accessed from the memory and dispatched to the PEs for processing. The input and weight vectors in the memory can be cached to exploit temporal locality throughout the entire computation. Similarly, for the VMM operation, the input vector is loaded and cached, and is paired across all weight vectors for processing.

CONV Operation: Fig. 4(c) shows the CONV operation between an $H \times W \times C$ input activation (IA) and an $R \times S \times C$ convolution weight (W) on a SIMD array. Due to the sliding window nature of the CONV operation, the IA and W are first converted into 2D matrices: 1) IA is converted to a $(X \times Y) \times (C \times R \times S)$ matrix, where each row matches the size of a $R \times S \times C$ weight kernel; and the IA values are partially replicated from one row to the next row to correspond to the sliding window from one step to the next step; and 2) W is converted to a $(C \times R \times S) \times K$ matrix, where each column corresponds to a weight kernel. Note the process of converting IAs and Ws in CONV into input and weight in MMMs, respectively, is often referred as Im2Col operation. An MMM operation can then be performed between the input and the weight matrix to produce the output matrix of size $(X \times Y) \times K$.

The advantage of a SIMD architecture is its flexibility and programmability to support diverse workloads. In general, for computation with straightforward data

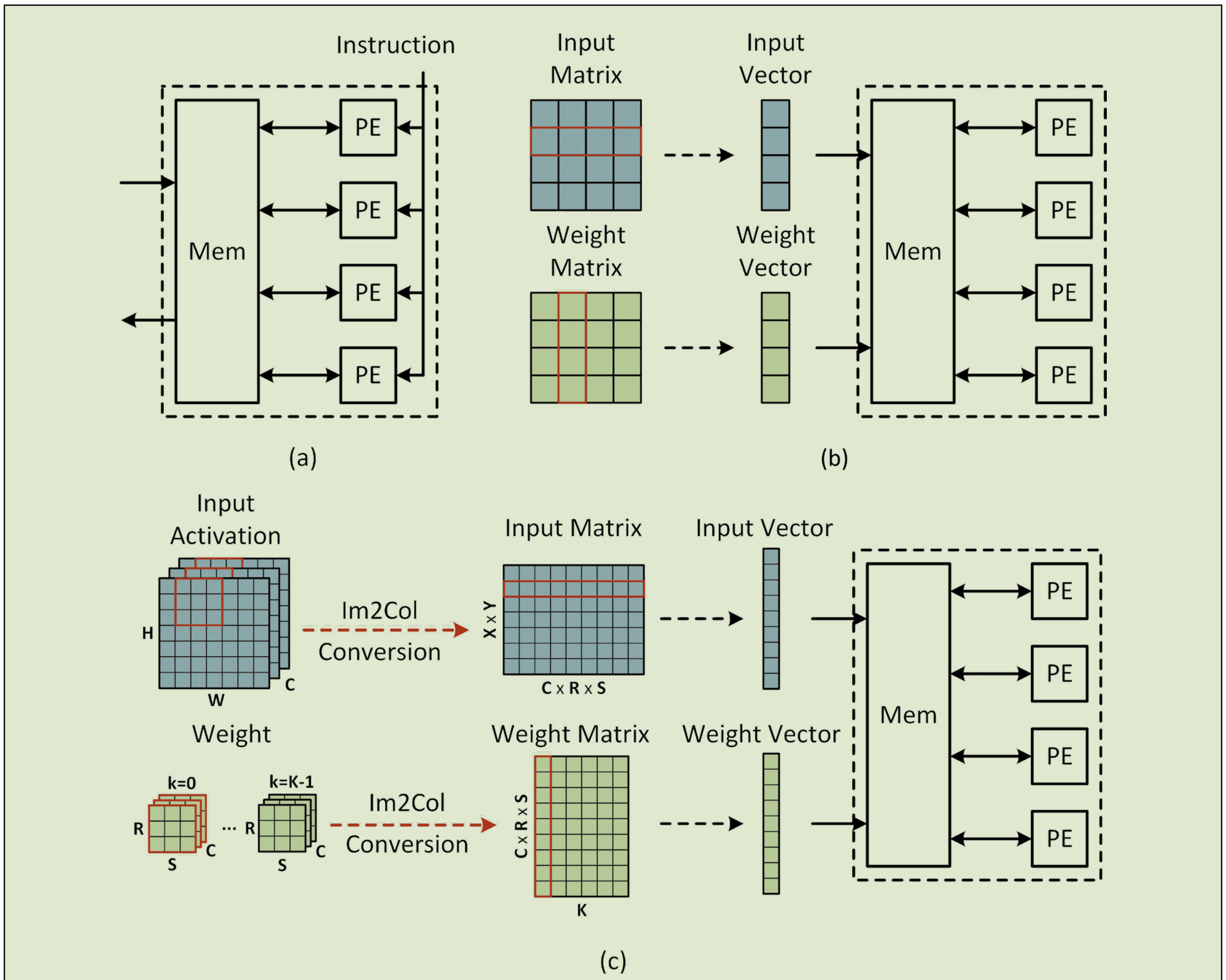

Figure 4. Illustration of the (a) SIMD array architecture, (b) matrix-matrix multiplication (MMM) operation, and (c) convolution (CONV) layer operations on SIMD array.

parallelism, e.g., VMM and MMM, a high compute utilization can be achieved. The SIMD architecture also provides opportunities to reuse weights or inputs by keeping them stationary at the PE's registers to reduce memory access. However, the memory size and bandwidth have to scale with the number of PEs in order to support the full vector processing.

In general, a 1D SIMD array can be tiled into a 2D SIMD array, where input and weight loading from the external memory can be shared between 1D tiles to reduce the bandwidth requirement while scaling up the total number of PEs for processing.

The SIMD architecture is used extensively in CPUs and GPUs [19], [20]. Fig. 5 shows a sub-partition of the streaming multiprocessor (SM) in Nvidia's A100 GPU [19]. The SM contains functional units (or CUDA cores) for arithmetic computation. In each cycle, an instruction is issued to a set of CUDA cores for parallel execution.

B. Systolic Array Architecture

A systolic array consists of a regular 2D array of PEs where each PE is connected to its immediate neighbors. Fig. 6(a) presents the architecture of the systolic array. The inputs are sent to the PEs through a PE array border, e.g., leftmost column, and the intermediate results are propagated across the PEs, e.g., horizontally to the right and vertically to the bottom. Finally, the output are sent out through another end of the PE array, e.g., bottom row.

A systolic array's PE microarchitecture and dataflow are illustrated in Fig. 6(b). A PE is commonly designed with a multiplier to compute the product of an incoming input and a cached weight value, and an adder to sum the computed product and an incoming partial sum (psum). The updated psum is sent vertically to the next PE down and the input is propagated horizontally to the next PE on the right.

To prepare for an MMM operation on a systolic array, the weights are first loaded to the array. The weight data are split into column vectors as shown in Fig. 7(a). Each vector is streamed to and stored in the corresponding column of the PE array column, as shown in Fig. 7(b).

The steps of an MMM operation are illustrated in Fig. 7(c)–(e). The input matrix is split into row vectors that are streamed sequentially to the PE array, as shown in Fig. 7(c). The inputs propagate from left to right, passing through one PE in a clock cycle. When an input enters a PE, the PE computes the product between the input and the cached weight, and sums the product with the psum that enters from top. Following the computation, the PE passes the input to the next PE on the right and the updated psum to the next PE down. Note that the inputs to the rows of PE must be arranged with a one cycle delay from one row to the next to ensure that the correct psum accumulation. Data move through the systolic array in waves. The wavefront propagates diagonally across the systolic array. The outputs are collected from the bottom row of PEs as shown in Fig. 7(e). The computation latency of a $H \times W$ systolic array is $H + W - 1$ cycles.

A systolic array allows efficient weight reuse. In a systolic array, the transfer of psums and inputs are restricted to efficient movements between neighboring PEs. Due to weight reuse and efficient data movements

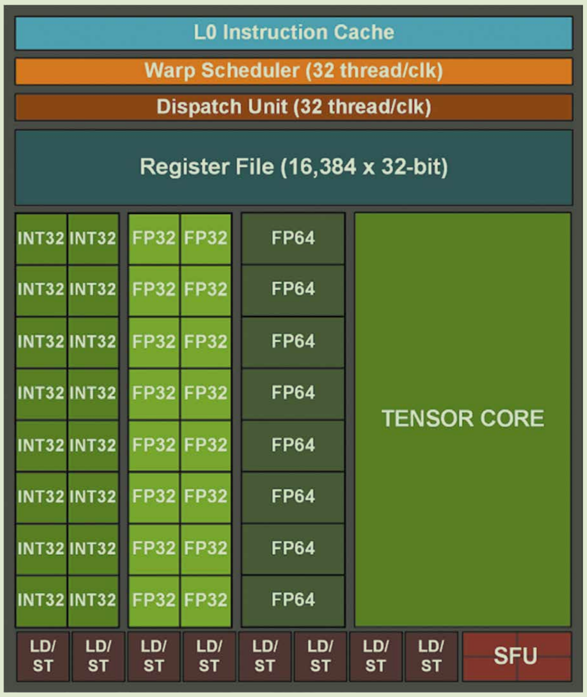
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU

Figure 5. Illustration of an example of SIMD architecture in Nvidia A100 GPU. Adapted from [19].

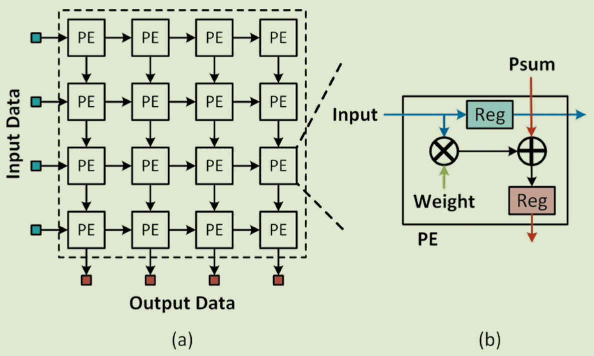

Figure 6. Illustration of the (a) systolic array architecture and (b) PE architecture in the systolic array.

within the array, a systolic array requires a lower data bandwidth. Systolic array is a data-driven architecture and has a low control overhead. These factors contribute to its high compute density.

Compared to a SIMD array, a systolic array is prewired for a defined dataflow and it is thus less flexible. A large-size systolic array provides a higher computation capacity, but it may also suffer from a low utilization when the operations do not utilize the entire array. The long latency is another drawback of a large-size systolic array.

TPU [13] is an example of systolic array. TPU is designed with a matrix multiply unit (MMU) that consists of a systolic array of 256 × 256 PEs. Fig. 8(a) and (b) show TPU's system architecture and dataflow in the MMU, respectively. The weight data are loaded from the weight FIFO into the MMU, and a systolic data setup module organizes the input data to ensure proper accumulation of the psums in the MMU. The MMU operates similarly to the description above where inputs are streamed in from the left to the right, and the psum accumulation happens vertically across columns.

Table 1 compares the SIMD architecture to the systolic array architecture. A SIMD array can be in the form of a 1D PE array to support VMM operations, and it can also be scaled up to a 2D PE array to support MMM operations. A systolic array is commonly designed as a 2D PE array to support MMM operations. In terms of data movement, a SIMD array needs to access memory to feed all its PEs, whereas a systolic array can rely on neighboring PE connections to reduce the bandwidth requirement and the number of memory accesses. A systolic array has a lower control overhead and can be easily scaled up. As such, a systolic array provides a

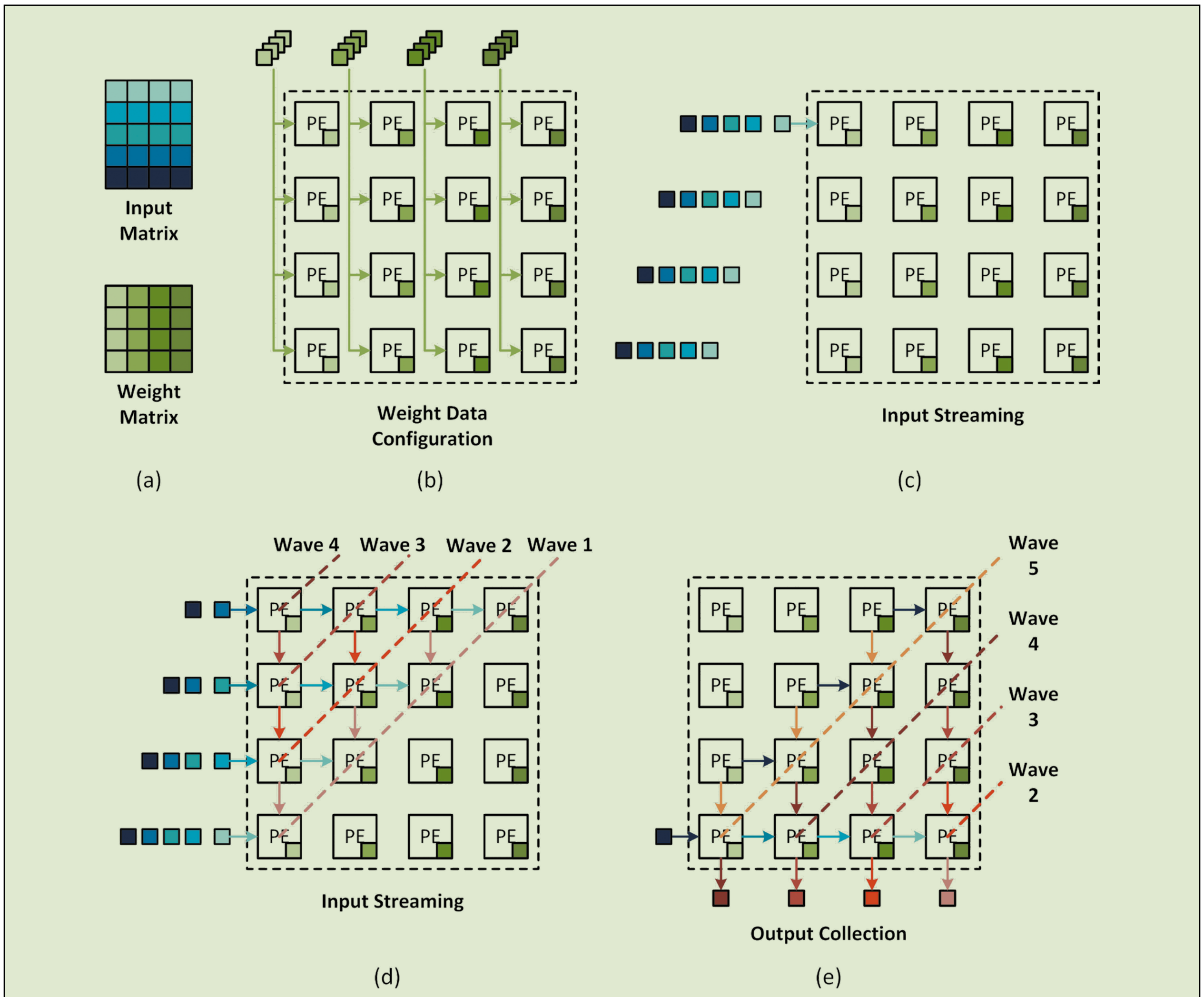

Figure 7. Illustration of the operations on systolic array: (a) input and weight matrices, (b) weight data configuration, (c) input streaming (early-stage), (d) input streaming (general), and (e) output collection.

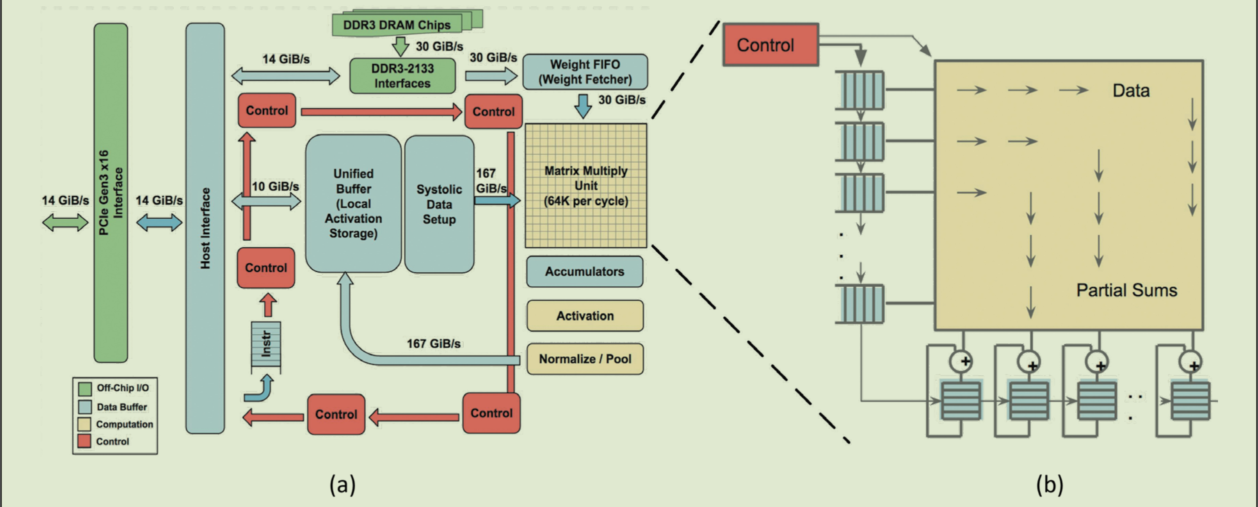

Figure 8. Illustration of the TPU (a) system architecture and (b) matrix-multiply engine architecture. Adapted from [13] ©2017 ACM.

Table 1. Processing architecture summary.

	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

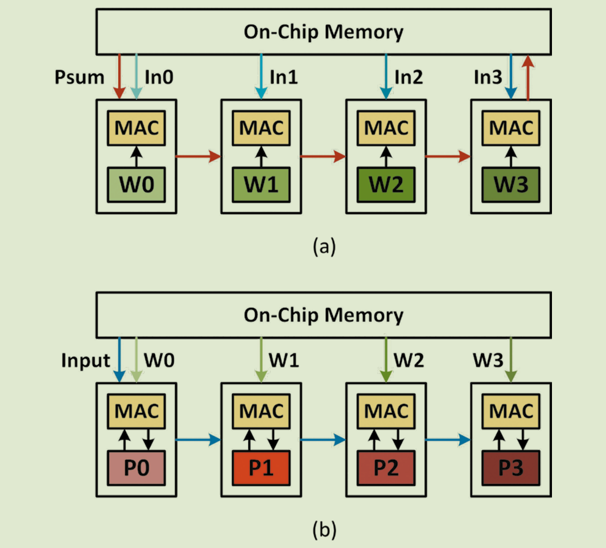

Figure 9. Illustration of (a) weight-stationary dataflow and (b) output-stationary dataflow. Adapted from [21] ©2017 IEEE.

higher compute density than a SIMD array. On the other hand, a systolic array is often designed for a fixed computation, whereas a SIMD array is more flexible to support a wide range of operations. The higher flexibility of a SIMD array over a systolic array leads to a higher hardware utilization, e.g., in flexibly handling inputs of various shapes.

C. Stationary Dataflows

DNN computation can be mapped onto the processing architectures in two common ways: weight stationary (WS) and output stationary (OS). These mapping methods provide data reuse opportunities and dictate the computation dataflows.

Weight Stationary (WS) Dataflow: In the WS dataflow, a PE stores a weight locally and reuses it for MAC computation with as many inputs as possible. The WS dataflow can effectively reduce the number of memory accesses required to fetch weights from memory, leading to a lower memory bandwidth and a lower power consumption. An example WS dataflow on a systolic array architecture is illustrated in Fig. 9(a). In the example, the weights (W_0, W_1, W_2, and W_3) are cached locally in the PEs. Inputs are accessed from memory and sent to the corresponding PEs for computation. The computed psums are passed along the PE array for accumulation. Lastly, the output data is written back to memory. The systolic array adopts the WS dataflow to reuse cached weights across all inputs. Jouppi et al. [13] is an example of ML accelerator that adopts the WS dataflow.

Output Stationary (OS) Dataflow: In the OS dataflow, a PE stores and accumulates a psum data locally. The OS dataflow can effectively reduce the amount of reading and writing of psums from and to the memory. An example OS dataflow mapped on a PE array is illustrated in Fig. 9(b). In this example, each PE accumulates a psum, $P_0, P_1, P_2,$ or $P_3,$ locally. In each cycle, new weights are fetched and sent to the PEs, and one input is broadcast across all PEs. Each PE computes a MAC and updates its local psum. Upon completion, the output data are written back from the PEs to memory. Du et al. [22] and Deng et al. [23] are examples of ML accelerators that adopt the OS dataflow.

IV. Sparse Architecture

The continued growth of model size and complexity has motivated research efforts in leveraging data sparsity to reduce the compute and storage requirements. In this section, we present an overview of network sparsity and how to exploit it to make more efficient processing.

A. Sparsity in Neural Networks

The sparsity in a network comes from both the model's weights (Ws) and input activations (IAs). For the model weights, network pruning and other sparsification techniques can be used to zero out a large number of weights in a model with only a small inference accuracy drop [24], [25], [26], [27], [28], [29]. For the input activations, some commonly-used operators like rectifier linear unit (ReLU) can clamp all negative activations to zeros, resulting in sparsity in output activations (OA), which become input activations (IA) of the next layer.

A CONV computation with IA and W sparsity is illustrated in Fig. 10. With network pruning [24], the typical W density (nonzero data over all data) ranges from 40% to 50% and the IA density ranges between 30% and 55% for well-known models, e.g., AlexNet, VGG-16, and ResNet-50 [30]. An up to 38% and 4% density for IA and W, respectively, is achieved by [24] on the FC layers of VGG-16. The CONV layers can be pruned down to 19% and 22% density for IA and W, respectively. Zhang et al. [26] reported a 95% W sparsity on AlexNet using ADMM. For an IA and a W with 50% density each, because the nonzero W and IA are nearly randomly distributed, the amount of effectual computation, i.e., computation that does not involve a zero, is only about 25%.

There are multiple benefits by exploiting sparsity in designing DNN compute. First, data sparsity can be exploited to save power. Accelerators e.g., Eyeriss [32] gate the computation, e.g., by turning off the clock, whenever a zero in the IA is detected during processing. This technique can effectively reduce the power consumption during DNN processing and can be conveniently incorporated into existing dense DNN accelerators. However, the throughput remains the same since PEs become idle during ineffective computation.

Second, data sparsity can be used to reduce off-chip memory storage and bandwidth usage. The sparse W and IA can be stored in a compressed format with only nonzero elements. They are loaded and decompressed for computation. The compressed storage reduces the storage size and memory bandwidth. However, the decompression can be difficult to parallelize and costly in power and area, leading to a bottleneck and additional overhead for DNN processing.

Lastly, data sparsity can be used to reduce latency by skipping the ineffectual computation. During processing, IA-W pairs are identified by searching through the sparse IA and W data and sent to the compute. The search step avoids wasting time on unnecessary computation, resulting in significant latency savings. State-of-the-art sparse DNN accelerators [31], [33], [34], [35], [36] process data directly in the compressed form, offering both low memory bandwidth and high degree of acceleration. However, supporting sparse processing can cost a high design complexity.

B. Sparse Compression Format

Sparse compression formats are used to store sparse data in compact ways to save storage space. A compressed format contains only nonzero data values and metadata to hold the information for locating the positions of nonzero values in the uncompressed vectors and matrices. During processing, the metadata is decoded to

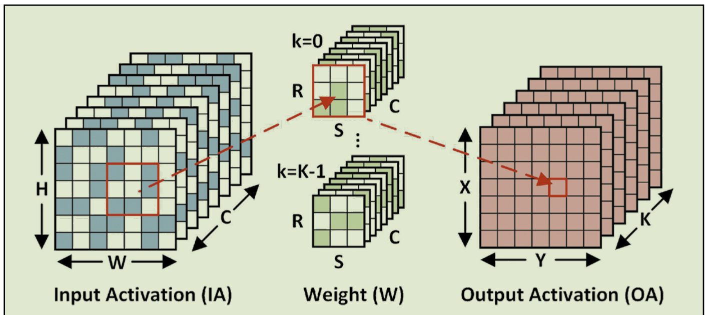

Figure 10. Convolution computation between unstructured sparse IA and W in a sparse DNN. The colored cells indicate nonzero entries, and the white cells indicate zero entries. Adopted from [31] ©2021 IEEE.

obtain the input address for fetching the nonzero data in the compressed format and to calculate the output address for writing back the computed result. Different sparse compression formats have different requirements in terms of total storage size (including nonzero data and metadata) and decoding complexity. Here, we present and discuss some common sparse compression formats used for sparse neural network processing.

Coordinate List (COO): In the COO format, nonzero data are stored along with their absolute indices in the original uncompressed vector or matrix. Fig. 11(a) shows an example of a matrix with zero and nonzero data. The COO format of the matrix stores all nonzero data in a 1D value array and records the (row, column) indices of each nonzero data, as shown in Fig. 11(b). The advantage of COO is its low decoding complexity, since the row and column indices can be directly used to locate the positions of the nonzero data in the uncompressed vector or matrix. However, the row and column indices may require significant amount of storage overhead which makes COO less efficient for data of medium or low sparsity.

Compressed Sparse Row (CSR): In the CSR format, nonzero data are stored first by row, then by column in a 1D value array. Different from COO, the metadata consists of a pointer (Ptr) array and a column index array. The Ptr array stores the row-by-row count of the total number of nonzero data. The first entry Ptr[0] is always 0; the second entry Ptr[1] stores the count of nonzero data in the first row; and Ptr[2] stores the count of nonzero data in the first two rows, etc. The column index array stores the column index of each nonzero data. Fig. 11(c) shows the CSR format of our matrix example. The CSR format requires a two-step decoding process. For instance, to access data in Row 1, the two steps are: 1) obtain the positions of the nonzero data of Row 1 stored in the value array: $\text{Ptr}[1], \text{Ptr}[1]+1,$ and so on and 2) obtain the column indices of the nonzero data in Row 1: Index[Ptr[1]], Index[Ptr[1]+1], and so on.

Compressed Sparse Column (CSC): The CSC format is similar to the CSR format, but nonzero data are first stored by column, then by row in a 1D value array. Fig. 11(d) shows the CSC format for our matrix example, where the Ptr array stores the column-by-column count of the total number of nonzero data and the row index array stores the row index of each nonzero data. The CSC format shares the same advantages and disadvantages as the CSR format.

Run-Length Coding (RLC): In the RLC format, nonzero data are stored in a 1D value array in either row major or column major, and a run array keeps track of the number of zeros before each nonzero data (known as the "run length"). Fig. 11(e) shows the RLC format of our matrix example using 2-bit run lengths. In this example, nonzero values a, b, c, and d are stored in the 1D value array, and they have 1, 3, 0, and 3 preceding zeros or run lengths, respectively, that are recorded in the run array. Note that the nonzero data e has four preceding zeros, which exceeds the two bits allocated to a run length. Therefore, an additional padding zero is inserted before e with a run length of 3. The RLC format can be decoded in one step. The position of *i*-th nonzero data in the value array can be calculated by accumulating all preceding run lengths in the run array.

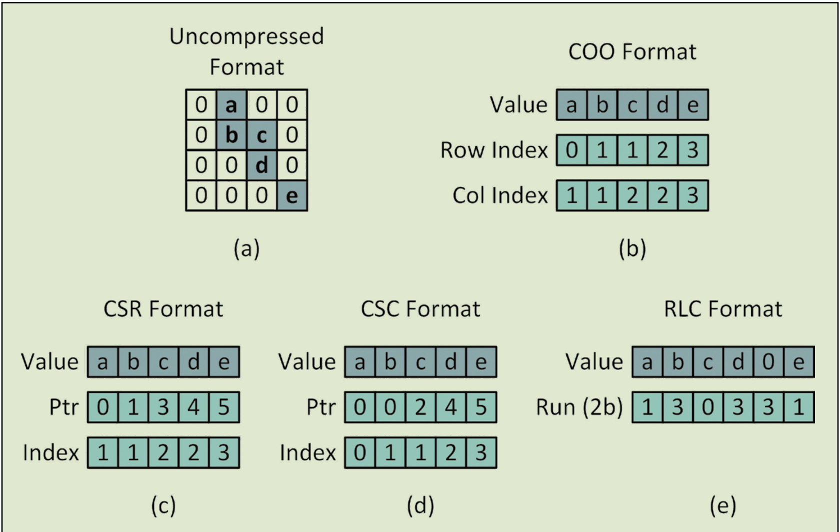

Figure 11. Examples of sparse compression formats: (a) sparse uncompressed tensor, (b) COO format, (c) CSR format, (d) CSC format, and (e) RLC format with a run of 2-bit.

C. Sparse Computation Pipeline

The high-level computation pipeline of sparse DNN processing in the compressed format is illustrated in Fig. 12. Following the computation pipeline, nonzero data and metadata arrays of Ws and IAs are first fetched on-chip for processing. The compressed W and IA pairs are then searched, paired and dispatched to a multiplier array for computation in the so-called frontend part of the pipeline. Finally, the computed psums are accumulated and written back to their respective OAs in output buffers in the so-called backend part of the pipeline.

The challenges of processing sparse data are two folds: 1) at the

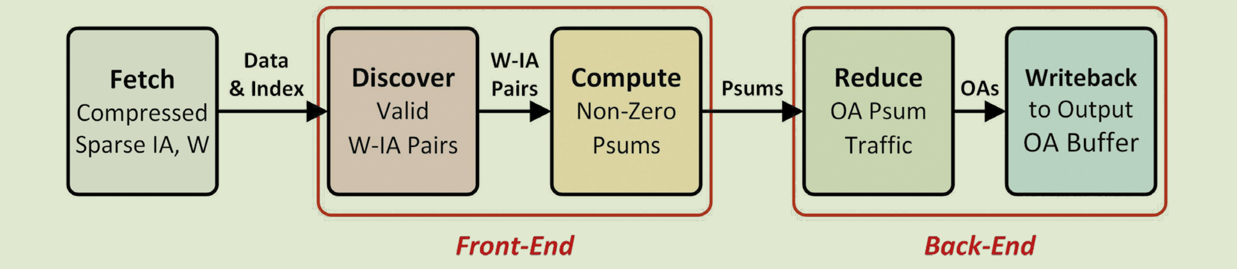

Figure 12. Processing pipeline of a sparse DNN processor. Adopted from [31] ©2021 IEEE.

front end, a sufficient number of IA-W pairs must be discovered and sent to the compute stage in order to maintain a high compute utilization and 2) at the backend: the irregular psum traffic out of the compute stage must be reduced and written back to the output buffer without costing excessive bandwidth.

We provide a high-level overview of the hardware design techniques and explain how they leverage sparsity in the following three subsections.

D. Single-Operand Sparsity

Some of the earliest sparse architectures leveraged sparsity from either IA, e.g., Cnvlutin [37], or W, e.g., Cambricon-X [38], but not both. By limiting the support to single-operand sparsity, these designs could adopt an existing dense DNN accelerator architecture and dataflow [39], and add a frontend to discover IA-W pairs for computation. Fig. 13 shows the frontend designs for Cnvlutin [37] and Cambricon-X [38]. Both used indirect access to fetch dense data (W in Cnvlutin, IA in Cambricon-X) using the indices of nonzero data (IA in Cnvlutin, W in Cambricon-X) decoded from the compressed format.

Cnvlutin supports IA sparsity, where the IA data are compressed in the COO format, as illustrated in Fig. 13(a). For each nonzero IA data, an IA offset is stored to represent the original location of the IA data in the uncompressed format. To discover IA-W pairs, the IA offset is used as the index to fetch W data from the W data array.

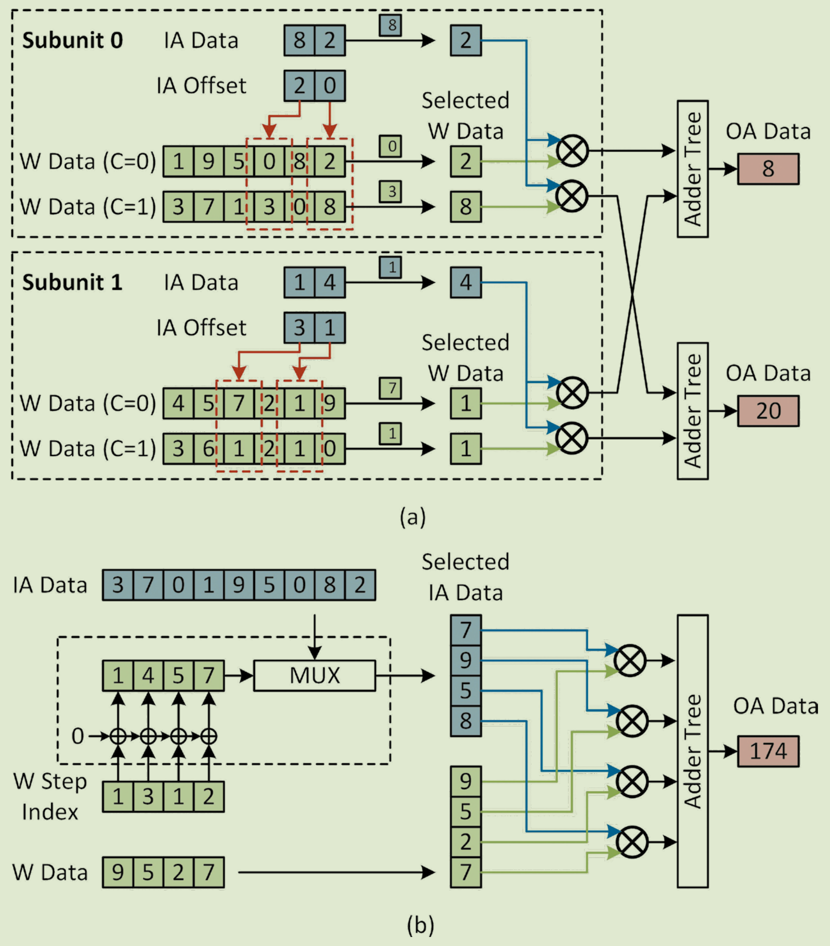

Figure 13. Sparse architectures for single operand sparsity: (a) Cnvlutin adapted from [37] and (b) Cambricon-X adapted from [38].

Cambricon-X supports W sparsity, where the W data are compressed in the RLC format. For each W data, a W step index stores the number of zeros preceding it, i.e., the run length, as shown in Fig. 13(b). To discover IA-W pairs, the run lengths are accumulated to recover the indices of the W data in the uncompressed format. The recovered indices are used to select the corresponding IA data to form IA-W pairs for computation.

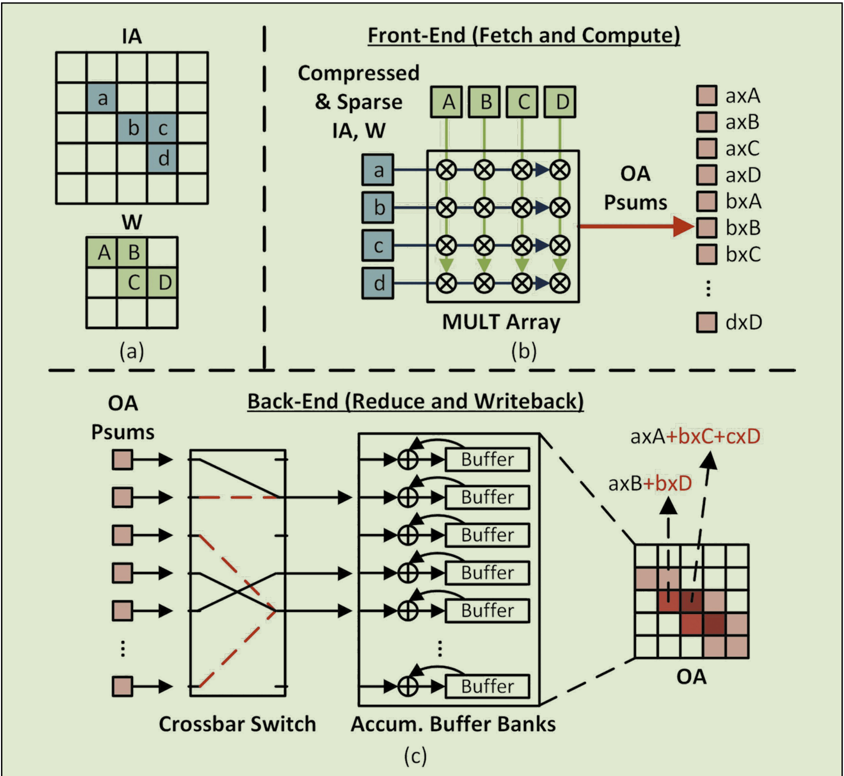

Figure 14. Illustration of channel-last dataflow for sparse DNN processing. (a) IA and W data in dense format, (b) front-end dataflow, and (c) back-end dataflow of channel-last processing. Adopted from [31] ©2021 IEEE.

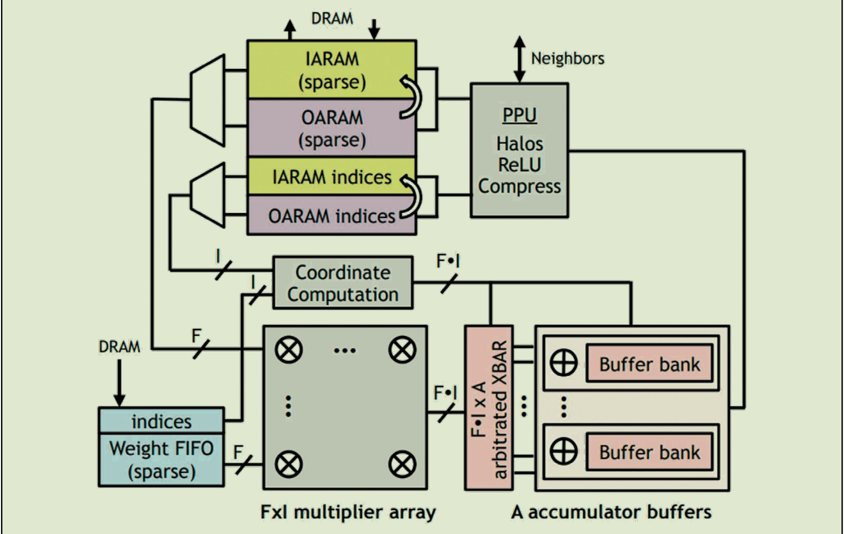

Figure 15. Architecture of SCNN, adopted from [34] ©2017 ACM.

E. Full Sparsity—Channel-Last Processing

There are generally two ways to handle full sparsity, i.e., sparsity in both W and IA: channel-last processing or channel-first processing. In this subsection, we will cover channel-last processing, an example of which is SCNN [34].

The channel-last dataflow is illustrated in Fig. 14. In the channel-last processing, the nonzero W and IA data are ordered in the $(R,S)/(H,W)$ dimension first and C dimension last for compressed storage and processing. Subsequently, as compressed W and IA data are fetched for processing, their channel indices are easily aligned. As long as a nonzero W's and a nonzero IA's channel indices are matched, they can be paired for multiplication.

Shown in Fig. 14(a) and (b), the compressed W and IA data of the same channel index can be cross paired and multiplied together using a 2D multiplier array. The advantage of the channel-last processing is the simple frontend, but the drawback is the complicated writeback because the OA addresses of the psums depend on the $(R,S)/(H,W)$ indices of the IA/W data, which are irregular for sparse data. There is little opportunity to reduce the psums before writeback, resulting in writeback traffic jam. It requires complex hardware or wiring, e.g., a crossbar switch, to resolve the contention, and it may cause pipeline stalls.

This backend challenge is illustrated in Fig. 14(c). The psums need to be distributed by a switch to the corresponding buffer bank. The red lines indicate the psum writebacks that lead to buffer contentions. To avoid contentions, conflicting psums need to be held. In the example, one output requires the accumulation of three psums, resulting in a three-cycle writeback and stalling the multiplier array for two cycles.

Fig. 15 shows the architecture of SCNN that adopts channel-last processing for sparse DNN processing.

F. Full Sparsity—Channel-First Processing

In the channel-first processing, the nonzero W and IA data are ordered in the C dimension first and $(R,S)/(H,W)$ dimension last. As compressed W and IA data are fetched, their channel indices are first matched to produce pairs of W and IA data to be multiplied together. Strings of resulting psums will share the same OA address, so they can be reduced before writeback. Compared to the channel-last processing, the channel-first processing incurs an overhead in the frontend due to the channel index matching, but it produces immediately-reducible psums to cut the writeback traffic, leading to more gain from simplifying the backend and a potential net improvement in the overall power and performance.

The channel-first dataflow is illustrated in Fig. 16. The W channel index is matched with the IA channel index to generate valid W-IA pairs. Valid W-IA pairs are fetched and multiplied to produce psums. The psums are to be accumulated to the OA address following the IA indices (h,w) and the W indices (r,s,k). Due to the channel-first input ordering, the (h,w) and (r,s) addresses will increment less frequently than the input channel index over the course of processing, allowing the OA address to stay constant for the majority of the time and the psums can be immediately accumulated before writeback.

An example of channel-first processing is SNAP [31]. SNAP utilizes associative index matching (AIM) units in the frontend to extract IA-W pairs for multiplication, as shown in Fig. 17. The AIM consists of a comparator array and each row is connected to a priority encoder. During operation, an AIM receives the W and IA channel index arrays and compares each W channel index to every IA channel index as shown in Fig. 17. A priority encoder encodes the match result in each row into a valid bit to indicate a match and the matched position in the IA channel index array. Upon completion, an AIM returns a list of valid-position pairs for processing.

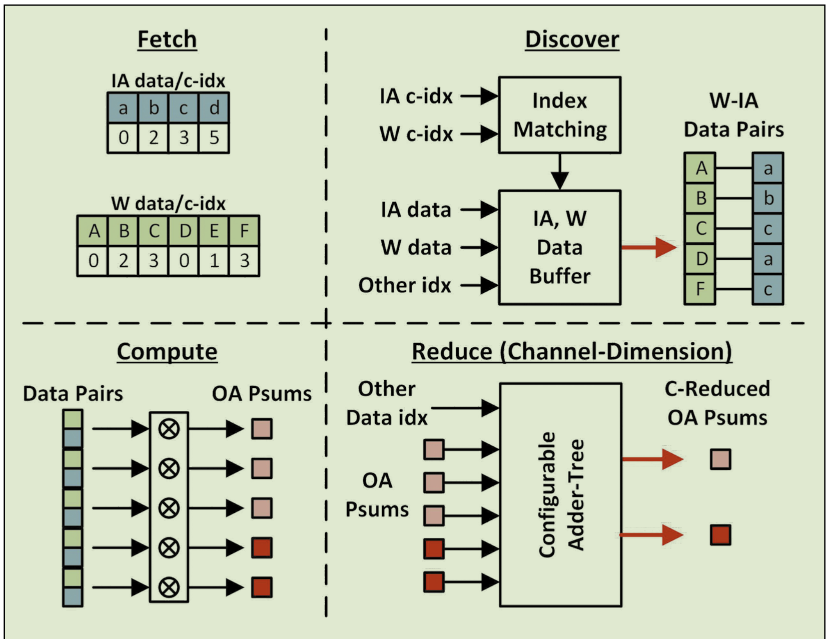

Figure 16. Illustration of channel-first dataflow for sparse DNN processing. Adapted from [31] ©2021 IEEE.

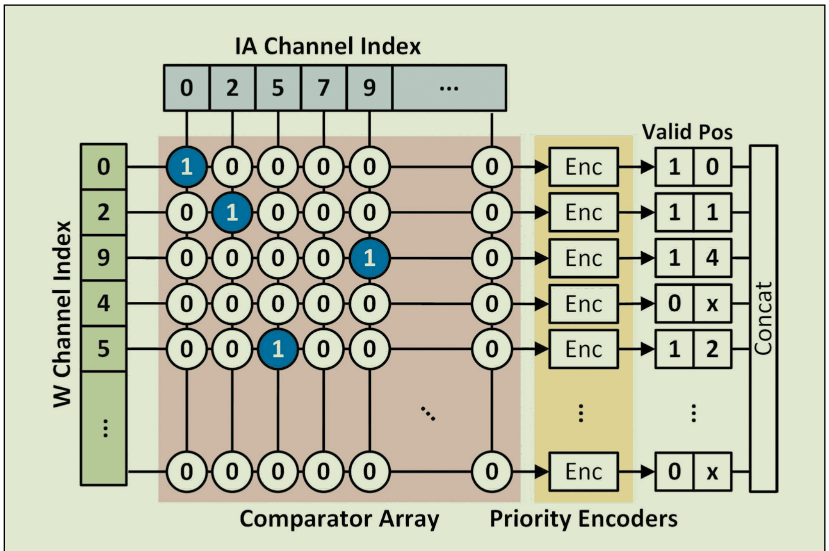

Figure 17. The associative index matching (AIM) unit in SNAP. Adopted from [31] ©2021 IEEE.

G. Structured Sparsity

Making use of full available sparsity can cost substantial hardware overhead. As a compromise, we can use a limited form of sparsity, such as coarse-grained or structured sparsity, that can provide a good enough gain in performance and efficiency without excessive hardware overhead [27], [40].

Different forms of sparsity are compared in Fig. 18. If pruning [24] is done without any constraints, it results in unstructured sparsity shown in Fig. 18(a). Pruning without any constraints generally produces a higher sparsity, but processing unstructured sparsity requires more fine-grained control and can cost excessive hardware overhead. Pruning can be done by block with a density upper bound. The approach produces density-bounded block sparsity [19], [41]. For example, Fig. 18(b) shows the result of a density-bounded block pruning with each 1×3 block containing at most one nonzero value. Pruning can be done by block [42], [43], e.g., by 2×2 blocks as shown in Fig. 18(c). Pruning can even be done by input and output channel [40], [44], [45] as shown in Fig. 18(d). More coarse-grained pruning produces more hardware-friendly structured sparsity, but it may sacrifice the model accuracy to some degree.

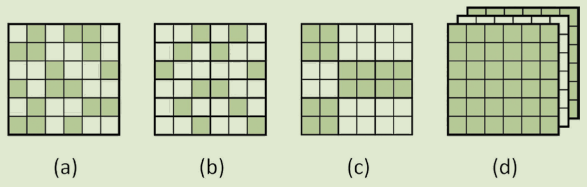

Figure 18. Common sparsity types: (a) fine-grained, (b) density structured, (c) block structured, and (d) filter structured.

One well-known example that leverages the density-bounded block sparsity is Nvidia A100 GPU [19]. As illustrated in Fig. 19, fined-grained structured pruning is applied to the trained model weights to create the so-called 2:4 sparsity, i.e., a 50% density bound for each block of 1×4 data. The sparse weights are compressed with COO indices that are used to access the dense inputs in processing, similar to the illustration in Fig. 13(b).

H. Bit-Level Sparsity

Besides sparsity at data level, bit-level sparsity can also be leveraged by bit-serial multipliers. One example that adopts this approach is bit-pragmatic [46], where the zero bits in one of the operands can be skipped in bit-serial multiplication. The bit-pragmatic processing is illustrated in Fig. 20 [46]. The IA is processed in a bit-serial fashion, and each nonzero bit is encoded by its position in the bit sequence similar to the COO format. In computation, the nonzero bit position of each IA data is used to set the configurable left shifter to shift the W data value, effectively acting as a bit-wise multiplier. Exploiting sparsity in bit-level reduces the number of computation cycles, and can increase both efficiency and throughput.

I. Sparse Architectures for RNNs and Transformers

Compared to the sparse architectures for CNNs, the sparse architectures for RNNs are focused on improving the performance for sparse matrix vector multiplication (SpMV) and the element-wise operations associated to

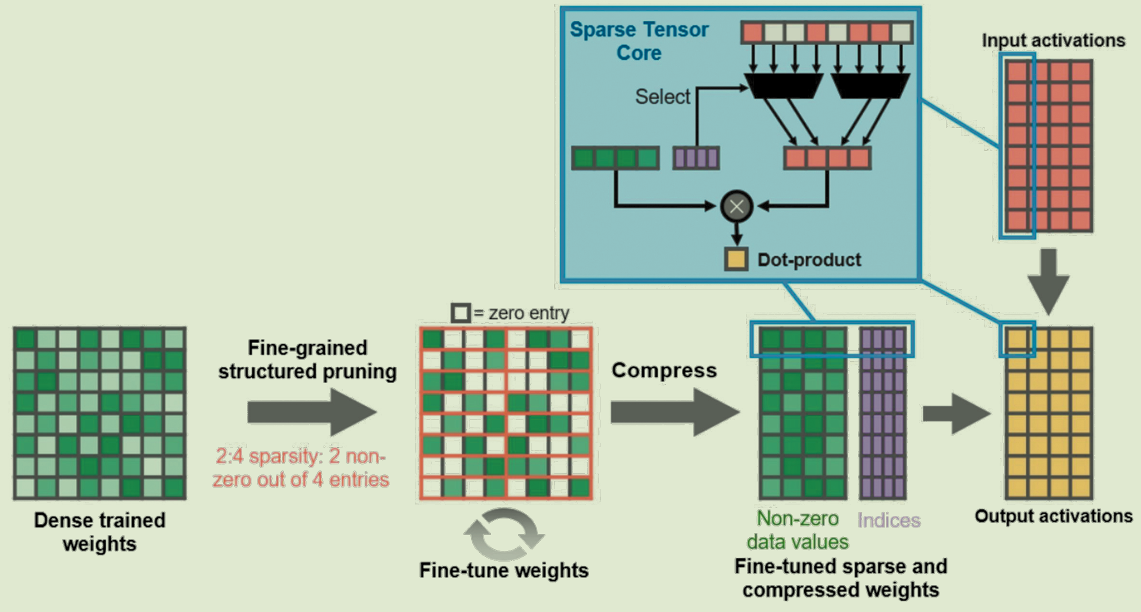

Figure 19. Processing mechanism of Nvidia A100 GPU for fine-grained structured sparse model weights. Adopted from [19].

the type of RNNs. For instance, LSTM requires element-wise multiplication, sigmoid, and tanh operations to compute the outputs. ESE is an example of a sparse architecture for LSTM [47]. It proposes a load-balancing pruning technique to reduce the workload imbalance in the sparse inputs and weights during pruning. Similar to EIE [33], ESE adopts the CSC format to store and compute the sparse data.

Different from the pruning techniques that eliminate the unimportant weights and inputs in CNNs to RNNs, the sparse architectures for transformers proposed to prune the unimportant connections (tokens or heads) in the self-attention matrix [48], [49]. Fig. 21 presents an example of the attention matrix. Several tokens have small contributions to the final result, thus can be pruned away without performance degradation. Spatten [48] proposed cascade head and token pruning techniques to eliminate the tokens and heads in the attention matrix. It uses a shifting mechanism to avoid irregular memory access from the sparse computation and a reconfigurable adder-tree to leverage the sparsity for speedup. DOTA [49] trains a decoder side by side to the Transformer to detect the weak connections in the attention matrix. To process the sparse attention matrix, DOTA adopts an out-of-order processing scheme to leverage the temporal locality and avoid unnecessary memory accesses.

V. Scale Up and Scale Out

The DNN model complexity grows at 1.5 times annually [8], [9], [29], [50], but it is unlikely to expect new custom chips to be built to respond to the rapid evolution of DNN models at the same rate. This lag is attributed to the high cost and effort to design new chips, especially ones that utilize large silicon area and advanced technology nodes needed to support the processing of more complex DNN models. Other important factors include the diverse use cases of DNN that diminish the space for custom chips, and the rapid evolution of DNN models that shortens the useful life of such custom chips.

Domain-specific accelerators for DNN, such as NVDLA [51] and TPU [13], represents a path forward by providing some degree of flexibility to support not only current models but also future models. However, without growing the raw compute and memory capacity, the performance of such accelerator will not be able to meet the demands of newer and more complex models.

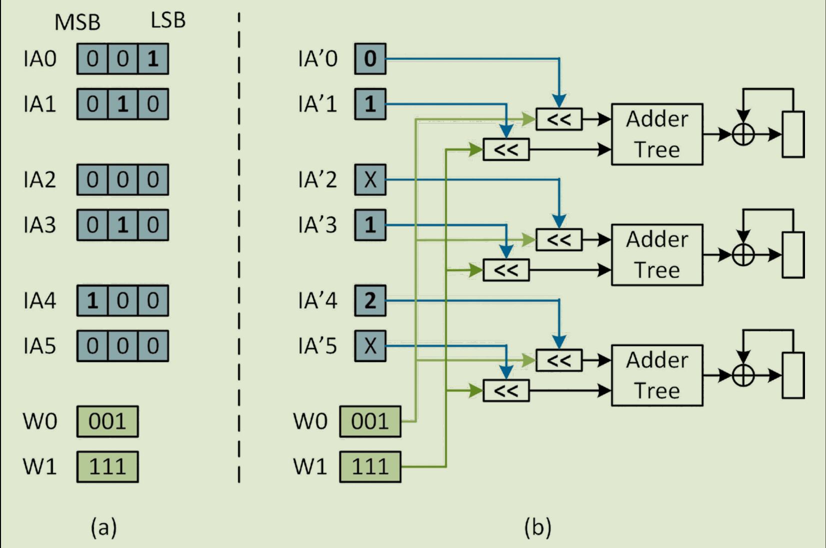

Figure 20. Frontend mechanism and processing example of bit-pragmatic [46]: (a) IA and W data for processing and (b) bit-serial processing using IA's nonzero bit position to control the shifter.

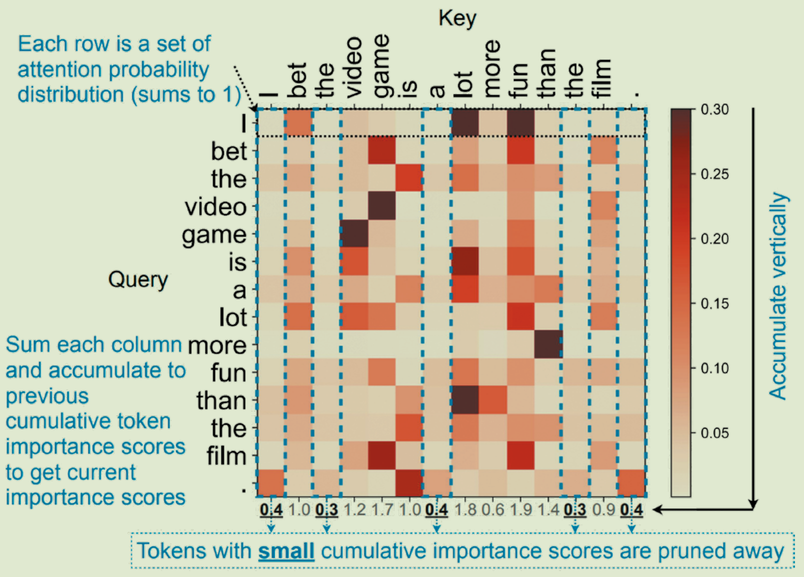

Figure 21. Illustration of the attention matrix with unimportant tokens. Adopted from [48].

Therefore, these domain-specific accelerators also need to be continuously upgraded, e.g., from the 28 nm TPUv1 in 2016 [13] to the 7 nm TPUv4 in 2021. Similar challenges in high cost and limited lifespan still remain.

We identify a growing trend to emphasize on hardware reuse and leverage advanced packaging to enhance the capability of hardware systems. Using this approach, a chip is designed to be a modular building block, called chiplet; and a system is constructed by reusing chiplets. To meet the requirements of different DNN models and use cases, systems can be constructed using the suitable number and types of chiplets. In other words, this approach takes advantage of chiplet reuse to construct systems in package (SiP). The premise of this approach is that designing, fabricating and assembling packages require lower cost and effort than designing and fabricating large monolithic chips.

For the SiP approach to succeed, we identify three basic requirements: 1) availability of reusable chiplets that are equipped with high-bandwidth and efficient I/O interfaces; 2) accessible advanced packaging and assembly process; and 3) methodology to map workloads to chiplet-based systems. Among the three requirements, a high-bandwidth and efficient I/O interface is necessary to ensure that the chiplets that constitute an SiP can be seamlessly integrated to match the performance of a monolithic chip; an accessible advanced packaging and assembly process ensure that high-density integration and high-bandwidth routing are feasible to construct an SiP at a reasonable cost; and a mapping methodology is needed to divide the workload and assign them appropriately to the chiplets to achieve high utilization and efficiency.

In the following, we use two recent designs as examples to outline the primary ways in constructing SiP for DNN compute acceleration. We categorize them into two classes, homogeneous integration and heterogeneous integration. In homogeneous integration, same chiplets are tiled to scale up the system to support models of larger size. In heterogeneous integration, different types of chiplets are put together to extend the functionality to cover new types of workloads.

A. Homogeneous Integration

The best example of homogeneous integration is Nvidia's DNN multi-chip package (MCP) shown in Fig. 22, where up to 36 DNN chiplets can be integrated in one MCP to scale up the system as needed [52]. The DNN chiplet measures 6 mm^2 in a TSMC 16 nm technology. It integrates tiles of SIMD-based PEs to provide up to 1,024 MACs/cycle (INT8) or 4 TOPS (INT8) [52].

Nvidia's DNN MCP is built on a 12-layer organic substrate. Organic substrate is generally of lower cost than substrates used for advanced packaging such as silicon interposers, but the routing density is generally lower too. Nvidia's DNN MCP adopts a serial link approach to achieve a high inter-chiplet bandwidth using fewer wires at very high speed, suitable for organic substrate. In particular, the Nvidia design used a 200 mV low-swing, short-reach serial link called ground-referenced signaling (GRS) to achieve up to 25 Gbps/lane at 0.82–1.75 pJ/b for a short reach of 3–7 mm [53]. A chiplet is equipped with four transmit lanes and four receive lanes for up to 100 Gbps of input and 100 Gbps of output bandwidth [52].

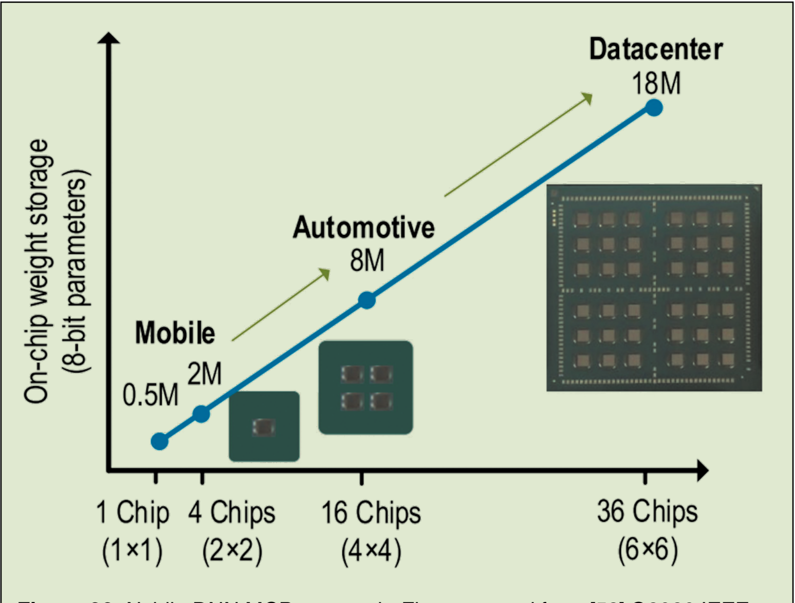

Figure 22. Nvidia DNN MCP approach. Figure reused from [52] ©2020 IEEE.

The compute and I/O specifications above shed light on key design considerations for a chiplet-based DNN accelerator: 1) the compute capacity of the DNN chiplet (4 TOPS in INT8) significantly exceeds the I/O bandwidth (100 Gbps, transmit or receive) and 2) the compute energy efficiency of the DNN chiplet (0.11 pJ/OP in INT8) is substantially lower than the I/O energy efficiency (0.82 pJ/b). The DNN chiplet must reuse the input data (input activations and weights) and reduce the output data (output activations) to minimize the I/O usage, or I/O can easily overtake compute to become the performance and energy bottleneck, rendering the chiplet approach impractical.

The contrast between compute and I/O also has an implication on the chiplet size choice. If a chiplet's *x*- and *y*-dimension are each scaled up by a factor of *S*, the compute capacity scales up by a factor of approximately S^2, but the chiplet I/O shoreline only scales up by a factor of *S*, allowing the I/O bandwidth to scale up by approximately *S*. This back-of-envelope calculation suggests that chiplet size may have to be kept smaller or the disparity between compute and I/O can become even larger.

A mapping strategy was developed for Nvidia's MCP to divide the weights into parts and allocate them to different chiplets [54]: 1) allocate output channels (different kernels) across columns of chiplets and 2) divide the input channels into parts and allocate them across rows of chiplets. To carry out the computation, the channels of the inputs are divided into parts and distributed to appropriate rows of chiplets. This mapping strategy provides data reuse and reduction: 1) weights are cached and reused within a chiplet; 2) input activations are reused between multiple kernels within a chiplet; and 3) output psums are reduced in the channel dimension before going out of the chiplet. Such a mapping strategy is essential for reducing the I/O usage and removing the I/O bottleneck in the DNN MCP.

B. Heterogeneous Integration

While homogeneous tiling of DNN chiplets solves the problem of scaling up DNN hardware to support larger DNN models, it does not address the problem of scaling out DNNs, i.e., extending DNNs to novel uses, e.g., DNNs used as a building block to support new applications. Besides scaling out DNNs, new operators can be added to DNNs in the future to enhance its capability, making it difficult to design a truly future-proof DNN chiplet.

We argue the importance of factoring computation into types, e.g., common operations and special operations, in considering chiplet-based system partitioning. As examples, CONV and FC layers are common and compute-heavy operations; and batch normalization and activation functions are special operations and relatively lightweight compared to CONV and FC layers. The control loops and data organization outside of NN processing to support different tasks are also special operations. This factoring exercise naturally leads to heterogeneous chiplets, e.g., an accelerator chiplet that supports common and compute-heavy operations, and a processor or FPGA chiplet that can be programmed to support special operations. Using this approach, accelerator chiplets can be made to target common kernels that are unlikely to change over time, allowing us to extend the useful lifetime of these chiplets. Processor and FPGA chiplets can be used to complement the accelerator chiplets to complete system implementations.

An example of heterogeneous integration is the MCP consisting of an FPGA with the PETRA systolic array chiplet [55] as illustrated in Fig. 23. The PETRA chiplet measures 3 mm^2 in an Intel 22 nm technology. It integrates tiles of systolic arrays to provide up to 1,024 MACs/cycle (FP16) or 1.43 TFLOPS (FP16) [55].

The PETRA MCP is built on Intel's embedded multi-die interconnect bridge (EMIB) [56], [57], a silicon bridge that connects an FPGA chiplet and an external chiplet. The silicon bridge provides a high routing density, enabling the use of parallel links of moderate speed. The I/O design for moderate-speed links can be made much simpler than high-speed serial I/Os, and it can even be made entirely digital [58]. A digital link is more reliable and can be ported to different technologies with ease. In the MCP design, a digital advanced interface bus (AIB) link [57], [58] was adopted with full swing, supporting a short-reach of 3 mm at 2 Gbps/pin. Thanks to the short reach and simple design, an AIB I/O consumes less than 1 pJ/b [58]. An AIB channel assembles 40 pins for an aggregate bandwidth of 80 Gbps. Using a dense bump pitch of 55 μm, an AIB channel occupies approximately 300 μm of die edge. The PETRA chiplet utilizes 8 AIB

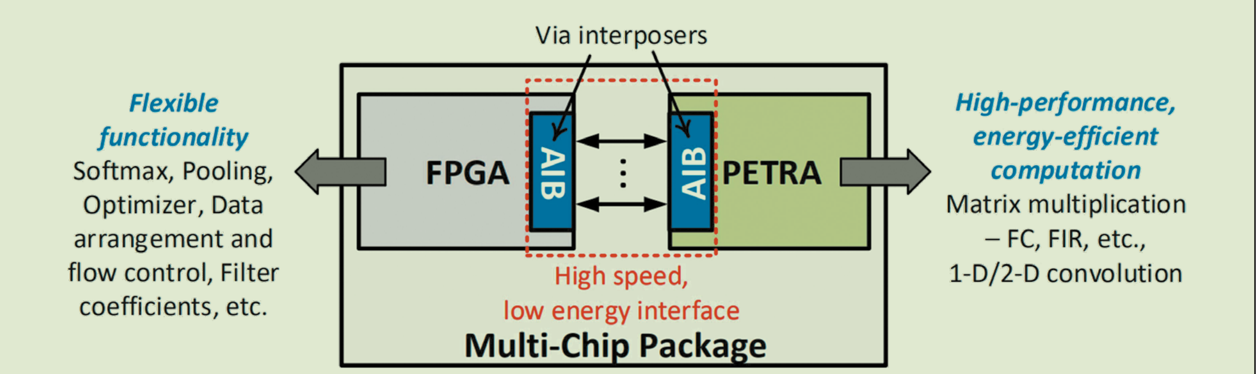
Via interposers
Flexible functionality
Softmax, Pooling, Optimizer, Data arrangement and flow control, Filter coefficients, etc.
FPGA
AIB
AIB
PETRA
High-performance, energy-efficient computation
Matrix multiplication – FC, FIR, etc., 1-D/2-D convolution
High speed, low energy interface
Multi-Chip Package

Figure 23. Illustration of the concept of integrating an FPGA with the PETRA chiplet. Figure reused from [55] ©2021 IEEE.

channels, or a total bandwidth of 640 Gbps, to communicate with the FPGA chiplet [55].

With heterogeneous integration, the FPGA can serve as the flexible platform that can be configured to serve as the host and handle control and data management, and the PETRA systolic array chiplet can perform the VMM and MMM that form the core part of DNN computation [55]. Operations that are not supported by the PETRA chiplet can always be covered by the FPGA chiplet. The heterogeneous platform can be further extended, e.g., by adding a front-end chiplet to make a complete sensor platform, and by adding another function accelerator chiplet to expand the capability of the system.

VI. Conclusion

DNN hardware design is a fast-evolving field. In this article we provide a survey and a tutorial on the basics of the DNN workloads, the essential processing architectures, and the promising directions in sparse DNN processing and multi-chip integration. First, we explain the two basic architectures for DNN processing, SIMD and systolic array, along with common WS and OS dataflows, to show the tradeoffs between flexibility and energy efficiency, and utilization and compute density. Next, we present designs that exploit data sparsity to improve both performance and energy efficiency with compressed storage and sparse processing. From partial sparsity to full sparsity, architectures can be designed with a range of overheads to gain from an array of benefits including lower energy, smaller memory, lower memory bandwidth and higher performance. Lastly, we show a path in scaling up and scaling out DNN hardware using multi-chiplet integration, either by tiling of modular DNN chiplets in constructing larger-scale systems or by heterogeneously pairing of DNN chiplets with CPU or FPGA to build a versatile platform.

Acknowledgment

This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) Program sponsored by DARPA.

Jie-Fang Zhang (Member, IEEE) received the B.S. degree in electrical engineering from National Taiwan University, Taipei, Taiwan, in 2015, and the M.S. degree in computer science and engineering and the 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 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.

Zhengya Zhang (Senior Member, IEEE) received the B.A.Sc. degree in computer engineering from the University of Waterloo in 2003, and the M.S. and Ph.D. degrees in electrical engineering from the University of California at 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, 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 been an Associate Editor of the IEEE Transactions on Very Large Scale Integration (VLSI) Systems since 2015, and serves on the Technical Program Committee of the IEEE Custom Integrated Circuits Conference (CICC) since 2018. He was an Associate Editor of the IEEE Transactions on Circuits and Systems—Part I: Regular Papers from 2013 to 2015 and the IEEE Transactions on Circuits and Systems—Part II: Express Briefs from 2014 to 2015, and served on the Technical Program Committee of the IEEE VLSI Symposium on Technology and Circuits from 2018 to 2022. He is an IEEE Solid-State Circuits Society Distinguished Lecturer.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.
[2] K. He et al., "Deep residual learning for image recognition," in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
[3] C. Szegedy et al., "Rethinking the inception architecture for computer vision," in *Proc. the Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 2818–2826.
[4] A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," 2020, *arXiv:2010.11929*.
[5] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2017, pp. 6000–6010.
[6] J. Devlin et al., "BERT: Pretraining of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Language Technol.*, vol. 1, Jun. 2019, Art. no. 41714186.
[7] T. Brown et al., "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2020, pp. 1877–1901.
[8] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," 2017, *arXiv:1605.07678*.

[9] S. Bianco et al., “Benchmark analysis of representative deep neural network architectures,” *IEEE Access*, vol. 6, pp. 64270–64 277, 2018.
[10] Y. Guo et al., “Deep learning for 3D point clouds: A survey,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 43, no. 12, pp. 4338–4364, Dec. 2021.
[11] G. Menghani, “Efficient deep learning: A survey on making deep learning models smaller, faster, and better,” 2021, *arXiv:2106.08962*.
[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2012, pp. 1097–1105.
[13] N. P. Jouppi et al., “In-datacenter performance analysis of a tensor processing unit,” in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2017, pp. 1–12.
[14] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–14.
[15] A. G. Howard et al., “MobileNets: Efficient convolutional neural networks for mobile vision applications,” 2017, *arXiv:1704.04861*.
[16] R. Girshick et al., “Rich feature hierarchies for accurate object detection and semantic segmentation,” in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2014, pp. 580–587.
[17] J. Redmon et al., “You only look once: Unified, real-time object detection,” in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 779–788.
[18] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation,” in *Proc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 3431–3440.
[19] R. Krashinsky et al. *Nvidia Ampere Architecture In-Depth*. Accessed: Dec. 10, 2022. [Online]. Available: https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/
[20] *Get Outstanding Computational Performance Without a Specialized Accelerator*. Accessed: Dec. 10, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-andtechnology/avx-512-solution-brief.html
[21] V. Sze et al., “Efficient processing of deep neural networks: A tutorial and survey,” *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
[22] Z. Du et al., “ShiDianNao: Shifting vision processing closer to the sensor,” in *Proc. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2015, pp. 92–104.
[23] C. Deng et al., “TIE: Energyefficient tensor train-based inference engine for deep neural network,” in *Proc. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2019, pp. 264–277.
[24] S. Han et al., “Learning both weights and connections for efficient neural network,” in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2015, pp. 1135–1143.
[25] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding,” in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2016, pp. 1–14.
[26] T. Zhang et al., “A systematic DNN weight pruning framework using alternating direction method of multipliers,” in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 184–199.
[27] S. Anwar, K. Hwang, and W. Sung, “Structured pruning of deep convolutional neural networks,” *ACM J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 1–18, 2017.
[28] S. Dave et al., “Hardware acceleration of sparse and irregular tensor computations of ML models: A survey and insights,” *Proc. IEEE*, vol. 109, no. 10, pp. 1706–1752, Oct. 2021.
[29] L. Deng et al., “Model compression and hardware acceleration for neural networks: A comprehensive survey,” *Proc. IEEE*, vol. 108, no. 4, pp. 485–532, Apr. 2020.
[30] J.-F. Zhang et al., “SNAP: A 1.67—21.55 TOPS/W sparse neural acceleration processor for unstructured sparse deep neural network inference in 16 nm CMOS,” in *Proc. Symp. VLSI Circuits (VLSI)*, Jun. 2019, pp. 306–307.
[31] J.-F. Zhang et al., “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.
[32] Y.-H. Chen et al., “Eyeriss: An energyefficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[33] S. Han et al., “EIE: Efficient inference engine on compressed deep neural network,” in *Proc. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 243–254.
[34] A. Parashar et al., “SCNN: An accelerator for compressed-sparse convolutional neural networks,” in *Proc. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 27–40.
[35] 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.
[36] Y.-H. Chen et al., "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.
[37] J. Albericio et al., “Cnvlutin: Ineffectual-neuron-free deep neural network computing,” in *Proc. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 1–13.
[38] S. Zhang et al., “Cambricon-X: An accelerator for sparse neural networks,” in *Proc. Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.
[39] Y. Chen et al., “DaDianNao: A machine-learning supercomputer,” in *Proc. Int. Symp. Microarchitecture (MICRO)*, Dec. 2014, pp. 609–622.
[40] W. Wen et al., “Learning structured sparsity in deep neural networks,” in *Proc. Adv. Neural Inf. Process. Syst. (NIPS)*, 2016, pp. 1–9.
[41] Z.-G. Liu, P. N. Whatmough, and M. Mattina, “Systolic tensor array: An efficient structured-sparse GEMM accelerator for mobile CNN inference,” *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 34–37, Jan./Jun. 2020.
[42] J. Yu et al., “Scalpel: Customizing DNN pruning to the underlying hardware parallelism,” in *Proc. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2017, pp. 548–560.
[43] S. Narang, E. Undersander, and G. Diamos, “Block-sparse recurrent neural networks,” 2017, *arXiv:1711.02782*.
[44] Z. Liu et al., “Learning efficient convolutional networks through network slimming,” in *Proc. Int. Conf. Comput. Vis. (ICCV)*, Oct. 2017, pp. 2755–2763.
[45] H. Li et al., “Pruning filters for efficient ConvNets,” 2016, *arXiv:1608.08710*.
[46] J. Albericio et al., “Bit-pragmatic deep neural network computing,” in *Proc. Int. Symp. Microarchitecture (MICRO)*, Oct. 2017, pp. 382–394.
[47] S. Han et al., “ESE: Efficient speech recognition engine with sparse LSTM on FPGA,” in *Proc. Int. Symp. Field Program. Gate Arrays (FPGA)*, Feb. 2017, pp. 75–84.
[48] H. Wang, Z. Zhang, and S. Han, “SpAtten: Efficient sparse attention architecture with cascade token and head pruning,” in *Proc. Int. Symp. High-Perform. Comput. Archit. (HPCA)*, Feb./Mar. 2021, pp. 97–110.
[49] Z. Qu et al., “DOTA: Detect and omit weak attentions for scalable Transformer acceleration,” in *Proc. Int. Conf. Architectural Support Program. Lang. Operation Systems (ASPLOS)*, Feb. 2022, pp. 14–26.
[50] N. P. Jouppi et al., “Ten lessons from three generations shaped Google's TPUv4i: Industrial product,” in *Proc. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2021, pp. 1–14.
[51] *Nvidia Deep Learning Accelerator (NVDLA). Accessed: Dec. 10, 2022.* [Online]. Available: http://nvdla.org/
[52] 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.
[53] J. W. Poulton et al., “A 1.17-pJ/b, 25-Gb/s/pin ground-referenced single-ended serial link for off-and on-package communication using a process-and temperature-adaptive voltage regulator,” *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 43–54, Jan. 2019.
[54] R. Venkatesan et al., “A 0.11 pJ/OP, 0.32-128 TOPS, scalable multi-chip-module-based deep neural network accelerator designed with a high-productivity VLSI methodology,” in *Proc. IEEE Hot Chips Symp. (HCS)*, Aug. 2019, pp. 1–24.
[55] S.-G. Cho et al., “PETRA: A 22 nm 6.97 TFLOPS/W AIB-enabled configurable matrix and convolution accelerator integrated with an Intel Stratix 10 FPGA,” in *Proc. Symp. VLSI Circuits (VLSI)*, Jun. 2021, pp. 1–2.
[56] R. Mahajan et al., “Embedded multi-die interconnect bridge (EMIB)—A high density, high bandwidth packaging interconnect,” in *Proc. IEEE Electron. Compon. Technol. Conf. (ECTC)*, May/Jun. 2016, pp. 557–565.
[57] D. Greenhill et al., “3.3 A 14 nm 1 GHz FPGA with 2.5D transceiver integration,” in *Proc. Int. Solid-State Circuits Conf. (ISSCC)*, Feb. 2017, pp. 54–55.
[58] C. Liu, J. Botimer, and Z. Zhang, “A 256 Gb/s/mm-shoreline AIB-compatible 16 nm 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)*, Apr. 2021, pp. 1–2.