Journal of Systems Architecture xxx (xxxx) xxx

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Accelerating CNN Inference on ASICs: A Survey

Diksha Moolchandani [a,*], Anshul Kumar [b], Smruti R. Sarangi [c]

[a] School of Information Technology, IIT Delhi, India
[b] Computer Science and Engg., IIT Delhi, India
[c] Computer Science and Engg. (joint appt. with Electrical Engg.), IIT Delhi, India

ARTICLE INFO

Keywords:
CNN
Inferencing
ASICs
Accelerators

ABSTRACT

Convolutional neural networks (CNNs) have proven to be a disruptive technology in most vision, speech and image processing tasks. Given their ubiquitous acceptance, the research community is investing a lot of time and resources on deep neural networks. Custom hardware such as ASICs are proving to be extremely worthy platforms for running such programs. However, the ever-increasing complexity of these algorithms poses challenges in achieving real-time performance. Specifically, CNNs have prohibitive costs in terms of computation time, throughput, latency, storage space, memory bandwidth, and power consumption.

Hence, in the last 5 years, a lot of work has been done by the scientific community to mitigate these costs. Researchers have primarily focused on reducing the computation time, the number of computations, the memory access time, and the size of the memory footprint. In this survey paper, we propose a novel taxonomy to classify prior work, and describe some of the key contributions in these areas in detail.

1. Introduction

In the last few years, we have seen great advances in the field of neural networks, particularly Convolutional Neural Networks (CNNs). The field has seen a huge upsurge since 2012, when CNNs proved their mettle by providing a significant improvement in the accuracy of the ILSVRC [1] image classification tasks [2]: the accuracy increased to 85% from 74% (best conventional method). This led to a revolution in the field of neural networks where every passing year these networks continuously got deeper and better. The field matured from the 7-layered AlexNet (proposed by Krizhevsky) to the 152-layered ResNet [3]. The latter had an error rate of 3.6%. Additionally, the structure of these networks was modified suitably to cater to different applications such as object recognition, face recognition, speech processing, and even self driving vehicles [4].

As of 2019, deep neural networks are the building blocks of many commercial applications that have captured public imagination such as Amazon's Alexa, and Google Deepmind. As the networks have grown deeper, the computational complexity of these networks has increased, thereby prohibiting their deployment for real-time applications. Some statistics by Sundaram [5] suggest that the complexity (defined as Flops/pixel) of these algorithms is increasing at the rate of roughly 10x per decade (since the early 70s). Furthermore, the amount of data that needs to be processed is increasing at the rate of 1000x [5] per decade. Thus, the computational throughput needs to scale by 10,000 times to match with the requirements of these algorithms. Also the number of parameters for these networks has been increasing at an exponential rate [6]. Fig. 1 gives an idea of the increasing computational and memory requirements of these networks since 2012: 3000X increase in the computational time, and 10X increase in memory requirements. Hence, the need to efficiently run these networks is urgent.

Additionally, these networks require millions of parameters [21,22] for their functioning, which translates to very high memory storage and bandwidth requirements. Thus, the job of hardware researchers involves making these algorithms affordable both in terms of computation and memory requirements. On the hardware side, the end of Dennard scaling has prohibited any increase in the processor frequency (saturated at 4 GHz) since 2006 [23]. Furthermore, the demise of Moore's law means that the number of cores per chip are not expected to increase significantly.

Hence, the scientific community has justifiably changed its focus to FGPAs and GPUs that provide greater throughput. GPU instruction throughput has increased 100x in the last decade [24]. FPGA technology has made similar strides, and as of date, the best single-board FPGAs in the market outperform single-GPU systems while inferencing neural networks [19].

For large neural networks, it is hard to say which platform, FPGA, ASIC or GPU, is better without creating an optimal implementation on all the platforms. Thus, we cite the general trends and the results from the related work that compare all the platforms. Table 1 shows the differences between a GPU, an FPGA, and an ASIC implementation in the

* Corresponding author.
E-mail addresses: diksha.moolchandani@cse.iitd.ac.in (D. Moolchandani), anshul@cse.iitd.ac.in (A. Kumar), srsarangi@cse.iitd.ac.in (S.R. Sarangi).

https://doi.org/10.1016/j.sysarc.2020.101887
Received 23 January 2020; Received in revised form 3 September 2020; Accepted 4 September 2020
Available online 12 September 2020

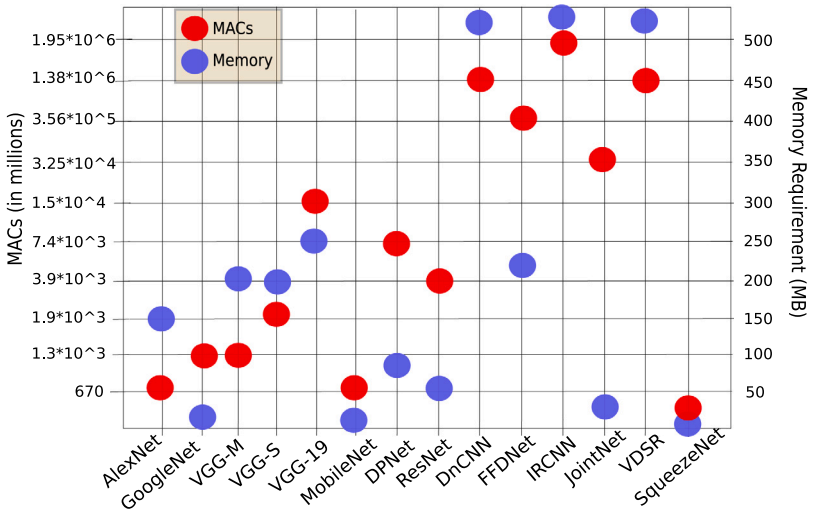

Fig. 1. Evolution of CNNs in recent years.
Source: Data taken from [7]

Table 1
Comparison of FPGAs and ASICs.
Source: Adapted from the observations in [8,9].

Metric	GPUs	FPGAs	ASICs	References
Time-to-market	Low	Medium	High	2 months for FPGA design, 30 months for ASIC design [10]. The number of lines of code in CUDA is lesser than its VHDL counterpart [11]
Frequency	High	Low	Medium	ASIC to FPGA frequency ratio is 3-4X [12] for a general suite of applications. TPU (ASIC) to Microsoft Catapult V1 (FPGA) frequency ratio is 3.5X [8]. GPU to FPGA frequency ratio is 10X [13]
NRE* cost	Low	Low	High	$350K-$1000K for ASIC [14,15], none for FPGA [9]
Power	High	Medium	Low	FPGA to ASIC dynamic power ratio of 7-14X [12] and GPU to FPGA power ratio of 10X [16,17]
Energy efficiency	Low	Medium	High	ASIC to FPGA performance per watt ratio is 10X [15,18] and FPGA (Stratix 10) to GPU (Titan X) performance per watt ratio is 2.3-4.3X [19]
Reconfigurability	N/A	High	Low	N/A
On-chip storage	Fixed memory hierarchy	Constrained by BRAMs	Flexible design	N/A
Area	Large	Large	Small	FPGA to ASIC area ratio is 18-35X [12] for a general suite of applications and 8.7X [8] for Chain-NN [20]
DSP blocks	N/A	Fixed precision	Customizable	N/A
Unit Cost	Medium	High	Low	$200-$3200 for FPGA, $30 for ASIC [9,14] (without NRE)
Performance	Low	Medium	High	ASIC to FPGA performance ratio of 6.3X [8]. FPGA (Stratix 10) to GPU (Titan X) performance ratio is 2.1-3.5X [19]

NRE is Non-Recurrent Engineering Cost

second, third and fourth columns respectively. In the fifth column, we cite the numbers from the original papers to support the comparison. For some cases, the comparison is qualitative and thus the fifth column is N/A (not applicable) for them.

1.1. Scope of the survey

In this survey paper, we focus on only ASIC based implementations of CNN accelerators and look at their design and optimizations, notably reduction of the computation time, memory access time and memory footprint. We shall exclude the discussion of CNN designs on other substrates such as FPGAs. There are already several survey papers on FPGA based CNN accelerators [25–29], and GPU based implementations [30,31].

There are a lot of non-conventional architectures for implementing CNNs (not covered in this paper). Let us briefly provide some pointers to relevant work in this area. For CNN inferencing processor-in-memory (PIM) architectures such as *Prime* [32] and *Pipelayer* [33] are very popular. They embed processing engines within memory such that it is easier to handle the memory requirements of modern CNNs. This line of thought can be extended to co-design CNN architectures and memory systems (Tu et al. [34]). Mittal [35] and Umesh et al. [36] provide comprehensive surveys on ReRAM based and spintronics based architectures for processing-in-memory respectively. There are many more proposals that use unconventional hardware such as a switched capacitor for MAC operations [37], and analog computing engines [38,39]. However, due to a lack of space our survey is limited to hardware made with conventional devices. Mittal [40] provides a detailed discussion of the impact of various design decisions on the reliability of the neural network accelerators.

We shall not be discussing the works pertaining to the acceleration of the training of CNNs. Even though training is carried out offline and is generally a one-time process, it is performance sensitive and can affect the overall turnaround time of the model. However, the training process is substantially different than the inferencing due to the presence of derivatives and continuous weight updates. Hence, it deals with a different regime of power, timing, and physical form factor of the device. There are several works that accelerate the training process,

however, discussing such works is out of the scope of this survey. Such a wide field of work needs a separate dedicated survey with the proper background. We restrict ourselves to discussing the works that accelerate the inferencing of CNNs.

Since the space of techniques that we shall present have not been implemented on a common platform, comparing the designs on the basis of performance or throughput is not feasible (see Section 3). We shall thus provide qualitative insights and intuitions based on first principles. We provide some basic form of quantitative comparison based on the data collected from the original papers in Section 8.

1.2. Organization

We start by providing a brief overview of a CNN (Section 2.1) and then present a generic architecture of an accelerator on custom hardware (Section 2.3). We also identify the components of the generic architecture that can be targeted for optimizations. We then present a novel taxonomy of different optimizations for accelerating CNNs in Section 3. We then discuss each optimization in detail. Section 4 presents the techniques employed by various papers to fit the computational needs of a CNN within the available computational resources. Then we move on to discuss methods to reduce the memory access time in Section 5. Next, we discuss the challenges of storing the parameters of all the neurons in the section on reducing the overall memory footprint (Section 6). We discuss the industrial designs of the accelerators in Section 7. In Section 8, we compare multiple works together both qualitatively and quantitatively on the basis of performance, energy, and accuracy. In Section 9, we discuss the challenges of designing these accelerators on ASICs. We finally conclude in Section 10.

2. Background

2.1. Overview of a Convolutional Neural Network (CNN)

An artificial neural network is inspired by the human brain and as the name implies it is a network of artificial neurons. The analogy of the neural network is drawn from the nervous system in the human body where the nerve cells communicate with other cells via synapses. A Convolutional Neural Network (CNN) is a type of artificial neural network that consists of multiple stages, where each stage is a small neural network in its own right. It is called a *layer*. Each layer finds a given feature of the data, and gradually as we move towards later layers, the complexity of the identified features increases till we can recognize full objects. A CNN generally consists of 4 types of layers – convolution layer, pooling layer, ReLU (rectified linear unit) layer and fully connected layer. There are some non-traditional layers such as deconvolution layer [41] and dilated convolution layer [42]. Each type of layer is discussed in detail in the following paragraphs.

(1) Convolution Layer: A 2D convolution operation between an input image matrix a (size $R \times C$) and a filter f (size $W \times L$) is a point-wise multiplication and addition of the corresponding pixels. The filter (usually smaller in size than the input image matrix) first multiplies with the $W \times L$ sized-block of the input image, accumulates the result, slides to the subsequent block of the input image, and repeats the process. It may be noted that the point-wise multiplication is performed between the matrices of same size, thus the input image (usually larger in size than the filter) is processed incrementally, one block at a time, till the entire $R \times C$ elements of the image are processed. Eq. (1) shows a 2D convolution operation. This convolution is a 2D convolution because the filter slides in two directions, along the height and the width of the input. Note that we will not be discussing 3D convolutions in this survey.

$$b = f \star a, \qquad b(r,c) = \sum_{w=0}^{W-1} \sum_{l=0}^{L-1} f(w,l) \times a(r+w-\lfloor \frac{W}{2} \rfloor, c+l-\lfloor \frac{L}{2} \rfloor) \quad (1)$$

Here, $b(r,c)$ represents an output pixel in the output matrix b, where the coordinates of each pixel are represented as (r,c). Also note that in the first sub-equation, the $\star$ operator stands for a convolution operation. l and w are the iterators over the length and width of the filter.

A convolution layer in a CNN performs the 2D convolution of N input matrices with the corresponding filters to produce M output matrices. For example, an RGB image consists of pixels that can be represented as a combination of three primary colors: red, green, and blue. Thus, the image matrix can be decomposed into three new matrices (also called *channels*), where the pixels in the new matrices indicate their contribution to the entire RGB image. This is also called the *depth* of the input. As explained earlier that in a 2D convolution, the filter slides in only two directions, thus for performing the convolution of an input of depth N, the filter should also be of depth N. The convolution of $R \times C \times N$-sized input with $W \times L \times N$-sized filter produces an output of size $R \times C \times 1$ (assuming padding is being done on the input appropriately). When M such filters are convolved with the input, we get M output matrices. These M output matrices become the input matrices for the next layer.

These matrices (except for the first set of input matrices) are called the feature maps. A *feature map*, as the name suggests, is a map of the features detected in the input matrix, produced as a result of sliding a filter over the entire input. It is also sometimes referred to as an *activation map* (a^i in Eq. (3)) and each element of this map is known as an *activation* (a^i_{rc} in Eq. (2)) or a *neuron*. It may be noted that the input image can have various features such as a straight line, a circle, or a curved line at various locations. The kind of feature detected in a feature map depends on the filter used.

The filter, also sometimes called the *kernel* (f^{ij} in Eq. (3)), is a matrix of values or weights (f^{ij}_{wl} in Eq. (2)), which when convolved with the input matrix, results in the detection of features (produces the feature map). As an example, say we have two filters, where the first one detects a straight line and the second one detects a circle, then a linear combination of the two generated feature maps may detect a curved line – simple detectors combine to detect higher level features. A mathematical definition is as follows.

(2)

$$b^i = \begin{bmatrix} b^i_{11} & b^i_{12} & \cdots & b^i_{1C} \\ b^i_{21} & b^i_{22} & \cdots & b^i_{2C} \\ \vdots & \vdots & \ddots & \vdots \\ b^i_{R1} & b^i_{R2} & \cdots & b^i_{RC} \end{bmatrix} f^{ij} = \begin{bmatrix} f^{ij}_{11} & f^{ij}_{12} & \cdots & f^{ij}_{1L} \\ f^{ij}_{21} & f^{ij}_{22} & \cdots & f^{ij}_{2L} \\ \vdots & \vdots & \ddots & \vdots \\ f^{ij}_{W1} & f^{ij}_{W2} & \cdots & f^{ij}_{WL} \end{bmatrix}$$

$$a^i = \begin{bmatrix} a^i_{11} & a^i_{12} & \cdots & a^i_{1C} \\ a^i_{21} & a^i_{22} & \cdots & a^i_{2C} \\ \vdots & \vdots & \ddots & \vdots \\ a^i_{R1} & a^i_{R2} & \cdots & a^i_{RC} \end{bmatrix} \quad (2)$$

Let us summarize the definitions of the key terms. b^i is the ith output feature map, f^{ij} is the convolution kernel that maps the ith input feature map to the jth output feature map, and a^i is the ith input feature map.

(3)

$$\vec{z} = \begin{bmatrix} b^1 \\ b^2 \\ \vdots \\ b^M \end{bmatrix} K = \begin{bmatrix} f^{11} & f^{12} & \cdots & f^{1N} \\ f^{21} & f^{22} & \cdots & f^{2N} \\ \vdots & \vdots & \ddots & \vdots \\ f^{M1} & f^{M2} & \cdots & f^{MN} \end{bmatrix} \vec{y} = \begin{bmatrix} a^1 \\ a^2 \\ \vdots \\ a^N \end{bmatrix} \quad (3)$$

Let $\vec{y}$ represent a vector of N input feature maps $(a^1, a^2, \ldots, a^N)$, where each a^i is an $R \times C$ matrix. Let $\vec{z}$ represent a vector of M output feature maps $(b^1, b^2, \ldots, b^M)$, where each b^i is an $R \times C$

matrix, and let K represent a matrix of $M \times N$ (mapping each output to input) convolution kernels ($f^{11}, f^{12}, \ldots, f^{MN}$), where each f^{ij} is a $W \times L$ matrix.

(4) Eq. (4) convolves N kernels with the N input feature maps to generate an output feature map. The total number of multiply-and-accumulate (MAC) operations are $O(WLRCN)$. This equation is calculated M times (with M different filters) to generated all the output feature maps (see Fig. 2), leading to a computational complexity of $O(WLMNRC)$.

$$\vec{z}_i = \sum_{j=1}^{N} K_{ij} \star \vec{y}_j \tag{4}$$

The process of convolution can be graphically visualized in the left side of Fig. 2.

(5) Pooling layer: The primary job of feature detection is done by the convolution layer. The pooling layer accentuates the detected features and introduces a degree of non-linearity (in the case of max-pooling). Specifically, a pooling layer introduces translational invariance in the feature map by replacing the activations in a window with a representative activation. For example, a max-pooling layer takes in an input feature map and replaces a window of activations in the feature map by the largest activation in that window. This size of the window of activations is equal to the size of the pooling filter. The pooling filter then slides by a certain length (also called *stride*) to traverse the entire feature map. Note that the filter slides in a way such that it first traverses right to cover all the columns of a row and then slides down to cover the subsequent rows in the same manner. Thus, the size of the output feature map is reduced by the size and the stride of the pooling filter (see Fig. 2).

(6) ReLU layer: ReLU stands for Rectified Linear Unit. This layer introduces non-linearity in the network by replacing negative outputs with 0. This also helps deeper networks in learning complex functions and ensures faster training.

(7) Fully Connected (FC) layer: This layer connects all pairs of neurons in the input and output layers. In the FC layer, we can proceed with the computations in two ways: *output-preferred* or *input-preferred*. We coin these names based on the requirement of the user. A full connection means that an output of a fully connected layer is formed by the contribution of all the input neurons in the input layer. Thus, to get the final output activation, we need to load all the inputs in the cache and perform a weighted sum. However, the on-chip cache is limited in capacity. If the user prefers a complete output (as opposed to a partial output), we need to load all the inputs contributing to an output activation, which has prohibitive memory requirements.
Alternatively, if the user prefers to use all the loaded inputs before loading the next batch of inputs, we need to produce all the partial output activations for which these inputs are contributing. Such a formulation will require us to have a large amount of memory to store the partial outputs. Note that the outputs are called partial outputs when there are more inputs contributing to them than the actually loaded ones. Thus, it can be deduced that this layer not only requires a large amount of memory (equal to the size of the network) but it is also associated with a significant number of memory accesses.

(8) Deconvolution/Transposed convolution layer: This layer is responsible for increasing the size of the feature map as opposed to the reduction in the size of the feature map done by a convolution layer [41]. This is done by inserting zero values in between the original values in the feature map (also called *upsampling*) and subsequently convolving it with a kernel. This is primarily used in depth estimation based tasks where a depth prediction is needed for every pixel or in super resolution based tasks where the size of an image is increased.

(9) Dilated convolution layer: Some tasks such as image segmentation require a wider global context to be captured in each output pixel. In such cases, the filter is up-sampled and convolved with the input image to capture a larger receptive field [42].

To summarize, CNNs are feed-forward neural networks that are simple neural networks, where the inputs of each layer come from the immediately preceding layer and the outputs are forwarded to the immediately following layer. Unfortunately, these networks are extremely complex, use millions of weights, and are intensive both in terms of the number of computations and memory bandwidth requirements. Qiu et al. [43] analyzed the requirements of the convolution layers and the FC layers. They found the former to be highly compute intensive, and the latter to be very memory intensive primarily because a large amount of memory is required to store $O(N^2)$ weights (N is the number of neurons in each layer) and a large number of memory accesses is required due to lower weight reuse as compared to the convolution layers. Both the transposed and the dilated convolution layers are special form of the convolution layer and hence most of the optimizations remain the same.

2.2. Running example: Convolution layer

The left side of Fig. 3 shows the code of a convolution layer (for batch size=1) using 6 nested loops. The outermost loop iterates through all the output feature maps (fm*ap_out*[]). Each such output feature map is the sum of the convolutions of each input feature map with its corresponding kernel (the figure is self explanatory). Depending on the size of the CNN, computing the result of such a deeply nested *for* loop can be computationally very expensive. Hence, it is essential to parallelize the process of computing all the output feature maps.

The parallelization can be exploited by employing the classical technique: *partitioning* (or tiling) the 6D space and *mapping* either the partitions or the computations in each partition to parallel PEs. If the partitions are computationally independent of each other then the PEs can progress independently. Thus, the 6D nested loop structure can be modified to form 12 nested loops, where the outermost 6 loops partition the 6D space and the innermost 6 loops perform the convolution operation for each partition (see right side of Fig. 3). The outermost 6 loops look the same as the innermost 6 loops; however, instead of incrementing by one in each iteration, they get incremented by b_i, where b_i is the block size of the ith loop (varies from 1 to 6). This is the classical way of partitioning the space. The innermost 6 loops compute the convolution for each partition – portion of the space of iterations assigned to it by the 6 outer loops. In each partition we compute the result for $b_1 \times b_2 \ldots \times b_6$ iterations.

The parallelization can be achieved in two ways: either map the different partitions generated by the outermost 6 loops to different PEs, or map the computations of each partition (innermost 6 loops) to different PEs. The simplest implementation of the first idea is to map each partition to a PE, where we have $(M/b_1) \times (N/b_2) \times (R/b_3) \times (C/b_4) \times (W/b_5) \times (L/b_6)$ PEs (or partitions). This is typically very expensive; hence, we choose a set of axes (loop iterations) that we want to parallelize. For example, if we choose to compute the output feature maps in parallel, then we create M/b_1 partitions, where each PE computes b_1 output feature maps.

The second idea requires $b_1 \times b_2 \ldots \times b_6$ computations to be mapped to PEs. Analogous to the previous example, this is very expensive in terms of the required computation units and hence we choose a subset of these iterations to map to the parallel PEs.

The quintessential problem that needs to be solved in this space is to decide the values of $b_1 \ldots b_6$. This has implications in terms of the data dependences across the PEs, data communication, and the amount of parallelism. Therefore, at the high level, we can create two broad classes of problems: ❶ decide the method of partitioning the convolution space, and ❷ for a given partitioning optimize computation and

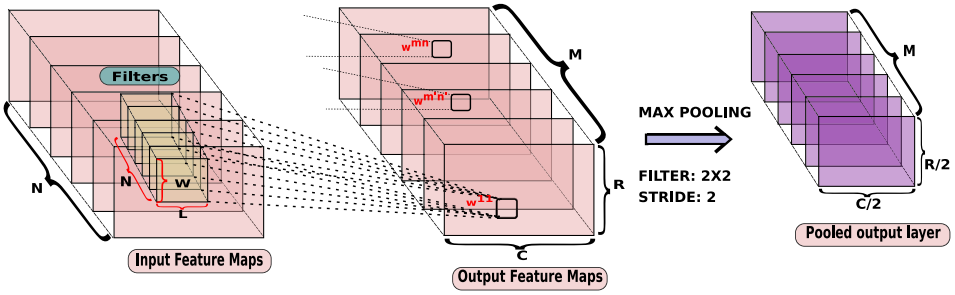

Fig. 2. A convolution layer and a pooling layer in a CNN.

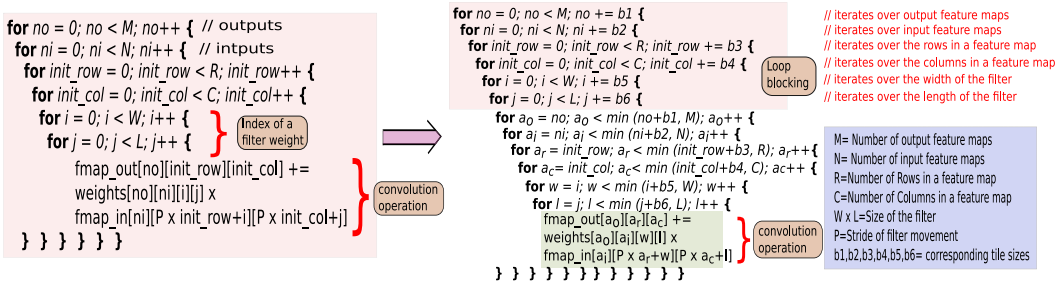

Fig. 3. Algorithm for a tiled convolution layer (adapted from Figure 5 in [44], names of variables have been changed).

communication. We can either reduce the computation time, reduce the amount of memory stored on-chip, or reduce the access latency. A typical optimization problem is as follows:

> *Minimize the latency in a given CNN architecture subject to a constraint on the amount of on-chip memory.*

2.3. Reference architecture of a CNN accelerator

Every CNN accelerator can be abstracted in the form of a vanilla/reference accelerator architecture as shown in Fig. 4. The basic accelerator architecture is characterized by multiple processing elements (PEs), where parts of the computation are mapped to each PE as discussed in Section 2.2. Furthermore, we can have connections between PEs to transfer data in the case of data dependences, and for transferring results from one layer of the CNN to the next: either within the same CNN layer or across layers. We would like to pass as much of data as possible without accessing memory structures.

Each such PE comprises an ALU, CU (control unit), and a register file (RF) (see Fig. 4). Additionally, there exist two global buffers each for input/output activations and weights. In some cases, the buffers may not be separate. The on-chip storage is typically not sufficient to hold millions of values across several layers. Thus we need to have off-chip memory to hold all the data. Apart from the global buffer, each PE stores the most frequently used data in its local register file. Additionally, there is an inter-PE network to stream the data directly from one PE to the other.

It is worth noting that the reference accelerator architecture presented above is a basic skeleton of an accelerator. The basic building blocks – PEs, MACs (multiply accumulate units), register files, interconnects, PE arrangements and the global buffers – are present in some or the other form in most of the proposals discussed in this survey. Let us provide examples regarding how they can be modified for better power and performance at a high level. For the MAC we can modify the operand values or their bit-widths. We can compress the data stored in the registers such as the weights, and in addition we can optimize the global buffer by using a combination of technologies such as SRAM and eDRAM arrays. Some architectures even experiment with different inter-PE interconnections, and try to optimize the flow of data.

2.3.1. Categorization of types of architectures

We can classify the architectures on the basis of the arrangement of PEs and the kind of dataflow exploited by them. Most of the architectures have either a 1D or 2D arrangement of PEs. The inter-PE network can be of two types: point-to-point connections between PEs, or all of them are connected to a global buffer. A direct point-to-point communication among the PEs assumes a producer–consumer relationship between the neighboring PEs. The data produced by a PE is passed to the neighboring PE in the next time step. This continuous production and transfer of data is analogous to the process of pumping of blood by the heart and is termed as *systolic* (refer to the literature on systolic arrays [47,48]).[1] If we consider all the possible combinations of the arrangement of PEs and the dataflow pattern, we get 4 different accelerator architectures. These are as follows (also see Fig. 5).

1D systolic: The characteristic of this architecture is the 1D arrangement of PEs allowing a systolic flow of data.

2D systolic: In this architecture, the PEs are arranged as a 2D matrix, and they receive data from both the horizontal and vertical directions.

1D array: The PEs are arranged in a 1D format with each PE receiving the data that has been broadcasted from the global buffer. There is no direct communication between the neighboring PEs.

2D matrix: This is similar in functionality to the 1D array except that it has a 2D arrangement of PEs.

3. Taxonomy

Each of the proposals discussed in this survey optimize one or more of the parameters of the basic building blocks of the reference architecture. There can be many parameters and building blocks that can be optimized, however classifying the proposals on this basis would not provide a concise and generic classification. Thus, we dissect the

[1] Note: For the purpose of this survey, we do not distinguish between the different types of systolic architectures.

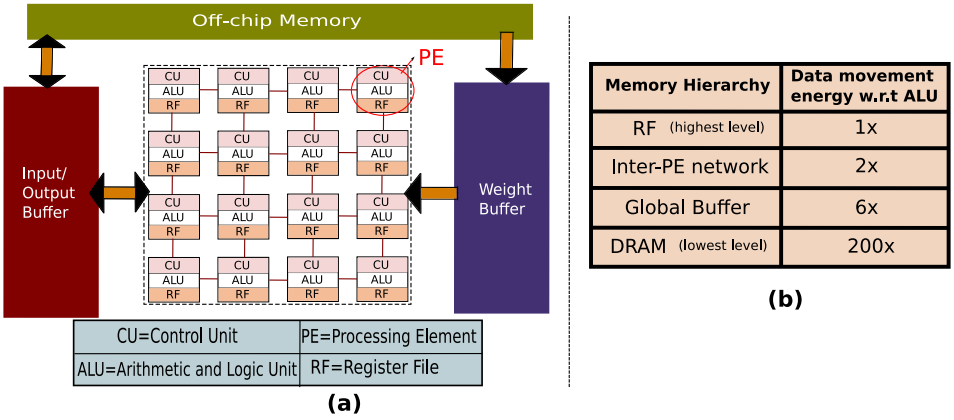

Fig. 4. (a) Reference dataflow architecture for CNNs (adapted from [21,45], and (b) Normalized energy (w.r.t ALU) for accessing data from the memory hierarchy.

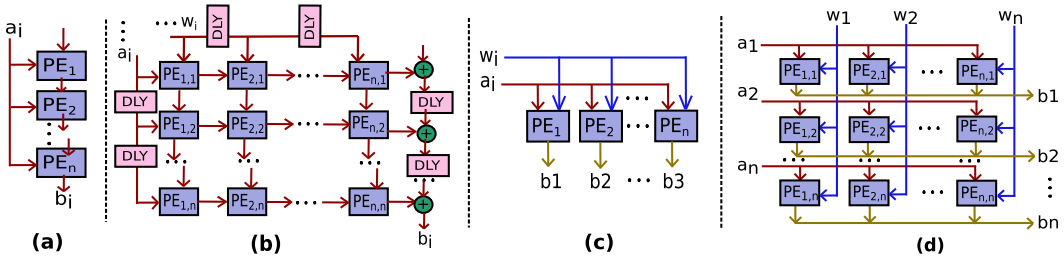

Fig. 5. Different types of architectures: (a) 1D Systolic, (b) 2D systolic, (c) 1D array, and (d) 2D matrix (adapted from [46]). DLY is the delay.

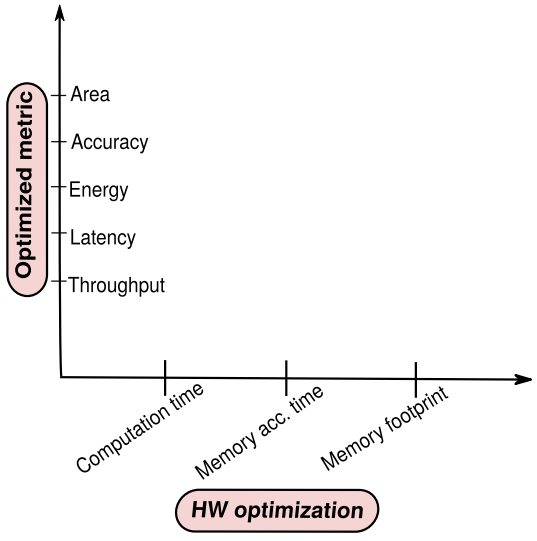

Fig. 6. Taxonomy of CNN accelerators.

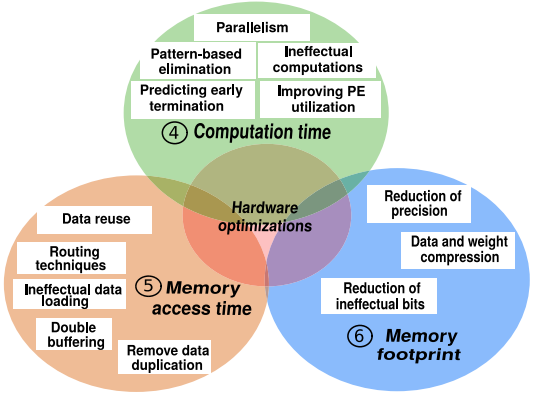

Fig. 7. Venn diagram of hardware optimizations.

space of techniques based on the type of architectural optimization (x axis of Fig. 6). We mostly target three kinds of optimizations: reduction of the computation time, reduction of the memory access time, and reduction of the memory footprint. The motivation for this breakup comes from the CPU-memory performance equation that has primarily three components: CPI (cycles per instruction), memory latency, and the number of memory accesses. There is a one-to-one correspondence between these terms of the performance equation, and our metrics.

The y axis of Fig. 6 focuses on the metric being optimized or affected: area, accuracy, energy, latency and throughput. We believe that a qualitative discussion of the techniques from the point of view of these metrics is important because the quantitative comparison of these numbers is almost impossible given that each research paper tests its design on a different set of parameters. Fig. 7 shows a Venn diagram of the techniques based on the type of optimization that they propose. We dedicate a section to each of these optimizations (shown in the diagram).

4. Reduction in computation time

Ever since the algorithmic complexity of CNNs started increasing, the computation time has also increased drastically. Cong et al. [49] studied the breakdown of the time taken by the different layers of AlexNet for an image recognition task. They showed that the runtime of the task was dominated (90.7%) by the time taken by the computations in the convolution layer (see Fig. 3). Given the applications of CNNs in image classification and object recognition, where real-time implementation plays a crucial role, elevated computation times pose serious questions with regards to their deployment. The **only intuitive** way to reduce the computation time is by ❶ *utilizing the parallel resources*

Table 2
Glossary of loop iterators in a tiled CNN.

Loop iterator	Iterates over
a_o, *no*	the number of output feature maps
a_i, *ni*	the number of input feature maps
a_r, *init_row*	the rows in the feature map
a_c, *init_col*	the columns in the feature map
w, i	the width of the filter
l, j	the length of the filter

Table 3
Loop variables targeted for parallelization.

Parallelism type	inner loop variables
Inter-output	a_o, a_i
Inter-kernel	a_r, a_c
Intra-kernel	w, l

to effectively reduce the total execution time, ❷ *reducing the number of computations*, ❸ *reducing the execution time of each computation* involved in a convolution operation, or ❹ *utilizing the idle cycles in a computation for some useful work*. Let us discuss research contributions in each of these categories.

4.1. Exploiting the inherent parallelism in CNNs

The operation of a convolutional layer is inherently parallel due to the nested loop structure as explained in Section 2.2. The process of convolution of a set of pixels with the kernel is independent of the convolution operations on another set of pixels, thus there is a scope for parallelism (see the work of Motamedi et al. [50]).

Since a CNN is a feed-forward network, each layer is data dependent on its previous layer. Thus, it is not possible to run them in parallel. Nevertheless, the different layers of a CNN can be pipelined subject to memory constraints. We do not consider inter-layer pipelining to be a type of parallelism per se and hence do not discuss it further.

Let us now provide a brief description of the types of parallelism opportunities in a convolution layer. Each kind of parallelism can be perceived as creating a different kind of partition in our 6D space of outermost loop iterators (see Fig. 3 – our running example). The key terms used in our running example are reproduced in the glossary in Table 2 in the same order.

Since we have 6 inner loop iterations in our running example, in theory we can parallelize a computation by choosing any subset of axes (iterations), and then appropriately choosing block sizes. However, we need to simultaneously look at data communication and storage constraints. Considering these limitations, researchers have primarily looked at three kinds of parallelism: inter-output, inter-kernel, and intra-kernel. The intra-kernel parallelism performs the $w \times l$ multiply-add operations of a convolution operation in parallel. The inter-kernel parallelism produces the activations of an output feature map in parallel. The inter-output parallelism produces the activations of multiple output feature maps in parallel. The loop variables chosen for each parallelization method are shown in Table 3. The block sizes depend on the number of available PEs for the convolutional layer.

4.1.1. Case studies

We shall discuss some recent work in more detail (refer to Table 4). The * marked entries in the table are subsequently discussed.

FlexFlow

Lu et al. [57] proposed *FlexFlow* to exploit all the three kinds of parallelisms on a 2D array of PEs. *FlexFlow*, enables this by adding extra interconnections between PEs and on-chip buffers so that there is some more flexibility to fetch any neuron from any feature map. Intuitively, this design reduces the interconnections among PEs but at the cost of *energy* because the data is now being forwarded from the on-chip buffers to the PEs instead of getting forwarded in a systolic fashion between neighboring PEs. The design achieves an increased *throughput* due to the parallel functioning of the PEs, and trades off interconnect latency for the memory access latency.

Table 4
Types of parallelism exploited by different architectural implementations.

Proposal	Parallelism Type			Architecture Type
	Intra-output		Inter-output	
	Inter-Kernel	Intra-Kernel		
CNP [51]		✓		2D systolic
DC-CNN [52]	✓		✓	1D array
MAPLE [53]	✓	✓		2D systolic
nn-x [54]		✓	✓	1D systolic
Jin et al.[55]	✓			1D systolic
C-Brain [56]	✓	✓	✓	1D array
FlexFlow* [57]	✓	✓	✓	2D matrix
Origami [58]			✓	1D array
SCNN* [59]	✓	✓	✓	2D systolic
YodaNN* [60]			✓	1D array

SCNN

A full fledged CNN accelerator (*SCNN*) proposed by Parashar et al. [59] also exploits all the kinds of parallelism. Their architecture arranges the PEs in a 2D array with systolic connections to transfer partial sums. Intra-kernel parallelism is exploited by arranging the multiplier array (in each PE) as a 2D matrix to perform the 2D multiplications in parallel. Intra-output and inter-output parallelism is exploited by distributing the input activations and weights across the PEs. Their proposal differs from *FlexFlow* in the sense that the number of global buffer accesses in *SCNN* is lower due to a streaming transfer of the partial sums and the stationarity of the input activations. This however reduces the flexibility of the design.

YodaNN

Another recent proposal by Andri et al. [60] exploits inter-output parallelism by replicating the PEs to form a 1D array with no interconnections among them. Each parallel PE gets its input data from the on-chip buffers, thereby increasing the amount of required memory bandwidth. Each PE is dedicated to produce an output pixel of the corresponding output feature map in a cycle. Recall that each output map is a linear combination of the convolutions of input feature maps.

This design stores a filter (matrix of weights) in each PE. A set of values is read from the input matrices, multiplied with the corresponding weights, and the value of an output pixel is computed in every clock cycle. Ideally, the number of PEs should be equal to the number of output feature maps. However, if the number of PEs is less, then this process will take several cycles till all the output maps have been generated.

Conclusion

The conclusion of this discussion is that parallelism improves the throughput of an implementation, however arranging these parallel units in a limited area and supplying them with appropriate data at appropriate times pose unique challenges to the memory system and interconnect designers. We can conclude from Table 4 that an architecture restricts the nature of optimizations and the type and degree of parallelism that can be exploited. Details such as the number of memory buffers, and the nature of the interconnects are very crucial, and also are the key determinants of the performance as we shall see in the subsequent sections.

4.2. Pattern-based computation reduction in convolution

Recall that a convolution operation consists of multiplication and addition operations between input activations and weights. Hegde

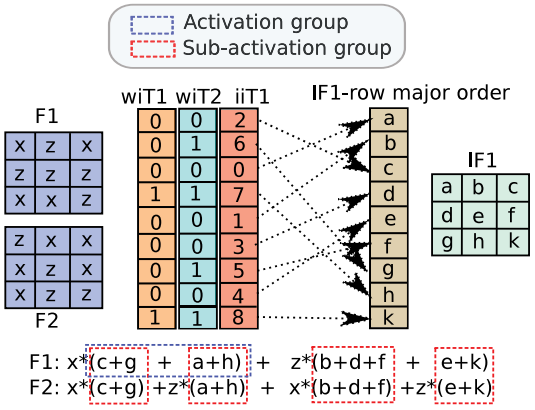

Fig. 8. Factorization of the dot product.
Source: Adapted from [61].

et al. [61] asserted that the number of unique weights in a CNN model is limited by the maximum number of bits used for their representation. For example, an 8-bit representation for weights will yield 256 unique weights. Given that the filter size and the number of filters in a CNN have been increasing with advancements in the design of CNNs, the values of weights will certainly get repeated. Thus, each common weight can be factored out of the multiplications it is involved in, thereby scaling down the number of multiplications by the weight's repetition count and clubbing together the activations to form an activation group [61]. As an example, the weight w_i in Eq. (5) was factored out to form the activation group $a_i + a_j + a_k$ in Eq. (6). This reduction is a result of the distributive property of the multiplication operation.

$$z = w_i \times a_i + w_i \times a_j + w_i \times a_k \tag{5}$$

$$z = w_i \times (a_i + a_j + a_k) \tag{6}$$

Let us describe a representative architecture by Hegde et al. [61], who propose to maintain two hardware structures: input indirection table, iiT, and weight indirection table, wiT (refer to Fig. 8). This figure shows two filters: $F1$ and $F2$. Assume that the weights are either x or z. $IF1$ is a feature map. The task is to compute a convolution between the filters and the feature map. While multiplying $F1$ with $IF1$, x will be multiplied with a, c, g, and h (element-wise multiplication). Similarly, z will be multiplied with b, d, e, f, and k. Similar results can be derived for the multiplication with $F2$. The exact expressions are shown at the bottom of the figure. Now, it can be noticed that the amount of work can be reduced by computing the repetitive expressions such as $a + h$, or $b + d + f$ only once.

This is recorded in the wiT table (one per filter). Consider $wiT1$ (for $F1$). Each entry contains a Boolean value, which is interpreted as follows. For $F1$, there is a 1 in position 4, which means that x will be multiplied with a sum of four elements of the feature map. The $iiT1$ table (one per feature map) stores the indexes of the elements in the feature map that need to be multiplied with x. Note that this is a 1D index (the paper shows a way to convert 2D indexes to 1D). Then in $wiT1$, the next 1 is at the ninth position. This means that the elements from positions 5 to 9 are multiplied with z. Their corresponding indexes are stored in $iiT1$. $wiT2$ is constructed on similar lines; however in this case there are four 1s, which means it divides the set of indexes into four groups. The first and third group are multiplied with x; the second and fourth group are multiplied with z.

Another work by Wang et al. [62] removes the redundant computations involved in a convolutional layer. They propose an architecture called *Cavoluche* that stores the repetitive patterns of parameters (weights and biases) of a layer in a buffer called the *pattern buffer*. The MAC of a weight pattern with an activation vector is computed in a SIMD-style PE array called a *P-tile* and stored in a cache called the P^2 cache. After the P^2 cache is filled, for any parameter pattern and activation vector combination, the P^2 cache is looked up first. If an entry is found, it is directly used else the computation is sent to the *P-tile*.

Discussion: The key idea in such approaches is a time and space tradeoff. We reuse partial results across filters. This reduces the amount of computation at the cost of increased storage space (wiT, iiT tables, P^2 cache, and pattern buffer). Such buffers cannot be made infinitely large and hence there is a need to design additional rules for storing and replacing the entries from the buffers. Additionally, this design increases the energy consumption by introducing additional buffer lookups; however the energy is also saved by reducing the number of computations.

4.3. Removal of ineffectual computations

Another way to reduce the number of computations is to remove the computations whose result is zero (or below a certain threshold), which are also known as *ineffectual* computations. This is a common phenomenon in algorithms for feature detection. Let us look at some of the important proposals in this area starting with a paper that describes the general approach.

Wang et al. [63] proposed a novel technique to prevent zero-valued kernel weights or input activations from being passed to the computation units, thereby, eliminating the number of ineffectual computations. They proposed an architecture called *Memsqueezer*, which inserts a two-level redundancy detector at both the input and the weight buffers (refer to Fig. 9). The redundancy detector detects and signals the computation units about the ineffectual computations. The first-level detector calculates the zero masks (see Fig. 9) for the inputs and the weights. The second-level of the detector calculates the effective number of zero-valued computations by computing a logical AND of the zero masks of both the inputs and the weights. This is then signalled to the address generation unit. This address generation unit is responsible for generating the addresses of weights and activations such that the weight and activation buffers can fetch data for computation. If all the addresses correspond to ineffectual data, then no data is sent to the PEs and the computation for all the PEs gets cancelled for that cycle. If some of the addresses are associated with zero masks, then the corresponding data is prevented from getting loaded to those PEs. In addition, the PEs become idle as a result of the zero masks, and are thus power gated to save power. These can also be utilized for some other useful work (see Section 4.5). Similar techniques have been used in hardware accelerators [64,65]. They prevent non-contributing computations from being performed, and also compress weights to reduce the storage requirements.

Discussion: This approach saves both energy as well as time. It saves energy because we have the option of quickly power or clock gating PEs when a computation need not be performed. It saves time when a PE is given a large number of multiplications to do, and the PEs need not complete their assigned tasks in lockstep. In this case PEs can quickly process their masks, and skip the multiplication step. There is a performance benefit because multiplication is much slower than checking bits in the masks. Let us now show some specific cases, where we can deliberately create more zero-valued entries.

Specific Cases

Kim et al. [66] introduced a kernel decomposition architecture, primarily tailored for binary weighted CNNs, where each element in the kernel is either +1 or −1. They proposed to decompose a binary kernel into two kernels: a base kernel and a filtered kernel. Base kernels are identical for all the binary kernels and are formed by replacing all the elements in the original kernel with −1. The information that is lost due to this simplification is captured in another decomposed kernel called the filtered kernel, which is formed by eliminating all the −1 weights from the original binary kernel and retaining only the +1 weights.

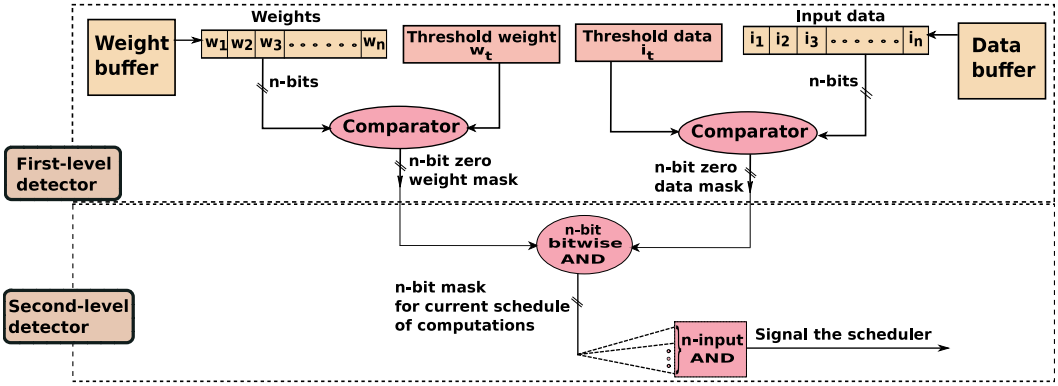

Fig. 9. Redundant data detection circuit.

Let us analyze the design for a computation with a single input feature map (see Fig. 10). The base kernels for all the filters are the same. For producing N output feature maps with N different filters, the base convolution is calculated once and is reused $N-1$ times, thereby saving energy. Then, the filtered convolution is calculated N times for N output feature maps, with the number of operations now reduced to half due to only 50% non-zero weights (assuming a 50-50 distribution of ± 1 weights) in the filtered kernel as compared to the original binary kernel. A naive implementation of this idea will actually increase the computation time because one convolution has been replaced by two; however, it is worth noting that the second convolution has a lot of zero-valued entries.

Another work in this area by Kim et al. [67] called *Zena* reads all the values required by the PE from the weight and activation buffers to the local register file. Thus, the memory accesses are not reduced at the global buffer level. The bit-vectors are then logically ANDed inside the PEs to generate the indexes for the non-zero activations and weights, which can then be read from the register file. Thus, this scheme reduces the computation time for each PE. However, *Zena* differs from *Memsqueezer* in the sense that the accesses to the global buffer are also reduced in the latter design.

Both *Zena* and *MemSqueezer* generate the bitmaps for both the activations and weights simultaneously, and then compute the zero masks. However, a proposal by Chole et al. [68] called *SparseCore* proposes to do this sequentially. They generate the bitmap for the elements read from the activation buffer and for each non-zero position in the bitmap, they fetch the corresponding weights from the weight buffer. If the weight is also non-zero, then only the pair of values is sent to the functional units. This saves the memory bandwidth of the weight buffer.

4.4. Prediction-driven computation reduction

The techniques discussed in Section 4.3 removed computations whose output is known to be zero. Another approach is to detect whether an activation from a preceding CNN layer contributes anything substantial to the succeeding CNN layer. Owing to the fact that the output of a convolutional layer passes through the ReLU and pooling layers, we can conclude that all the output activations from the previous layer do not contribute equally to the next layer. This is because the ReLU and pooling layers reduce the size of the feature map considerably by filtering out many activations (see their working in Section 2.1). Thus, the computations that contribute to the activations, which get filtered out later are *ineffectual* [69].

Song et al. [69] proposed to perform the computations of the higher-order bits and lower-order bits separately. Since the higher-order bits of an output activation are responsible for the final sign and have a greater role in determining the final value, they are solely used in the first stage called the *prediction stage*. The generated higher-order output activations are passed through a dedicated *Sign unit* and *Comparator unit* to predict the activations that would be rendered ineffectual by the ReLU and Pooling layers respectively. The predicted ineffectual activations are then marked as zeros in both the *ReLU table* and *Max table* (see Fig. 11(a)). These tables indicate the ineffectuality of the output neurons using one bit per output neuron. These tables are further used as an input by the *execution stage* for performing the remaining computations corresponding to the lower-order bits to generate the final result.

Discussion: Note that if the predictor stage has predicted certain activations to be ineffectual, the lower-order bits of such activations will not be processed in the execution stage. This introduces idleness among the PEs because their architecture is a 2D array with the same input activations broadcasted to all the PEs in the same column and the same weights broadcasted to all the PEs in the same row. Thus, if any PE is rendered idle due to an ineffectual value in the RELU or Max table, it has to wait for its neighboring PEs to finish their computations. This scheme saves power but does not reduce the computation time.

Thus, to reduce the computation time, we can increase the flexibility of data fetching by allowing only 1D sharing: share either weights or activations. The advantage is that if we allow sharing in just one dimension, the data for the other dimension (activation or weight) can be fetched from the memory without any synchronization with the neighboring PEs. This reduces the idleness and hence optimizes *energy*, however, this would increase the pressure on the memory system since the effectual activations needed by neighboring PEs may be located at random positions in the main memory. There are two components that lead to an increased *area* in this design: the two tables maintaining the prediction results and the area for two different bit-width multipliers for the execution and predictor stages. The area of the multiplier is reduced further by introducing a common serial multiplier that can be reused by both the predictor and execution stages for variable bit-widths. This however increases the *latency* because it becomes a function of the bit-width of the serial multiplier.

Similar work was done by Akhlaghi et al. [70] called *SNAPEA*. However, their approach is fundamentally different from the previous one (Song et al. [69]) in the sense that they classify and rearrange the weights. The ideal procedure to know that the sign of a computation is going to be negative at the earliest is as follows: ❶ first, perform all the multiply-add operations with the positive weights, ❷ subsequently, perform the multiplication with one negative weight per cycle and check for the sign after every cycle. As soon as the sign turns negative, the computation can be terminated and the output can be made zero (early ReLU). Note that to detect the termination condition at the earliest, the multiplication with the negative weights should be performed in the decreasing order of the magnitude of the weights: multiply with the highest magnitude negative weight first. This mode is called the *exact mode* of operation because it is known precisely if the computation will be ineffectual in future. However, this mode can lead to diminishing

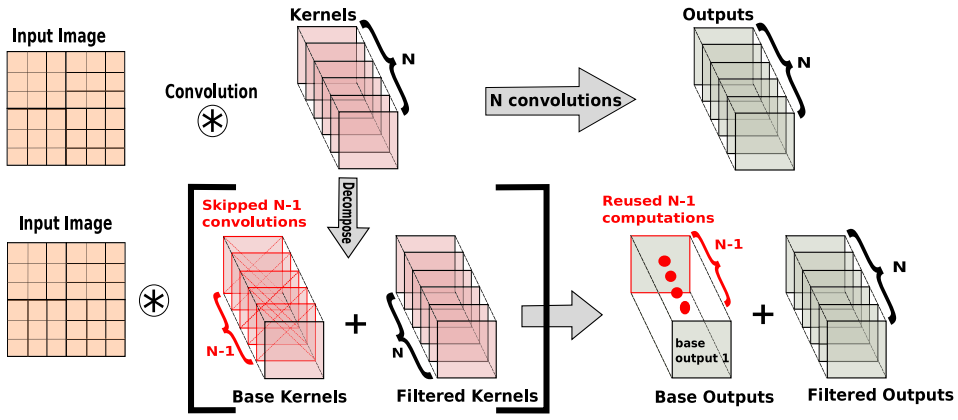

Fig. 10. Kernel decomposition scheme for the single input and multiple output case.

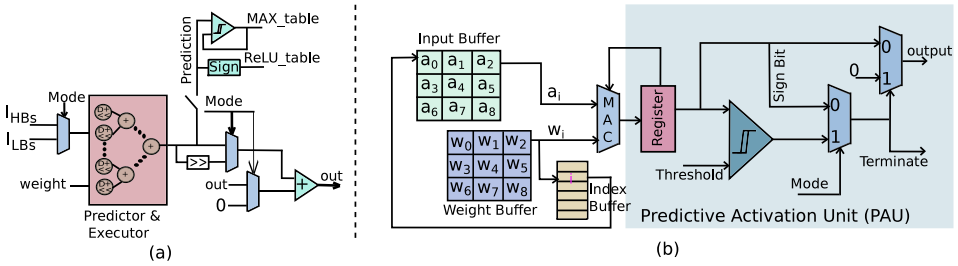

Fig. 11. (a) Prediction based (adapted from Song et al. [69]), and (b) SNAPEA (adapted from Akhlaghi et al. [70]).

returns if the number of negative weights is large while their magnitude is small.

Thus, a *predictive mode* exists in which a suitable threshold value of the computations is pre-determined and the number of MAC operations needed to achieve that threshold. Subsequently, the weights are arranged in three classes: predictive, positive, and negative. The predictive weights are those, which when multiplied by the activations and compared to the threshold value, give a prediction whether the computation will be ineffectual or not. If the computation is determined to be effectual, it enters the exact mode, else it terminates instantly. Note that this can lead to false positives when the predictive weights are not chosen appropriately. In many such cases, this strategy may result in producing a value that is less than the threshold, while further computations with the positive and negative weights could make it exceed the threshold.

The added advantage is that if the prediction is correct, we save $K \times K - N$ computations, where K is the size of the filter and N is the number of predictive weights. Note that as soon as the ineffectuality is detected, the PEs become idle and are hence power gated to save *energy*. In terms of the architecture, the reordering of weights at the time of convolution requires an *index buffer* to store the corresponding indexes for the activations and hence requires more *area* (see Fig. 11(b)).

4.5. Improving the PE utilization

As discussed in Section 4.3, the removal of ineffectual computations may render the PEs idle. In such a case, there are two options: either power gate the PEs, or utilize these PEs for some useful work. In the case of utilizing the PEs for useful work, the computation time can be further reduced. This is because we can get future computations to execute on these idle PEs. Thus, utilizing the idle PEs to execute the future computations can effectively reduce the overall computation time of the workload.

One such work by Delmas et al. [71,72] efficiently exploited the computation cycles involving zero-valued filter weights (sparse weights) by performing computations with the non-zero weights that were scheduled for a later time instant (promoting the weights in time). Specifically, they proposed two novel designs: *weight lookahead* and *weight lookaside*. These techniques are explained in Fig. 12. Please refer to this figure as we explain the techniques in the rest of this section.

The authors define a lookahead window, where the width of this window corresponds to the time steps. The *weight lookahead* technique promotes the non-zero weights from the left end of the same horizontal row in the lookahead window. Here, the leftward direction signifies a later point in time. This technique can eliminate the sparsity of the weights in this window with the promotion of weights within this window. Note that such a promotion in time would require activations corresponding to the promoted weight to be available in the input buffer. This technique may not prove to be beneficial, if a particular row has all non-zero weights. Additionally, since the rows are executed in parallel on different PEs, the row with the maximum number of effectual weights will decide the execution time. It may be possible that other rows have no effectual weights. Such a load imbalance can be overcome by the *weight lookaside* technique that allows weight stealing (or computation stealing) from the neighboring rows (within the lookaside window). Specifically, if the currently executing row has an ineffectual weight in the current time step, the row can steal a non-zero weight from a later time step of its adjacent row within the lookaside window.

Apart from the sparsity of the network, there is another factor that can lead to idle PEs. Suppose the mapping scheme allows the $height \times width$ dimension of the input feature map (2D Planar mapping) to be mapped to the PE array or the $depth$ dimension (channel direction mapping) to be mapped to the PE array. In both these cases, the PEs will go idle if the size of the PE array is larger than the number of activations in the mapped feature map dimensions. Liu et al. [73] propose an adaptive algorithm that dynamically maps the dimensions to the PE array such that the PE utilization is maximized. They employ a hybrid mapping that is a combination of both the planar and the channel direction mapping when both the mappings individually are inefficient.

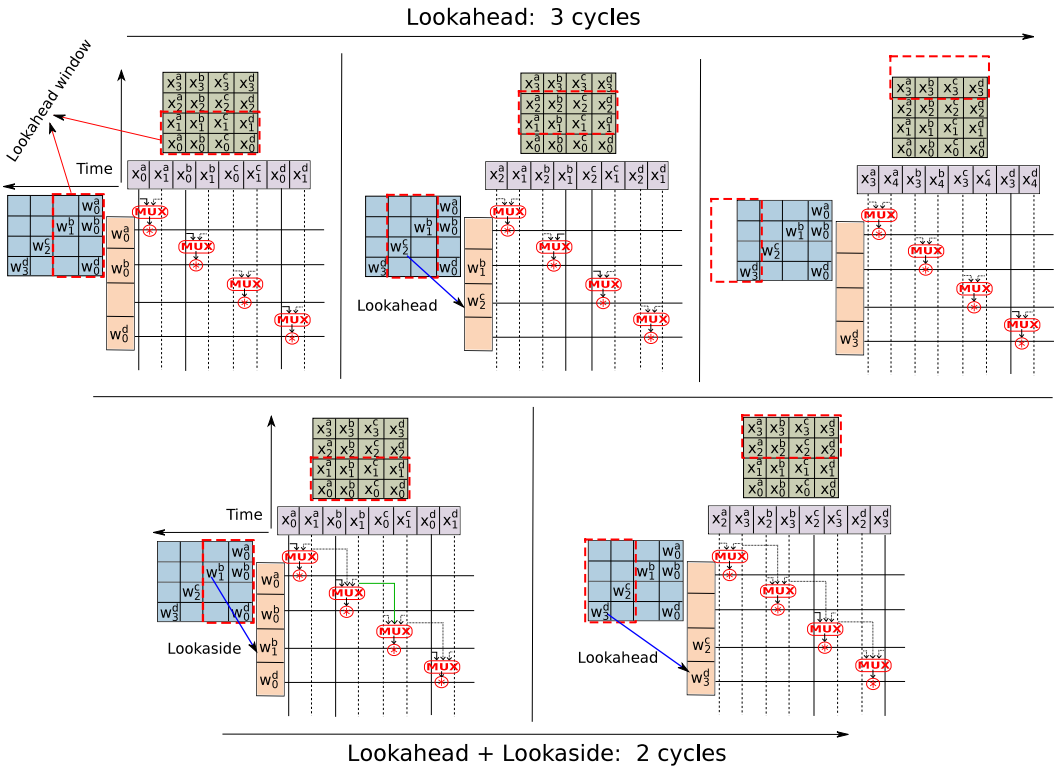

Fig. 12. Weight lookaside and Weight lookahead.
Source: Adapted from [71].

5. Reduction of the memory access time

Let us start with an example: Sze et al. [21] reported that AlexNet [2] requires 724 million MAC operations and 60 million weights to identify a 227 × 227 image. All the data including intermediate results cannot be stored in on-chip buffers and caches. It is thus necessary to make many accesses to main memory. For this convolution, Sze et al. estimated that the DRAM is accessed 3 billion times to fetch and process all this data. The important point to note is that the number of accesses is 50 times the total number of parameters. This means that the memory access time is dominated by accesses to main memory, which is significant even if modern optimizations such as streaming and pipelining are considered. The average memory access time needs to be reduced for realizing better performance.

Most proposals leverage the unique patterns of data accesses in CNN workloads, which are as follows: ❶ The same data is frequently accessed and can be reused in the future (temporally) by storing it in local buffers or can be reused at the same time (spatially) by a different computation unit, ❷ the accessed data leads to ineffectual computation, that is, the computed output is zero and hence it need not be fetched, and ❸ the data accesses can be made in parallel. Note that we had looked at ineffectual computations in the previous section as well; however, that was only from the point of view of reduction of computation time. We shall revisit many such issues in the next few sections, albeit from different points of view. Finally, note that in this section we will use the code in Fig. 3 as the running example.

5.1. Data reuse: Temporal reuse

This technique reuses the already fetched data from the off-chip memory by caching it in local on-chip buffers. Thus, the number of accesses to the off-chip memory is reduced. Effective caching of data in on-chip buffers has been shown to reduce the number of off-chip accesses by up to 500X [21].

As explained in Section 4.1, the mapping of different loop iterators to parallel PEs allows the exploitation of different types of parallelisms. Hence, these parallel computations decide the data access pattern. Thus, the iterators chosen for parallelization decide the kind of dataflow. For example, if the a_r and a_c loops (Fig. 3) are mapped to the parallel PEs, then each PE is responsible for the generation of one output pixel. Additionally, the partial outputs generated by each PE should remain in the PE's local storage such that it is reused till the final output is produced. Since there are three levels in our memory hierarchy of the reference accelerator architecture (see Fig. 4), data reuse can be exploited at all the three levels: register file, inter-PE network, and global buffer. Dataflow architectures use these storage structures to locally cache weights, partial sums and inputs. This is also referred to as keeping the parameters *stationary*. In more precise terms, if we say that a parameter is *stationary* in a memory structure, it means that the parameter is changing at the slowest rate in this memory structure while the rate of change of other parameters is much more.

To keep the data stationary we can take any number (usually <= 4) of loop iterators and partition them across the 2D PE array. This will lead to many such combinations of the loop iterators. Because of architectural considerations, only four such combinations are widely used as described by Chen et al. [74] (refer to Fig. 13). They result in four kinds of distinct dataflows.

The *weight stationary* (WS) dataflow [52,54,75,76] is exploited by mapping the w and l loop iterators to the parallel PEs. This architecture reuses the weights by storing them in the local register file of the PEs. In each cycle, the input activations are broadcasted to the PEs and partial sums are accumulated spatially over the PE array to form one output pixel. The weights are cached till all the computations with those weights are completed. Additionally, as can be observed that this mapping of the iterators will lead to one-by-one computation of the output pixels and hence will be slow. Thus, the partitioning can be done along the filters, where each PE corresponds to a_o, a_i-pair. However, this would require the entire filter of $w \times l$ to be cached in a PE, and hence a bigger register file will be needed.

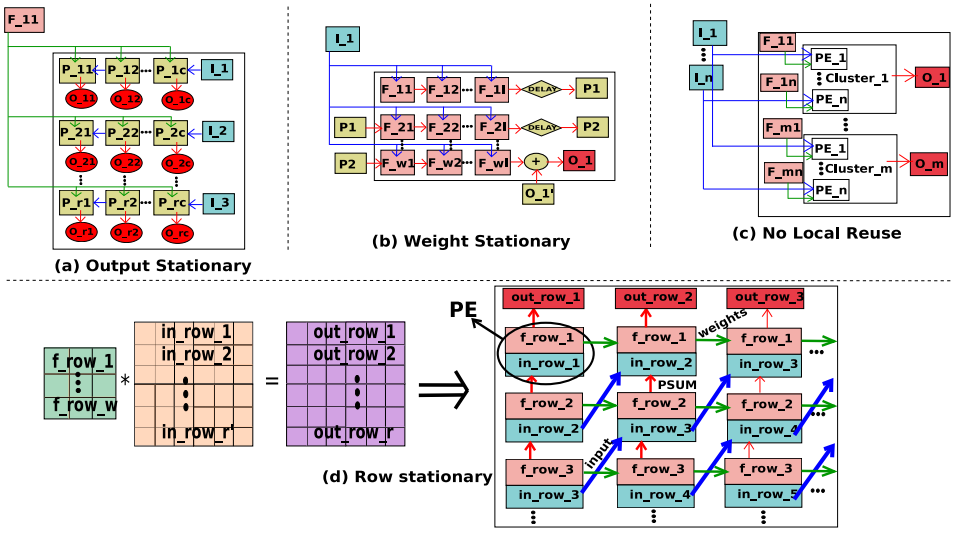

Fig. 13. Dataflow architectures for (a) Output stationary, (b) Weight stationary, (c) No local reuse, and (d) Row stationary. *Source:* Adapted from [21].

Similarly, the *output stationary* (OS) dataflow [46,77] is exploited by mapping the a_r and a_c loop iterators to the parallel PEs. It reuses the partial outputs in each PE. Since each input activation contributes to multiple output pixels in a convolution, it is reused across the neighboring PEs via the inter-PE array. The weights are broadcasted to all the PEs. Note that in cases where $a_r \times a_c$ is smaller than the PE array size, the parallelism can be exploited along the a_o dimension to improve PE utilization. It will still be output stationary, albeit the output activations of different output feature maps will be generated in the PE array.

The dataflow that does not keep any of the data, weight, input, and output, stationary at the local register file is called *No Local Reuse (NLR)* dataflow [44,78–80]. The data can flow between the PEs in a systolic fashion. The PE array can be parallelized either along a_i or a_o iterator to compute one or multiple output maps respectively. In Fig. 13(c), the a_o iterator is parallelized for a cluster of PEs and a_i iterator is parallelized within the cluster of PEs. Within each cluster of PEs, the partial sum flows between the PEs systolically to generate one output map.

Another dataflow architecture that keeps the input activations stationary at the RF level is called the *input stationary* (IS) dataflow. Here, the parallelism on the PE array is along the a_r and a_c iterators. This appears similar to OS, however the important point is that the partial sums are cached in the RF in OS dataflow, while the input activations are stationary in RF in IS dataflow. Here all the input activations are multiplied with the weights and accumulated to get one partial sum. If the parallelization is done along a_i iterator too (given that there are sufficient resources), the entire output pixel can be generated. It will still be input stationary albeit those will be input activations from different channels.

Finally, the all-in-one reuse architecture called the *Row stationary (RS)* dataflow [74] (see Fig. 13(d)) reuses all the types of data – weights, activations and partial sums – at the register file. This is done by dedicating a PE for processing a particular row of the input feature map and the filter. If the size of the filter is $W \times L$, W rows of the input feature map and the filter are processed to produce one row of the output feature map and hence W PEs are required for producing one row of the output feature map. Thus, $W \times R$ PEs in total are required to process all R rows of the output feature map. This can be exploited by mapping the a_r and w loops in the running example.

We observe from Fig. 13(d) that the reuse is being done at two levels: inside each PE and across the PEs. Inside each PE, the filter row is reused till all the output elements of a row are generated, and the overlapping input activations are reused between the two sliding windows. Across the PEs, the filter rows are reused horizontally across all the columns (alternatively all the rows of the output feature map). The rows of the input feature map are reused in a diagonal manner and the partial sums are accumulated vertically to get one row of the output feature map per column. We show the relationship of the loops that need to be unrolled with the different dataflows in Table 5.

Table 5
Types of reuse exploited by different proposals.

Dataflow	Acronym	Reuse	Loop iterators
Weight stationary	WS	weights	w, l
Output stationary	OS	partial sums	a_r, a_c
No local reuse	NLR	no reuse in the register file	a_o, a_i
Row stationary	RS	filters, activations, and partial sums	a_r, w

Discussion: Due to varying degree of reuse at different levels of the local memory hierarchy, the number of DRAM accesses is reduced substantially and hence the *energy* consumption is reduced. Additionally, the cycles spent in accessing data from the memory is reduced, thereby reducing the CPI (memory) and hence improving the *throughput*. However, this technique is heavily dependent on the reuse at on-chip buffers and hence the *area* for the on-chip storage structures increases. A quantitative comparison of the four dataflow architectures in terms of *energy* has been given by Sze et al. [21]. Our qualitative analysis suggests that the NLR dataflow should have the maximum energy consumption given that it does not reuse data at the highest level of the memory hierarchy (most energy-efficient). Also, since the RS dataflow has the maximum possible data reuse at the highest level of the memory hierarchy, the energy spent is typically the least. Comparing the WS and OS dataflows, the energy consumed while accessing the global buffer is more for the WS dataflow as compared to the OS dataflow because for each partial sum, the WS dataflow writes the data to the buffer and reads it back at the time of accumulation. While in the case of the OS dataflow, the partial sums are held stationary in the register files.

We perform a quantitative analysis of the different dataflows using Timeloop [81] in Section 8.2. Our quantitative results corroborate the qualitative intuition given in the previous paragraph. We find that the NLR dataflow is the most energy-inefficient while RS dataflow is the most energy-efficient dataflow.

5.2. Data reuse: Spatial reuse

In the previous section, we discussed techniques to reuse data by storing it in local buffers. We can also leverage spatial locality by

routing data between the PEs such that we can avoid costly memory accesses. Good routing techniques achieve an equitable tradeoff between bandwidth and latency, where the latter is also determined by the congestion in the network. The congestion in turn is a function of the number of accesses that the PEs make to fetch the filters weights and input activations from the memory. Consider an example. Assume we are convolving a set of input feature maps with the same filter to generate a batch of outputs. In this case the filter's weights can be shared across the PEs, and there is no need for PEs to make independent memory accesses to the lower level of the memory hierarchy.

5.2.1. Pinned input feature maps

In this regard, Dundar et al. [82] presented a routing scheme that maximizes the ratio between the number of output feature maps and the number of accesses to an input feature map. To maximize this ratio, when an input feature map is accessed, it should be convolved with all the filters to produce multiple output feature maps in parallel. The method to do this is to route the input map to all the PEs such that they can simultaneously compute all the output feature maps. However, if the feature map is a sparse matrix and we wish to eliminate zero-valued computations, then distributing it among PEs is non-trivial, and consequently such optimizations lose a lot of their benefits. In this case, to get the same benefits we can run multiple CNN inferencing applications for a batch of inputs in parallel.

5.2.2. Flexible NoCs and dataflow patterns

Let us compare two architectures in this space: *flexflow* and *Eyeriss-v2*.

As explained in Section 4.1, partitioning and mapping different loops to the parallel units can exploit different kinds of parallelism depending on the loop iteration that is mapped to the PEs. It is worth noting that the data reuse pattern of an accelerator is highly correlated with the mapping of the loop iterators to the PEs. For example, mapping the innermost iterators, w and l, to the PEs would exploit weight stationary dataflow and require the $w \times l$ weights of the filter to be cached locally in the register file of the respective PEs.

Inputs, weights and intermediates (partial sums) have varying degrees of data reuse opportunities [83]. Thus, to maximally exploit all the types of data reuse, *FlexFlow* finds optimal values of the tile sizes in our running example, $b_1 \ldots b_6$, and hence follows a mixed dataflow approach. The dataflow required by different groups of PEs in the PE array is different, hence the routing of data becomes a challenge. Since the interconnects are designed to just broadcast the reusable data, this design performs sub-optimally when there is low data reuse because all the broadcast links cannot be fully utilized. Nevertheless, this design almost never leads to *congestion* due to its simplified routing scheme.

Eyeriss-v2 [84] tackles such uncertainty in the data reuse opportunities by adopting a flexible NoC architecture that can adapt to unicast, multicast or broadcast patterns. Note that unicast favors low data reuse where each PE needs a different data word, multicast favors a fair amount of data reuse where multiple PEs work simultaneously on the same data, and broadcast favors the highest data reuse where all the PEs work simultaneously on the same data. Additionally, they do not exploit a mixed dataflow pattern unlike *FlexFlow*, and instead follow an improved row stationary dataflow pattern that uses larger tile sizes.

5.2.3. Forwarding data between PEs

Another hurdle that arises when generating an output feature map is that the number of convolutions needed to produce an output feature map is much larger than the number of processing units. Thus, there is a need to temporarily store the partial sums. Storing them in large on-chip or off-chip memory structures incurs high read and write latency overheads. Jin et al. [55] proposed a routing scheme that sends the stream of partial sums produced by a PE to its neighboring PEs. Thus, the partial sums get processed immediately by effectively routing them to the destination unit, without having to store them in memory.

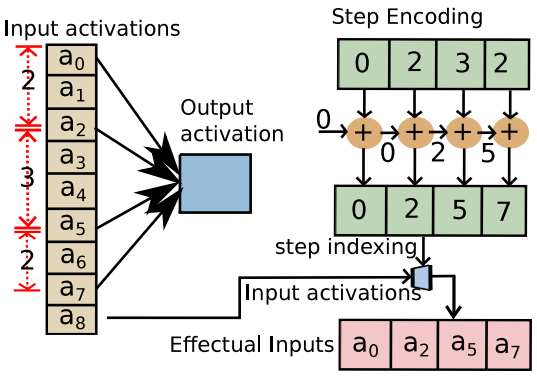

Fig. 14. Step Indexing. *Source:* Adapted from [85].

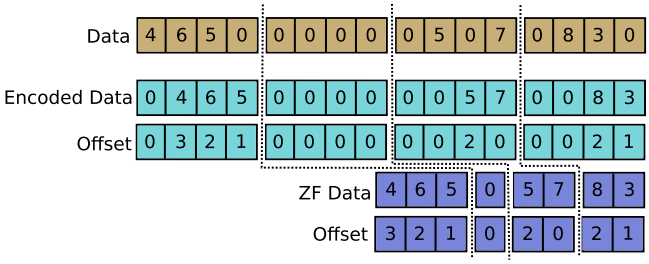

Fig. 15. ZFNAf encoding format. *Source:* Adapted from [86].

5.3. Eliminate loading of ineffectual data

Values that are close to zero, and are too small to make a difference in the computations are termed *ineffectual*, and they need not be loaded from memory. Most proposals replace them with 0s. The computations (also termed as *ineffectual*) that use those values can be skipped. Feature maps where these small values have been converted to 0s are known as sparse feature maps. If different PEs process different parts of these feature maps, they are expected to take different amounts of time because of varying degrees of ineffectual computations. This presents an opportunity for accelerator designers to make their PEs aware of the level of sparsity in the feature maps.

Let us proceed assuming a conventional architecture, where we discuss different optimizations to eliminate the need for loading ineffectual data from memory. We shall first discuss methods to generate indexes that point to ineffectual data, and then we shall discuss methods to store them efficiently.

5.3.1. Generating indexes to effectual data

Zhang et al. [85] proposed an architecture called *Cambricon-X* that does not fetch those activations that correspond to zero-valued weights. The idea is that for producing an output element, since the filters are known in advance, the input activations corresponding to the zero-valued weights need not be accessed. Hence, the memory access time for fetching the input activations can be reduced.

For an element in the output feature map, the indexes of the input activations that can influence the output's value is needed. These indexes depend on the value of the corresponding weight: zero or non-zero. The information of the non-zero weights is stored in a vector. Each element of the vector contains the difference in the indexes of the two consecutive non-zero weights. This vector is then used by the *indexing module* to generate the indexes for the input activations and store them in an *indexing buffer* (one per PE). A variant of run-length

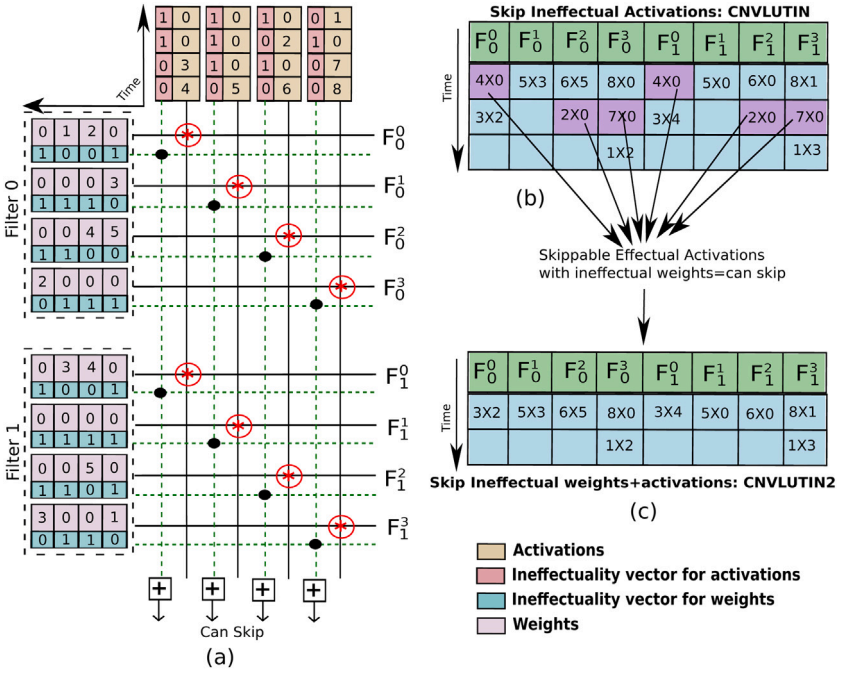

Fig. 16. (a) Working of Cnvlutin and Cnvlutin2.
Source: Adapted from [87].

encoding known as step-indexing approach is used (as shown in Fig. 14) to generate the offsets for the input activations in the *indexing buffer*.

The reduction in *latency* and the improvement of *throughput* comes at the cost of additional *area*, which is required for the indexing module and indexing buffer. We need as many indexing buffers as the number of convolutions being performed concurrently. In terms of *energy*, the memory access energy is much more than the energy required to access the on-chip indexing buffer [21].

On similar lines, Han et al. [65] also proposed a method called *EIE* to take advantage of sparsity in the activations and filter weights. However, their indexing method is based on a variation of the CSC (compressed sparse column) format [88] that tries to compress the bitmap directly instead of converting it to a bit vector first. This indexing method compresses and stores the non-zero weights using two tables: one containing the index of the next row and the other containing the index of the next column.

Note that the elimination of ineffectual data in *EIE* is driven by the zero-valued elements in both the weight and the activation buffers. This is done by using the CSC based indexing technique to detect zero-valued weights. Here zero-valued activations are detected when they are read from memory. The indexes that correspond to ineffectual data are broadcasted to all the PEs such that they can avoid processing or fetching them. As compared to *Cambricon-X*, *EIE* exploits more features namely weight sparsity, and input sparsity. This gives it additional opportunities to reduce time and energy.

Another recent work by Zhou et al. [89], *Cambricon-S*, exploited the ineffectuality of both weights and activations by calculating bit vectors for activations and weights called *activation indexes* and *synapse indexes* respectively. These vectors are calculated by an indexing module called the *Neuron selector module (NSM)*. This technique differs from *Cambricon-X* in the sense that it exploits both activation and weight sparsity. Additionally, it exploits activation sparsity that arises at runtime due to ReLU layers. For this purpose, there is an additional indexing module at each PE that tracks ineffectuality at runtime. Furthermore, the footprint is reduced by storing a compressed version of the weights in the synapse buffer. The weights are compressed using Huffman encoding that assigns a class to each weight based on its probability of occurrence. Thus, any access to the effectual weights has to first pass through a decoder module (weight decoder module or WDM) before being used for computation.

5.3.2. Storing effectual data along with indexes

There are two broad approaches for storing sparse data: store indexes to effectual data, and store a bitmap where 1 indicates that the corresponding element is effectual, and 0 indicates that it is ineffectual. Depending upon the degree of sparsity, we prefer one approach over the other.

Storing Explicit Indexes

Albericio et al. [86] propose *Cnvlutin* that uses a *Zero-Free Neuron Array format (ZFNAf)*-encoding to determine and encode only the effectual activations. In ZFNAf, each non-zero activation in an accessed block of activations is encoded in the form of a value and an offset, where the offset represents the index of the non-zero activation. This encoding is done at the output of the layer and subsequently the encoded output is stored in memory (see Fig. 15).

Assume that each row of the input matrices is divided into a set of blocks with n elements each. The offset of an element in a block thus requires $log_2(n)$ bits. For m effectual activations (where $m \leq n$), $mlog(n)$ bits are required. In addition, $16m$ bits are needed to represent all the values (assuming a 16-bit number system). In case of uncompressed data, $16n$ bits (16 bits per value) are needed. Thus, area is saved if $16m + mlog(n) < 16n$. Lastly, note that this approach encodes the input data prior to execution.

Similarly, Liu et al. [90] proposed an architecture called *Swan* that stores the non-zero weights in the compressed form in the weight buffer along with their indexes. They introduce a selector circuit just before the MAC unit that selects the activations from the activation buffer based on the indexes of the non-zero weights. Thus, the ineffectual weight accesses are removed. Additionally, if the accessed activation is zero, the MAC unit is power-gated.

Storing Bitmaps

Recently, *Cnvlutin2* [87] was proposed that detects the ineffectuality of both the activations and weights on-the-fly while reading them from the buffers. This approach is different from *Cnvlutin* where the effectual activations along with the offsets are first stored in memory

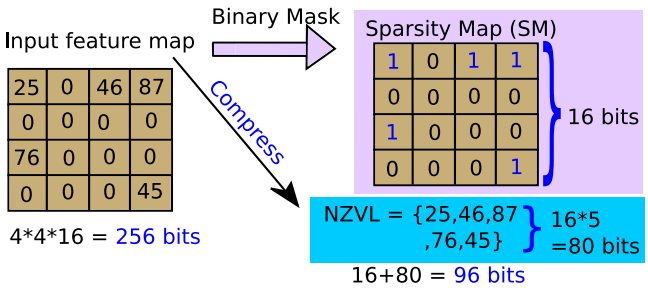

Fig. 17. NZVL encoding scheme.
Source: Adapted from [91].

and are then read by the next layer. Since *Cnvlutin2* exploits the ineffectuality of both the inputs and weights, it saves more computation cycles as shown in Fig. 16. Specifically, *Cnvlutin2* employs the circuit shown on the left side of Fig. 16 that calculates the logical AND of the bit vectors for both the activations and weights. A '1' in the result indicates an effectual computation and the activation/weight corresponding to this value is read from the buffers. Thus, *Cnvlutin2* saves memory access time by preventing the access of ineffectual activations and weights while *Cnvlutin* only saves in terms of ineffectual activations. Additionally, the on-the-fly calculation in *Cnvlutin2* saves the memory storage overhead of the offsets needed in the case of *Cnvlutin*. Fig. 16(a) shows the corresponding circuits and a sample execution for both the designs is shown in Figs. 16(b) and (c) respectively.

The total storage overhead with approaches that store indexes [86] (*Cnvlutin*) was calculated to be $16m + mlog(n)$, where m is the number of non-zero activations, and n is the number of elements in each block. Now assume that we wish to store a bitmap, or a bit vector for a set of activations, where a single bit indicates if the corresponding element (with the same index) is effectual or not. In this case the number of bits that we require is T, where T is the total number of activations in the matrix that we wish to compress (filter or feature map). Thus, the mathematical tradeoff is very simple; if $T > 16m + mlog(n)$, we use bitmaps, otherwise we explicitly store indexes.

For CNNs that will benefit by this storage method, Aimar et al. [91] proposed *NullHop*. It uses the NZVL encoding to store the non-zero activations in a list and also maintains a sparsity map (SM) for the bitmaps (refer to Fig. 17). Since the SM is for the entire feature map and NZVL has only effectual activations of the feature map, an additional circuit is needed to determine the exact positions of the effectual activations. The insight is that the energy required for a computation is typically lower than a few memory accesses [92].

5.4. Miscellaneous techniques

This section covers some miscellaneous optimizations that lead to a reduction in the memory access time.

5.4.1. Processing multiple images

Even though reusing weights is in general very effective; however, it can be counter-productive if there is very little repetition in the values of the weights. Since a weight is a characteristic of the network and not the input, processing a batch of input images will ensure a greater degree of weight reuse. Thus, this technique allows for better reuse of data by controlling the number of input images being processed simultaneously. This technique is always the preferred approach when any other source of reuse is not available. However, processing multiple images at once has its pitfalls in terms of a higher requirement of on-chip area, and additional memory bandwidth.

5.4.2. Double buffering

To optimize filter reuse, Shen et al. [93] proposed a buffering mechanism that achieves a favorable tradeoff between on-chip storage, and off-chip bandwidth. They use a formal approach by proposing a set of constraints and objective functions. Buffering is a good approach in array based CNN accelerators with flexible interconnects; it is not very useful in architectures that rely on a streaming pattern such as systolic arrays. To overcome this Jouppi et al. [94] introduced *double buffering* in their systolic *TPU-v1* architecture to hide the memory access latency. They create an overlap between computation in the systolic array, and fetching data for the next set of computations. This strategy effectively hides the time it takes to buffer a large chunk of data.

5.4.3. Large on-chip buffers

Sim et al. [95] propose to use a large unified on-chip buffer for storing both input and output feature maps. Since the entire operation is on-chip, double buffering is not needed and the saved area can be used to increase the unified buffer size. Such a design allows for inter-layer feature map reuse. However, in cases where the unified buffer is not able to accommodate the feature maps, there is a need to tradeoff the amount of the two maps that can be stored on-chip or accessed externally. The tradeoff depends on the size of the feature maps, the size of the unified buffer, and the amount of reuse needed.

5.4.4. Remove data duplication

Gao et al. [96] propose to remove the data duplication in the on-chip buffers of the PE array clusters. Their setting assumes a large PE array that is divided into smaller PE array clusters, with each cluster having a local on-chip buffer. When these clusters compute the different output feature maps, they will require the same input feature map at some point of time. A more common implementation would fetch an input feature map and broadcast it to all the clusters so that it is not accessed multiple times by multiple clusters, however this leads to data duplication and reduces the total effective on-chip memory available. The proposed architecture, *Tangram*, brings in different feature maps for the clusters and each cluster processes different feature maps at a time. The feature maps are then rotated among the neighboring PEs to be processed by all the clusters. This approach is called *buffer sharing dataflow* and it maximizes the overall effective on-chip buffer capacity. However, the communication latency is added due to the rotation of data.

6. Reduction in the memory footprint

CNN models have evolved significantly over time and have grown in size, thereby leading to increased memory requirements. An estimate of the memory requirements can be obtained from the sizes of the Caffe models of AlexNet and VGG16. The size of AlexNet's [2] base model is approximately 233 MB while for VGG16 [97] it is more than 500 MB. Das et al. [98] did further analysis of the memory required by the Caffe model of VGG16. It is 528 MB (as per their estimate), and if we consider the storage required for the intermediate data, then the cumulative total is 732 MB. Given the large memory requirements of these models, it is not possible to retain all the data in the on-chip caches, which makes it difficult to achieve good performance due to continuously fetching data from off-chip memory.

This is primarily due to the following reasons: ❶ a large number of intermediate values, ❷ full-precision storage of the input activations and filter weights, and ❸ storing all the effectual and ineffectual data. Thus, there is a need to reduce the memory required by these algorithms. Researchers have proposed to ❶ *reduce the precision of inputs and weights or compress them*, ❷ *not store intermediates and instead consume them (this is an overlapping technique with the previous section as this also reduces the memory access time)*, and ❸ *not store ineffectual data or bits*. Note that in Section 5, our focus was on reducing the memory access time by keeping data closer to the PEs, however, in this section our aim is to reduce on-chip storage of data as much as possible.

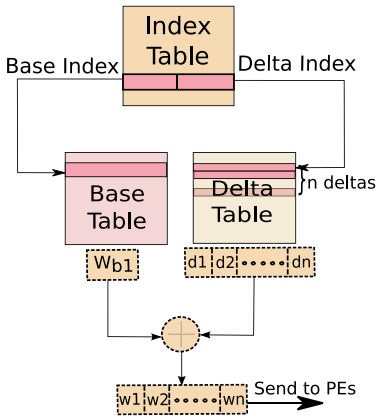

Fig. 18. Weight storage architecture.

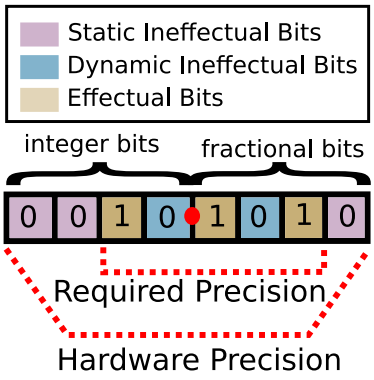

Fig. 19. Ineffectual bits in the precision.

6.1. Data and weight compression

Most techniques compress the input data and filter weights using lossless compression algorithms. This data is decompressed when it is required for computation. Notably, Wang et al. [63] designed a memory architecture called *Memsqueezer* that employs compression techniques for both data and weights. They define the *active weight buffer set* to be a hardware structure that contains all the weights. It uses the *base-delta partitioning* method to store the weights in a compressed form. In this method, a subset of weight values are decomposed into two values: a mean value, and a set of differences. Since the differences are expected to be small if the values are centered around the mean, a fewer number of bits are required to store the difference values as compared to the number of bits required in an uncompressed form. Consider an example. We want to compress the set $\langle 5, 5, 6, 7\rangle$ in a system with a 4-bit number system. We would ideally require 16 bits (4 bits per number). However, if we assume that the base is 6, then we require 4 bits to represent it, and the remaining values can be expressed as offsets. They would be $-1, -1, 0, 1$. We require a 2-bit number system to represent the offsets (deltas). Thus, the total number of bits required in the encoded version is 4 bits for the base and (2×4) bits for the deltas, making it 12 bits. We thus reduce the storage requirements from 16 bits to 12 bits (25% reduction).

These partitioned and compressed values are stored in two tables: Base Table and Delta Table, which are indexed by an Index Buffer (see Fig. 18). A weight value is generated by accessing the values from the base and delta tables and adding them. Here, the overheads are in terms of both time and space; however, in most cases the storage overheads can be reduced significantly.

Additionally, Wang et al. [63] define a *data buffer set* for storing the intermediate data. Since the intermediate data in a CNN dynamically changes (unlike weights that are shared across the layer), the base-delta method does not work well. In this case, a frequent pattern mining [99] method is used to compress the data. This is done by replacing the frequently seen patterns with shorter identifiers (similar to Huffman coding). This method finds patterns in a block of intermediates, and stores the frequently encountered patterns in a compact pattern table by assigning a unique index to each pattern. The *area* overhead is reduced due to the storage of encoded data but this introduces additional computational overheads to decode the data before processing. In terms of *latency*, there is an improvement because the partitioning method uses a fewer number of cycles to access the compressed repetitive data from the pattern table as compared to accessing the original uncompressed data.

6.2. Reduction of ineffectual bits in the binary representation

The ineffectual bits in a number's representation (input or filter weight) are of two types: statically ineffectual and dynamically ineffectual (see Moshovos et al. [100–102]). Fig. 19 explains the difference between the two forms. Specifically, statically ineffectual bits are those that are not necessary to represent the number.

In Fig. 19 we see that the three zero bits – one in the suffix and two in the prefix – are not needed for the representation. These are statically ineffectual bits. In contrast, there are two zero bits in the remaining five bits (necessary for representation). Any multiplication with these zero bits will result in a zero and hence these are also ineffectual from the point of view of multiplication. These are dynamically ineffectual bits.

6.2.1. Removal of statically ineffectual bits from activations

A recent proposal by Judd et al. [103] called (*STR*) exploited the desired precision at each CNN layer. Specifically, the precision of a layer is decided by the maximum number of statically effectual bits needed for the representation of all the activations in the layer. Such a precision reduction technique can result in different precision requirements for different layers and hence different multipliers are needed for different layers with different bit-widths. Thus, they introduced a serial multiplier unit that performs bit-wise multiplication and hence can be reused across all the layers. However, the computation time of this unit is proportional to the precision of the layer. Using a serial multiplier increases the latency of the multiplication operation by the bit-width (b) of the serial data, thus they used b such parallel units to compensate for the increased latency. Hence, the performance over the fixed bit-width (f) implementation improved by f/b, assuming that the b parallel units are able to efficiently hide the increased latency due to the bit-serial multiplication. An additional observation is that each layer has a different desired precision depending on the static ineffectual bits in all the activations, the buffer size at the output of each layer can be customized accordingly to reduce the memory footprint.

STR was further advanced by the introduction of *Dynamic STR* [104]. The primary difference between the two is that *STR* already knows the per-layer precision based on profiling methods, while *Dynamic STR* determines the precision of a group of activations at runtime. The precision considered by the *Dynamic STR* is a further refinement over *STR* because it takes into account only the activations currently being processed unlike *STR* that considers all the activations of a layer. They introduced a circuit to determine the required precision (on-the-fly) for the current batch of executing activations. Dynamic approaches are always more effective than static approaches because they use more information, however, they have significantly more overheads also at the same time.

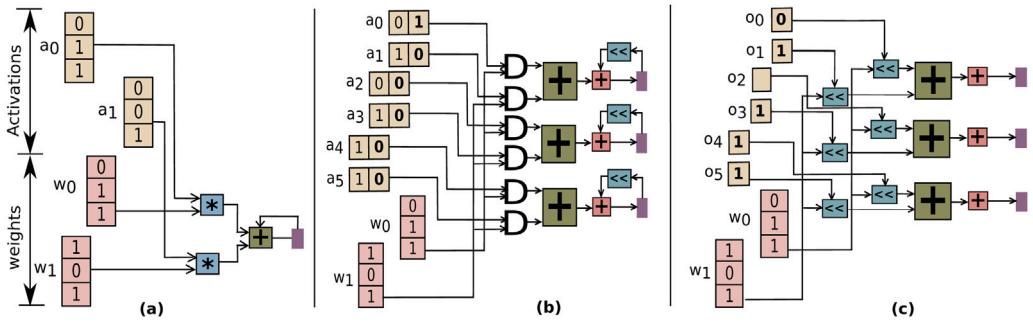

Fig. 20. (a) Parallel multiplier unit, (b) Stripes with equivalent throughput, and (c) Pragmatic with equivalent throughput.
Source: Adapted from [105].

6.2.2. Removal of dynamically ineffectual bits from activations

An advancement of STR was proposed by Albericio et al. [105] called *Pragmatic*. *Pragmatic* exploits dynamic ineffectuality to determine the precision (see Fig. 19). Similarly, Sharify et al. [106] propose an extension to *Pragmatic* by exploiting the dynamic ineffectuality in both activations and weights. An extension over *Pragmatic* was proposed by Mahmoud et al. [107] that performs differential convolution. For consecutive windows of an image it stores the input image in its original form only for the first convolution and stores the difference of the pixel values for the other windows w.r.t to the corresponding pixels in the first window. Due to the presence of spatial correlation, these difference values are small and hence require even lower precision.

As explained earlier dynamic ineffectuality refers to those bits in the precision that are necessary for representation but are redundant in the computations. If we take such an ineffectuality into account, we can further reduce the buffer sizes required for the storage of the intermediates produced by each layer. Fig. 20 explains the difference between *STR* and *Pragmatic*. Note that all the three implementations in Fig. 20 have the same throughput. As a multiplier migrates from a bit-parallel implementation in Fig. 20(a) to a bit-serial implementation in Fig. 20(b) and (c), the throughput of the implementation scales down by the number of bits. Since the activation originally has a 3-bit representation, Fig. 20 implements three instances of the activation-weight multiplication for both the bit-serial designs: STR and Pragmatic. As shown in Fig. 20(b), *STR* reduced the precision of all the activations from 3-bits to 2-bits because all the activations ($a_0 = 001, a_1 = 010, a_2 = 000, a_3 = 010, a_4 = 010, a_5 = 010$) have at least 1 statically ineffectual bit and hence it is removed from the representation. Similarly, Fig. 20(c) further reduces the representation by encoding only the positions of the effectual bits in the activations in Fig. 20(b).

Despite the increased on-chip storage requirement for the parallel computation units (for equivalent throughput) in all the three works, the *area* is not adversely affected due to the reduced precision of the data being stored. As compared to *STR*, the removal of dynamic ineffectual bits by *Pragmatic* reduces the precision even further, thereby reducing the *area* required for the storage. Similarly, the *throughput* and *latency* are improved in the case of *Pragmatic* because of the complete elimination of computations involving the ineffectual bits. The bit-serial multipliers also introduce additional energy efficiency.

6.2.3. Removal of ineffectual bits from weights

All the three bit-serial designs discussed till now exploited the ineffectuality in the representation for the activations only. Sharify et al. [108] advanced the field by exploiting the per-layer ineffectuality in the weights too. They used bit-serial multipliers as well. However, the difference with respect to prior implementations is that those implementations fixed the bit-width of the weights and fed input activations serially bit-by-bit while *LOOM* [108] exploited serial processing for both the operands of the MAC operation: weights and activations. They thus reduced the *area* requirements even further at the cost of additional latency. For example, in *STR* one of the 16-bits of an activation was multiplied with all the 16-bits of a weight per cycle, resulting in a total of 16 cycles for a 16-bit x 16-bit multiplication. However, *LOOM* multiplies a 1-bit weight with a 1-bit activation per cycle. The 16-bit x 16-bit multiplication requires 256 cycles in the worst case. Specifically, the *performance* advantages for *STR* and *LOOM* over a baseline (fixed-precision) design will be $16/a_p$ and $256/(a_p \times w_p)$ respectively (assuming a fixed 16-bit precision for weights and activations in the baseline design), where a_p and w_p are the reduced precisions for the activations and weights respectively.

6.2.4. Optimizing the processing of the effectual bits:

In this section, we discuss the works that replace the bit-serial multipliers by a flexible multiplier architecture that adapts itself according to the given design. The limitation of a bit-serial multiplier is that it works well when the bit-widths are in a given range. This restricts the amount of effectuality that can be exploited by all the layers of a CNN. An easy solution to this could be to have multiple bit-serial multipliers for all the layers of a CNN such that they can be adapted to different bit-widths. However, bit-serial multipliers are slow, and thus they limit the final performance. Sharma et al. [109] proposed to use a 2×2 multiplier as the basic unit and called it a *BitBrick*. It is possible to fuse these bricks to create larger multipliers on demand. This depends on the values of the weights and activations in a given layer.

Hence, the main challenge is to determine the method to fuse these *BitBricks*. Recall that we have described two paradigms for reusing data: *spatial* and *temporal*. These concepts can be adapted for this case as well. First an n-bit multiplication is broken down into smaller 2×2 multiplications. Subsequently, these multiplications are either distributed across different clusters of *BitBricks* (spatial) or are scheduled on the same cluster in a time-multiplexing manner (temporal). Sharma et al. describe an approach to first create optimal clusters of *BitBricks* on demand, and then reuse them both temporally as well as spatially.

Lee et al. [110] proposed an optimization to reduce the number of computations done via bit-serial arithmetic in their architecture called *UNPU*. The main insight is as follows (refer to Fig. 21). In a 16-bit x 16-bit MAC, for each 1-bit x 1-bit bit-serial operation between a weight and an activation, the corresponding input activations say, X, Y, and Z (each 1-bit) are multiplied with the corresponding weights say, $w_x[MSB : LSB]$, $w_y[MSB : LSB]$, and $w_z[MSB : LSB]$, bit-serially over $MSB-LSB$ cycles. These multiplications are accumulated to form a 1-bit output activation per cycle. Thus, the input activations are reused over $MSB-LSB$ cycles. Since each of the input activations X, Y and Z is the same for $MSB - LSB$ cycles, and the number of unique combinations of (w_x, w_y, w_z) is limited, the number of unique output activations is also limited. In our case of 3 weights (w_x, w_y, and w_z), the number of such unique combinations will be $2^3 = 8$ and hence 8 possible values are stored in the LUT, which is indexed by the unique weight combinations. Thus, upon subsequent MAC operations, if the value exists in the LUT table, the bit-serial multiplication is replaced by an LUT access. This design reduces the *area* overhead of bit-serial processing unit by introducing a small lookup table (LUT) inside the PEs. Note that the idea of introducing the LUT at the highest level of the memory hierarchy improves the *energy* requirements.

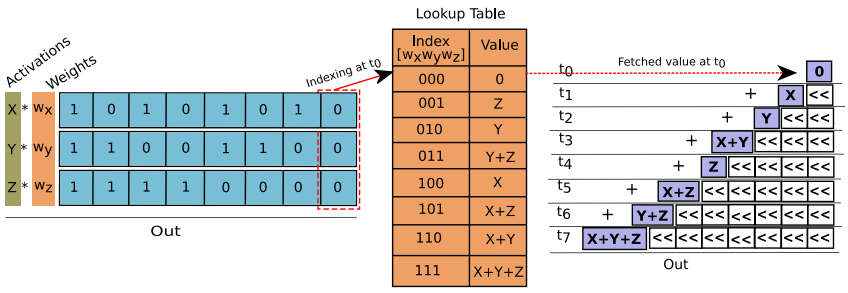

Fig. 21. Concept of LUT based PE (LBPE).
Source: Adapted from [110].

6.3. Reduction of precision

Recently a set of techniques have been proposed that convert the encoding of values to reduce the number of bits required, possibly at the cost of precision. For example, when we convert double precision numbers to single precision numbers, we are reducing their precision. The advantage of reducing precision is a higher computational performance, and a significantly reduced memory footprint but the cost is accuracy. Fortunately for CNNs, this approach can be used very effectively to compress different parts of the model such that the reduction in accuracy is minimized.

Let us trace the evolution of this area. In the initial CNN implementations [44,80,111], a fixed-precision was used for the entire network. Subsequently, researchers realized the performance and storage advantages of precision reduction and started using only the minimum required precision per layer. This sadly necessitated multiple designs for the functional units and hence, increased the design and verification complexity significantly. Then came the era of bit-serial computations (see Section 6.2) to support any level of precision with just one multiplier design. The bit-serial multiplication also helped in eliminating multiplications with ineffectual bits, thereby leading to further speedups. The precision was decided using either static or profile-driven analysis; later approaches did this dynamically as discussed in Section 6.2.

In this regard, Qiu et al. [43] provide a comprehensive analysis of different data precision reduction strategies and their effect on accuracy. They experimented with a dynamic-precision based strategy, where weights and feature maps belonging to different layers of a CNN have different degrees of precision in their binary encoding. Their experiments have shown that reducing the precision of weights and feature maps from 16 bits to 8 or 4 bits can double the performance of the convolution layer with only a 0.04% loss in accuracy for the VGG16 model. Concomitantly, the authors show a 50% reduction in the storage space required for the intermediates, which is significant.

Similar works by Chen et al. [78,79] showed that using a 16-bit multiplier instead of a 32-bit multiplier increased the error rate of the dot product computation by just 0.26% with a 16.4% reduction in the memory footprint. Park et al. [112] experimented with reducing the precision of weights to only 3 bits such that all the weight values of the network fit in the on-chip memory. At the cost of accuracy, this approach resulted in significantly reduced time and energy.

Another work by Park et al. [113] reduces the precision of majority of non-critical computations, while performing the full precision computation of the outlier activations in parallel. This introduces complexity in terms of introducing different multiplier units for outliers and non-outliers. However, the parallel computation, and data reuse allow for significant savings.

7. Industrial designs of CNN accelerators

Unlike academic research where a fair mix of systolic and SIMD designs have been explored for CNN accelerators, recent trends in industry have seen an upsurge in SIMD (single instruction, multiple data), MIMD (multiple instruction, multiple data), and VLIW paradigms. These designs do not prefer off-chip memory because of its high energy and latency requirements. The processing units in a SIMD architecture are arranged in an array of lanes that cannot communicate with each other and hence a global on-chip buffer feeds all the lanes with data. Moreover, instead of designing just a CNN accelerator on the chip, industrial chips are equipped with multiple, heterogeneous processing units to support both inference and training. In order to maximize parallelism, these units are fed with their corresponding instructions simultaneously and hence the independent instructions are packed in a very large instruction word (VLIW). Additionally, some designs are generic enough to provide program level parallelism, where all the processing units are full-fledged processors with their own share of local memory for storage (MIMD paradigm).

Let us discuss some of the recent designs along with their pros and cons.

7.1. SIMD-based design

One of the recent works by Jiao et al. [114] propose a Neural Processing Unit (NPU) primarily for datacenter applications that has four Tensor Engines (TEs) dedicated to accelerating the convolution operations. Each TE is based on a SIMD architecture, containing B lanes of dot product and accumulator units spread across K channels. Apart from the on-chip buffers (present in the earlier designs too) on each TE, there is a large banked on-chip local memory (LM) that is shared among the four TEs. Thus, the introduction of an extra on-chip level in the memory hierarchy in a SIMD design reduces the energy consumption significantly, which would have been huge if all the data was fetched from the external memory. The weights are stored in a compressed form to fit all the data on-chip and are decompressed before their corresponding MAC operations.

7.2. VLIW + SIMD

One of the recent proposals, Groq TSP (tensor streaming processor) [115,116], implements a VLIW processor that follows the SIMD paradigm. The Groq TSP has seven different functional units arranged horizontally. There are a total of 20 such horizontal arrangements (also called a *superlane*) of the functional units stacked over one another. Here, 20 instances of each functional unit need to operate on different data values and each of the seven types of functional units in a superlane fetch and execute a set of VLIW instructions.

They save on the instruction/data memory bandwidth, energy, time, and synchronization logic by following a systolic communication architecture. Each VLIW instruction enters the first superlane and ripples down to the next stacked superlane. Similarly, data enters the leftmost functional unit and ripples to the right.

Hence, the entire burden lies on the compiler to dispatch instructions and data at the right time such that the corresponding instruction

(travelling from top to bottom vertically) as well as its operands (travelling left to right) reach the functional unit at the same time. This leads to a deterministic flow of execution, which is leveraged to reduce the off-chip memory accesses by pre-filling the on-chip memory with the temporally local data.

Similarly, Habana Goya [117] employs eight TPCs (tensor processing cores) for accelerating the convolution operations. Each TPC is a VLIW based SIMD engine. A large on-chip SRAM is shared among all the TPCs, and other processing units to minimize the accesses to the external memory. Additionally, banking allows the designers to effectively partition the large SRAM and perform simultaneous accesses from different banks.

7.3. MIMD paradigm

A recent industrial example of the MIMD paradigm is Graphcore's Intelligent Processing Unit (IPU) [118]. It contains 1216 IPU processing tiles, where each tile is composed of a full-fledged core, and a local SRAM scratchpad with no external memory. It allows for the exploitation of fine-grained parallelism by allowing fast irregular data accesses from the local scratchpad.

7.4. VLIW with systolic computation

Google's TPU-v1 [94] consists of a systolic PE array that performs 8-bit integer multiplications instead of 32-bit floating point multiplications, which saves area, compute time, and energy with a negligible loss in accuracy. In general, a systolic array is able to cut down the energy spent in reading data from SRAM by half [119], and hence systolic arrays are in many cases preferred over SIMD.

To accelerate the training of CNNs, Google developed a supercomputer called *TPU-v2* [119]. The matrix multiply unit follows the same systolic architecture as in TPU-v1, however there are multiple units to accelerate other linear and non-linear operations involved in training a CNN. These units (scalar ALU, vector ALU, vector load, vector store, matrix multiply, and transpose) are supplied with their corresponding instructions packed in a 322-bit VLIW word. Additionally, it uses High Bandwidth Memory (HBM) that provides 20X higher data bandwidth than the SRAMs in TPU-v1. TPU-v3 [119] improves over TPU-v2 by using a higher clock frequency and 2X more HBMs than TPU-v2.

Discussion: In a SIMD design, a large chunk of on-chip area is dedicated to a sophisticated data distribution network that is capable of multicasting, unicasting and broadcasting data. Hence, a systolic array can accommodate more PEs per unit area as compared to a SIMD design. Systolic designs are, however, inferior when it comes to exploiting irregular access patterns because of the rigid data movement patterns between the PEs. On the other hand, SIMD/VLIW designs are able to exploit irregular parallelism.

In general, VLIW is used when multiple units can be run in parallel and there are independent instructions to be executed on these units. For an architecture with a single type of units, systolic arrays and SIMD lanes are the popular architectures. Furthermore, for VLIW based designs, the compiler needs to do the main job of finding the independent instructions and packing them into a single large instruction. Such compiler-dependent paradigms allow deterministic flow of the execution. Owing to this reason, Groq TSP does not take advantage of sparsity in the network, which creates irregularity in the execution.

The large on-chip SRAM in these designs leads to inefficiency in data distribution due to the wiring and interconnect delays; it also leads to thermal issues. Thus, most of these designs with large on-chip SRAMs follow one of the two approaches: divide the large on-chip SRAM into small SRAM arrays and distribute them evenly [118], or allow banking in the large on-chip SRAM to enable concurrent accesses [114,115].

8. Discussion

Table 6 classifies recent work on the basis of the technique that is used: orchestration of dataflow, data reuse, loop unrolling, and the method used for the exploitation of sparsity of data and weights. As discussed in previous sections, sparsity can be exploited by locating either zero-valued activations, weights or zero bits in the binary representation of weights and activations. In general, such sparsity exploitation techniques are also called value-based acceleration techniques because they need to take into account the exact values of the parameters. Our taxonomy in Section 3 classified these works on the basis of three optimization objectives; in this section we present a far more specific classification based on the techniques that are used (refer to Table 6).

Dataflow: In the third column, we classify the works on the basis of the dataflow exploited by them. We observe that early CNN accelerators were developed primarily for dense CNNs and hence followed a uniform dataflow strategy as discussed in Section 5.1, while the latter accelerators were developed for sparse CNNs. In a sparse CNN, the effectual data is scattered and thus new dataflow paradigms are required. These accelerators require efficient routing of effectual data. This non-uniformity of data can be exploited in the following ways: ❶ *Dot-product*: identify the activation vector corresponding to a non-zero weight vector and then compute the dot-product, ❷ *Vector-scalar1*: identify a non-zero activation and multiply it with the weight vector, ❸ *Vector-scalar2*: identify a non-zero activation and multiply it with a non-zero weight vector, and ❹ *Cartesian-product*: identify both non-zero activations and non-zero weights and perform an all-to-all multiplication.

In addition, RS+ is an improved and flexible version of RS that allows data tiling to improve the utilization of the PEs. RS provides good mappings and utilization for dense CNNs while RS+ performs well for both dense as well as sparse CNNs.

Reuse: In the fourth column, we identify the input that is reused by different accelerator designs. As explained in Section 5.1, there is a strong affinity between certain dataflows and types of reuse. For example, WS allows weight reuse, OS allows the reuse of partial sums and RS allows the reuse of inputs, weights and partial sums at the register file. In the case of a sparse CNN, we need to find effectual weights and reuse them for all effectual activations. There are different designs in this space that have different levels of aggressiveness when it comes to locating and eliminating computations involving ineffectual values.

Unrolling: The nested loop structure of a CNN allows us to partition and map different combinations of loop iterators to parallel computation units. The accelerators developed till date decide the loop unrolling based on the kind of parallelism that is required and the type of data to be reused. For the first part of the table (*Early CNN accelerators*), the unrolling factors are discussed in detail in Section 5.1. For the second part of the table involving accesses to sparse CNNs, only the RS dataflow uses a constant unrolling factor, wheres for the rest of the approaches these factors are decided dynamically. For them the aim is to just route the effectual data to the computation units and this data can belong to any loop iterator.

For the third part of the table where we list accelerators that exploit ineffectuality at the level of bits (inside the representation of a number), all the four designs are an extension of the DaDianNao architecture. They use different versions of bit-serial multipliers. Specifically, later works compensate for the performance loss due to bit-serial multipliers by increasing the block size, b_i (defined in Section 2.2). This allows for a higher degree of parallelism, while simultaneously retaining the flexibility to eliminate arithmetic operations involving the ineffectual bits.

Table 6
Types of optimizations exploited by different proposals.

Proposal	Year	Dataflow	Reuse	Unrolling	Architecture	Sparsity exploitation	
			Early CNN accelerators				
NN-X [54]	2014	WS	weights	w, l	1D systolic		
DianNao [79]	2014	NLR	-	a_o, a_i	1D array		
DaDianNao [120]	2014	NLR	-	a_o, a_i	1D array		
ShiDianNao [46]	2015	OS	psums	a_r, a_c	2D matrix		
UCLA [44]	2015	NLR	-	a_o, a_i	1D array		
TPU [94]	2017	WS	weights	w, l	2D systolic		
FlexFlow [57]	2017	flexible	adaptive	flexible	2D matrix		
Origami [58]	2017	WS	weights	w, l	1D array		
YodaNN [60]	2018	WS	weights	w, l	1D array		
			CNN accelerators exploiting sparsity				
						Sparsity exploitation	
						Indexing	**Bitmap**
Eyeriss [121]	2016	RS	weights, act., psums	a_r, w	2D array	act.	-
Cnvlutin [86]	2016	Vector-scalar1	act.	-	1D array	act.	-
Cambricon-X [85]	2016	Dot-product	-	-	1D array	weights	-
Cnvlutin2 [87]	2017	Vector-scalar1	act.	-	1D array	-	act.
SCNN [59]	2017	Cartesian product	act.+weights	-	2D systolic	act.+weights	-
ZeNA [67]	2017	Vector-scalar2	act.	-	1D array	-	act.+weights
Cambricon-S [89]	2018	Dot-product	-	-	1D array	act.+weights	-
SparseCore [68]	2018	Vector-scalar2	act.	-	1D array	-	act.+weights
NullHop [91]	2018	Vector-scalar1	act.	-	1D array	-	act.
TCL [71]	2019	Dot product	-	-	1D array	-	weights
Eyeriss-v2 [84]	2019	RS+	weights, act., psums	a_r, w	2D array	act.+weights	-
Swan [90]	2020	Dot product	act.+weights	-	2D systolic	weights	-
			CNN accelerators exploiting ineffectuality in representation				
						Sparsity exploitation	
						Static Ineff.	**Dynamic Ineff.**
STR [103]	2016	NLR	-	a_o, a_i	1D array	act.	-
D-STR [104]	2017	NLR	-	a_o, a_i	1D array	act.	-
Pragmatic [105]	2017	NLR	-	a_o, a_i	1D array	-	act.
Loom [108]	2018	NLR	-	a_o, a_i	1D array	act. + weights	-

WS → weight stationary, OS → output stationary, NLR → no local reuse, RS → row stationary, RS+ → row stationary plus act. → activations, ineff. → ineffectuality, psums → partial sums/intermediates

Architecture: In the sixth column, we classify the works on the basis of the accelerator architecture employed by different works. These were introduced in Section 2.3.1.

Sparsity exploitation: The last group of columns characterizes the proposals on the basis of the technique employed for the exploitation of sparsity in the network. The primary aim in such architectures is to prevent the computation of ineffectual data and utilize the PEs for the effectual computations only. The first part of the table consists of early accelerators that were not tailored to take care of sparse networks. The second part describes stand alone accelerators that record the locations of effectual data by using either indexing or bitmaps (see Section 5.3). Subsequently, they propose methods to eliminate those computations. The third part of the table (explained in Section 6.2) lists papers that have proposed optimizations to the basic bit-serial multiplier architecture.

8.1. Accuracy of the competing architectures

The accuracy of a CNN accelerator refers to the accuracy of the task for which the CNN is being used. Let us divide our set of optimizations in two classes: ❶ optimizations that perform all the effectual computations and ❷ optimizations that eliminate some of the effectual computations. Effectual computations are those that actually contribute to the value of an output pixel. For example, the multiplication of a zero-valued weight and a non-zero valued activation is ineffectual.

The optimizations such as parallelism, data reuse, sub-expression elimination, removal of ineffectual computations, and removal of redundant operations fall within the first class. These optimizations will have no effect on the accuracy of the CNNs because they rearrange, and remap the computations to the underlying hardware, however the effectual computations remain the same and hence the final result is the same.

The accuracy will be impacted only when the MAC operation or its operands are perturbed in some way. This is done by either changing the operands, removing a few operands, reducing the precision of the operands, truncating the operands, changing the bit-width of the multiplier, or replacing the multiplication operation with some approximate operation. Such kind of optimizations fall within the second class of optimizations. All such proposals are presented in Table 7, where the loss in accuracy is obtained from the results published in the corresponding original papers.

It can be observed from Table 7 that the accuracy loss is within 3% in almost all the cases. Another observation is that the accuracy loss is more pronounced if the small-weight values are removed as compared approximating them to a nearby value.

Some proposals use fixed-point multiplication as opposed to floating point multiplication. This optimization is preferred because it leads to a reduction in the area of the chip with negligible or limited accuracy loss in the classification tasks. Cambricon-X is one such example (see Table 7). Another recent proposal called *Cavoluche* [62] employs a lookup based technique for frequently occurring patterns in the weight-activation pair. However, the accuracy reduction is due to approximating near-identical patterns as the already stored patterns in order to reduce the computations.

8.2. Comparison of the schemes

In this section, we quantitatively compare the discussed proposals subject to some simplifying assumptions. We classified the proposals on the basis of multiple metrics as discussed in Section 8. The type of dataflow is the distinguishing feature among all the accelerators, so we use it for our comparison. Note that we will be comparing the dataflows and not the exact architecture of different proposals because there are many unique features in each proposal and such features cannot be captured by state-of-the-art CNN modeling tools [81].

Table 7
Accuracy loss of several recently proposed techniques.

Year	Proposal	Accuracy loss (%)	Reason
2016	Mem-squeezer [63]	<1.6%	Removal of non-zero valued small weights
2018	SNAPEA [70]	<=3%	Predict the early termination of the computation.
2018	Cambricon-S [89]	<2%	Pruning technique to reduce the irregular sparsity
2018	Outlier-aware [113]	3% for ResNet	Use 4-bit quantization
2018	Cambricon-X [85]	negligible	16-bit fixed-point multiplier
2019	Cavoluche [62]	0.44%	Stores output corresponding to weight-input patterns in a lookup table. Use the lookup results for the approximate patterns too.

8.2.1. Experimental setup

We use the Timeloop [81] tool to find an optimal mapping of the computations in a CNN layer to the PE array. However, Timeloop is not able to model sparse dataflows and hence it can be used for the comparison of only IS, WS, NLR, RS, and OS dataflows. It takes as input the specifications of the hardware architecture and the model of the neural network. It formulates an optimization problem with the desired metric – delay or energy – to compute the optimal mapping. The constraints of the optimization problem are specified in terms of the six loop iterators (see Section 2). The generated mapping is then passed to an analytical model (in Timeloop) to get the PE utilization, energy efficiency, and performance of the mapping.

Our experimental setup is as follows. For comparing these dataflows, the specifications of the hardware architecture and the neural network layer are kept the same. We perform the experiments for two sizes of the PE array: 16×16 and 32×32. These are representative numbers for the PE array; they have been sourced from [81,121,122]. The global buffer is 128 KB in the 16×16 array and 512 KB in the 32×32 array. Each PE has a register file of 16 entries. Since there is no register file in the NLR dataflow, the size of the global buffer is increased to keep the total storage area the same across the dataflows. We simulate the second layer of VGG-16 [97], where the kernel size is 3×3 pixels and the size of the output feature map is 56×56 pixels. The number of input and output channels is 256 each. The batch size and the stride are set to 1.

8.2.2. Performance comparison of dataflows

Table 8 shows the energy efficiency (pJ/MAC), performance (MACs/cycle), and PE utilization of the five dataflows discussed in Section 5.1. We observe in Table 8 that the RS dataflow has the lowest energy per MAC because it has a high degree of data reuse and hence the DRAM accesses are significantly reduced. The NLR dataflow has the highest energy per MAC because it has no local reuse and hence it has the highest number of DRAM accesses. Additionally, the WS dataflow has a higher value of the energy per MAC than the OS dataflow because for each output partial sum, there is a global buffer access while for an OS dataflow these accesses are not there. Another observation is that even with an extensive design space exploration and sufficient relaxations in the constraints, the WS dataflow was not able to achieve full PE utilization. This lower PE utilization for the WS architecture (specifically in TPU) was also observed in [73]. The reason is the inflexible nature of the spatial mapping: either channel direction or planar mapping (refer to Section 4.5).

Similar trends for energy efficiency were obtained by Sze et al. [21] using their in-house energy model. Additionally, the performance in terms of MACs per cycle is the highest for the OS dataflow because it has a higher PE utilization and hence the idle PE cycles are minimized.

Table 8
Comparison of dataflows (data obtained by running Timeloop [81]).

Dataflow	Energy (pJ/MAC)		Performance (MACs/cycle)		PE Utilization
	16 x 16 array	32 x 32 array	16 x 16 array	32 x 32 array	
NLR	96.2	69.3	144	512	0.56
WS	49.1	30.4	144	576	0.56
IS	28	13	254	1009	0.9
OS	16.1	10.2	256	1024	1
RS	6.30	6.27	192	768	0.75

As we increase the size of the PE array and the global buffer, the energy per MAC decreases. This is because a larger amount of data can now fit in the global buffer and hence redundant DRAM accesses for the same data are minimized. To summarize, a higher PE utilization leads to better performance and a dataflow with better reuse opportunities leads to lesser energy consumption.

8.2.3. Performance comparison of sparsity-aware accelerators

To the best of our knowledge, there is no tool (as of 2020) to model the dataflows that exploit sparsity. Thus, we compare these works by deriving and scaling [123] the performance numbers from the original papers and normalizing them with respect to a common baseline, DaDianNao [120]. We choose DaDianNao as the baseline because most of the sparsity-aware accelerators build on the basic DaDianNao architecture. Table 9 shows the performance and energy comparison of the sparsity-aware accelerators with respect to DaDianNao [120]. We observe from Table 9 that the exploitation of dynamic ineffectuality leads to better performance as compared to just using static ineffectuality. Also, the exploitation of static ineffectuality of activations yields better performance if the statically ineffectual bits are determined at runtime as in DynamicSTR [104]. This is because DynamicSTR takes in only the current working set of activations unlike STR that takes into account all the activations of a layer to determine the static ineffectual bits. Thus, DynamicSTR eliminates ineffectuality at a much finer granularity. Another observation is that the performance improvement achieved by exploiting the sparsity in the activations and weights is at par with the improvement achieved by exploiting the ineffectuality in the bits. This implies that even the non-zero values in a neural network have significant ineffectuality. Similar are the trends for energy efficiency. To summarize, exploiting the sparsity of weights and activations along with exploiting the ineffectuality in the non-zero weights and activations can maximize the performance and energy efficiency.

9. Challenges in ASIC design of CNN accelerators

In general, FPGAs are flexible and reconfigurable in nature. GPUs also offer some sort of flexibility due to a large number of cores, caches, and scratchpads. As opposed to FPGAs and GPUs, ASICs are not flexible. Once fabricated, the design is fixed and it cannot be reconfigured. Thus, multiple design decisions need to be made to build an architecture that generalizes well and hence compensates for the large turnaround time and lack of reconfigurability in an ASIC. Let us elaborate.

Size of the PE array: The compiler plays an important role in mapping the computations to the PE array such that it is efficiently utilized for most layers and feature map sizes. For this purpose, the software/compiler and the PE array have to address the following design concerns: ❶ support for pipelining, and ❷ support for folding and scheduling the computations. If multiple layers are executed on a PE array concurrently as in [38,124], a synchronization mechanism among the PEs is necessary to support the pipelining of the layers. It will typically be the case that the feature map sizes are not the same as the size of the PE array. In such cases, we need to partition the set of computations

Table 9
Performance comparison of the proposals w.r.t. [120] (taken from original papers).

Proposal	Performance	Energy Eff.	Sparsity	Methodology
STR [103]	1.8X	1.5X	Static ineffectuality of bits from activations	offline
DynamicSTR [104]	2.6X	2.1X	Static ineffectuality of bits from activations	runtime
Loom [108]	2.8 - 3.2X	2.6 - 2.95X	Static ineffectuality of bits from activations and weights	offline
Pragmatic [105]	2.25 - 4.31X	1.30 - 1.71X	Dynamic ineffectuality of bits from activations	runtime
Cnvlutin [86]	1.37X	1.07X	Sparsity of activations	runtime
Zena [67]	2.74X	N/A	Sparsity of activations and weights	runtime
N/A -> not found in the original paper, Eff. -> efficiency				

and serially execute the partitions (this is known as folding). If some PEs are idle, we can schedule computations from another channel. Such decisions will necessitate the development of complex algorithms in both software and hardware.

Systolic vs SIMD: In an ASIC chip, the interconnects are not flexible and are decided at design time. There are certain area, power, and bandwidth constraints associated with designing an NoC. The NoC design is highly dependent on the type of dataflow supported by the PE array: systolic, semi-systolic, or SIMD. A systolic array allows only the PEs at the edges to access data from the global buffers and hence a lesser bandwidth is required as compared to a SIMD array that allows all the PEs to access data from the global buffers. This increases the area of the NoC in the SIMD design. Thus, given the area constraint, more PEs (and hence compute) can be packed in a systolic array as compared to a SIMD array. Nevertheless, the SIMD array allows the exploitation of irregular parallelism. Thus, a trade-off exists in deciding the appropriate architecture.

Functionality within a PE: There is a design choice involved in deciding the level of functionality in a PE: a single MAC [57,79] or multiple computation units such as a tree of adders or a multiplier array [59]. While a PE with more computation units is able to process a larger chunk of data, the complexity and hence the area and power of the PE increase. Additionally, larger PEs have less synchronization requirements, however, the startup delay and the bandwidth of prefilling the PEs increases. Thus, the advantage of systolic dataflow is not fully exploited. On the contrary, a less complex PE is able to minimize the bandwidth requirement and maximize the reuse among the PEs.

Memory issues: Given the constraints on the on-chip area, it is important to decide the nature of on-chip buffers, their bit-widths, and their size. The size of the on-chip memory decides the degree of temporal locality that can be exploited. However, a large on-chip SRAM increases the power consumption and the access latency. Another important issue is the bit-width of the different types of data involved in a MAC operation: inputs, outputs, and weights. If there is a unified SRAM and the bit-widths of the data types are not integer multiples of the bit-widths of the SRAM, it leads to wasted bandwidth during data accesses. Thus, many works [79] split the on-chip SRAM to accommodate the bit-width requirements of different types of data.

Secondly, the on-chip SRAMs need to support banking to allow simultaneous data accesses by the PE array. This involves a design decision regarding the number of banks for each type of data. Another important issue in designing the memory system is to prevent the stalling of the computation array due to the memory accesses. This requires double buffering, a technique that has a secondary buffer along with the primary buffer and uses them interchangeably for computation and prefetching. This increases the on-chip area at the cost of improved performance.

When the on-chip area is limited, circular buffers [79] are sometimes used instead of double buffering [94]. A circular buffer allows us to access data from one end and prefetch the data to the other end of the queue. Such buffers have to be carefully designed because of the conflicting requirements of accessing data and prefetching. In general, it is sufficient to prefetch as much of data as is needed to process the next row of the output. However, this is dependent on the neural network model's parameters.

Level of reconfigurability: Finally, since the development cycle of an ASIC is large, researchers want to design an ASIC that caters to a large variety of CNNs or ANNs. Owing to this, a certain degree of dynamic reconfigurability is required to be embedded in the chip. However, it would lead to increased area and power owing to the extra logic and wiring.

10. Conclusion

In this paper, we presented a comprehensive survey of CNN accelerator architectures on custom hardware. We proposed two ways to classify related work. In Section 3 we classified the related work based on the performance-enhancing optimization that they target. We took a different look at related work in Section 8 where we classify CNN accelerator architectures on the basis of their design. These are two different ways of looking at the same body of work and convey very different insights. The point to note is that the same design decision can often be used to implement different performance-enhancing optimizations. For example, we can use a dynamic ineffectuality detector to either eliminate computations, or to reduce the memory footprint. In many ways both the taxonomies are orthogonal yet supplementary in nature.

Let us now comment on the directions in which research is progressing. Early research was focused on developing accelerators for CNNs assuming that all the data is effectual and the CNN is dense. Designers thus created rigid designs and dataflows – exploiting the data reuse and deciding the computations to be unrolled and mapped to different PEs. The second phase saw an upsurge of sparse CNNs. The networks grew a lot sparser and hence following a rigid dataflow was a suboptimal choice. This was because these dataflows were not concerned about the number of ineffectual computations that the sparsity leads to. They followed the same rules everywhere for bringing in and multiplying the data. Thus, this phase of research focused on developing dataflows such that only the effectual data is multiplied and read from memory. Most of these were greedy approaches where decisions were taken when a value was read from memory. They were not deferred to a later point in time, and most of these decisions were local to the layer. The interactions between layers was not considered.

The third phase of research focused more on the number of effectual bits in the representation of the data: weights and activations. These bit-level techniques can be used over and above any of the previously proposed techniques. One common observation across the second and the third phase of accelerators was that they allowed sparsity exploitation, be it data sparsity or value sparsity, across both weights and activations.

There is a lot of work that still needs to be done to ensure that we have proper programming models, debugging tools, and compiler infrastructure to ensure that CNN accelerators can be seamlessly integrated in contemporary computing systems. Most of these issues are expected to be addressed in the coming years.

11. Future directions

Some of the open research problems in this domain are described below in brief.

(1) **Novel Memory Hierarchies:** Most contemporary industrial designs have opted for an all-SRAM architecture where the off-chip memory is either non-existent or has a minimal presence. SRAMs have scalability issues [125], and thus sooner or later there will be a need to rethink the memory hierarchy of accelerators. A novel memory hierarchy will most likely be a combination of embedded DRAM, hybrid memory cube technologies, and NVM (non-volatile memory) technologies.

(2) **Processing-in-Memory Paradigms:** Over the next few years, processing-in-memory (PIM) technologies for deep learning accelerators are expected to become very popular. The key idea here is to perform an analog computation to compute the partial sums, where the weights are coded as the resistances of transistors or memristive elements such as ReRAMs.
This paradigm greatly reduces the data movement and allows us to quickly get an estimate of the value of each output pixel. While using NVM elements, we do not need to read in a large array of weights from memory whenever the system is started; the weights will already be stored as the state of the NVM elements and thus the overhead of reading and initializing the network is almost nil.
Analog computation of this nature has its challenges as well. We need an analog to digital converter, which can have quantization errors, and such signals are increasingly susceptible to noise. Additionally, implementing max-pooling in the analog domain is difficult [35].
Sadly, the power consumption of DACs and ADCs (digital to analog converters) has been found to be roughly 50% of the total power consumption in some systems [125]. Some proposals have handled this by breaking the computation into smaller chunks at the level of bits and distributing them over the entire 2-D memory array such that the ADCs require lesser precision; this reduces their power consumption.

(3) **New Algorithms:** There are many new deep learning algorithms that are becoming increasingly popular such as RNNs [126], GANs [127] and Transformers [128]. Current work has primarily focused on inferencing in CNNs. Modern networks such as Transformer networks have an additional self-attention mechanism to embed the information from the neighboring values in the sequence in the current prediction, thereby leading to a complicated network of feed-forward and nonlinear layers. The current architectural optimizations focus mainly on the feed-forward layers while the upcoming networks demand specific optimizations for the non-linear and normalization layers too. Many innovations are required to implement such futuristic networks.

(4) **Multi-tenancy:** In a realistic setting we will have multiple applications that will share a single accelerator. Modern accelerators do not support multi-tenancy and thus their use in a cloud setting is limited. There is a need to enable this and make appropriate provisions for virtual memory and security to enable multiple concurrent applications.

(5) **CNN accelerators for IoT devices:**
Another challenging direction is the development of accelerators for ultra-lighweight IoT devices that are limited by the computational resources and battery capacity. There is a need to create extremely power-efficient versions of CNN inferencing accelerators that provide acceptable levels of accuracy.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments. This work has been funded by the Science and Engineering Research Board (SERB) via grant number CRG/2018/000431.

References

[1] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al., ImageNet large scale visual recognition challenge, Int. J. Comput. Vis. 115 (3) (2015) 211–252.

[2] Krizhevsky Alex, Sutskever Ilya, Hinton, Geoffrey E., ImageNet classification with deep convolutional neural networks, in: Advances in Neural Information Processing Systems, NIPS, 2012, pp. 1097–1105.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Deep residual learning for image recognition, in: IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2016, pp. 770–778.

[4] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, Jason Mars, The architectural implications of autonomous driving: Constraints and acceleration, in: ACM SIGPLAN Notices, Vol. 53, No. 2, ACM, 2018, pp. 751–766.

[5] Narayanan Sundaram, Making Computer Vision Computationally Efficient (Ph.D. dissertation), UC Berkeley, 2012.

[6] Xiaowei Xu, Yukun Ding, Sharon Xiaobo Hu, Michael Niemier, Jason Cong, Yu Hu, Yiyu Shi, Scaling for edge inference of deep neural networks, Nat. Electr. 1 (4) (2018) 216.

[7] Kevin Siu, Dylan Malone Stuart, Mostafa Mahmoud, Andreas Moshovos, Memory requirements for convolutional neural network hardware accelerators, in: IEEE International Symposium on Workload Characterization, IISWC, IEEE, 2018, pp. 111–121.

[8] Andrew Boutros, Sadegh Yazdanshenas, Vaughn Betz, You cannot improve what you do not measure: FPGA vs. ASIC efficiency gaps for convolutional neural network inference, ACM Trans. Reconfig. Technol. Syst. 11 (3) (2018) 20.

[9] Falih S.M. Alkhafaji, Wan Z.W. Hasan, M.M. Isa, N. Sulaiman, Robotic controller: ASIC versus FPGA-a review, J. Comput. Theor. Nanosci. 15 (2018) 1–25.

[10] Moein Khazraee, Lu Zhang, Luis Vega, Michael Bedford Taylor, Moonwalk: Nre optimization in asic clouds, ACM SIGARCH Comput. Archit. News 45 (1) (2017) 511–526.

[11] Shuai Che, Jie Li, Jeremy W Sheaffer, Kevin Skadron, John Lach, Accelerating compute-intensive applications with GPUs and FPGAs, in: 2008 Symposium on Application Specific Processors, IEEE, 2008, pp. 101–107.

[12] Ian Kuon, Jonathan Rose, Measuring the gap between FPGAs and ASICs, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 26 (2) (2007) 203–215.

[13] Shuichi Asano, Tsutomu Maruyama, Yoshiki Yamaguchi, Performance comparison of FPGA, GPU and CPU in image processing, in: 2009 International Conference on Field Programmable Logic and Applications, IEEE, pp. 126–131.

[14] DeepChip, FPFA vs ASIC, 2011, https://www.deepchip.com/downloads/fpga-vs-asic.pdf.

[15] William J. Dally, Yatish Turakhia, Song Han, Domain-specific hardware accelerators, Commun. ACM 63 (7) (2020) 48–57, http://dx.doi.org/10.1145/3361682.

[16] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, Eric S. Chung, Accelerating deep convolutional neural networks using specialized hardware.

[17] Linley Gwennap, Groq rocks neural networks, 2020, https://tractica.omdia.com/automation-robotics/fpgas-challenge-gpus-as-a-platform-for-deep-learning/.

[18] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, Debbie Marr, Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC, in: 2016 International Conference on Field-Programmable Technology, FPT, IEEE, 2016, pp. 77–84.

[19] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al., Can FPGAs beat GPUs in accelerating next-generation deep neural networks? in: International Symposium on Field-Programmable Gate Arrays, FPGA, ACM, 2017, pp. 5–14.

[20] Shihao Wang, Dajiang Zhou, Xushen Han, Takeshi Yoshimura, Chain-NN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks, in: Design, Automation & Test in Europe Conference & Exhibition, DATE, 2017, IEEE, 2017, pp. 1032–1037.

[21] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel S. Emer, Efficient processing of deep neural networks: A tutorial and survey, Proc. IEEE 105 (12) (2017) 2295–2329.

[22] Song Han, Jeff Pool, John Tran, William Dally, Learning both weights and connections for efficient neural network, in: Advances in Neural Information Processing Systems, NIPS, 2015, pp. 1135–1143.

[23] Karl Rupp, 42 years of microprocessor trend data, 2018, https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[24] Nvidia, NVIDIA CUDA C programming guide, version 10.1, 2019, https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[25] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, François Berry, Accelerating CNN inference on FPGAs: A survey, 2018, arXiv preprint arXiv:1806.01683.

[26] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, Huazhong Yang, A survey of FPGA based neural network accelerator, 2017, arXiv preprint arXiv:1712.08934.

[27] Sparsh Mittal, A survey of FPGA-based accelerators for convolutional neural networks, Neural Comput. Appl. (2018) 1–31.

[28] Ahmad Shawahna, Sadiq M. Sait, Aiman El-Maleh, FPGA-based accelerators of deep learning networks for learning and classification: A review, IEEE Access 7 (2018) 7823–7859.

[29] Ahmed Ghazi Blaiech, Khaled Ben Khalifa, Carlos Valderrama, Marcelo AC Fernandes, Mohamed Hedi Bedoui, A survey and taxonomy of FPGA-based deep learning accelerators, J. Syst. Archit. 98 (2019) 331–345.

[30] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, Weimin Zheng, Performance analysis of GPU-based convolutional neural networks, in: International Conference on Parallel Processing, ICPP, IEEE, 2016, pp. 67–76.

[31] Sparsh Mittal, A survey on optimized implementation of deep learning models on the NVIDIA jetson platform, J. Syst. Archit. 97 (2019) 428–442.

[32] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, Yuan Xie, PRIME: a novel processing-in-memory architecture for neural network computation in ReRAM-based main memory, in: ACM SIGARCH Computer Architecture News, Vol. 44, No. 3, IEEE Press, 2016, pp. 27–39.

[33] Linghao Song, Xuehai Qian, Hai Li, Yiran Chen, PipeLayer: A pipelined reram-based accelerator for deep learning, in: International Symposium on High Performance Computer Architecture, HPCA, IEEE, 2017, pp. 541–552.

[34] Fengbin Tu, Weiwei Wu, Shouyi Yin, Leibo Liu, Shaojun Wei, RANA: towards efficient neural acceleration with refresh-optimized embedded DRAM, in: International Symposium on Computer Architecture, ISCA, IEEE Press, 2018, pp. 340–352.

[35] Sparsh Mittal, A survey of ReRAM-based architectures for processing-in-memory and neural networks, Mach. Learn. Knowl. Extr. 1 (1) (2018) 75–114.

[36] Sumanth Umesh, Sparsh Mittal, A survey of spintronic architectures for processing-in-memory and neural networks, J. Syst. Archit. 97 (2019) 349–372.

[37] Boris Murmann, Daniel Bankman, E Chai, Daisuke Miyashita, Lita Yang, Mixed-signal circuits for embedded machine-learning applications, in: Asilomar Conference on Signals, Systems and Computers, ACSSC, IEEE, 2015, pp. 1341–1345.

[38] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, Vivek Srikumar, ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars, in: International Symposium on Computer Architecture, ISCA, IEEE Press, 2016, pp. 14–26.

[39] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, Lin Zhong, RedEye: analog convNet image sensor architecture for continuous mobile vision, in: ACM SIGARCH Computer Architecture News, Vol. 44, No. 3, IEEE Press, 2016, pp. 255–266.

[40] Sparsh Mittal, A survey on modeling and improving reliability of DNN algorithms and accelerators, J. Syst. Archit. 104 (2020) 101689.

[41] Jung-Woo Chang, Suk-Ju Kang, Optimizing FPGA-based convolutional neural networks accelerator for image super-resolution, in: 2018 23rd Asia and South Pacific Design Automation Conference, ASP-DAC, IEEE, 2018, pp. 343–348.

[42] Dongseok Im, Donghyeon Han, Sungpill Choi, Sanghoon Kang, Hoi-Jun Yoo, Dt-cnn: Dilated and transposed convolution neural network accelerator for real-time image segmentation on mobile devices, in: 2019 IEEE International Symposium on Circuits and Systems, ISCAS, IEEE, 2019, pp. 1–5.

[43] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al., Going deeper with embedded FPGA platform for convolutional neural network, in: International Symposium on Field-Programmable Gate Arrays, FPGA, ACM, 2016, pp. 26–35.

[44] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, Jason Cong, Optimizing FPGA-based accelerator design for deep convolutional neural networks, in: International Symposium on Field-Programmable Gate Arrays, FPGA, ACM, 2015, pp. 161–170.

[45] Jian Cheng, Pei-song Wang, Gang Li, Qing-hao Hu, Han-qing Lu, Recent advances in efficient computation of deep convolutional neural networks, Front. Inf. Technol. Electr. Eng. 19 (1) (2018) 64–77.

[46] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, Olivier Temam, ShiDianNao: shifting vision processing closer to the sensor, in: ACM SIGARCH Computer Architecture News, Vol. 43, No. 3, ACM, 2015, pp. 92–104.

[47] Nikolay Petkov, Systolic parallel processing, in: Advances in Parallel Computing, Vol. 5, North-Holland, Elsevier Sci. Publ., Amsterdam, ISBN: 0444887695, 1993.

[48] H.T. Kung, Why systolic architectures? in: Advanced Computer Architecture, IEEE Computer Society Press, 1986, pp. 300–309.

[49] Jason Cong, Bingjun Xiao, Minimizing computation in convolutional neural networks, in: International Conference on Artificial Neural Networks, Springer, 2014, pp. 281–290.

[50] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, Soheil Ghiasi, Design space exploration of FPGA-based deep convolutional neural networks, in: Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, IEEE, 2016, pp. 575–580.

[51] Clément Farabet, Cyril Poulet, Jefferson Y Han, Yann LeCun, CNP: An FPGA-based processor for convolutional networks, in: International Conference on Field Programmable Logic and Applications, FPL, IEEE, 2009, pp. 32–37.

[52] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, A dynamically configurable coprocessor for convolutional neural networks, in: ACM SIGARCH Computer Architecture News, Vol. 38, No. 3, ACM, 2010, pp. 247–257.

[53] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, Hans Peter Graf, A programmable parallel accelerator for learning and classification, in: International Conference on Parallel Architectures and Compilation Techniques, PACT, ACM, 2010, pp. 273–284.

[54] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, Eugenio Culurciello, A 240 g-ops/s mobile coprocessor for deep neural networks, in: IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPRW, 2014, pp. 682–687.

[55] Jonghoon Jin, Vinayak Gokhale, Aysegul Dundar, Bharadwaj Krishnamurthy, Berin Martini, Eugenio Culurciello, An efficient implementation of deep convolutional neural networks on a mobile coprocessor, in: International Midwest Symposium on Circuits and Systems, MWSCAS, IEEE, 2014, pp. 133–136.

[56] Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, Xiaowei Li, C-Brain: A deep learning accelerator that tames the diversity of CNNs through adaptive data-level parallelization, in: Proceedings of the Design Automation Conference, DAC, IEEE, 2016, pp. 1–6.

[57] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, Xiaowei Li, FlexFlow: A flexible dataflow accelerator architecture for convolutional neural networks, in: International Symposium on High Performance Computer Architecture, HPCA, IEEE, 2017, pp. 553–564.

[58] Lukas Cavigelli, Luca Benini, Origami: A 803-GOp/s/w convolutional network accelerator, IEEE Trans. Circuits Syst. Video Technol. 27 (11) (2017) 2461–2475.

[59] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, William J Dally, SCNN: An accelerator for compressed-sparse convolutional neural networks, in: International Symposium on Computer Architecture, ISCA, IEEE, 2017, pp. 27–40.

[60] Renzo Andri, Lukas Cavigelli, Davide Rossi, Luca Benini, YodaNN: An architecture for ultralow power binary-weight cnn acceleration, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 37 (1) (2018) 48–60.

[61] Kartik Hegde, Jiyong Yu, Rohit Agrawal, Mengjia Yan, Michael Pellauer, Christopher W Fletcher, UCNN: exploiting computational reuse in deep neural networks via weight repetition, in: International Symposium on Computer Architecture, ISCA, IEEE Press, 2018, pp. 674–687.

[62] Ying Wang, Shengwen Liang, Huawei Li, Xiaowei Li, A none-sparse inference accelerator that distills and reuses the computation redundancy in CNNs, in: Proceedings of the 56th Annual Design Automation Conference 2019, 2019, pp. 1–6.

[63] Ying Wang, Huawei Li, Xiaowei Li, Re-architecting the on-chip memory subsystem of machine-learning accelerator for embedded devices, in: Proceedings of the International Conference on Computer-Aided Design, ICCAD, ACM, 2016, p. 13.

[64] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al., ESE: efficient speech recognition engine with compressed LSTM on FPGA, 2016, arXiv preprint arXiv:1612.00694.

[65] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, William J Dally, EIE: Efficient inference engine on compressed deep neural network, in: International Symposium on Computer Architecture, ISCA, IEEE Press, 2016, pp. 243–254.

[66] Hyeonuk Kim, Jaehyeong Sim, Yeongjae Choi, Lee-Sup Kim, A kernel decomposition architecture for binary-weight convolutional neural networks, in: Proceedings of the Design Automation Conference, DAC, ACM, 2017, p. 60.

[67] Dongyoung Kim, Junwhan Ahn, Sungjoo Yoo, ZeNA: Zero-aware neural network accelerator, IEEE Des. Test 35 (1) (2018) 39–46.

[68] Sharad Chole, Ramteja Tadishetti, Sree Reddy, SparseCore: An accelerator for structurally sparse CNNs, in: The Conference on Systems and Machine Learning, SysML, 2018.

[69] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, Tao Li, Prediction based execution on deep neural networks, in: International Symposium on Computer Architecture, ISCA, IEEE, 2018, pp. 752–763.

[70] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K Gupta, Hadi Esmaeilzadeh, SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks, in: International Symposium on Computer Architecture, ISCA, IEEE, 2018, pp. 662–673.

[71] Alberto Delmas, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Andreas Moshovos, Bit-tactical: Exploiting ineffectual computations in convolutional neural networks: Which, why, and how, 2018, arXiv preprint arXiv:1803.03688.

[72] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, Andreas Moshovos, Bit-tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks, in: International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, ACM, 2019, pp. 749–763.

[73] Bosheng Liu, Xiaoming Chen, Ying Wang, Yinhe Han, Jiajun Li, Haobo Xu, Xiaowei Li, Addressing the issue of processing element under-utilization in general-purpose systolic deep learning accelerators, in: Proceedings of the 24th Asia and South Pacific Design Automation Conference, 2019, pp. 733–738.

[74] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, Vivienne Sze, Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks, IEEE J. Solid-State Circuits 52 (1) (2016) 127–138.

[75] Hans P Graf, Srihari Cadambi, Venkata Jakkula, Murugan Sankaradass, Eric Cosatto, Srimat Chakradhar, Igor Dourdanovic, A massively parallel digital learning processor, in: Advances in Neural Information Processing Systems, NIPS, 2009, pp. 529–536.

[76] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, Yann LeCun, NeuFlow: A runtime reconfigurable dataflow processor for vision, in: IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPRW, IEEE, 2011, pp. 109–116.

[77] Maurice Peemen, Arnaud AA Setio, Bart Mesman, Henk Corporaal, Memory-centric accelerator design for convolutional neural networks, in: International Conference on Computer Design, ICCD, IEEE, 2013, pp. 13–19.

[78] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, Olivier Temam, Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning, in: ACM Sigplan Notices, Vol. 49, No. 4, ACM, 2014, pp. 269–284.

[79] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, Olivier Temam, DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning, in: International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, ACM, 2014, pp. 269–284.

[80] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, Olivier Temam, A high-throughput neural network accelerator, IEEE Micro 35 (3) (2015) 24–32.

[81] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, Joel Emer, Timeloop: A systematic approach to dnn accelerator evaluation, in: 2019 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, IEEE, 2019, pp. 304–315.

[82] Aysegul Dundar, Jonghoon Jin, Vinayak Gokhale, Berin Martini, Eugenio Culurciello, Memory access optimized routing scheme for deep networks on a mobile coprocessor, in: High Performance Extreme Computing Conference, HPEC, IEEE, 2014, pp. 1–6.

[83] Yu-Hsin Chen, Joel Emer, Vivienne Sze, Eyeriss v2: A flexible and high-performance accelerator for emerging deep neural networks, 2018, arXiv preprint arXiv:1807.07928.

[84] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, Vivienne Sze, Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices, IEEE J. Emerg. Sel. Top. Circuits Syst. 9 (2) (2019) 292–308.

[85] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, Yunji Chen, Cambricon-x: An accelerator for sparse neural networks, in: International Symposium on Microarchitecture, MICRO, IEEE, 2016, pp. 1–12.

[86] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Andreas Moshovos, Cnvlutin: ineffectual-neuron-free deep neural network computing, in: ACM SIGARCH Computer Architecture News, Vol. 44, No. 3, IEEE Press, 2016, pp. 1–13.

[87] Patrick Judd, Alberto Delmas, Sayeh Sharify, Andreas Moshovos, Cnvlutin2: Ineffectual-activation-and-weight-free deep neural network computing, 2017, arXiv preprint arXiv:1705.00125.

[88] Richard Wilson Vuduc, Automatic Performance Tuning of Sparse Matrix Kernels, Vol. 1, University of California, Berkeley, 2003.

[89] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, Yunji Chen, Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach, in: International Symposium on Microarchitecture, MICRO, IEEE, 2018, pp. 15–28.

[90] Bosheng Liu, Xiaoming Chen, Yinhe Han, Ying Wang, Jiajun Li, Haobo Xu, Xiaowei Li, Search-free accelerator for sparse convolutional neural networks, in: 2020 25th Asia and South Pacific Design Automation Conference, ASP-DAC, IEEE, 2020, pp. 524–529.

[91] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, et al., Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps, IEEE Trans. Neural Netw. Learn. Syst. 30 (3) (2018) 644–656.

[92] Mark Horowitz, 1.1 Computing's energy problem (and what we can do about it), in: International Solid-State Circuits Conference Digest of Technical Papers, ISSCC, IEEE, 2014, pp. 10–14.

[93] Yongming Shen, Michael Ferdman, Peter Milder, Escher: A CNN accelerator with flexible buffering to minimize off-chip transfer, in: International Symposium on Field-Programmable Custom Computing Machines, FCCM, IEEE, 2017, pp. 93–100.

[94] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al., In-datacenter performance analysis of a tensor processing unit, in: International Symposium on Computer Architecture, ISCA, IEEE, 2017, pp. 1–12.

[95] Hyeonuk Sim, Jason H. Anderson, Jongeun Lee, XOMA: Exclusive on-chip memory architecture for energy-efficient deep learning acceleration, in: Proceedings of the 24th Asia and South Pacific Design Automation Conference, 2019, pp. 651–656.

[96] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, Christos Kozyrakis, Tangram: Optimized coarse-grained dataflow for scalable NN accelerators, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 807–820.

[97] Karen Simonyan, Andrew Zisserman, Very deep convolutional networks for large-scale image recognition, 2014, arXiv preprint arXiv:1409.1556.

[98] Subhasis Das, Song Han, Neuraltalk on embedded system and GPU-accelerated RNN, in: CVA Group, Stanford University, 2018.

[99] Charu C. Aggarwal, Mansurul A. Bhuiyan, Mohammad Al Hasan, Frequent pattern mining algorithms: A survey, in: Frequent Pattern Mining, Springer International Publishing, 2014, pp. 19–64.

[100] Andreas Moshovos, Jorge Albericio, Patrick Judd, Alberto Delmás Lascorz, Sayeh Sharify, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Value-based deep-learning acceleration, IEEE Micro 38 (1) (2018) 41–55.

[101] Andreas Moshovos, Jorge Albericio, Patrick Judd, Alberto Delmas, Sayeh Sharify, Mostafa Mahmoud, Tayler Hetherington, Milos Nikolic, Dylan Malone Stuart, Kevin Siu, et al., Identifying and exploiting ineffectual computations to enable hardware acceleration of deep learning, in: International New Circuits and Systems Conference, NEWCAS, IEEE, 2018, pp. 356–360.

[102] Andreas Moshovos, Jorge Albericio, Patrick Judd, Alberto Delmás Lascorz, Sayeh Sharify, Zissis Poulos, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, Exploiting typical values to accelerate deep learning, Computer 51 (5) (2018) 18–30.

[103] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, Andreas Moshovos, Stripes: Bit-serial deep neural network computing, in: International Symposium on on Microarchitecture, MICRO, IEEE, 2016, pp. 1–12.

[104] Alberto Delmas, Patrick Judd, Sayeh Sharify, Andreas Moshovos, Dynamic stripes: Exploiting the dynamic precision requirements of activation values in neural networks, 2017, arXiv preprint arXiv:1706.00504.

[105] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, Andreas Moshovos, Bit-pragmatic deep neural network computing, in: International Symposium on Microarchitecture, MICRO, ACM, 2017, pp. 382–394.

[106] Sayeh Sharify, Alberto Delmas Lascorz, Mostafa Mahmoud, Milos Nikolic, Kevin Siu, Dylan Malone Stuart, Zissis Poulos, Andreas Moshovos, Laconic deep learning inference acceleration, in: International Symposium on Computer Architecture, ISCA, ACM, 2019, pp. 304–317.

[107] Mostafa Mahmoud, Kevin Siu, Andreas Moshovos, Diffy: A Déjà vu-free differential deep neural network accelerator, in: International Symposium on Microarchitecture, MICRO, IEEE, 2018, pp. 134–147.

[108] Sayeh Sharify, Alberto Delmas Lascorz, Kevin Siu, Patrick Judd, Andreas Moshovos, Loom: exploiting weight and activation precisions to accelerate convolutional neural networks, in: Proceedings of the Design Automation Conference, DAC, ACM, 2018, p. 20.

[109] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, Hadi Esmaeilzadeh, Bit fusion: bit-level dynamically composable architecture for accelerating deep neural networks, in: International Symposium on Computer Architecture, ISCA, IEEE Press, 2018, pp. 764–775.

[110] Jinmook Lee, Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sangyeob Kim, Hoi-Jun Yoo, UNPU: An energy-efficient deep neural network accelerator with fully variable weight bit precision, IEEE J. Solid-State Circuits 54 (1) (2018) 173–185.

[111] Murugan Sankaradas, Venkata Jakkula, Srihari Cadambi, Srimat Chakradhar, Igor Durdanovic, Eric Cosatto, Hans Peter Graf, A massively parallel coprocessor for convolutional neural networks, in: International Conference on Application-Specific Systems, Architectures and Processors, ASAP, IEEE, 2009, pp. 53–60.

[112] Jinhwan Park, Wonyong Sung, FPGA based implementation of deep neural networks using on-chip memory only, in: International Conference on Acoustics, Speech and Signal Processing, ICASSP, IEEE, 2016, pp. 1011–1015.
[113] Eunhyeok Park, Dongyoung Kim, Sungjoo Yoo, Energy-efficient neural network accelerator based on outlier-aware low-precision computation, in: International Symposium on Computer Architecture, ISCA, IEEE, 2018, pp. 688–698.
[114] Yang Jiao, Liang Han, Rong Jin, Yi-Jung Su, Chiente Ho, Li Yin, Yun Li, Long Chen, Zhen Chen, Lu Liu, et al., 7.2 A 12 nm programmable convolution-efficient neural-processing-unit chip achieving 825TOPS, in: 2020 IEEE International Solid-State Circuits Conference, ISSCC, IEEE, 2020, pp. 136–140.
[115] Linley Gwennap, Groq rocks neural networks, 2020, https://www.linleygroup.com/newsletters/newsletter_detail.php?num=6110&year=2020&tag=3.
[116] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, et al., Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads.
[117] Linley Gwennap, Groq rocks neural networks, 2020, https://habana.ai/wp-content/uploads/pdf/habana_labs_goya_whitepaper.pdf.
[118] Zhe Jia, Blake Tillman, Marco Maggioni, Daniele Paolo Scarpazza, Dissecting the graphcore IPU architecture via microbenchmarking, 2019, arXiv preprint arXiv:1912.03413.
[119] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, David Patterson, A domain-specific supercomputer for training deep neural networks, Commun. ACM 63 (7) (2020) 67–78, http://dx.doi.org/10.1145/3360307.
[120] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al., Dadiannao: A machine-learning supercomputer, in: International Symposium on Microarchitecture, MICRO, IEEE, 2014, pp. 609–622.
[121] Yu-Hsin Chen, Joel Emer, Vivienne Sze, Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks, in: International Symposium on Computer Architecture, ISCA, IEEE, 2016, pp. 367–379.
[122] Nvidia, NVDLA open source project, 2017.
[123] Wei Huang, Karthick Rajamani, Mircea R Stan, Kevin Skadron, Scaling with design constraints: Predicting the future of big chips, IEEE Micro 31 (4) (2011) 16–29.
[124] Manoj Alwani, Han Chen, Michael Ferdman, Peter Milder, Fused-layer CNN accelerators, in: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, IEEE, 2016, pp. 1–12.
[125] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, Efficient Processing of Deep Neural Networks, Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2020.
[126] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, Yoshua Bengio, Learning phrase representations using RNN encoder-decoder for statistical machine translation, 2014, arXiv preprint arXiv:1406.1078.
[127] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio, Generative adversarial nets, in: Advances in Neural Information Processing Systems, 2014, pp. 2672–2680.
[128] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, Illia Polosukhin, Attention is all you need, in: Advances in Neural Information Processing Systems, 2017, pp. 5998–6008.

Diksha Moolchandani received a bachelor's degree in electronics and communication engineering from Indian Institute of Information Technology Jabalpur, Jabalpur, MadhyaPradesh, India.

She is currently a Research Scholar with the School of IT, Indian Institute of Technology Delhi, New Delhi, India. Her current research interests span the areas of architecture design and acceleration techniques for computer vision and image processing applications, designing convolutional neural network accelerators, and machine learning applications to computer architecture. She is a student member of the IEEE.

Anshul Kumar received the Ph.D. degree in Computer Aided Design of Digital Systems from Indian Institute of Technology Delhi, New Delhi, India and is currently an Emeritus Professor with the Computer Science and Engineering Department at IIT Delhi. He has held visiting appointments at USC, University of Edinburgh, KTH Stockholm, and EPFL.

Prof. Kumar's current research interests are VLSI synthesis, embedded systems design methodology and high performance computer architectures and he has published more than 100 research papers in reputed journals and proceedings of refereed international conferences. He has been a consultant to Gateway Design Automation (now Cadence Design Systems), Technology Parks Ltd, ST Microelectronics and Poseidon Design Systems. Prof. Kumar has been associated with the annual International Conference on VLSI Design since its inception in 1985 and has served as its General Co-Chair in 2009.

Prof. Kumar co-founded the start-up company called Kritikal Solutions and served as its Hon. Chairman, Hon. Director and Mentor. He serves as the Technical Advisory Board Member of a start-up company VirtuQ. He received the ACM Transaction on Design Automation of Electronic Systems (TODAES) 2007 Best Paper Award.

Smruti R. Sarangi received the B.Tech. degree in computer science from the Indian Institute of Technology Kharagpur, Kharagpur, India, and the M.S. and Ph.D. degrees from the University of Illinois at Urbana–Champaign, Champaign, IL, USA.

He is currently an Usha Hasteer Chair Professor with the Computer Science and Engineering Department, Indian Institute of Technology Delhi, New Delhi, India, where he holds a joint appointment with the Department of ELectrical Engineering and the School of Information Technology. He has published extensively in peer reviewed conferences and journals, holds 5 U.S. patents, and has filed 3 Indian patents. He has authored the popular undergraduate textbook on computer architecture entitled Computer Organisation and Architecture (McGrawHill). His current research interests include processor reliability, architectural support for operating systems, and processors for the Internet of Things. Dr. Sarangi is a member of the ACM and IEEE.